LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

KRIŠS RAUHVARGERS

PROGRAMMATŪRAS IZPILDES VIDES TESTĒŠANA

PROMOCIJAS DARBS
DATORZINĀTNES DOKTORA GRĀDA IEGŪŠANAI

Apakšnozare: datu apstrādes sistēmas un datortīkli

Zinātniskais vadītājs
profesors Dr. sc. comp. JĀNIS BIČEVSKIS

RĪGA - 2010

Anotācija

Promocijas darbā risināta praktiska programmatūras dzīves cikla uzlabošanas problēma, tiecoties samazināt programmatūras uzturēšanas fāzē nepieciešamo cilvēka darba un zināšanu ieguldījumu. Darbā piedāvāta jauna programmatūras būves tehnoloģija, kas sniedz iespēju programmatūru izstrādāt tā, lai tā varētu patstāvīgi pārbaudīt savu „dzīvotspēju" dotajā izpildes vidē.

Darbā izstrādātā tehnoloģija paredz programmatūras papildināšanu ar vides konfigurācijas aprakstu jeb profilu, kas satur nepieciešamās prasības programmatūras veiksmīgai izpildei. Izmantojot profilā aprakstītās prasības, programmatūra patstāvīgi un pilnībā automātiski veic izpildes vides pārbaudes katrā tās lietošanas sesijā un gadījumā, ja vide nav piemērota darbam, darbu nemaz neuzsāk. Piedāvātā tehnoloģija sastāv no trīs daļām: prasību aprakstīšanas valodas; moduļiem prasību pārbaudei un metodikas ārējās vides pārbaudes veikšanai.

Atšķirībā no programmatūrā iebūvēto testu metodēm, kur paredzēta fiksētu vides parametru pārbaude iepriekš definētos punktos, šajā darbā piedāvātais risinājums ļauj vides testēšanas pārbaudes algoritmus aprakstīt ārpus darījumu programmatūras. Tādējādi ir iespējams veidot vienotu programmatūras izstrādātājiem pieejamo vides pārbaužu bāzi, kas atkārtoti izmantojama dažādu programmatūras risinājumu būvē.

Darbā apskatīti visi piedāvātās tehnoloģijas strukturālie aspekti – sintakse jeb valodas, kas lietojamas izpildes profila aprakstīšanai, semantika, kas valodā definētajām konstrukcijām piekārto funkcionālo jēgu un metodika, kas definē veidus, kā vides pārbaudes lietojamas un kā vides testēšanas metodika ieviešama.

Tehnoloģija galvenokārt paredzēta programmatūras izstrādes uzņēmumiem, kas nodarbojas ar pasūtītāja vēlmēm pielāgotas programmatūras izstrādi, taču pielietojama arī plaša patēriņa programmatūras izstrādes vajadzībām.

Piedāvātā metodika praksē aprobēta divos programmatūras izstrādes uzņēmumos, kur vienā no tiem tiek veikta programmatūras izstrāde lietošanai uzņēmuma iekšienē, taču izmantots vēsturiski plašs pielietojamo tehnoloģiju klāsts, savukārt otrs uzņēmums izstrādā daudziem lietotājiem paredzētu programmatūru, kuras mērķa instalācijas vide nav iepriekš zināma. Tehnoloģijas praktiskā aprobācija apliecina tās dzīvotspēju.

Abstract

The promotion thesis is a research on practical improvement of software life-cycle, which aims to decrease the intellectual and labour inputs of human operators during the software maintenance phase. The paper proposes a new software development technology, which allows for development of independent software that can perform independent tests to verify its "viability" in the given execution environment.

The proposed solution is to supplement the software with an execution profile – a document containing requirements on environmental settings that are required for successful execution of the software. Using the knowledge contained in the profile document, the software can independently and fully automatically perform tests on the execution environment at every execution session. In case if the environment is not suitable for execution, the software interdicts its business functionality from execution. The proposed technology consists of three parts: the language for requirements' description, test modules for environment validation, and a methodology for execution of environmental tests.

In contrast to regular built in testing methods, which are constructed to test a fixed set of environmental parameters at known life-cycle points, the method proposed in this paper allows to describe the test algorithms outside of the business software. By doing so, the reuse of environment testing functions becomes feasible and hence a common collection of environment tests can be created, which can later be used across all the systems developed in a software development organization.

The paper is a study on all the structural aspects of the proposed technology – the syntax of languages that may be used for description of software execution profiles; the semantics which fit functional meaning to the constructs of description language; and the methodology which defines ways of use of the environment tests and ways of implementation of the technology in a software development environment.

The proposed methodology is mainly focused on organizations that develop custom software, but can also be adapted for development of off-the-shelf software.

The method has been approbated in practice in two software development organizations, one of which mainly develops software for in-house use while the other one is a customised software developer. The results of the approbation show the usefulness of the proposed technology.

Satura rādītājs

Ievads **6**

1. Izvēlētās problēmas identifikācija un aktualitāte **11**

1.1. Izvēlētās problēmas aktualitāte **11**

1.1.1. Vides homogenitāte 11

1.1.2. Sistēmu sadarbība 12

1.1.3. Tehnoloģiju attīstība 15

1.1.4. Datoru konfigurāciju atšķirības 16

1.2. Problēmas identifikācija **17**

2. Saistītie pētījumi **19**

2.1. Autonomās un *paš-* sistēmas **19**

2.2. Viedo tehnoloģiju programmatūra **25**

2.3. Iebūvēto testu metode **28**

2.3.1. Apgalvojumi (assertions) 28

2.3.2. Automatizētā vienībtestēšana 29

2.3.3. Citi iebūvēto testu veidi 30

2.4. Instalāciju un konfigurāciju aprakstīšana **32**

2.4.1. Instalāciju deskriptors 33

2.4.2. Programmatūras izvietošanas deskriptors 37

2.5. Servisa kvalitātes mērīšanas metodes **40**

2.6. Saistīto pētījumu rezultātu pielietojamība vides automātiskai testēšanai **41**

3. Piedāvātais vides testēšanas problēmas risinājums **45**

3.1. Vispārīgie principi **45**

3.2. Izpildes vides testēšanas arhitektūra **48**

3.3. Metodes izklāsts **51**

3.3.1. Prasību aprakstīšanas valoda 51

3.3.2. Izpildes prasības un prasību pārbaudes moduļi 56

3.3.3. Izpildes parametru pielietošana prasību aprakstīšanai 62

3.3.4. Izpildes profila versiju veidošana 66

3.4. Programmatūras izpildes profila veidošana **68**

3.5. Metodes adaptācijas principi **72**

4. Praktiskie metodes pielietojumi **75**

4.1. Pielietojums organizācijā, kurā tiek veikta iekšēja IS izstrāde **75**

4.1.1. Par organizāciju 75

4.1.2. Pašlaik izmantotie uzturēšanas procesu uzlabojumi 76

4.1.3. Pārmaiņu nepieciešamība 77

4.1.4. Izvēlētais uzlabojumu ieviešanas modelis 79
4.2. Metodes pielietošana sadalītu sistēmu integritātes pārbaudei 81
4.2.1. SharePoint platforma 81
4.2.2. SharePoint infrastruktūra 82
4.2.3. SharePoint vidē darbināmu lietotņu izstrāde 83
4.2.4. Programmatūras uzstādīšanas principi apskatāmajā uzņēmumā 84
4.2.5. SharePoint bāzētu sistēmu uzstādīšanas problēmas 84
4.2.6. Programmatūras izpildes profila lietošana aprakstīto problēmu risināšanai 86
4.2.7. Vides pārbaužu pielietošana serveru fermā 88
4.2.8. Turpmākā izpēte 90
4.3. Vides testu pielietošana problēmu ziņotāja programmatūrā 90
4.3.1. SysTest rīka lietošanas scenāriji 91
4.3.2. SysTest rīks kā vides pārbaudes tehnoloģijas realizācija 92
4.3.3. Vides testētāja ieviešana uzņēmumā 93
Rezultāti 95
Darbā lietotie apzīmējumi un saīsinājumi 98
Saīsinājumi 98
Termini 99
Bibliogrāfija 101
Pielikumi 108
1. pielikums **Programmatūras prasību profila realizācija, izmantojot *SDD* sintaksi 109**
Koda piemērs 1.1. Prasību faila instance 110
Koda piemērs 1.2. Pielāgots resursu deskriptors 112
2. pielikums **Programmatūras prasību profila realizācija, izmantojot speciāli definētu valodu 114**
Koda piemērs 2.1. Prasību dokuments 115
Prasību aprakstīšanas valodas sintakse, izmantojot XML Schema 116
Koda piemērs 2.3. XML shēma 116
3. pielikums **Parametrizējamu konfigurācijas lielumu saistību aprakstīšana, izmantojot XML 119**
Koda piemērs 3.1. Parametru aprakstīšana 121
4. pielikums **Testu koordinatoram pieejamo vides pārbaudītāju apraksts ar XML 124**

IEVADS

Programmatūras sarežģītības pieaugums neapšaubāmi atstāj iespaidu uz programmatūras izstrādi un uzturēšanu, palielinot nepieciešamo darba ieguldījumu abos šajos procesos. Operētājsistēmu un izstrādes platformu veidotāji piedāvā dažādus karkasus un citas tehnoloģijas, kas paātrina datorsistēmu izstrādi. Izmantojot izstrādes karkasu, programmētāji var vairāk uzmanības pievērst darījuma prasību realizēšanai un mazāk rūpēties par fona servisiem, kas lietotājam tiešā veidā nav pieejami. Piemēram, Java platformas *EJB* konteineri vai *Microsoft COM+* tehnoloģija sniedz transakciju servisus, datu saglabāšanas (angl. *persistence*) risinājumus un citas iespējas, kuru izstrāde, neizmantojot šīs tehnoloģijas, būtu darbietilpīga. Līdzīgi – pastāv arī daudzi servisi, kuriem nav piešķirti specifiski nosaukumi, tie ir iekļauti operētājsistēmā vai izpildes vidē – piemēram, datortīkla protokola *TCP/IP* piekļuves bibliotēka ir operētājsistēmas sastāvdaļa, bet augstāka abstrakcijas līmeņa tehnoloģijas kā *.Net Remoting* vai *CORBA* ir realizētas kā izpildes karkasa funkcija.

Būvējot datorsistēmas, kas paļaujas uz dažādu karkasu sniegtiem servisiem, tiek vienkāršota sistēmu izstrāde, bet vienlaikus novērojams, ka tiek sarežģīta sistēmu uzturēšana. Katras tehnoloģijas pielietošana izraisa arvien jaunu konfigurējamu atribūtu rašanos. Piemēram, ja programmatūras būvē izmantoti *COM+* servisi, tā pakļauta šīs tehnoloģijas konfigurācijas iestatījumiem, kādi būs izvēlēti mērķa datorā. Piemēram, tikai vienas „izvēles rūtiņas" vērtības izmainīšana var nozīmēt to, ka attiecīgais dators vairs neatbalsta attālinātos *COM* izsaukumus Kaut arī šķietami neliela, šāda nianse atstāj būtisku iespaidu uz programmas izpildi. Mēģinot programmatūru darbināt vidē, kas neatbilst izstrādes laikā iecerētajai, vairs nav iespējams garantēt sistēmas normālu izpildi. Sastopoties ar šāda tipa kļūdām, programmas bieži vien rezultātus interpretē nepareizi vai sniedz neatbilstoši vienkāršus kļūdu paziņojumus (piemēram, „Kļūda pieslēdzoties datubāzei" – fakta konstatācija ir pareiza, taču sistēma nesniedz priekšstatu par kļūdas iemesliem).

Neveiksmīgās *COM+* konfigurācijas un līdzīgu piemēru esamība liecina par to, ka netriviāla izmēra datorsistēmu uzturēšanai nepieciešamas dziļas zināšanas par sistēmu arhitektūru un sistēmā izmantotajām tehnoloģijām. Tādējādi datorsistēmu administratoru zināšanas par dažādu organizācijā izmantotu sistēmu nepieciešamajām darba konfigurācijām kļūst par vērtīgu organizācijas īpašumu. Lai vienkāršotu sistēmu administratoru darbu un nodrošinātu uzņēmuma darbības nepārtrauktību arī personāla sastāva izmaiņu gadījumos, sistēmu darba konfigurācijas mēdz uzglabāt arī rakstiski – gan kā sistēmu uzstādīšanas dokumentāciju, gan specifiski pielāgotu strukturētu dokumentu veidā (piemērs – *MS Excel* tabula, kurā pārskaitītas sistēmām nepieciešamās failu sistēmas takas un lietotāji, kuriem

nepieciešama piekļuve šīm takām). Šādu dokumentu uzturēšana atbilstoši reālajai situācijai prasa sistemātisku pieeju, turklāt jāņem vērā saspringtie darba apstākļi, kādos tiek veikts sistēmu administratora ikdienas darbs [**1**].

Atsevišķi pētījumi liecina, ka tieši sistēmu konfigurācijas sarežģītība bieži vien kļūst par iemeslu datorsistēmu atteicēm. Piemēram, veicot pētījumu, kurā vairākiem datorsistēmu administratoriem tiek uzdots atjaunot *RAID* disku masīvu, var novērot, ka, kaut arī izpildītājiem bijušas nepieciešamās zināšanas šāda darba izpildei, ne vienmēr darbs noritējis veiksmīgi, turklāt ap 40 procentos neveiksmju par iemeslu ir bijusi tieši operatora, nevis aparatūras kļūda. Pētījuma autori secina – „ticami, ka problēmas iemesls ir tas, ka mūsdienu datorsistēmu sarežģītā uzbūve ir pārāk grūti izprotama, bet IT operatoriem vai pārvaldniekiem lēmumi jāpieņem dažu sekunžu laikā" [**2**,**3**].

Šajā darbā pētītas iespējas, kā vienkāršot programmatūras sistēmu uzturēšanu, samazinot datorsistēmu administratoriem nepieciešamo zināšanu apjomu un pēc iespējas lielu zināšanu daļu pārnesot uz pašu datorsistēmu. Šis pētījums organiski iekļaujas un papildina Latvijas Universitātes Datorikas fakultātē veiktos pētījumus *viedo tehnoloģiju* jomā [**4**,**5**,**6**], kuru mērķis ir, iebūvējot programmatūrā dažādus palīgrīkus, uzlabot atsevišķas programmatūras dzīves cikla fāzes. Šis darbs padziļināti pēta vienu no viedo tehnoloģiju jomām – izpildes vides konfigurācijas automātisku izvērtēšanu.

Darbā veiktais pētījums sniedz praktisku risinājumu vienai no *IBM* autonomo sistēmu manifestā [**7**] aprakstītajām vīzijām – pašdiagnostikai jeb programmatūras spējai analizēt savas darba spējas, lai, novērojot grūtības, varētu piemērot atbilstošu stratēģiju normālu darba spēju atjaunošanai. Pašdiagnostikas virziens sastāv no vairākiem apakšvirzieniem, to skaitā arī datu kvalitātes pārbaudes, drošības pārraudzība u.c., taču šajā darbā kā galvenais virziens apskatīta programmatūras „pašsajūtas" atkarība no apkārtējās vides konfigurācijas piemērotības programmatūras prasībām.

Galvenā nostāja šajā darbā ir – programmatūrai pašai jāspēj noteikt, vai apkārtējā vide tai ir piemērota. Lai to noteiktu, programmatūrai jāveic izpildes vides testi, turklāt to vēlams darīt pilnīgi automātiski, neprasot lietotāja mijiedarbību ar programmatūru. Atšķirībā no pašlaik praksē bieži lietotā modeļa, kad programmatūras vide tiek pārbaudīta tikai instalācijas laikā, šādas pārbaudes jāveic regulāri – ja iespējams, katrā izpildes sesijā pirms darba uzsākšanas. Šādas pieejas ieviešana ļauj izvairīties no datorsistēmu infrastruktūrā veiktu izmaiņu negatīva iespaida uz tajā izmantotās programmatūras darbību.

Lai programmatūra varētu veikt izpildes vides testus, tai nepieciešams „etalons", ar kuru salīdzināt izpildes vides atsevišķu parametru mērījumus, turklāt tai nepieciešamas zināšanas, kuri vides parametri var atstāt iespaidu uz tās darbu. Faktu, ka kāda vides parametra

neatbilstība etalonam var atstāt iespaidu uz programmatūras darbību, šajā darbā apzīmē ar jēdzienu *atkarība* (no izpildes vides), savukārt etalonā iekļautu ierobežojumu, kas attiecināma uz kāda parametra vērtību – par programmatūras *prasību* (angl. *requirement*) pret izpildes vidi. Automātiski veicot apkārtējās vides parametru salīdzināšanu ar etalonā noteiktajiem, tiek aizstāts ilglaicīgs, monotons cilvēka manuāls darbs, kas būtu nepieciešams vides īpašību pārbaudei.

Piedāvātā izpildes vides testēšanas tehnoloģija kā dabisku pieņem faktu, ka programmatūras atkarības no izpildes vides tiek atklātas pamazām visa programmatūras dzīves cikla laikā – daļa no tām ir zināmas jau projektēšanas fāzē, būtiskāko daļu (galvenās prasības) iespējams noskaidrot un fiksēt izstrādes laikā, taču daļa atkarību var tikt noskaidrotas arī vēlākos dzīves cikla posmos. Lai saglabātu programmatūras spēju patstāvīgi testēt izpildes vidi, arvien paplašinot pārbaudāmo prasību klāstu, šajā darbā piedāvāts tās prasību etalonu glabāt ārpus programmatūras koda. Tiek piedāvāts prasības apkopot programmatūras izpildes prasību *profilā* – dokumentā, kas pievienots programmatūras izpildāmajiem moduļiem un tiem seko visā programmatūras dzīves cikla laikā. Šī ideoloģija detalizēti aprakstīta [**8**,**9**].

Process, kurā programmatūra patstāvīgi testē apkārtējo vidi (tas ir, salīdzina vides parametrus ar izpildes profilā pieprasītajām vērtībām), nav vienkāršs un ir grūti unificējams. To viegli saprast, iedomājoties analoģiskas cilvēka veiktas manuālas darbības – piemēram, vides parametru „teksta redaktora *OpenOffice* versija" un „vai dators pieslēgts datortīklam" vērtību noskaidrošanas algoritmi ir pilnīgi dažādi. Līdzīgi atšķiras arī pārbaužu algoritmi katra noteikta prasību veida patiesuma noskaidrošanai. Tādēļ šajā darbā piedāvāts katras prasību klases pārbaudei pielietot atsevišķu pārbaudes moduli, kas saturētu attiecīgās prasību klases pārbaužu algoritmiskos risinājumus. Veicot vides testēšanu, programmatūra automātiski piemeklētu, kurš testēšanas modulis piemērots kurai izpildes prasībai.

Kopumā piedāvātā metode aprakstāma šādi:

- programmatūras izstrādes laikā tiek fiksētas prasības pret izpildes vidi,
- nododot programmatūru tālāk lietošanas vai testēšanas vidē, fiksētās prasības tiek apkopotas vienā programmatūras izpildes prasību profilā,
- tiek izstrādāti nepieciešamie vides pārbaudes moduļi, lai varētu programmatiski pārbaudīt profilā definētās prasības,
- lietošanas vidē, uzsākot programmatūras darba sesiju, tiek veiktas automātiskās vides pārbaudes; šīs pārbaudes realizē, izmantojot vides testēšanas moduļus.

Šādas tehnoloģijas praktiska ieviešana programmatūras izstrādes uzņēmumā ir netriviāls uzdevums, kam nepieciešams relatīvi liels sākotnējais ieguldījums – jāizstrādā

programmatūras bāze, kas ļautu viegli veikt vides testēšanu, jāspecificē valoda, kādā tiks veidoti programmatūras izpildes profili, jāizstrādā pārbaudes moduļi populārākajām programmatūras prasību klasēm. Būtiskākie soļi un apsvērumi, kas jāņem vērā, ieviešot tehnoloģiju uzņēmumā, aprakstīti [**10**].

Sagaidāms, ka piedāvātā risinājuma ieviešana vislabākos rezultātus sniegs uzņēmumos, kur tiek veikta individuālās programmatūras izstrāde, savukārt vide, kurā programmatūra tiek izvietota, ir nehomogēna un vides konfigurācija – mainīga. Pilotprojekta veidā šis risinājums jau ieviests kādā Latvijas finanšu institūcijā, kur programmatūras izstrāde tiek veikta organizācijas iekšienē. Šī sākotnējā pieredze un izvēlētie risinājumi aprakstīti [**11**].

Promocijas darba pamatdaļa ir strukturēta četrās nodaļās.

Darba pirmā nodaļa apskata nehomogēnā vidē izvietotas programmatūras problēmu cēloņus, tos klasificējot vairākās būtiskākajās problēmu grupās – atsevišķu datorsistēmu infrastruktūras resursu koplietošana, sistēmu savstarpējās sadarbības organizēšana, tehnoloģiju attīstības rezultātā arvien pieaugošais konfigurējamo vienību skaits un datoru konfigurāciju atšķirības. Tāpat pirmajā nodaļā identificētas galvenās problēmas, kas risināmas šajā promocijas darbā – programmatūras atteiču skaita samazināšana, ieviešot preventīvu vides pārbaudes mehānismu, kurā programmatūra patstāvīgi veic izpildes vides testus.

2. nodaļa sniedz detalizētu pārskatu par saistītajiem pētījumiem programmatūras patstāvības uzlabošanā, kā būtiskāko no tiem apskatot *IBM* korporācijas definēto autonomās skaitļošanas iniciatīvu, kas definē vispārēju virzību uz patstāvīgas programmatūras būvi, bet atsevišķu uzmanību pievēršot arī Latvijā radītajai *viedās* programmatūras ideoloģijai, kas tiecas rast praktiskus risinājumus patstāvīgas programmatūras būvei. Kā saistītas tēmas tiek apskatītas arī vairākas iebūvēto testu metodes, kas tehniski līdzīgas, taču ideoloģiski atšķiras no šajā pētījumā piedāvātās metodes. Otrs saistīto tehnoloģiju bloks ir programmatūras konfigurāciju aprakstīšanai veltītie pētījumi. Šeit kā radniecīgi virzieni apskatīti divi programmatūras instalāciju rīku izstrādātāju definēti secīgi standarti – *IUDD* (angl. *Installable Unit Deployment Descriptor*) un *SDD* (angl. *Solution Deployment Descriptor*). Nodaļas nobeigumā apkopotas tās saistīto tehnoloģiju īpašības, kas varētu rast pielietojumu šajā darbā pētītās problēmas risinājumā.

3. nodaļa sniedz detalizētu piedāvātās tehnoloģijas izklāstu, aprakstot gan vispārējo vides testēšanas infrastruktūras uzbūvi, gan detalizēti katru tās komponenti. Nodaļas nobeigumā apkopoti principi, pēc kuriem vadoties, metodi iespējams ieviest kā programmatūras dzīves cikla sastāvdaļu.

4. nodaļa veltīta darbā formulētās tehnoloģijas aprobācijas rezultātu aprakstīšanai. Tiek apskatīti trīs dažādi metodes aprobācijas scenāriji.

Pirmais scenārijs apraksta metodes ieviešanu uzņēmumā, kur visa programmatūras izstrāde tiek veikta „uz vietas" un programmatūra tiek izvietota labi strukturētā uzņēmuma datorsistēmu vidē, kurā pastāv liels skaits datorsistēmu savstarpējo saišu. Galvenais mērķis, ieviešot metodi šajā vidē – izvairīties no programmatūras atteicēm, ko izraisījušas vides konfigurācijas izmaiņas, piemēram, atsevišķu sistēmas daļu pārnešana uz citu datoru.

Otrais scenārijs apskata piedāvātās metodes izmantošanu citam mērķim – atsevišķas programmatūras komponentes integritātes pārbaudei pēc programmatūras jauninājumu uzstādīšanas. Šajā gadījumā, pielietojot darbā piedāvāto tehnoloģiju, paredzēts izvairīties no attiecīgās platformas īpatnībām un darījumu programmatūras versiju atjaunošanas procesu padarīt paredzamāku.

Trešajā tehnoloģijas aprobācijas scenārijā tā pielietota programmatūras izstrādes uzņēmumā kā viedo tehnoloģiju infrastruktūras sastāvdaļa. Sistēmas pārbaudītājs – viena no datorsistēmu komponentēm, kas iekļauta visās organizācijas izstrādātajās sistēmās – būvēts tā, lai veicamās sistēmas pārbaudes realizētu pēc šajā darbā aprakstītās tehnoloģijas. Rezultātā iegūtā programma ir viegli pielāgojama jebkurai organizācijas izstrādātajai darījumu sistēmai.

1. IZVĒLĒTĀS PROBLĒMAS IDENTIFIKĀCIJA UN AKTUALITĀTE

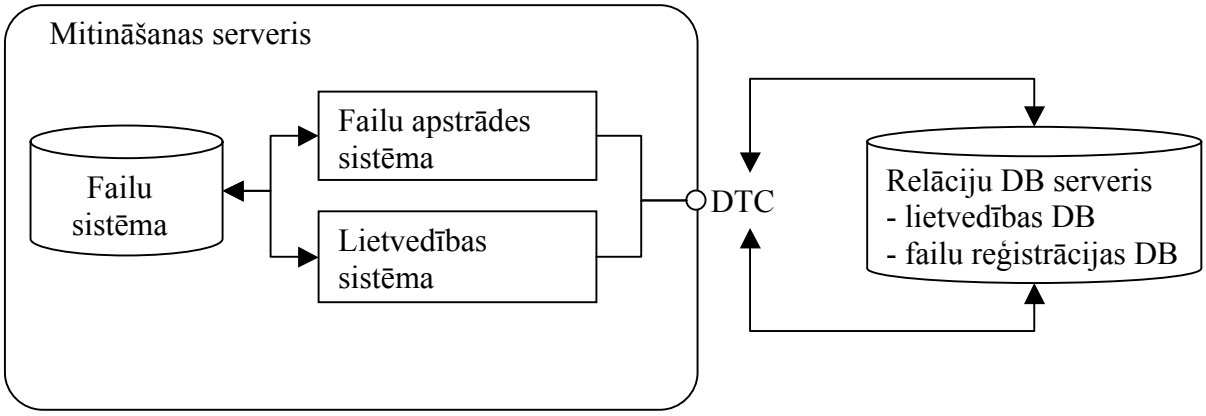
1.1. Izvēlētās problēmas aktualitāte

1.1.1. Vides homogenitāte

Gandrīz jebkuram uzņēmumam nobriestot savā darījumu sfērā, aug vai paplašinās tā veikto funkciju skaits. Ja sākotnēji gandrīz visi pieejamie resursi tiek patērēti pamatfunkciju izpildei, vēlāk pamazām attīstās arī lietas, kas mazam uzņēmumam bijušas sekundāras – piemēram, nelielā uzņēmuma algu aprēķins un atvaļinājumu uzskaite var tikt veikta, izmantojot vienkāršu elektronisko izklājlapu, taču, pieaugot darbinieku skaitam, šāds risinājums kļūst neefektīvs. Sastopoties ar dažādām „augšanas grūtībām", uzņēmēji tiecas vienkāršot un automatizēt blakus uzdevumus, lai varētu efektīvāk veikt galvenos darbus. Līdzīgi, lai uzņēmumu padarītu konkurētspējīgāku, bieži vien nepieciešams uzlabot un automatizēt arī uzņēmuma pamatfunkcijas. Viens no populāriem automatizācijas risinājumiem ir informācijas tehnoloģiju pielietošana, iegādājot vai no jauna izstrādājot attiecīgajam mērķim piemērotu programmatūru. Veiksmīgi pielietojot iegūto infrastruktūru un attīstot uzņēmumu tālāk, ar laiku rodas jaunas papildu funkcijas vai jaunas darba uzlabošanas un automatizācijas iespējas, kuru pielietošana iepriekš nav bijusi racionāla vai pat nav bijusi iespējama.

Iepriekšminētais liecina, ka jebkura uzņēmuma IT infrastruktūra ir augoša – pieaug gan aparatūras, gan pielietotās programmatūras daudzums. Pat tad, ja lietoti adekvāti IT vadības risinājumi, kas panāk kontrolētu un plānotu infrastruktūras attīstību [**12**], iegūtā vide bieži vien nav homogēna, tās atsevišķi segmenti veidojas no dažādu IT sistēmu atsevišķām struktūrām. Piemēram, apskatīsim uzņēmumu, kura pamatfunkcija ir citu uzņēmumu grāmatvedības veikšana. Lai nodrošinātu klientu datu konfidencialitāti, datu apmaiņa ar klientiem tiek veikta šifrēti. Uzlabojot pakalpojumu kvalitāti, uzņēmums pasūta un tam tiek izstrādāta specializēta darījumu sistēma, kas veic saņemto failu pārbaudi, atšifrēšanu, interpretēšanu un saglabāšanu relāciju datubāzē, no kuras dati pieejami tālākai apstrādei. Tāpat sistēma veic pretējo darbību, sagatavotos atskaišu failus šifrējot un automātiski nosūtot klientam. Tādējādi tiek aizstāts agrāk nepieciešamais manuālais atšifrēšanas darbs un būtiski samazināti datu noplūdes riski, jo šifrēšanas atslēgas turpmāk nav nepieciešams izsniegt darbiniekiem individuālai lietošanai. Vienlaicīgi ar šīs sistēmas izstrādi, pieaugot organizācijas iekšējo dokumentu aprites plūsmai, uzņēmumā tiek ieviesta centralizēta lietvedības un dokumentu aprites sistēma. Arī šī sistēma datus glabā relāciju datubāzē. Lai

taupītu līdzekļus, abas šīs sistēmas var tikt mitinātas vienotā sistēmu uzturēšanas vidē, piemēram, instalējot tās vienā un tajā pašā serverī un datu glabāšanai izmantojot vienu un to pašu relāciju datubāzu serveri, tādējādi optimizējot aparatūras resursu patēriņu (skat. ***1.1****. att.*). Šādā modelī atsevišķie infrastruktūras elementi ne tikvien darbojas vienotā vidē, bet arī izmanto vienus un tos pašus vides servisus. Piemēram, apskatīsim gadījumu, kad abas sistēmas ir būvētas tā, ka tās izmanto operētājsistēmas piedāvāto transakciju koordinēšanas servisu (attēlā – DTC). Mainot DTC konfigurāciju ar mērķi ietekmēt vienas sistēmas (piemēram, lietvedības sistēmas) darbu, automātiski arī otra sistēma tiks pakļauta mainītajai konfigurācijai. Ja sistēmas būvētas pēc vienotiem principiem, ir iespējams, ka iegūtā konfigurācija nebūs pretrunīga un failu apstrādes sistēmas darbs tādējādi netiks traucēts. Tomēr, veicot šādas izmaiņas, rodas jautājums, kā pārliecināties par to, vai otra jaunā konfigurācija būs tāda, kas neietekmē citu sistēmu darbu.



1.1. att. **Sistēmu mitināšana vienotā izpildes vidē**

Tādējādi, ne tikai atsevišķu komponenšu, bet arī sistēmas attīstība kopumā, izvirza problēmu par ārējās vides atbilstību katras komponentes prasībām.

1.1.2. Sistēmu sadarbība

Turpinot attīstīt uzņēmuma elektronisko vidi, bieži rodas nepieciešamība veikt horizontālu sistēmu integrāciju, veidojot dažādas „saites" starp uzņēmumam piederošajiem resursiem. Par saiti šajā kontekstā var uzskatīt tādu dažādu sistēmu konfigurāciju, kur vienas sistēmas izejas dati kļūst par citas sistēmas ieejas datiem (piemēram, iepriekšējā nodaļā aplūkotajā organizācijas IT infrastruktūrā – ja dokumenti, kas glabāti lietvedības sistēmā, tiktu šifrēti un nosūtīti uzņēmuma klientam ar failu apmaiņas sistēmas palīdzību, starp abām sistēmām veidotos saite, jo sistēmas būtu jākonfigurē tā, lai lietvedības sistēma saglabātu datus kādā noteiktā takā, savukārt šifrēšanas sistēma – tos no turienes saņemtu). Līdzīgi, par saiti uzskatāma arī situācija, kur kāda informācijas sistēma būvēta, paļaujoties uz citas

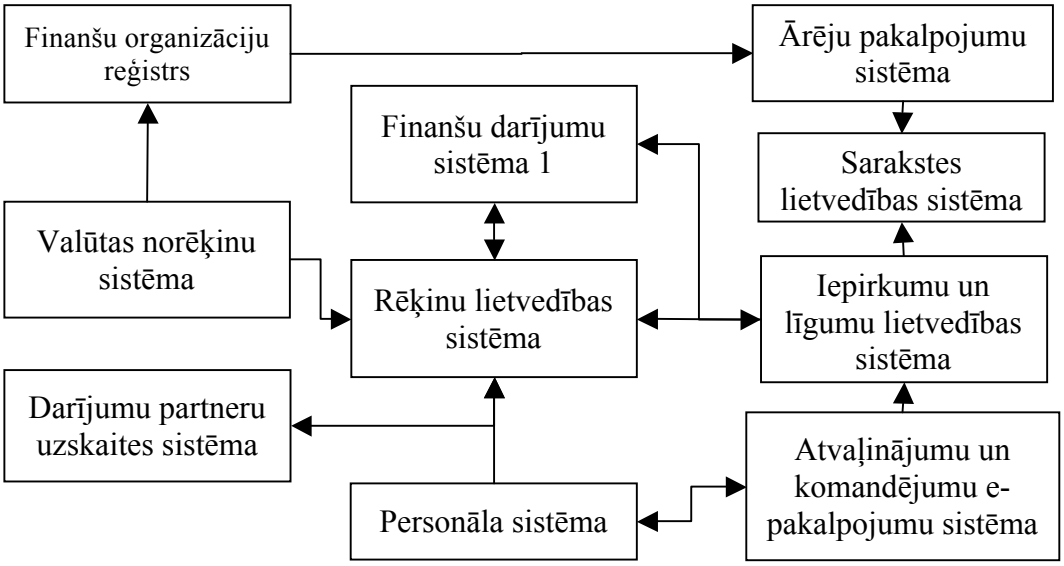
sistēmas noteiktu konfigurāciju vai arhitektūru. (Šeit netiek apskatītas datorsistēmas, kas jau būvētas, apzinoties nepieciešamību tās savstarpēji saistīt ar citām, t.i., apzinoties SOA (angl. *Service-Oriented Architecture*) ideoloģiju, bet ņemot vērā, ka praksē eksistē daudz sistēmu, kas būvētas kā individuālas informācijas krātuves).

Praktiski veicot sistēmas arhitektūras auditu kāda uzņēmuma dokumentu aprites sistēmas rēķinu lietvedības daļā, tika konstatēts, ka tās darbam nepieciešamas pat vairākas ārējas sistēmas:

- norēķinu sistēma, kurai nosūta sagatavoto rēķinu izpildei; divvirzienu datu apmaiņa, kur norēķinu sistēma apstiprina failu saņemšanu un reģistrēšanu, tiek veikta failu sistēmas līmenī,
- cita finanšu sistēma, no kuras tiek nolasīti aktuālie valūtu kursi (nepieciešams rēķinu summu konvertēšanai latos); saite starp sistēmām veidota kā SQL līmeņa tabulu replikācija,
- organizācijā lietotā personāla uzskaites sistēma (nepieciešama, lai noskaidrotu darbinieku, kurš drīkst apstiprināt attiecīgo rēķinu – amatpersonu prombūtnes laikā šo uzdevumu pilda amatpersonu vietnieki vai pienākumu izpildītāji); saite starp sistēmām veidota kā SQL līmeņa tabulu replikācija,
- klientu uzskaites sistēma, kurā reģistrētas visas partnerorganizācijas (nepieciešama, lai nebūtu atkārtoti jāreģistrē organizācijas, no kurām saņemti rēķini); saite ar šo sistēmu veidota failu sistēmas līmenī.

Organizācijā, kurā tika veikts šis audits, vienlaikus tiek darbinātas daudzas informācijas sistēmas – gan tādas, kas paredzētas organizācijas pamatfunkciju atbalstam, gan palīgfunkciju atbalsta sistēmas, kā iepriekšējā piemērā apskatītās. Organizācijā kopumā veidojas konfigurācijas struktūra, kas līdzīga ***1.2***. *att.* parādītajai (attēlā redzams tikai neliels fragments no auditētās organizācijas informācijas sistēmu infrastruktūras).

Var secināt, ka informācijas sistēmu savstarpējo saišu veidošanās ir dabiska uzņēmumos, kas veic relatīvi specifiskas funkcijas un tādēļ visu sistēmu nepieciešamo sistēmu komplektu nav iespējams iegūt vienlaicīgi (piemēram, iegādājoties integrētu uzņēmuma vadības sistēmu). Automatizējot darījumu procesus, kas ir savstarpēji saistīti, arī atbalstošās informācijas sistēmas var būt nepieciešams savstarpēji saistīt, lai izvairītos no datu vairākkārtējas reģistrēšanas.



1.2. att. **Organizācijā izmantotu sistēmu savstarpējo saišu shēma (fragments)**

Šādu informācijas sistēmu saišu īpatnība ir tā, ka to pastāvēšana nav acīmredzama – tās eksistē dažādu atsevišķu sistēmu darba konfigurācijas aprakstos (gan datoram lasāmos – konfigurācijas faili, konfigurācijas tabulas datubāzē utt., gan cilvēkam lasāmos – sistēmu uzstādīšanas apraksti, sistēmu administratoru pieraksti par sistēmu konfigurācijām), tās var ilglaicīgi veiksmīgi darboties, bet ar laiku var zust zināšanas par to, ka (un – kādēļ) attiecīgā saite pastāv.

Veicot uzturēšanas darbus, saiti iespējams netīši likvidēt – piemēram, pārvietojot kādu sistēmu uz serveri ar lielāku veiktspēju, jāņem vērā, ka var pastāvēt citas sistēmas, kas ar to saistītas un meklē sistēmu pēc iepriekšējā servera nosaukuma. Situāciju sarežģī fakts, ka neeksistē algoritmisks pārbaudes mehānisms, kas ļautu noskaidrot, vai tikko veiktās izmaiņas nav atstājušas nevēlamu iespaidu uz kādu no programmatūras sistēmām, kas tiek darbinātas attiecīgajā mitināšanas vidē. Tādējādi iespējams, ka sistēmu saišu problēmas tiek novērotas tikai brīdī, kad saistītā programmatūra pēc izmaiņu veikšanas tiek iedarbināta pirmo reizi. Šķietami vienkāršs risinājums būtu katru reizi pēc izmaiņu veikšanas iedarbināt visas saistītās sistēmas, taču šāda prakse nebūtu pieņemama no darījumu viedokļa. Daudzās darījumu procesu atbalsta sistēmās, jo sevišķi tajās, kas veidotas paša procesa automatizēšanai, nevis procesu iznākumu (dokumentu) apstrādei, nav iespējams veikt „izpildi testa režīmā", lai pārliecinātos, vai sistēma korekti darbojas. Šādi testi varētu atstāt neparedzamu iespaidu uz attiecīgo darījuma procesu. Tādējādi, ja darījuma process tiek izmantots reti, tomēr tas ir būtisks procesa īpašniekam, ir nepieciešams veids, kā automātiski noskaidrot, vai programmatūras konfigurācija ir adekvāta, nedarbinot pašu informācijas sistēmu.

Kaut arī IS saišu pašreizējo konfigurāciju iespējams noskaidrot relatīvi viegli (analizējot sistēmu konfigurācijas failus vai uzstādīšanas aprakstus), kopējās zināšanas par infrastruktūras konfigurāciju ir dziļi saistītas ar personāla zināšanām par atsevišķo sistēmu uzbūvi, tādējādi pakļaujot organizāciju dažādiem ar personālu saistītiem riskiem. Rakstā par programmatūras mitināšanas pagātni, tagadni un nākotni [**13**] A. Dearle secina – sistēmas uzturēšanas laikā ir būtiski zināt pēc iespējas daudz par sistēmas uzbūvi, pat – ka zināšanas par sistēmu savstarpējām saitēm ir tikpat būtiski programmatūras metadati. Kaut arī pastāv daudzi praksē lietojami sistēmu instalācijas procesu aprakstīšanas rīki kā, piemēram, *Macrovision InstallShield* vai *NullSoft NSIS* (angl. *NullSoft Scriptable Install System*) vai vispārīgāki sistēmu struktūras un uzstādīšanas procesu aprakstīšanas standarti, kā *IUDD* vai *SDD* (aprakstīti attiecīgi šī darba 2.4.1 un 2.4.2 nodaļās), tie paredzēti atsevišķas sistēmas saistību ar instalācijas vidi aprakstīšanai, daļēji ignorējot sistēmu saistību faktu.

Tādējādi, starp datorsistēmām pastāvošās loģiskās informācijas saites var viegli kļūt par sistēmu uzturēšanas problēmu cēloni.

1.1.3. Tehnoloģiju attīstība

Organizācijai pamazām nobriestot un augot tās izmēriem, palielinās arī tās pamatfunkciju un tās iekšienē notiekošo palīgprocesu skaits. Lai uzlabotu organizācijas veiktspēju, arī klāt nākušie procesi tiek automatizēti, izstrādājot tiem piemērotas atbalstošās informācijas sistēmas. Taču jāņem vērā, ka jaunie palīgprocesi bieži vien ir sarežģīti, jo tie saistīti ar citu procesu pārraudzību vai uzlabošanu (piemēram, kvalitātes un risku pārvaldība u.c.), tādējādi funkcionālās prasības šo procesu atbalsta informācijas sistēmām kļūst arvien sarežģītākas.

Lai ar esošajiem programmatūras izstrādes resursiem (organizācijas iekšienē veiktas izstrādes gadījumā) adekvāti ātri izstrādātu nepieciešamo funkcionalitāti, tiek izmantotas arvien jaunas palīgkomponentes un tehnoloģijas, kas vienkāršo atsevišķas sistēmas dzīves cikla daļas. Iespējams izmantot gan atsevišķas programmatūras bibliotēkas, piemēram, datu šifrēšanai, programmatūras datu slāņa vienkāršošanai, transakciju atbalstam u.c., gan arī veselas sistēmu sagataves, kuras tikai jāpapildina ar nepieciešamo darījumu funkcionalitāti, piemēram, universālās dokumentu glabātuves, tādas kā *DocLogix* [**14**] vai *Microsoft Sharepoint Server* [**15**].

Tā kā nav iespējams vienlaicīgi pārveidot visu jau eksistējošo darījumu sistēmu kodu, lai tas turpmāk izmantotu tikai jaunākās tehnoloģijas, darbināšanas vidē vienlaicīgi ir jāuztur gan agrāk, gan nesen izstrādātas sistēmas. Tādējādi sistēmu uzturētājiem nepieciešamas ļoti

specifiskas zināšanas, turklāt atmiņā jāpatur katras atsevišķas darījumu sistēmas ieviešanas un uzturēšanas īpatnības.

Situāciju var raksturot ar sekojošu piemēru – organizācija jau ilgstoši lieto dokumentu pārvaldības satura indeksēšanas un meklēšanas dzinēju ar tīmekļa saskarni, kas realizēta kā *Microsoft* ASP.Net 1.0 tīmekļa lietojumprogramma. Programmatūras izstrādes firma, kas savulaik veidojusi šo sistēmu, ir tikusi likvidēta un produkts nekad netiks pārveidots uz jaunāku vidi. Pēc programmatūras produkta iegādes tas papildināts ar organizācijai būtisku papildu moduli, lai būtu iespējams indeksēt arī organizācijā izmantotās lietvedības dokumentu glabātuves datus. Tādējādi šī meklēšanas sistēma kļuvusi par būtisku un organizācijā bieži izmantotu sistēmu, pie kuras turklāt lietotāji ir labi pieraduši un prot to izmantot (šādas sistēmas dēvē par mantotām sistēmām [**16**]). Vienlaicīgi šajā pašā organizācijā tiek izmantotas arī citas sistēmas, kas būvētas, izmantojot jaunāko ASP.Net 3.5 versiju, kurā lietojumprogrammu konfigurēšanas principi ir nedaudz citādi. Tātad, lai vienlaikus uzturētu gan novecojušo sistēmu, gan jaunās sistēmas, ne tikvien nepieciešams dubults daudzums aparatūras, bet arī sistēmu administratoriem jāpārzina abas tehnoloģijas un to konfigurēšanas principi.

Tādējādi, organizācijā saglabājot morāli novecojušas datorsistēmas, pieaug sistēmu uzturēšanai nepieciešamo zināšanu apjoms.

1.1.4. Datoru konfigurāciju atšķirības

Iepriekšējās nodaļās vairāk apskatītas tās datorsistēmu infrastruktūras daļas, kas saistītas ar centralizētu serveru infrastruktūru, taču sistēmu uzturēšana vēl vairāk sarežģījas, ja to papildina plašs lietotāju loks, kur vienu un to pašu datorsistēmu uzstāda dažādas konfigurācijas lietotāju darbstacijās (šeit netiek apskatīta vispārējas pieejamības plaša patēriņa programmatūra, bet specifiski izstrādāta programmatūra ar relatīvi lielu lietotāju skaitu).

Kā piemēru šeit iespējams minēt Latvijā izstrādātu datorsistēmu FIBU, kas paredzēta darbam ar valsts organizāciju budžeta pārskatiem. Šo sistēmu galvenokārt izmanto pašvaldību organizācijas un tās lietotāji izvietoti visā Latvijas teritorijā. Šīs sistēmas uzturēšanu no izstrādātāju un uzturētāju viedokļa detalizēti apraksta [**17**,**18**]. Apraksta autors norāda, ka programmatūra bieži vien tiek darbināta vidē, kas neatbilst tās lietošanas prasībām – piemēram, atšķirīgi MS Windows reģionālie iestatījumi, instalēta biroja programmatūras nepilnvērtīga versija, programma tiek darbināta datorā, kam nav datortīkla pieslēguma u.c. Analizējot iespējas uzlabot FIBU programmatūras uzticamību, autors norāda, ka „Pastāv arī risks, ka klients ievēro kļūdu programmas darbībā tikai tad, kad ir sabojājušies vai pat pazūd lietotāja dati. Šādas situācijas ir ļoti nepatīkamas klientam, kas pat var radīt klientam vēlmi

meklēt kādu citu uzticamāku programmatūru. Tādēļ vides pārbaudes būtu nepieciešams veikt automātiski pirms programmas startēšanas vai programmas darbības laikā, lai laicīgi identificēt radušās problēmas lietotāja datorā un brīdināt lietotāju par tām, un, iespējams, neļaut lietotājam strādāt ar programmu līdz piefiksētās problēmas tiks novērstas." Raksta autors piedāvā minētās neatbilstības novērst, izmantojot [**17**] izstrādāto vides automātiskās testēšanas rīku.

1.2. Problēmas identifikācija

Iepriekšējās nodaļās minētais ļauj spriest – programmatūras uzturēšana mūsdienās ir kļuvusi sarežģīta un darbietilpīga, lai to veiktu kvalitatīvi, nepieciešams daudz cilvēka darba, turklāt jāpiesaista augsti kvalificēti darbinieki. Ja uzņēmumā netiek lietota kāds speciāls konfigurācijas pārvaldības risinājums, kas ļauj detalizēti aprakstīt sistēmu pašreizējos iestatījumus (kā aprakstīts, piemēram, [**19**]), sistēmu savstarpējās saites un citas uzturēšanas īpatnības, faktiskā sistēmu konfigurācija pastāv tikai kā par uzturēšanu atbildīgo darbinieku zināšanas, tādējādi radot risku, ka šādas zināšanas var viegli „aiziet" no uzņēmuma. Turklāt, pieaugot sistēmu skaitam un izmantoto tehnoloģiju klāstam, konfigurācijas sarežģītība progresīvi pieaug. Tā, piemēram, [**20**] apskata neliela tīmekļa servisa konfigurēšanā veicamos soļus. Serviss izmanto *IBM DB2* datubāzi un *WebSphere Application Server* mitināšanas serveri un, lai to padarītu darboties spējīgu, jāveic konfigurēšanas process, kas sastāv no vairāk nekā 50 atsevišķiem konfigurēšanas soļiem. Raksta autori norāda – „ja turpināsim izstrādāt informācijas sistēmas, kas prasa tik sarežģītu konfigurēšanu, mēs drīz nonāksim pie pilnībā nepārvaldāmām sistēmām".

Šajā darbā kā būtiskākā problēma apskatītas tieši datorsistēmu atteices vai precīzāk – iespēja tās prognozēt un veikt preventīvas darbības to novēršanai. Datoru sistēmas, kas ir būvētas, pieņemot, ka izpildes vides konfigurācija būs atbilstoša vēlamajai, sastopoties ar darbam nepiemērotiem apstākļiem, var „uzvesties" nepiemēroti vai pilnībā atteikties darboties. Atteices patiesie cēloņi var būt dažādi, to skaitā arī programmatūras koda kļūdas, kas radušās, sistēmas izstrādātājam neapzināti izdarot pieņēmumus par izpildes vides konfigurāciju. Tomēr pastāv arī iespēja, ka datorsistēma darba sesiju uzsāk atbilstoši vēlamajam, bet sesijas vidū sastopas ar situāciju, kas traucē tās normālai izpildei.

Būtiskākās problēmas, ko risina šis pētījums, ir:

- izstrādājot programmatūru relatīvi nelielam lietotāju lokam, turklāt ja izstrādei pieejamie resursi ir tikai daļēji pietiekami, programmatūrā galvenokārt ir izstrādātas tās pamatfunkcijas un pieņemts, ka izpildes vide būs atbilstoša vēlamajai, nevis

izstrādātas detalizētas „aizsardzības” funkcijas, turklāt programmatūra nesniedz informāciju par pieņēmumiem, kas tajā iekļauti,

- programmatūrā „iebūvētajiem” pieņēmumiem par izpildes vidi nepiepildoties, programmatūra var nespēt darboties dotajā vidē,
- atteices gadījumā programmatūra nesniedz pietiekami daudz informācijas par notikušo,
- programmatūru nav iespējams iedarbināt, lai pārbaudītu tās darba spējas attiecīgajā vidē, jo tādējādi tā uzsāktu darījumu funkcionalitātes izpildi,
- pārbaude, vai programmatūras pieņēmumi par izpildes vidi atbilst realitātei, ir ilglaicīgs manuāls darbs, turklāt ir sarežģīti ilgtermiņā uzturēt zināšanas par dažādiem programmatūras pieņēmumiem,
- neveicot plašas un ilglaicīgas pārbaudes, nav iespējams noskaidrot, no kādiem datorsistēmu infrastruktūras resursiem ir atkarīga dotā programmatūra,
- nav iespējams viegli identificēt to programmu kopumu, kas atkarīgas no noteikta resursa (var kļūt būtiski brīdī, kad, piemēram, nepieciešams attiecīgo resursu likvidēt).

Tādējādi šī darba galvenais mērķis ir vienkāršot ilglaicīgi uzturamu sistēmu dzīves ciklu un pēc iespējas samazināt atkarību no personāla zināšanām par datorsistēmas uzbūvi un ekspluatācijas nosacījumiem.

Identificētās problēmas pastāv jebkurā uzņēmumā, kas sasniedzis pietiekamu tehnoloģisku briedumu, lai gribētu atbrīvoties no atkarības no personāla zināšanām, taču nav gatavs finanšu un resursu investīcijām, ko prasa pilnībā autonomu sistēmu izveides prakse.

2. SAISTĪTIE PĒTĪJUMI

2.1. Autonomās un *paš-* sistēmas

1990. gadu beigās un 2000. gadu sākumā kļuva arvien skaidrāks, ka pašreizējās datoru sistēmas ir sarežģītas un to pārvaldīšana pieprasa milzīgu cilvēka darba ieguldījumu. Tā, piemēram, [**21**] autori analizē kāda eksperimenta rezultātus, kur kvalificētam sistēmu administratoram tiek uzdots relatīvi vienkāršs darbs – aizvietot bojātu disku specifiskā RAID disku masīvā. Vairākos gadījumos šis uzdevums izpildīts neveiksmīgi tieši cilvēciskā faktora dēļ, turklāt šādu gadījumu skaits ir relatīvi liels – pat līdz 20% gadījumu. Autori secina, ka, veidojot uzticamas sistēmas, nav pieļaujami neievērot cilvēciskā faktora nozīmi.

Līdzīgi secinājumi atrodami arī [**2**], kur secināts, ka cilvēciskās kļūdas faktors sistēmu uzturēšanas uzdevumos var sasniegt pat 40% (relatīvi pret visu kļūdu skaitu).

Šajā laikā (200. gadu sākumā) programmatūras ražošanas industrijā attīstījās prasība pēc vienkāršāk uzturamām sistēmām, ko tālāk attīstīja arī pētniecība. Dažādu platformu izstrādātāji sāka piedāvāt ideoloģiski radniecīgus metodoloģiskos karkasus, kas paredzēti sistēmu uzturēšanas vienkāršošanai un attiecīgi – kopējo uzturēšanas izmaksu samazināšanai. Tā, piemēram, *IBM* to apzīmēja ar jēdzienu „autonomā skaitļošana" (angl. *autonomic computing*), *Microsoft* termins bija „uzticamā skaitļošana" (angl. *trustworthy computing*), bet *Hewlett-Packard* nosaukums – „adaptīvais uzņēmums" (angl. *adaptive enterprise*) [**22**]. Jēdziens „uzticamā skaitļošana" vēlāk kļuvis nedaudz neprecīzāks, uzsākot to lietot gan lai definētu uzlabotu programmatūras dzīves cikla procesu [**23**], gan lai aprakstītu uzlabotu operētājsistēmas arhitektūru [**24**]. Līdzīgi, arī termins „adaptīvais uzņēmums" mūsdienās ir izplūdis un neskaidrs.

IBM izstrādāto un 2001. gadā publicēto autonomās skaitļošanas iniciatīvu var uzskatīt par daudz veiksmīgāku nekā *Microsoft* un *HP* piedāvātās. Iniciatīvas veiksmīgo attīstību var skaidrot ar to, ka tā jau sākotnēji ir tikusi dokumentēta, skaidri definējot iniciatīvas mērķus un mazāku uzsvaru liekot uz līdzekļiem, kas pielietojami mērķu sasniegšanai. Tādējādi *IBM* definēto autonomās skaitļošanas ideju var uzskatīt par atskaites punktu konkrētiem autonomo sistēmu risinājumiem.

Autonomās skaitļošanas ideja sākotnēji aprakstīta *IBM* izstrādātā dokumentā „Autonomās skaitļošanas manifests" (angl. *Autonomic computing manifesto*) [**7**], bet vēlāk to

detalizēti aprakstījuši arī ar *IBM* saistīti zinātnieki: [**2**,**25**,**26**]. Manifests definē strukturētu autonomās skaitļošanas vīziju, uzstādot iniciatīvas ilgtermiņa mērķus.

Termins „autonomā sistēma" un citi manifestā lietotie termini ir apzināti izvēlēti tā, lai radītu asociācijas ar dzīvo būtņu autonomajām nervu sistēmām un tādējādi vienkāršotu ideoloģijas izpratni, lietojot analoģijas.

Manifests definē īpašības, kādām jāpiemīt „autonomām" programmatūras sistēmām, taču neskaidro, kā veidot šādas sistēmas, tādējādi ļaujot izstrādātājiem pašiem definēt tālākas prasības. Sākotnēji manifests definē astoņas svarīgākās autonomu sistēmu *paš-** īpašības, taču vēlākajos darbos [**2**] šis skaits samazināts, minot tikai četras īpašības.

Paškonfigurēšanās apraksta programmatūras sistēmas spēju pielāgoties dinamiski mainīgai videi, ar jēdzienu „vide" šeit saprotot gan aparatūras, gan programmatūras vidi. Labi paškonfigurēšanas piemēri ir *karstās pārnešanas* (angl. *hot-swapping*) un „*Plug and Play"* tehnoloģijas. [**2**] šo īpašību skaidro šādi: „*Paškonfigurēšanās ietver ne tikai atsevišķas sistēmas spēju izpildes laikā mainīt savu konfigurāciju, bet arī visām uzņēmumā esošajām sistēmām patstāvīgi mainīt savu konfigurāciju, lai veidotu uzņēmumam nepieciešamo e-biznesa infrastruktūru. Autonomās skaitļošanas mērķis ir panākt, situāciju, kurā ne tikai atsevišķi serveri, programmas vai datu glabāšanas ierīces, bet pilnībā visas IT infratruktūras komponentes ir paškonfigrējošas*".

Pašārstēšanās ir īpašību kopums, kas sistēmai ļauj pēc atteices atjaunot savu darbību iespējami īsā laikā. Šī īpašība sevī ietver pašdiagnostiku, bojāto elementu noteikšanu un aizstāšanu. Pašārstēšanās mērķis ir „*maksimizēt katras aparatūras un programmatūras produkta uzticamību un pieejamību, tiecoties uz nepārtrauktu pieejamību*" [**2**].

Pašoptimizācija tiek izmantota, lai, pārraugot (angl. *monitor*) dažādu sistēmai nepieciešamu resursu parametrus, automātiski pielāgotu sistēmas darbu. Pie pašoptimizācijas tiek pieskaitīti tādi uzdevumi kā darba slodzes pārvaldīšana, optimāla resursu pārvaldīšana u.c. darbības, kas ļauj palielināt sistēmas veiktspēju un/vai samazināt darbam nepieciešamos resursus.

Pašaizsardzība definē sistēmas spēju aizsargāt pašai sevi, tas ir, paredzēt, pamanīt un identificēt apdraudējumus, pielietot adekvātus līdzekļus risku novēršanai.

Autonomās skaitļošanas manifesta autori min, ka pilnvērtīga autonoma sistēma ir vienādi spēcīga jebkurā no šiem virzieniem, taču jāsaprot, ka ir grūti vienlaikus pārveidot pilnībā visu uzņēmuma infrastruktūru, lai atbilstu visiem mērķiem. Tādēļ autonomās skaitļošanas manifests piedāvā identificēt sistēmas „neatkarības" brieduma pakāpes. Brieduma pakāpju klasifikācija ir daļēji līdzīga CMMI (R) modelim. Brieduma pakāpes tiek

definētas kā pieci dažādi autonomijas līmeņi [**27**]: pamata, vadītais, paredzošais adaptīvais un autonomais līmenis.

Pamata līmenis. „Šajā līmenī par visu sistēmas elementu uzturēšanu atbild IT profesionāļi. Komponenšu konfigurēšana, optimizēšana un aizsardzība tiek veikta manuāli"

Vadītais līmenis. „Šajā līmenī, lai apkopotu informāciju no dažādām sistēmām, tiek pielietotas sistēmu pārvaldības tehnoloģijas. Tas palīdz administratoriem apkopot un analizēt informāciju. Lielāko daļu analīzes veic IT profesionāļi. Šis līmenis uzskatāms par IT uzdevumu automatizācijas sākuma līmeni."

Paredzošais līmenis. „Šajā līmenī katra komponente atsevišķi pārrauga pati sevi, analizē izmaiņas un sniedz padomus. Tādējādi tiek samazināta atkarība no cilvēka un ir uzlabots lēmumu pieņemšanas process."

Adaptīvais līmenis. „Šajā līmenī IT komponentes individuāli vai pa grupām pārrauga un analizē savu darbu un sniedz ieteikumus. Nepieciešama tikai neliela cilvēka iejaušanās."

Autonomais līmenis. „Šajā līmenī sistēmas darbu pārvalda ar cilvēka definētu darījuma politiku līdzekļiem. Atšķirībā no adaptīvā līmeņa, šeit nav nepieciešama tieša cilvēka iejaukšanās, sistēmu pārvaldība tiek veikta tikai ar darījuma politiku līdzekļiem."

IBM autonomās skaitļošanas manifestā izteiktās idejas uztvēruši un savus risinājumus piedāvājuši gan industrijas, gan pētniecības pārstāvji. Paveiktais apkopots vairākos vairāku pētnieku veiktos pārskatos.

D. Tosi 2004. gada ziņojums par autonomo sistēmu aktuālajiem pētniecības virzieniem [**28**] sniedz plašu literatūras apskatu, vienlaikus skaidrojot autonomo sistēmu uzbūvi, sevišķu uzmanību pievēršot pašārstēšanās īpašībai. Ziņojumā tiek nodalītas piecas būtiskas pašaizsardzības sistēmām nepieciešamas fāzes:

- pārraudzība, kuras mērķis ir, veicot dažādus komponentes iekšējā stāvokļa un apkārtējās vides mērījumus, „pamanīt" novirzes no vēlamās izpildes,
- interpretācija, kas, izmantojot pārraudzības fāzē novērotās īpatnības, mēģina piemeklēt līdzīgus novērojumus sistēmas zināšanu bāzē un tādējādi atpazīt problēmu un piemeklēt iespējamos problēmas risinājumus,
- diagnostika, kuras uzdevums ir noskaidrot atrastās problēmas cēloņus un pārliecināties, ka zināmie pretlīdzekļi, kurus paredzēts izmantot, novērsīs atrastos cēloņus,
- adaptācijas fāzē tiek veiktas izmaiņas, kuru noderīgums pārbaudīts diagnostikas laikā,

- apmācības fāzē sistēmas zināšanu bāze tiek papildināta ar informāciju šo pašārstēšanās ciklu, t.sk. pārraudzības laikā novērotajām izmaiņām, piemērotajiem pretlīdzekļiem un darba rezultātiem pēc adaptācijas.

D. Tosi pārskats norāda uz vairākiem avotiem, kas būtiski arī šajā darbā veiktajam pētījumam. Atsaucoties uz [**29**], pārskatā minēts, ka „Orso un citi piedāvā papildināt komponentes ar metadatiem, lai padarītu ziņas par komponentes uzbūvi pieejamas citām programminženierijas disciplīnām, neatklājot pašas komponentes pirmkodu". Atšķirībā no šajā disertācijā paredzētā pielietojuma, Orso metodika paredz metadatus izmantot komponenšu bāzētas programmatūras izstrādes, nevis gatavas programmatūras uzturēšanas fāzē. Tādējādi tiek piedāvāts risinājums metadatus glabāt pašā komponentē un definēt specifisku programmatūras interfeisu, kas ļautu izstrādes rīkiem nolasīt metadatus no šādas komponentes.

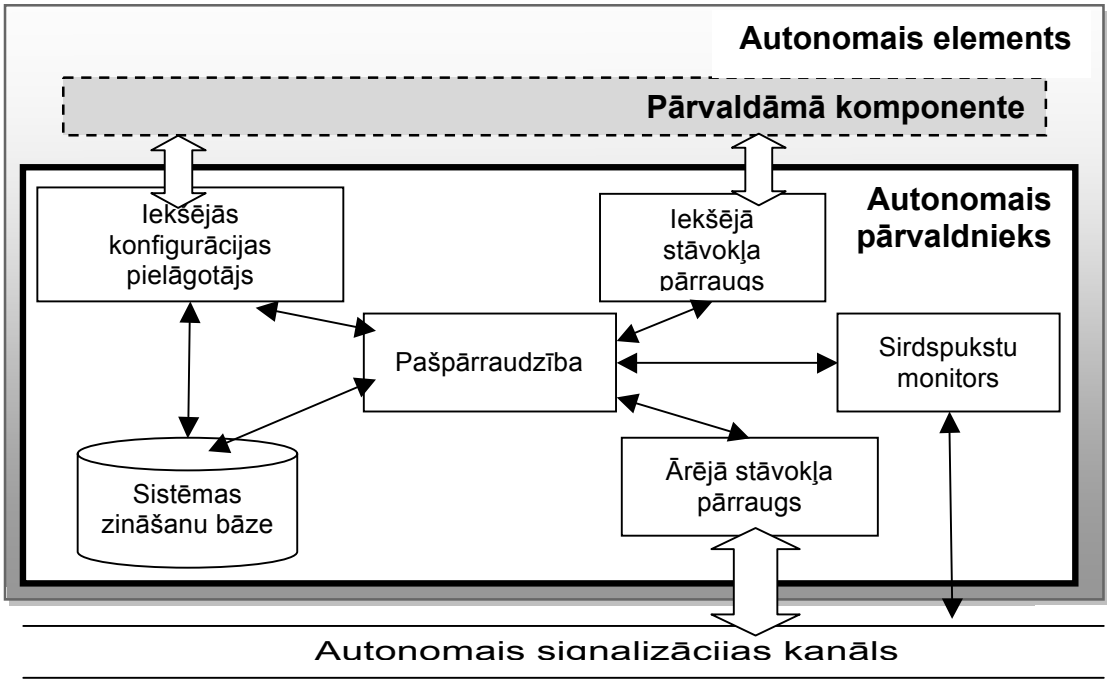
2005. gadā tiek publicēts J. O. Kepharta (*J.O. Klephart*) veikts pētījums [**30**] par autonomās programmatūras sfērā veicamajiem pētījumiem. Šis autors piedāvā iedalīt veicamos pētījumus trīs virzienos – autonomie elementi (kurus var uzskatīt par autonomu sistēmu pamatjēdzienu), autonomās sistēmas un cilvēka-datora sadarbība autonomajās sistēmās. Tālāk rakstā detalizētāk analizēts katrs no šiem pētījumu virzieniem. Raksta turpmāk veicamo pētījumu sadaļā atrodamas norādes, kas saistošas apkārtējās vides pētījumam: „Viens no pašārstēšanās risinājumiem ir apkopot lielu datubāzi ar zināmajiem simptomiem un veicamajām korektīvajām darbībām. Lai problēmu lokalizētu vai diagnosticētu, tiek izmantota simptomu datubāze un, ja tajā atrodams saderīgs simptoms, piemēro saistīto risināšanas metodi". Šī norāde atsaucas uz M. Brodija (*M. Brodie*) pētījumu [**31**], kas analizē programmatūras izsaukuma informāciju, fiksējot to kļūdas notikšanas brīdī. No izsaukuma informācijas tiek svītrota nesvarīgā informācija, atstājot tikai būtisko, kas identificē sistēmas stāvokli. Šī stāvokļa informācija tiek salīdzināta ar datubāzē jau esošiem vēsturisko stāvokļu pierakstiem. Metode ir sevi pierādījusi kā veiktspējīgu gadījumos, kad nepieciešams identificēt bieži atkārtojošās sistēmas kļūdas. Autori arī norāda, ka labākus rezultātus būtu iespējams iegūt, ja kļūdas notikšanas brīdī būtu iespējams iegūt papildus informāciju par izpildes vidi. „*Salīdzinot ar citām pieejām*, *mēs paļaujamies tikai uz programmas izsaukuma informāciju no iepriekšējām atteicēm, neņemot vērā informāciju no izpildes vides*" (šeit ar "citām pieejām" domāti citu autoru piedāvātie risinājumi – apkopot izpildes stāvokļa informāciju, modelēt kļūdu vai identificēt programmas invariantu).

Tikuši veikti arī atsevišķi pētījumi, kuru mērķis ir specificēt vispārīgu autonomo sistēmu arhitektūru. Autonomās izpildes vide, kā to apraksta R. Srterits (*R. Sterritt*) [**25**], sastāv autonomajiem elementiem, kurus savieno autonomais signalizācijas kanāls. Katrs

autonomais elements sastāv no (skat. ***2.1***. *att.*) pārvaldāmās komponentes (t.i., komponentes, kas faktiski realizē darījuma funkcionalitāti) un autonomā pārvaldnieka, kas ir atbildīgs par pārvaldāmās komponentes darba vides nodrošināšanu. Savas funkcijas autonomais pārvaldnieks realizē, veicot vairākas funkcijas:

- **Pārvaldāmās komponentes iekšējā stāvokļa pārraudzība** (tiek pieņemts, ka pārvaldāmā komponente būvēta tā, lai būtu spējīga sniegt nepieciešamo informāciju pārvaldniekam). Šo funkciju realizē ar iekšējā stāvokļa pārrauga palīdzību.
- **Apkārtējās vides stāvokļa pārraudzība**, izmantojot informāciju, kas saņemta no autonomās signalizācijas kanāla. Šo funkciju realizē ārējā stāvokļa pārraugs.
- **Iekšējās konfigurācijas pielāgošana**, mainot pārvaldāmās komponentes parametrus tā, lai sasniegtu vēlamo darba režīmu. Šo funkciju realizē iekšējās konfigurācijas pielāgotājs
- **Sistēmas zināšanu bāzes uzkrāšana** un analīze. Pārvaldnieks uzkrāj dažādu informāciju par sistēmas stāvokli, tā izmaiņām, apkārtējo vidi un izmanto to turpmāko lēmumu pieņemšanai, piemēram, iepriekš notikušu kļūdu atkārtošanas novēršanai.
- **Ziņošana** „sirdspukstu monitoram". Šo funkciju realizē atsevišķa apakškomponente, kuras uzdevums ir, izmantojot autonomo signalizācijas kanālu, informēt „sirdspukstu pārvaldnieku" par stāvokli sistēmā un būtiskām tā pārmaiņām.

Sirdspukstu monitors aprakstīts kā specifiska komponente, kas pārrauga citu autonomo elementu darbu un izmanto saņemto informāciju, lai pielāgotu cilvēka noteiktās darījuma politikas sistēmas darbam.



2.1. att. **Autonomo elementu arhitektūras risinājums**

Interesantu pieeju autonomo komponenšu savstarpējo saistību realizācijai piedāvā [**32**], kur komponenšu saziņa tiek decentralizēta, autonomās saziņas kanālu papildinot ar zināšanām par „kaimiņiem", sagrupējot komponentes funkcionāli radniecīgās komponenšu grupās. Tādējādi daļu veicamo mērījumu komponentes var izpildīt, neiesaistot centrālo sirdspukstu pārvaldnieku.

Autonomo sistēmu ideja tikusi arī kritizēta vairāku iemeslu dēļ. K. Hermans (*K. Hermann*) 2005. gadā identificē vairākus autonomo sistēmu ideoloģijas trūkumus [**33**].

Precīzu **definīciju trūkums**. Autori norāda, ka salīdzinājums ar autonomajām nervu sistēmām var palīdzēt izprast autonomo sistēmu vīziju kopumā, bet esošās definīcijas ir nepietiekamas. Piemēram, pat pieredzējušam sistēmu administratoram reizēm nav viegli noteikt, vai attiecīgā komponente ir bojāta, vai arī tā nespēj pietiekami labi darboties kādu ārēju apstākļu dēļ. Autonomo sistēmu ideja paredz, ka bojājuma gadījumā jāpielieto pašārstēšanās, bet ārēju apstākļu ietekmes gadījumā– pašoptimizācijas stratēģijas. Tiek formulēts jautājums, vai sistēma spēs automātiski identificēt šos apstākļus labāk nekā cilvēks un patiešām pielāgos pareizo stratēģiju?

Kā būtisku autonomo sistēmu trūkumu Hermans min publikācijām pārāk vienkāršotos piemērus un **izvairīšanos no reālās problēmas sarežģītības** aprakstīšanas. Viena no būtiskām autonomo sistēmu sastāvdaļām ir t.s. „slēgtās vadības cilpas" (angl. *closed control loop*) princips, kas, saņemot ieejas parametrus, kontrolē sistēmas stāvokli tā, lai norādītie parametri tiktu ievēroti. Autori sniedz salīdzinājumu starp automašīnas termostatu, ko ir relatīvi viegli izveidot, un bremžu pretbloķēšanas sistēmu (ABS), kurai ir daudz parametru, kas specifiski jāpielāgo katram automašīnas modelim. Tiek apgalvots, ka ABS sistēmu parametru pielāgošana ir grūts uzdevums, jo pat nelielas izmaiņas automašīnas kravnesībā vai citos parametros var atstāt iespaidu uz nepieciešamo ABS konfigurāciju. Šī iemesla dēļ nopietnākie autoražotāji veido parametru datubāzes, lai kaut nedaudz vienkāršotu uzturēšanu. Tiek uzdots jautājums, vai autonomā sistēma patstāvīgi spēs izveidot un uzturēt atbilstošu datubāzi un vai cilvēka noteikto darījuma politiku izmaiņas neatstās katastrofālu iespaidu uz nepieciešamo parametru komplektu.

Rakstā tiek minēts, ka autonomo sistēmu ideoloģija pārāk zemu novērtē **starpkomponenšu saites**. Tiek apskatīts piemērs, ka divas dažādas sistēmas komponentes, reaģējot uz kāda sistēmas parametra izmaiņu, vienlaicīgi uzsāk pašoptimizāciju, bet katra no tām vēlas sasniegt citādu mērķi. Piemēram, novērojot sistēmas lēndarbību, datu nolasīšanas komponente izvēlas pēc iespējas vairāk datu ielasīt sistēmas kešatmiņā, savukārt datu apstrādes komponente, reaģējot uz lēndarbību un pamanot atmiņas aizpildītību, uzsāk atmiņas

pārnešanu uz cieto disku. Raksta autori min, ka vienīgais veids, kā apiet šādu situāciju, būtu ieviest atsevišķu komponenti, kas pārvaldītu katru resursu un novērstu pretrunīgas darbības, taču tas vēl vairāk sarežģītu sistēmu uzbūvi.

Ar autonomo sistēmu lauku saistītās literatūras apskats sniedz noderīgas idejas šajā darbā veiktajam pētījumam. Tomēr saskaņā ar autoram zināmajiem pašlaik publicētajiem avotiem, darbā apskatītā problēma tiešā veidā nav atrisināta, izmantojot autonomo sistēmu idejas. Galvenie apsvērumi no autonomo sistēmu ideoloģijas, kas jāņem vērā, risinot programmatūras izpildes vides pārbaudes problēmas, apkopoti nākamajās rindkopās.

Lai vienkāršotu sistēmu uzturēšanu, arī pašām sistēmām ir jābūt ar tam pielāgotu arhitektūru. Daudzas pašreizējās sistēmas nav pielāgotas darbam paš-* modelī, jo tajās nav iebūvēta nepieciešamā funkcionalitāte un tām arī nav ārēju interfeisu, kas ļautu citiem rīkiem veikt pārbaudes.

Izstrādājot komponentes, kas pielāgotas autonomo sistēmu ideoloģijai, tās ieteicams veidot kā autonomās komponentes, kas paredz tādas darbības kā iekšējo pārraudzību un atbildes „sirdspukstu monitora" komponentei.

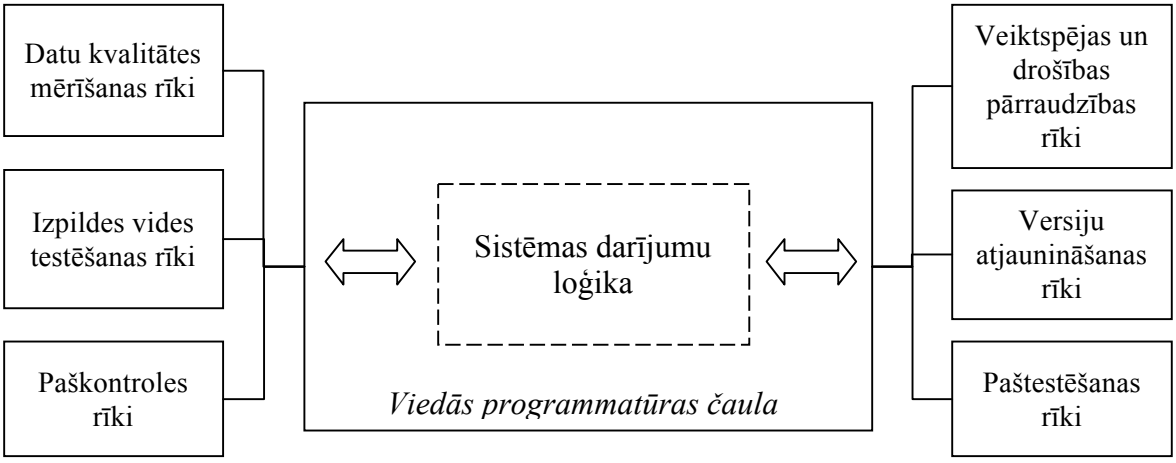
Pievienojot programmatūras komponentei metadatu aprakstu par šīs komponentes uzbūvi un prasībām pret izpildes vidi, tālākajos sistēmas dzīves cikla posmos tos var izmantot gan izstrādes, gan uzturēšanas vienkāršošanai.

2.2. Viedo tehnoloģiju programmatūra

Šajā darbā veiktais pētījums ir daļa no viedo tehnoloģiju programmatūras (angl. *Smart Technology Compatible Software*, STSW) ideoloģijas, ko definē Z. Bičevska un J. Bičevskis [**4**,**5**,**6**]. Līdzīgi kā *IBM* autonomās skaitļošanas vīzija, arī viedo tehnoloģiju programmas ideja ir – meklēt veidus, kā vienkāršot programmatūras uzturēšanas un darbināšanas procesus. Atšķirībā no autonomās skaitļošanas, kas definē tālu mērķi, uz kuru tiekties datorsistēmu izstrādātājiem, viedo tehnoloģiju pieeja definē tiešākus attīstības virzienus un ir pielāgojama arī pašlaik pastāvošām programminženierijas praksēm. Tas galvenokārt izskaidrojams ar faktu, ka viedo tehnoloģiju programmatūras ideja nākusi no datorsistēmu izstrādes industrijas un balstīta uz praktiskiem programmatūras procesu uzlabošanas mēģinājumiem. Viedo tehnoloģiju programmatūras ideja galvenokārt attiecināma uz specifiskiem mērķiem izstrādātas, nevis plaša patēriņa programmatūras inženierijas lietojumiem.

Viedo tehnoloģiju idejas autoru mērķis ir tālākminētās programmatūras īpašības padarīt par darījumu programmatūras aspektiem, kuru izstrāde nav jāatkārto no sistēmas uz sistēmu. Tādējādi darījumu sistēmu būvei pieejama jau gatava rīku un iegultās programmatūras infrastruktūra, kurā nepieciešams izstrādāt tikai to loģiku, kas nepieciešama attiecīgās

darījumu sistēmas programmatūrai, bet pārējo ar šo loģiku nesaistītos aspektus saņemt kā viedās programmatūras čaulas sniegtus servisus (skat. ***2.2****. att.*, [**6**]).



2.2. att. **Viedās programmatūras čaulas pieejamība darījumu sistēmu loģikai**

Līdzīgi kā autonomās skaitļošanas gadījumā, arī viedo tehnoloģiju ideoloģija definē vairākas *viedās* programmatūras īpašības, kas uzlabo programmatūras uzturamību.

Automātiska versiju atjaunošana [**5**] apraksta programmatūras spēju automātiski sazināties ar centralizēto versiju glabātuvi, lai pārliecinātos, vai lietotājs izmanto jaunāko programmatūras versiju. Šī īpašība ir ļoti būtiska gadījumos, kad programmatūra tiek izstrādāta iteratīvi un programmatūras laidieni mainās relatīvi bieži. Automātiska versiju atjaunošana jau sastopama dažādās plaša patēriņā lietojumprogrammās kā, piemēram, *Skype* (tiešsaistes saziņa), *Mozilla Firefox* (tīmekļa pārlūkprogramma), *Microsoft Office* (biroja lietojumprogrammu pakete) un citās. Plaša patēriņa programmās, kur izplatīto dažādo versiju skaits katram produktam ir relatīvi neliels, salīdzinot ar lietotāju skaitu, kas attiecīgo versiju izmanto, šāda automatizācija ir dabiska un tās pielietojamība ir pašsaprotama, kaut arī šādas funkcionalitātes izstrāde un uzturēšana ir resursietilpīga un dārga. Specifiski pielāgotas programmatūras gadījumā situācija ir citāda, jo produkta versiju skaits tuvojas atsevišķo pasūtītāju skaitam, tātad lieliem specifiskās programmatūras izstrādātājiem ir vienlaicīgi jāuztur liels skaits dažādu programmatūras versiju. Šī iemesla dēļ programmatūras spēja patstāvīgi automātiski atrast jauninājumus tiek uzskatīta par būtisku uzturēšanas procesu atvieglojumu.

Programmatūras **izpildes vides analīze** raksturo programmatūras spēju patstāvīgi izvērtēt, vai pašreizējā izpildes vide atbilst vēlamajiem nosacījumiem. Šajā darbā aprakstītais pētījums ir attiecināms tieši uz šo viedās programmatūras ideoloģijas aspektu.

Paštestēšana, dinamiska kontroles plūsmas, notikumu un vērtību trasēšana. Viedo tehnoloģiju ideoloģijas autori [**4**] to definē kā „spēja kontrolēt pašam sevi un izprast savu

spēju robežas". Viedās programmatūras ideju autori piedāvā paštestēšanas rīkus iebūvēt programmatūrā (līdzīgi kā aprakstīts šī darba nodaļā „2.3. Iebūvēto testu metode"). Paštestēšana ir testēšanas paņēmiens, kurā programmatūra patstāvīgi, izmantojot tajā iebūvētus līdzekļus, pārbauda savu kvalitāti. Šī pieeja detalizēti aprakstīta [**6**], kur autori piedāvā programmatūru būvēt tā, lai tajā iebūvētie testi tiktu izpildīti tikai gadījumā, kad programmatūra iedarbināta specifiskā paštestu izpildes režīmā. Tas ļautu iebūvētos testus izmantot arī lietošanas vides sistēmā, jo speciālajā režīmā programmatūra izvairītos no sistēmas stāvokļa (piemēram, datubāzes ierakstu) mainīšanas, visus testu rezultātā iegūtos datus saglabājot kādā citā, speciāli testiem paredzētā datu repozitorijā (tehnoloģija cieši sasaucas ar [34]). Tādējādi tiek iegūts risinājums, kas ļauj lielākajai daļai sistēmas komponenšu darboties tā, it kā sistēma tiktu darbināta parastajā izpildes režīmā, taču rezultātā vides stāvoklis paliek nemainīgs.

Darījumu modeļa (angl. *business model*) **integrēšana programmatūrā**. Šīs īpašības mērķis ir papildināt programmatūras darbināšanas infrastruktūru ar starpniekvidi, kas nodrošina programmatūras procesu atbilstību darījumu procesiem, kurus tā apkalpo. Saskaņā ar [**4**] definēto – „šajā gadījumā darījuma procesu modelis kalpo kā metainformācija programmatūras funkciju aprakstīšanai." Šādas starpniekvides piemērs ir aprakstīts [**35**] – rīks ar nosaukumu „IS Tehnoloģija", kas kalpo kā lietotāja darba virsma un ļauj iedarbināt darījumu procesus lietotājam saprotamā veidā. „IS Tehnoloģija" saskarne piedāvā dinamiski mainīgus pieejamo darījumu objektu un tiem piemērojamo darbību sarakstus, tādējādi ierobežojot darbības, kas noteiktā sistēmas stāvoklī katram objektam pieejamas. Šādā veidā ar starpniekvides palīdzību tiek nodrošināta arī darījumu darba plūsmu izpilde.

Datu kvalitātes kontrole. Šī viedās programmatūras īpašība attiecas gan uz datiem, ko sistēma saņem, gan tiem, ko tā rada. Viedo tehnoloģiju ideoloģijas autori piedāvā veidot *viedkomponentes*, kas veiktu datu kvalitātes pārraudzību - to veicot vai nu pēc pieprasījuma, vai darbojoties autonomā režīmā un regulāri veicot datu saderības pārbaudes, vadoties pēc iepriekšdefinētiem likumiem. Datu kvalitātes kontrole pieder pie autonomās programmatūras pašdiagnostikas un pašaizsardzības principiem.

Pārraudzība: veiktspējas, drošības un pieejamības pārraudzība. Visu šo pārraudzības jomu apzināšana jau programmatūras cikla sākotnējās fāzēs ir būtiska, lai nodrošinātos pret reālā lietojuma problēmām, tādējādi ļaujot vieglāk atklāt sistēmas veiktspējas vājās vietas, citu pušu uzbrukumus sistēmām, serveru nepieejamību u.c. Piemēram, ja nepieciešama spēja automātiski atklāt paroļu automatizētas skenēšanas uzbrukumus, sistēmas autentifikācijas modulis jāpapildina ar piemērotiem šādu darbību atklāšanas līdzekļiem, turklāt ņemot vērā serveru klasterēšanas un sinhronizācijas īpatnības. Šī programmatūras īpašību virziena mērķis

ir radīt kopīgu infrastruktūru, ko jebkura komponente var izmantot pārraudzībai, kļūdu paziņošanai un „trauksmes" radīšanai.

2.3. Iebūvēto testu metode

Paštestējošas programmatūras ideja vēsturiski ir pārņemta no aparatūras sistēmām, kas jau sen spēj, uzsākot darbu, veikt dažādas pārbaudes. Piemēram, RAM atmiņa, kas pārbauda visas atmiņas šūnas un tikai tad sāk pildīt savu pamatuzdevumu – datu glabāšanu. Atšķirībā no aparatūras, programmatūras sistēmu izpildes vide ir dinamiski mainīga, tādēļ nepieciešami citi pārbaudes principi. Šī iemesla dēļ programmatūras sistēmas nevar veikt tik plaša apmēra pārbaudes kā datoru aparatūras gadījumā, bet ir iespējams noteikt atsevišķu pārbaužu apakškopu, ko ir iespējams veikt algoritmiskā ceļā neatkarīgi no konfigurācijas. Nākamajās nodaļās apskatīti vairāki iebūvēto testu veidi.

2.3.1. Apgalvojumi (assertions)

Daudzas programmēšanas valodas satur priekšrakstu (angl. *statement*) „assert", kas ļauj programmētājam deklarēt faktu, kam vienmēr jābūt patiesam. Piemēram, *ANSI C* valoda, kas standartizēta jau 1989. gadā [**36**], piedāvā makrodefinīciju „*assert*" .Šī makrodefinīcija ļauj definēt nosacījumu, kam programmas izpildes laikā jābūt spēkā un kas aptur programmas izpildi, ja tā nav. Līdzīgas konstrukcijas iekļautas arī daudzās citās valodās, piemēram, *Eiffel*, *Java*, *Visual Basic*.

Piemēram, ja programma atver lasīšanai kādu failu, nākamais priekšraksts varētu būt pieņēmums, ka faila atvēršana ir noritējusi veiksmīgi (tas ir, ir iegūts rādītājs, kura vērtība nav *null*). Šāda pieņēmuma pieraksts parādīts koda piemērā ***2.3*** *att.*

```
input = fopen ("optimal.in", "r");
assert (input != null);
```

2.3 att. ***Assert*** **lietojuma piemērs C valodā**

Aprakstītā metode novērtēta kā būtisks uzlabojums programmatūras izstrādes un pārstrādes (angl. *refactoring*) mērķiem – tiek uzskatīts, ka tā uzlabo iespējas identificēt kļūdas un to izcelsmi, sevišķi tādas kļūdas, kas saistītas ar nekorektām funkciju parametru vērtībām un citu veidu datu saderības pārbaudēm [**37**]. Tomēr pastāv uzskats, ka tehnika nav pielietojama produkcijas vidē, galvenokārt veiktspējas uzlabošanas un potenciālo koda blakusefektu dēļ. Atsevišķi kompilatori programmas baitkodā, kas paredzēts lietošanai produkcijas vidē (C kompilatoru gadījumā - ja pielietots *NDEBUG* kompilācijas slēdzis), *assert* priekšrakstus nemaz neiekļauj.

Atsevišķi pētījumi paredz apgalvojumu pielietošanu kā programmatūras izstrādes, tā darbināšanas ciklā. [**37**] norāda uz iespēju lietot apgalvojumus, lai programmā samazinātu zināmo kļūdu atstāto efektu. Autori sniedz piemēru, kur funkcija apstrādā lielus dažādu tipu datu daudzumus. Funkcijā realizētas gandrīz visas iespējamās datu vērtību kombinācijas, bet ne visas. Tas nozīmē, ka, ja funkcijai padotas kādas specifiskas datu vērtības, tajā nebūs koda, kas šādas vērtības varētu apstrādāt. Ir skaidrs, ka „normālā gadījumā" šādas vērtību kombinācijas nevarētu parādīties. Tādēļ, lai taupītu cilvēkresursus izstrādes laikā, šādu vērtību korektai apstrādei tiek piešķirta zema prioritāte, bet programmatūras kodā var iekļaut atbilstošu apgalvojumu, lai tādējādi spētu novērtēt, vai šāda kļūda praksē ir iespējama. Kad attiecīgā kļūda ir notikusi, saistītajam programmēšanas darba uzdevumam var paaugstināt prioritāti.

2.3.2. Automatizētā vienībtestēšana

Tālāk attīstot programmatūrā iekļautu apgalvojumu ideju, ir cits programmatūrā iebūvētu testu princips – automatizētā vienībtestēšana. Ja apgalvojums ir vienkārša Būla tipa izteiksme, kuras patiesumvērtības noskaidrošana neiekļauj sarežģītu aprēķinu veikšanu, tad automatizētie vienībtesti ļauj veikt daudz plašāku testēšanu, pievienojot programmatūras koda modulim specifisku testu komplektu. (Lai novērstu neskaidrības – jēdziens „vienībtestēšana" tā sākotnējā nozīmē apraksta programminženierijas procesu, kā aprakstīts IEEE/ANSI standartā [**38**]. Šajā darbā un dažādos citos avotos termins lietots, lai aprakstītu metodi vienībtestēšanas procesa automatizācijai, kas radusies, pielietojot ekstrēmo un citas spējās izstrādes metodes).

Piemēram, .NET saimes valodās lietotais karkass NUnit[1][**39**] ir balstīts uz atsevišķu testa klašu, sauktu arī par testa armatūru (angl. *test fixture*), izstrādi. Katram testējamajam modulim izstrādā un pievieno atsevišķu testa armatūru. Katra armatūra sastāv no viena vai vairākiem testiem. Tests parasti satur kodu, kas analizē pētāmā moduļa vienu vai vairākas funkcijas. Atšķirībā no „*assert*" apgalvojumiem, vienībtesti parasti tiek iekļauti programmatūras baitkoda būvējumos, tomēr arī tie nav paredzēti lietošanai programmas normālas izpildes laikā produkcijas vidē. Lai iedarbinātu vien automātiskos vienībtestus, tiek izmantota speciāla programma, testu izpildītājs, kas pārskata kodu (lietojot .NET koda refleksiju) . Testu izpildītājs programmas kodā meklē testu armatūras (kas tehniski veidotas kā specifiskas koda klases, kam pielietots deklaratīvais atribūts[2] „*TestFixture*"). Pēc tam testu

[1] Atsevišķa koda balstītu testēšanas karkasu klase, kuru pārstāv NUnit, ir pazīstama ar apzīmējumu xUnit. Šie karkasi ir cēlušies no Kenta Beka (*Kent Beck*) sākotnēji izstrādātā SUnit karkasa, kas paredzēts SmallTalk koda testēšanai [en.wikipedia.org]

[2] Atribūta klase piekārto mērķa elementam sistēmas vai lietotāja iepriekš definētu informāciju. Informāciju, ko sniedz attiecīgais atribūts, sauc arī par metadatiem [Microsoft Developer's network library]

izpildītājs izpilda katru testu, kas atrasts testu armatūrā, un pārķer izņēmumsituācijas, kuras rada testu izpilde.

Vienībtestēšana tiek uzskatīta par risinājumu, kas vienkāršo programmatūras izstrādi, sevišķi koda moduļu regresijas testēšanu. Šī pieeja ir galvenais testu bāzētās programmatūras izstrādes pieejas rīks [**40**], kas programmatūras izstrādi sāk tieši ar testa moduļu veidošanu, pirms ticis izstrādāts reālais sistēmas kods. Protams, lai testus definētu, ir jādefinē koda klašu struktūra, metožu un atribūtu aizbāžņi, taču tas arī ir testu bāzētās pieejas mērķis – fokusēties uz komponenšu savstarpējām saitēm un to striktu definēšanu, lai apzinātu koda pieejamības robežas.

Tomēr jāsecina, ka vienībtestēšanas metodika tiešā veidā nevar tikt piemērota programmatūras izpildes vides pārbaudīšanas mērķiem. Viena no vienībtestēšanas pamatidejām ir – testēt katru programmatūras moduli atsevišķi, tas ir, tests nedrīkst pārkāpt pašreiz apskatāmā moduļa robežas. Gadījumos, kad tomēr nepieciešama reakcija no „apkārtējās pasaules", vienībtestēšanas metodika iesaka pielietot atdarinātājobjektus (angl. *mock object*) un viltojumus (angl. *fake object*)[**41**]. Abu veidu objekti tiek izmantoti, lai simulētu testējamajam modulim nepieciešamos ārējo moduļu interfeisus. Šāda tipa objektu izmantošana liecina, ka vienībtestēšana nav piemērota moduļu savstarpējo saišu testēšanai un jo vairāk – tādu saišu testēšanai, kas veidojas starp programmatūras sistēmu un apkārtējo vidi.

2.3.3. Citi iebūvēto testu veidi

1999. gadā Vangs (*Wang*) un citi [**42**] demonstrē pieeju, kas programmatūrai pievieno iebūvētus testus. Iebūvēts tests ir koda funkciju kopums, ko izmanto, lai pārbaudītu, vai koda komponente darbojas „kā paredzēts". Šī ideja ir zināmā mērā līdzīga automātiskajai vienībtestēšanai, kas aprakstīta 2.3.2 nodaļā, taču testi netiek sadalīti pa atsevišķām armatūrām un tiek izmantoti „vienību testēšanai" veselas komponentes līmenī (šeit autori acīmredzami nodala moduli kā programmatūras klasi un komponenti kā klašu kopumu, kas pilda atsevišķu darījuma funkciju).

Autori norāda, ka katrā programmatūras komponentē būtu jāiekļauj viens vai vairāki izpildāmi iebūvētie testi (angl. *built-in test*, *BIT*) un šajā pašā komponentē arī jārealizē programmatūras interfeiss, ko varētu izmantot, kad programma tiek iedarbināta „uzturēšanas režīmā". Šeit autori faktiski definē jēdzienu „uzturēšanas režīms", kas sakrīt ar šajā darbā lietoto formulējumu – specifisks programmas darbināšanas režīms, kad tiek izpildīti programmā iebūvētie testi, bet netiek izmantota sistēmas darījumu funkcionalitāte.

Programmatūras komponentēs iebūvēto testu ideja tiek attīstīta tālāk arī [**43**], kur autori norāda, ka katra komponente realizē savu interfeisu, uz kura pieejamību un korektu darbu var

paļauties citas komponentes. To var apzīmēt ar jēdzienu „kontrakts" starp komponentēm. Autori iesaka izmantot iebūvēto testu metodi, lai pārbaudītu, vai komponente faktiski ir spējīga sniegt solītos interfeisus, tas ir, vai nav pārkāpti kontrakti starp komponentēm.

Tāpat autori norāda – šādus iebūvētus testus var izmantot apkārtējās vides testēšanai vai savietot ar servisa kvalitātes (QoS) testēšanu (taču detalizētāk šīs idejas neapraksta, minot, ka šī tēma raksta tapšanas brīdī nav pētīta).

Komponenšu savstarpējo kontraktu testēšanas ideja plaši aprakstīta [**44**], kas apskata programmatūras komponenšu savstarpējo kontraktu testēšanu no modeļu bāzētās izstrādes perspektīvas. Šī pieeja jau pieņem, ka iebūvētie testi tiek izpildīti „uzturēšanas režīmā". Šeit gan autori piemin, ka, darbinot sadalītu klienta-servera lietojumprogrammu, servera puses komponentei var nebūt informācijas par to, ka klienta komponente pašlaik darbojas paštestēšanas režīmā. Tādējādi, lai izvairītos no iespējas „sabojāt" reālās sistēmas izmantotos datus, katrai servera komponentei būtu jārealizē savs „servera testēšanas" interfeiss, ko lietotu uzturēšanas režīmā, atšķirībā no parastā interfeisa. Autori arī piemin, ka „tādējādi komponentei tiek sniegta iespēja pacelt "sarkano karodziņu", ja tā izvietota nepiemērotā programmatūras izpildes vidē", tādējādi vēlreiz uzsverot iespēju pielietot iebūvētos testus programmatūras izpildes vides testēšanai. Jāņem vērā, ka šādi iespējams testēt tikai tās izpildes vides īpašības, kuru specifiskie stāvokļi zināmi programmatūras izstrādes laikā.

Cits komponenšu savstarpējo kontraktu testu pielietojums piedāvāts [**45**], kur autori apskata komponentes sniegtā interfeisa dažādu metožu secīgu pielietošanu, tādējādi veidojot testēšanas scenāriju. Šādus testēšanas scenārijus var viegli pierakstīt datoram lasāmā formā un vēlāk „atspēlēt", tādējādi faktiski veidojot pēc „melnās kastes" testēšanas principa veidotu regrestestēšanas risinājumu.

Kā viens no iebūvēto testu veidiem tiek pieminētas arī pēc COMPONENT+ principa būvētas komponentes [**44**,**46**]. Šī principa pamatā ir novērojums, ka „parasto" iebūvēto testu gadījumā izpildāmā testa būtība ir iebūvēta pašā komponentē, tādējādi potenciāli palielinot komponentes resursu patēriņu, kas, ņemot vērā to, ka iebūvētie testi tiks reti izmantoti produkcijas vidē, varētu kļūt par iemeslu, kādēļ iebūvētos testus nelietot. COMPONENT+ arhitektūrā šī nepilnība tiek apieta, definējot trīs komponenšu veidus – BIT komponentes, testerus un apdarinātājus (angl. *handlers*). BIT komponentes ir komponentes, kas pielāgotas iebūvēto testu arhitektūrai – tās realizē kādu vienu vai vairākus obligātus interfeisus. Testeri ir komponentes, kurās glabājas testu realizācijas, un kas, izmantojot BIT komponentēs iebūvētos interfeisus, realizē testu izpildi. Citās iebūvēto testu sistēmās testeri ir faktiski iekļauti BIT komponentē, taču COMPONENT+ tos nodala, tādējādi ļaujot testus izstrādāt un uzturēt atsevišķi no BIT komponentes. Visbeidzot, apdarinātāji ir komponentes, kas tiešā

veidā nepiedalās testēšanā, bet ko var izmantot, piemēram, lai nodrošinātu darbības atjaunošanas mehānismu gadījumā, ja kāds no iebūvētajiem testiem nav beidzies sekmīgi.

Saskaņā ar autoram zināmo informāciju, pašlaik neeksistē efektīva COMPONENT+ realizācija. Kā viens no loģiskiem šīs arhitektūras turpinājumiem ir MORABIT projekts [**47**], kas iebūvēto testu ideju izmanto mobilu sistēmu testēšanai. Šo sistēmu īpatnība ir to dinamiskā daba, proti, to izpildes vide veidojas pēc pieprasījuma un to darbināšanas laiks nav paredzams iepriekš. Šādas sistēmas tiek izmantotas mobilajos sakaros, plaukstdatoros u.c. MORABIT piedāvā ne tikai iebūvēto testu metodiku, bet arī praktisku realizāciju EJB komponenšu konteinera veidā. Atšķirībā no iepriekšējām metodikām, MORABIT precizē klienta-servera attiecības testu izpildes laikā, veidojot savu atbildību modeli (kura komponente veic testu un kura – tiek testēta, iekļaujot versiju, kad centralizēta ārēja komponente testē gan klienta, gan servera komponenti). Tāpat MORABIT ievieš testu izpildes politikas jēdzienu, katram testa aprakstam piekārtojot vienu vai vairākas izpildīšanas politikas. Tās ir:

- tukšgaitas testi – laikā, kad komponente neveic savas pamatfunkcijas, komponente drīkst veikt šī tipa testus;
- servera komponentes atrašanas brīža testi – pirms klienta komponente uzsāk sadarbību ar tai piedāvāto servera puses komponenti, tā drīkst izpildīt šī tipa testus, tādējādi nodrošinot, ka komponentes tiek savienotas tikai gadījumā, ja testi izpildījušies veiksmīgi;
- izpildes laikā veicamie testi – kad klienta komponente izsauc kādu servera komponentes metodi, tā drīkst izpildīt šī veida testus;
- periodiskie testi - kad ir izdevies izveidot savienojumu ar servera komponenti, klients drīkst ik pa laikam veikt šādus testus;
- topoloģijas izmaiņu testi - ja tiek mainīta komponenšu savienojumu topoloģija, klienta komponente drīkst veikt šādus testus servera komponentē.

Kaut arī MORABIT tehnoloģija piemērojama citam problēmu apgabalam nekā šajā darbā pētītā problēma, atsevišķas tajā pielietotās idejas iespējams izmantot arī programmatūras izpildes vides testēšanas mērķiem.

2.4. Instalāciju un konfigurāciju aprakstīšana

Kopš autonomās skaitļošanas iniciatīvas formulēšanas 2001. gadā, tikuši izstrādāti dažādi programmatūras inženierijas standarti un rekomendācijas, kuru galvenais uzdevums ir padarīt autonomu sistēmu izstrādi par praktisku disciplīnu. Kā viens no būtiskiem pētījumu virzieniem ir autonomo sistēmu darba vides (angl. *autonomic environment*) standartizācija,

kas turklāt vidi aprakstītu no ražotāja un izpildes platformas neatkarīgā veidā. Līdzīgi standarti tikuši veidoti arī pirms 2001. gada, piemēram, 1997. gadā publicētais *Marimba Corporation* (sadarbībā ar *Microsoft, Corp*) izstrādātais atvērtais programmatūras aprakstu formāts (angl. *Open Software Description Format*), kas ticis paredzēts Java programmatūras pakotņu savstarpējo atkarību aprakstīšanai [**48**]. Formāts ir XML bāzēta valoda (jāņem vērā, ka piedāvātais formāts radies aptuveni vienlaikus ar paša XML standarta pirmo versiju [**49**]), kas izmanto jēdzienus „programmatūras pakotne", (savstarpēja divu komponenšu) „atkarība", kā arī ļauj aprakstīt dažas prasības pret programmatūras izpilde vidi (brīvās diska vietas apjoms, operētājsistēma u.c.). Minētais formāts gan nav apstiprināts kā standarts vai rekomendācija, tomēr uzskatāms par tālākajās nodaļās aprakstīto deskriptoru idejas aizsācēju.

2.4.1. Instalāciju deskriptors

Sadarbojoties vairākām programmatūras izstrādes un pētniecības organizācijām, kā *IBM, InstallShield, Novell* un citām, ir izstrādāts un *W3C* konsorcijam 2004. gadā iesniegts priekšlikums standartizēt programmatūras uzstādīšanas aprakstu valodu un programmatūras pakotņu satura aprakstīšanas valodu [**50**]. Līdzīgi kā autonomo sistēmu manifests, arī izvietošanas deskriptors pēc publicēšanas skaidrots vairākos rakstos un citu veidu publikācijās [**51**,**52**].

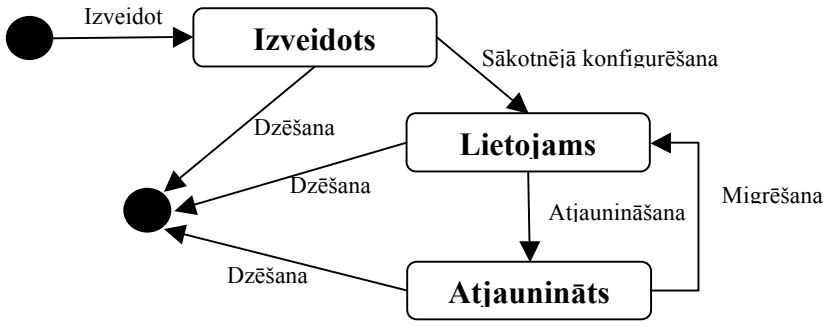
Piedāvātais risinājums – instalējamo vienību izvietošanas deskriptors jeb „*Installable Unit Deployment Descriptor, IUDD*" (angl.) – apraksta programmatūras sistēmu kā individuālu komponenšu kopumu, sniedzot iespēju aprakstīt ar sistēmu kopumā vai kādu komponenti atsevišķi veicamās darbības jebkurā identificējamā sistēmas uzturēšanas posmā. Viens no svarīgākajiem *IUDD* paredzētajiem lietojumiem ir automātisko instalācijas rīku būves vienkāršošana.

Risinājums paredz apskatīt programmatūras sistēmu kā atsevišķu instalējamu un konfigurējamu vienību kopumu (lietots termins „mazākā instalējamā vienība" – angl. „*smallest installable unit*", *SIU*), un aprakstīt šo vienību savstarpējās saistības un ar tām veicamās darbības, izmantojot standartizētu *XML* formātu, kas turpmāk viegli lasāms dažādiem sistēmu pārvaldības rīkiem. *IUDD* specifikācija apraksta [**50**] *XML* struktūru un sniedz atbilstošus piemērus.

IUDD paredz programmatūras sistēmas ar sadalītu arhitektūru un ļauj vienā deskriptorā aprakstīt sistēmas komponenšu izvietojumu visās mērķa vidēs (tiek lietots termins „mērķis", angl. *target*). Piemēram, mērķis var būt klienta dators, tīmekļa serveris, noslodzes balansēšanas serveris. Mērķa vides faktiskā konfigurācija tiek noskaidrota tikai sistēmas instalācijas laikā, bet deskriptorā tiek aprakstīta tikai mērķa vižu *topoloģija*. Katra

komponente var tikt uzstādīta vienā vai vairākās mērķa vidēs (piemēram, programmas ar grafisko saskarni – klienta datorā, tīmekļa lapas – tīmekļa serverī, bet koplietošanas funkciju bibliotēka – abās mērķa vidēs).

IUDD apskata programmatūras dzīves ciklu kā atsevišķu notikumu virkni, kur katrs notikums maina sistēmas kopējo stāvokli. Tā, piemēram, pēc notikuma „izveidot" sistēma nokļūst stāvoklī „izveidots". No šī stāvokļa, veicot notikumu „sākotnējā konfigurēšana" iespējams pāriet uz stāvokli „derīgs lietošanai" (Skat. ***2.4****. att.*, [**52**])



2.4. att. **Instalējamās vienības dzīves cikla stāvokļu diagramma**

Katru ar sistēmu veicamo operāciju iespējams ar *IUDD* līdzekļiem aprakstīt, iekļaujot tajā dažādas pārbaudes, ar instalējamo vienību veicamās darbības, citas darbības, kas veicamas instalācijas laikā. Izmantojot atjauninašanas mehānismu, tiek rasta iespēja veikt dažādas pārbaudes arī sistēmas jauninājumu uzstādīšanas laikā.

Lai instalējamā vienība tiktu uzstādīta mērķa vidē, tā var izvirzīt dažādas prasības attiecībā uz mērķa vidi un pieprasīt veikt dažādas pārbaudes mērķa vidē (angl. *checks*). *IUDD* nodala prasības no pārbaudēm – prasība ir nosacījums, kura patiesumu var noskaidrot, veicot vienu vai vairākas pārbaudes. Tā, piemēram, lai nepieļautu komponentes uzstādīšanas mēģinājumus citā operētājsistēmā nekā tā paredzēta (piem., *MS Windows XP*), komponente var izvirzīt prasību „*mērķa vides nosaukums = 'Windows XP'*". Šīs prasības patiesumvērtības noskaidrošanai tiek veikta pārbaude, vai datoram, kurā komponenti instalē, operētājsistēmas nosaukums ir „Windows XP". Deskriptors piedāvā sekojošus veicamo pārbaužu veidus:

- izmitināšanas vides (angl. *hosting environment*) *ietilpība* specificē kāda noteikta programmatūras izpildes vides atribūta (piemēram, procesora ātruma) minimālo vai maksimālo vērtību;
- *resursu patēriņš* izmitināšanas vidē apraksta to, cik lielu kāda resursa (piemēram, brīvās diska vietas) kvantumu instalējamā vienība dotajā vidē patērēs;

- *atribūtu pārbaudes* paredzētas, lai salīdzinātu kāda noteikta izmitināšanas vides atribūta vērtību ar to, kas norādīta deskriptorā (piemēram, „operētājsistēmas nosaukums”);
- *programmatūras pārbaude* paredzēta, lai pārbaudītu, vai izmitināšanas vidē uzstādīts noteikts programmatūras resurss (ko identificē pēc nosaukuma un versijas numura);
- *instalējamo vienību pārbaude* ļauj aprakstīt atkarības starp instalējamajām vienībām;
- *attiecību pārbaude* (angl. *relationship*) specificē attiecību, kādai jāpastāv starp divām mērķa vidēm;
- *pašu definētas pārbaudes* ļauj veikt brīvā formā aprakstītas pārbaudes attiecībā pret mērķa vidi.

Iebūvētajiem prasību veidiem *IUDD* paredzēti atbilstoši *XML* elementi, kas ļauj aprakstīt nepieciešamo prasību. Koda piemērā ***2.5****. att.* parādīta prasība pret instalācijas vidi „*Valid_Platform_With_DB*”, kas liedz instalēt aprakstīto sistēmu nepiemērotā vidē. Prasība sastāv no divām pārbaudēm – atribūta pārbaude, kas noskaidro, vai atribūta „*OsType*” vērtība sakrīt ar „*Windows XP*” un programmatūras pārbaude, kas noskaidro, vai mērķa vidē ir instalēta nepieciešamā datubāzu pārvaldības sistēmas *DB2* versija. Abas prasības apvienotas vienā „*alternative*” elementā. *IUDD* paredz, ka vienu un to pašu prasību var apmierināt vairākos veidos, t.i., ja kaut viena no aprakstītajām alternatīvām ir spēkā, prasība tiek uzskatīta par izpildītu. Katrai prasībai jānorāda arī sistēmas operāciju saraksts, kuras veicot prasībai jābūt spēkā (piemēram, pieejamās diska vietas ierobežojums ir spēkā darbības „Izveidot” laikā, bet vēlākām atjaunināšanas darbībām prasība vairs var nebūt būtiska).

```
<requirement name="Valid_Platform_With_DB" operations="Create">
  <alternative name="WindowsDB2">
    <property checkId="Windows_XP_Check">
      <propertyName>OsType</propertyName>
      <value>Windows XP</value>
    </property>
    <software checkVarName="db2_for_Windows_check">
      <UUID>22345678901234567890123456789012</UUID>
      <name pattern="true">(DB2|Universal Database)</name>
      <minVersion>7.2</minVersion>
      <maxVersion>8.1</maxVersion>
    </software>
  </alternative>
</requirement>
```

2.5. att. ***IUDD*** **shēmā aprakstītas prasības pret instalācijas vidi**

Gadījumā, ja nepieciešams aprakstīt specifisku prasību pret instalācijas vidi, kas nav iekļauta standarta prasību sarakstā, *IUDD* ļauj prasībā iekļaut pašu veidotas pārbaudes. Šim

mērķim lietojams pārbaužu sarakstam pievienojams elements CUSTOM. Koda piemērā (skat. ***2.6****. att.*, [**50, 89**]) parādīts pašu definētas pārbaudes „atslēgas vērtība noteiktā *MS Windows* reģistra koka zarā" izsaukums.

```
<custom artifactIdRef="winRegCustomCheck"
        checkId="GSK4_Win_Registry_Check">
     <parameter variableNameRef="Win_Reg_Hive">
       HKEY_LOCAL_MACHINE</parameter>
     <parameter variableNameRef="Win_Reg_Key">
       SOFTWARE\IBM\GSK4\CurrentVersion\Version</parameter>
     <parameter variableNameRef="Win_Reg_Type">
       REG_SZ</parameter>
     <parameter variableNameRef="Win_Reg_Value">
       4.0.2.49</parameter>
</custom>
```

2.6. att. **Pašu definētu pārbaužu izsaukšana *IUDD* deskriptorā**

Viegli pamanīt, ka koda piemērā netiek parādīts, kā attiecīgo pārbaudi veikt, tikai definētas prasības pret mērķa vidi. To, kāds kods izmantojams attiecīgās pārbaudes veikšanai, jāpiemeklē programmai, kas interpretē *IUDD* deskriptoru. Viens no CUSTOM elementa atribūtiem ir *artifactIdRef*, kas sasaista norādīto pārbaudi ar *artefaktu*, kas atbildīgs par šādu pārbaužu veikšanu.

Pārbaudes artefakts jāapraksta, lietojot *IUDD* konstrukciju „*customCheckArtifact*", kas specificē gan to, kāds ir minētā artefakta interfeiss (parametri, ko artefakts var saņemt vai atgriezt), gan to, kurš koda fails realizē attiecīgo pārbaudi (skat. koda piemēru ***2.7****. att.* piemēru, [**50, 89**]).

```
<customCheckArtifact artifactId="winRegCustomCheck">
   <fileIdRef>WinRegistryCheckArtifact</fileIdRef>
   <parameterMaps>
       <map>
         <internalName>Win_Reg_Hive</internalName>
         <externalName>Reg_Hive</externalName>
       </map>
       <map>
         <internalName>Win_Reg_Key</internalName>
         <externalName>Reg_Key</externalName>
       </map>
       <map>
         <internalName>Win_Reg_Type</internalName>
         <externalName>Reg_Type</externalName>
       </map>
       <map>
         <internalName>Win_Reg_Value</internalName>
         <externalName>Reg_Value</externalName>
       </map>
   </parameterMaps>
</customCheckArtifact>
```

2.7. att. **Parametru nosaukumu un faila piesaiste pašu definētiem pārbaudes artefaktiem**

IUDD nesniedz precīzāku specifikāciju tam, kā panākt, lai norādītais artefakts veiktu vēlamo pārbaudi. Praksē attiecīgas vides pārbaudes varētu veikt gan ar operētājsistēmas komandu skriptiem (piemēram, *Windows* operētājsistēmā – *BAT* fails), gan ar speciāli veidotu koda bibliotēku (piemēram, *Windows* operētājsistēmā – *DLL* bibliotēka). Kaut gan tiek skaidri noteikts, kā attiecīgajam artefaktam nodot parametru vērtības, nav noteikts, kā tiek atrasts un iedarbināts nepieciešamais pārbaudes mehānisms. Piemēram, ja pārbaudi veic kāda *DLL* bibliotēkā iekļauta programmatūras koda klases metode, būtu nepieciešams veids, kā specificēt koda klases un metodes nosaukumu.

2.4.2. Programmatūras izvietošanas deskriptors

Turpinot attīstīt instalāciju deskriptora ideoloģiju, vairākas programmatūras izstrādes organizācijas kopīgi izstrādājušas jaunu standartu – risinājumu izvietošanas deskriptoru (angl. *Solution Deployment Descriptor*, *SDD*) [**53**]. Standarta atbalstītāji – *IBM, Novell, SAP, Macrovision* un citas organizācijas – apvienojušies darbam kā *OASIS* (angl. *Organization for the Advancement of Structured Information Standards*) [**54**] tehniskā komiteja. Šī komiteja laikā no 2004. gada līdz 2008. gadam izstrādājusi vairākus dokumentus, kas kopā veido risinājumu izvietošanas deskriptora standartu. 2008 .gada 1. septembrī šīs tehniskās komitejas izstrādātais deskriptors atzīts par *OASIS* līmeņa standartu.

Skaidrojot nepieciešamību pēc *SDD*, [**55**] autori norāda: „Padarot uz ārpusi pieejamu izvietošanas informāciju, kas līdz šim bijusi noslēpta programmatūras kodā, rodas dažādi ieguvumi. *SDD* lietotāji – gan cilvēki, gan programmatūra – var izmantot *SDD* iekļautās ziņas par programmatūras prasībām un instalēšanas rezultātiem, gan lai veiksmīgi izvietotu programmatūru mērķa vidē, gan arī lai vienkāršotu izvietošanas un citu uzturēšanas operāciju plānošanu."

Līdzīgi kā *IUDD*, arī *SDD* apraksta programmatūras pakotni kā atsevišķu instalējamu vienību komplektu, starp kurām var pastāvēt savstarpējas saites un kas jāizvieto noteiktā vidē. Atšķirībā no priekšgājēja, *SDD* vairāk pielāgots reālajam paredzētajam lietojumam. Divi būtiskākie uzlabojumi, salīdzinot ar *IUDD*, ir selektīvas instalēšanas iespēja un profila jēdziena ieviešana.

Selektīvā instalācija paredz, ka programmatūras pakotne var sastāvēt no pamatkomponentēm (angl. *base content*), komponentēm, ko uzstāda pēc izvēles (angl. *selectable content*) un lokalizētā satura (angl. *localized content*).

SDD apraksta programmatūru un tās prasības pret programmatūras izpildes vidi, pēc iespējas izvairoties norādīt platformu nosaukumus, ražotāju u.c. Deskriptora standarta izstrādātāji īpaši norāda, ka standarts paredzēts, lai aprakstītu sasniedzamos mērķus, nevis

veidu, kā tie sasniedzami vai kā mērīt, vai attiecīgais mērķis sasniegts. Lai programmatūras izstrādātāji varētu savstarpēji vienoties par vienotu nomenklatūru un terminoloģiju, *SDD* izmanto *profila* jēdzienu. Profils apraksta attiecīgajā deskriptora realizācijā izmantoto vārdnīcu un tādējādi nodrošina iespēju aprakstīt platformas sniegtos pakalpojumus un programmatūras paketes pieprasīto vidi vienotā valodā. [**55**] minēts piemērs – ja instalējamās vienības prasība pret apkārtējo vidi ir „nepieciešama operētājsistēma *Linux*", šis jēdziens tiks interpretēts, vadoties pēc deskriptora profilā izmantotās definīcijas attiecīgajam terminam. Lai vienkāršotu profilu izstrādāšanas turpmāko praksi, *SDD* autori piedāvā instalācijas profila sākotnējo paraugu (angl. *starter profile*) [**56**], kura pamatā ir *DMTF*[1] izstrādātais vispārīgās informācijas modelis *CIM* [**57**].

Būtiskas atšķirības novērojamas *SDD* sadaļās, kas attiecas uz prasībām pret instalācijas vidi. *IUDD* gadījumā tika skaidrots, ka prasība „sastāv no" pārbaudēm, turklāt prasība tiek uzskatīta par izpildītu tad, ja visas pārbaudes beigušās veiksmīgi. *SDD* standarts [**53, 84**] apraksta jēdzienu „prasība" kā „prasība sastāv no atsevišķiem resursu ierobežojumiem, kuriem saskaņā ar attiecīgā *SDD* dokumenta instancē definētām prasībām jābūt spēkā, lai veiksmīgi uzstādītu vai lietotu programmatūru, kas aprakstīta šajā *SDD* dokumentā". Viegli pamanīt, ka *SDD* arī šajā gadījumā pielieto principu „pateikt, ko darīt, bet nepateikt, kā tas veicams".

SDD joprojām saglabā daļu veicamo pārbaužu veidu, kā tie aprakstīti *IUDD* (bet tos apzīmē ar jēdzienu „ierobežojums"): pieejamās ietilpības ierobežojums, vietas patēriņa ierobežojums, atribūta vērtības ierobežojums, programmatūras versijas ierobežojums, komponenšu savstarpējo attiecību ierobežojums. Papildus iepriekšējiem ierobežojumu veidiem *SDD* apraksta arī „unikalitātes ierobežojumu", kas ļauj pieprasīt, ka viens un tas pats resurss drīkst vai nedrīkst būt unikāls programmatūras pakotnē.

Piemēram, lai aprakstītu uzstādāmās sistēmas prasību, ka nepieciešami vismaz 15 megabaiti brīvas vietas un failu sistēmas tipam jābūt „*NTFS*", *SDD* deskriptors jāveido, kā parādīts ***2.8****. att.* [**58**]. Prasība ar nosaukumu „*DiskSpace*" attiecināma uz programmatūras pakotni gan instalācijas, gan vispārējās lietošanas laikā (*SDD* papildina *IUDD* definēto operāciju sarakstu ar jaunu operāciju „*use*", kas apzīmē jebkuru laika momentu, kad attiecīgais resurss var būt nepieciešams). Prasība sastāv no diviem ierobežojumiem – diska patēriņa ierobežojuma un atribūta vērtības ierobežojuma, abus šos ierobežojumus piesaistot resursam „*Filesys*". Katrs resurss, uz kuru attiecināmi jebkādi ierobežojumi, atsevišķi jāapraksta „*Topology*" sadaļā (skat. piemēra beigu daļu). Tādējādi jēdziens „*Filesys*" tiek

[1] DMTF – Distributed Management Task Force, 1992. gadā dibināta organizācija, kas izstrādā un uztur ar datorsistēmu pārvaldību saistītus standartus.

sasaistīts ar pakotnei pieejamo resursu – failu sistēma. Savukārt „failu sistēma" tiek sasaistīta ar kādu konkrētu uzstādīšanas profilā minētu šī jēdziena interpretāciju.

```
<SDD-dd:DeploymentDescriptor
      xmlns:sp="http://docs.oasis-open.org/SDD/ns/starterProfile"
  ... >
  ...
  <SDD-dd:Topology>
     <SDD-dd:Resource id="os" type="sp:CIM_OperatingSystem">
        <SDD-dd:HostedResource id="Filesys" type="sp:CIM_FileSystem"/>
     </SDD-dd:Resource>
  </SDD-dd:Topology>
  ...
  <SDD-dd:Requirement id="DiskSpace" operation="install use">
     <SDD-dd:ResourceConstraint
            id="DiskSpaceRequirement" resourceRef="Filesys">
        <SDD-dd:ConsumptionConstraint>
           <SDD-dd:PropertyName>
               sp:CIM_FileSystem.AvailableSpace
            </SDD-dd:PropertyName>
           <SDD-dd:Value unit="megabytes">15</SDD-dd:Value>
        </SDD-dd:ConsumptionConstraint>
        <SDD-dd:PropertyConstraint>
           <SDD-dd:PropertyName>
               sp:CIM_FileSystem.Type
            </SDD-dd:PropertyName>
           <SDD-dd:Value>NTFS</SDD-dd:Value>
        </SDD-dd:PropertyConstraint>
     </SDD-dd:ResourceConstraint>
  </SDD-dd:Requirement>
  ...
</SDD-dd:DeploymentDescriptor>
```

2.8. att. **Diska vietas un failu sistēmas veida prasību aprakstīšana *SDD* deskriptorā**

Koda piemērs veidots saskaņā ar *SDD* piedāvāto „*starter profile*", tādējādi jēdziens „*File System*" uztverams kā *CIM* modelī [**57**] definētais atbilstošais jēdziens. Tādējādi tiek nodrošināts, ka „REQUIREMENTS" sekcijā minētās prasības apraksta jēdzienus, izmantojot to pašu vārdnīcu, ko definē *CIM* modelis – piemēram, prasība „pieejamā diska vieta" attiecināma uz *CIM_FileSystem* tipa objekta atribūtu *CIM_FileSystem.AvailableSpace*. Deskriptora dokumentā CIM jēdzieni tiek izmantoti, atsaucoties uz *XML* vārdkopu „*sp*" – *XML* shēmu, kas *XML* valodā pārskaita visus „*starter profile*" izmantotos *CIM* jēdzienus.

Kā redzams no piemēra, *SDD* veiksmīgi sasniedz mērķi abstrahēties no operētājsistēmu, izpildes platformu un citiem lietojumam specifiskiem jēdzieniem – tas izmanto instalācijas profilu, kas nodrošina piesaisti reālās vides objektiem un to metrikām. Programmatūras izstrādātāji un instalācijas rīku izstrādātāji var vienoties par noteikta profila izmantošanu (vai izstrādāt jaunu specifisku profilu), lai savstarpēji saskaņotu jēdzienu interpretācijas.

Tādējādi, ja nepieciešams izstrādāt pārbaudes mehānismu pašu definētām prasībām pret izpildes vidi, jāveic sekojoši soļi:

- deskriptora topoloģijā jādefinē resurss, uz kuru attieksies nepieciešamais ierobežojums,
- ja resurss nav jēdzieniski aprakstīts līdz šim izmantotajā instalācijas profilā, jāpaplašina profila jēdzienu saraksts, iekļaujot arī jauno resursu, tā atribūtus un atribūtu mērvienības,
- prasībai jābūt aprakstāmai ar kādu no ierobežojumu veidiem, ko definē *SDD* standarts – ietilpības vietas patēriņa, atribūta vērtības (un pārējie).

Līdzīgi kā *IUDD*, arī *SDD* tālāk nespecificē, kā instalācijas programmatūrai izpildīt pārbaudes, lai noskaidrotu prasību atbilstību realitātei.

2.5. Servisa kvalitātes mērīšanas metodes

Viens no programminženierijas virzieniem, kas šķietami orientēts uz šajā darbā pētītajām problēmām līdzīgu situāciju apstrādi, ir servisa kvalitātes mērīšanas metodes. Šī disciplīna vēsturiski cēlusies no datoru komunikāciju tīkliem, kur termins *QoS* (angl. *quality of service*) visbiežāk tiek saistīts ar resursu pieejamības rezervēšanas un kontroles mehānismiem, nevis attiecīgo resursu sniegtā servisa kvalitātes mērīšanu.

Atsevišķi pētījumi šo terminu pielieto arī kā datorsistēmu, sevišķi autonomo sistēmu īpašības aprakstu, tomēr, saskaņā ar autoram zināmo informāciju, šis virziens ir relatīvi maz izpētīts. Viens no pētījumiem, kas apskata iespējas servisa kvalitātes kontraktus pielietot datorsistēmu būvē [**59**], piedāvā veidot komponenšu mitināšanas serveri, kurā komponentes varētu, lietojot *QoS* līdzekļus, pieprasīt noteikta resursa pieejamību un servera puse garantētu pieprasījumu izpildi vai informētu klienta komponenti par prasītā resursa nepieejamību.

S.Ansaloni un citi [**60**] attīsta ideju par servisa kvalitātes metriku pielietošanu savienojamu ierīču vai datorsistēmu kontraktu aprakstīšanai. Šis princips galvenokārt attiecināms uz mobilo ierīču savienošanu, kur divām vai vairākām ierīcēm jāveic kāds kopīgs uzdevums. Tiek piedāvāta valoda, kas ļauj aprakstīt „kvalitātes standartus" – nosacījumus, kādiem jāatbilst saistītajai ierīcei, lai ar to būtu iespējams nodibināt savienojumu – piemēram, saistītās ierīces procesora takts frekvencei jābūt lielākai par 266MHz un procesora noslodzei mazākai par 70%. Autori gan nenorāda, kā tiek veikta attiecīgo atribūtu nolasīšana, kas programmatūras izpildes vides testēšanai ir būtiskākā šī pētījuma daļa.

Izmantojot līdzīgu valodu kā Ansaloni piedāvātā, [**61**] apraksta metodiku, kas ļautu programmatūrai pētīt apkārtējās vides atsevišķu atribūtu vērtības un saņemtās zināšanas izmantot programmatūras darba optimizācijai. Atšķirībā no šajā promocijas darbā piedāvātā

risinājuma, rakstā tiek apskatītas sistēmas, kam ir jau iepriekšējas zināšanas par tiem vides atribūtiem, kas var atstāt iespaidu uz programmatūras darbību. Tādējādi programmatūra mēra tikai tai būtisko atribūtu vērtības un vienlaikus pielāgo savu darbību nolasītajām vērtībām. Tiek apskatīts piemērs, ka video straumēšanas programmatūra, novērojot datortīkla ātruma krišanos vai procesora noslodzes pieaugšanu, samazina video izšķirtspēju.

2.6. Saistīto pētījumu rezultātu pielietojamība vides automātiskai testēšanai

Pārskatot iepriekšējās nodaļās aplūkotos saistītos pētījumu virzienus, tos var iedalīt nosacīti divās grupās – ideoloģiskie pētījumi, kas formulē mērķus, uz kādiem būtu vēlams tiekties, lai izstrādātu vieglāk uzturamu programmatūru, un tehnoloģiskie pētījumi, kas apraksta konkrētas metodes kāda mērķa sasniegšanai. Pie ideoloģiskajiem pētījumiem iekļaujama gan autonomās skaitļošanas ideoloģija (skat. 2.1 nodaļu), kas apraksta visai tālus un ar pašreiz lietotiem programmatūras dzīves cikla arhitektūru veidiem grūti savienojamus mērķus, gan viedo tehnoloģiju programmatūras ideoloģija (skat. 2.2 nodaļu), kas apraksta ar pašreizējo programmatūras izstrādes praksi saderīgu metožu ideoloģiju. Šis pētījums ir viens no viedo tehnoloģiju idejas pamata aspektiem, tādējādi var uzskatīt, ka viedo tehnoloģiju un šajā darbā apskatīto metožu mērķi sakrīt.

Salīdzinot šajā darbā izvirzītos mērķus ar autonomās skaitļošanas mērķiem – izstrādāt programmatūru, kas spēj patstāvīgi analizēt programmatūras izpildes vidi un saistīt savu darbu ar analīzes rezultātiem – var secināt, ka metodēm ir vairāki kopīgi mērķi. Šādas programmatūras īpašības atbilst vairākiem pašārstēšanās principa uzdevumiem – pārraudzībai, interpretācijai, diagnostikai [**28**]. Promocijas darbā pētītās problēmas risinājumam būtu vēlams atturēties no mērķa dinamiski (no programmatūras koda puses) mainīt konfigurāciju, jo ārējās vides konfigurācijas izmaiņas var atstāt negatīvu iespaidu uz citām programmatūras sistēmām, kas tiek darbinātas šajā pašā vidē. Par šādu izmaiņu veikšanu jālemj cilvēkam, balstoties uz programmatūras sniegtajām ziņām par vēlamajām izmaiņām.

Pārējās iepriekšējās nodaļās apskatītās ar šo pētījumu saistītās metodes – dažādas iebūvēto testu tehnikas, programmatūras pakotņu deskriptori un servisa kvalitātes mērīšanas metodes uzskatāmas par tehnoloģiskiem pētījumiem, kas apraksta kāda izvēlēta mērķa sasniegšanai pielietojamās metodes.

Var secināt, ka neviena no tehnoloģiskajām metodēm tiešā veidā viena pati nerisina šajā darbā izvirzīto problēmu – panākt, ka darījumu programmatūra patstāvīgi veic izpildes vides pārbaudes, turklāt veicamo pārbaužu daudzums un veids ir brīvi maināms jebkurā programmatūras dzīves cikla fāzē, nemainot pašu darījumu programmatūru. Tomēr arī katras

apskatītās tehnoloģiskās metodes pamatā ir sava ideoloģiskā bāze, ko iespējams daļēji pārņemt, arī lai rastu risinājumu šajā darbā apskatītajai problēmai.

Programmatūrā iebūvēto testu ideja daļēji sader ar šajā darbā meklēto risinājumu – tā piedāvā pašā programmas kodā iekļaut pārbaudes, kas palīdzētu noskaidrot, vai programma ir gatava darbam. Pārbaudes paredzētas pašas komponentes darbības testēšanai vai vairāku komponenšu savstarpējo „kontraktu" izpildes testiem, taču šī ideja sākotnēji iecerēta kā programmatūras testēšanas metode, t.i., kā risinājums, kas palīdz identificēt programmēšanas vai projektēšanas kļūdas. Sevišķi labi šo mērķi ilustrē testu bāzētās izstrādes princips un automatizētās vienībtestēšanas princips ([**39**], [**40**]). Līdzīgi kā komponenšu savstarpējo kontraktu testu gadījumā, programmatūrā varētu iebūvēt arī testus, kas pārbauda programmatūras izpildes vides piemērotību komponentei. Tomēr šeit jāņem vērā, ka tādā gadījumā iespējamo pārbaužu loks tiek ierobežots tikai ar tām pārbaudēm, par kurām zināms jau programmatūras izstrādes laikā, bet vēlāk parādījušies ierobežojumi nav automātiski pārbaudāmi. Piemēram, var iedomāties programmatūras sistēmu, kas datu glabāšanai izmanto *Microsoft SQL Server*. Sistēma izstrādāta un veiksmīgi testēta ar „*SQL Server 2000*" versiju. Dažus gadus vēlāk, parādoties „*SQL Server 2005*", noskaidrojies, ka sistēma ir tikai daļēji saderīga ar jauno datubāzu platformu, to konfigurējot „atpakaļejošās saderības" režīmā [**62**]. Tā rezultātā apskatītā programmatūras sistēma ir apdraudēta gadījumā, ja to mēģinātu uzstādīt vidē, kur instalēta jaunā datubāzu vadības sistēma vai gadījumā, ja atjauninātu esošo datubāzu serveru programmatūru. *SQL* servera versijas pārbaude ir relatīvi viegli veicama programmiski. Tomēr, tā kā veicamās vides pārbaudes tiek „iekodētas" pašā darījumu sistēmā, to nav iespējams pievienot jau esošajam pārbaužu sarakstam, nemainot pašu programmatūru. Šis ir viens no galvenajiem iemesliem, kādēļ šajā darbā tiek meklēts veids, kā pārbaudes „iznest" ārpus darījumu programmatūras koda, lai tādējādi iegūtu iespēju veicamo pārbaužu sarakstu mainīt atkarībā no lietojuma un turklāt vienu un to pašu pārbaudes rīku varētu pielietot dažādām darījumu sistēmām. Atšķirībā no iebūvēto testu metodes, šajā darbā piedāvāts risinājums, kur komponente definē interfeisa punktus iebūvēto testu izsaukšanai, bet paši izpildāmie testi glabājas ārpus komponentes.

Vairāki autori, aprakstot programmatūrā iebūvēto pārbaužu ideoloģiju, norāda uz nepieciešamību pārbaužu veikšanas laikā darbināt programmatūru kādā speciālā izpildes režīmā (uzturēšanas režīmā). Šī režīma būtība ir – nodrošināt, ka sistēma neveic nekādas darbības, kas varētu ietekmēt tās izmantotās datu glabātuves vai citu sistēmu darbību. Tāpat tiek minēts, ka gadījumā, kad iebūvēto testu metode tiek pielietota sadalītu sistēmu būvei, servera tipa komponentēm jāveido atsevišķs interfeiss, kas paredzēts tikai iebūvēto testu izpildīšanai (pat vairāki – viens, ko klienta puses komponente var izsaukt, lai servera puses

komponente patstāvīgi veiktu paštestēšanu, otrs – ko klienta komponente drīkst izmantot laikā, kad tā veic paštestēšanu, bet servera komponentei jāimitē normāls darbs). Šo ideju izmanto arī šajā darbā piedāvātais programmatūras izpildes vides testēšanas risinājums.

Instalāciju un programmatūras pakotņu deskriptoru standarti (attiecīgi - *IUDD* un *SDD*) definē veidus, kā aprakstīt uzstādāmo programmatūras komponenšu prasības pret programmatūras izpildes vidi. Tas liek jautāt, vai šie mehānismi (galvenokārt *SDD*, kas aizstāj *IUDD* kā vēsturiski jaunākais) nav viegli izmantojami šī darba pētījuma mērķiem? Tomēr, mēģinot pielāgot kādu no deskriptoriem nepieciešamajam lietojumam, atklājas tehnoloģiski ierobežojumi, kas neļauj veikt vai ierobežo iecerēto risinājumu.

Tā, piemēram, ar *IUDD* līdzekļiem iespējams aprakstīt sešu veidu standarta pārbaudes un veidot pašu definētas pārbaudes (detalizētāk skat. 2.4.1 nodaļā). Taču standartā netiek detalizētāk norādīts, kā interpretēt „*customCheckArtifact*" elementa saturu. Līdzīgi arī *SDD* standarts ļauj aprakstīt jebkādus ierobežojumus programmatūras lietošanai, ja tie attiecināmi uz kādu no pakotnes topoloģijā pieminētajiem resursiem, taču nenorāda veidu, kā pārbaudes veicamas. Tas saistīts ar deskriptoru vēsturisko izcelsmi – tie radušies, cenšoties izveidot formālu valodu programmatūras instalāciju aprakstīšanai – šī procesa ieejas un izejas datus, instalēšanai veicamās darbības un pārbaudes. Deskriptoru galvenokārt paredzēts izmantot instalācijas rīkos, kā, piemēram, *InstallShield* vai *NSIS*, tādēļ deskriptora korekta interpretācija (*kā* veikt to vai citu darbību) tiek „uzticēta" šīm programmām.

SDD standarts paredz, ka katrs instalācijas rīku izstrādātājs var definēt atsevišķu instalācijas profilu, kurā aprakstīti dažādi videi raksturīgie resursu veidi un to metrikas. Tātad, ja tiktu mēģināts *SDD* pielietot programmatūras izpildes vides testiem, būtu jāievieš pašu definēts instalācijas profils, kas dinamiski jāsaista ar piemērotiem vides pārbaudes rīkiem. Gadījumā, ja būtu nepieciešams aprakstīt tāda veida prasības pret programmatūras izpildes vidi, kas saistītas ar kādu līdz šim neizmantotu resursu, tā atribūti un metrikas būtu jāpievieno izstrādātajam profilam, bet pats resurss – jāiekļauj attiecīgās programmatūras pakotnes deskriptora topoloģijas sadaļā. Kaut arī šāds risinājums tehniski ir iespējams, tā lietošana ne vienmēr būs vienkārša.

Piemēram, varam apskatīt sistēmu, kas datu glabāšanai izmanto *SQL* serveri un kuras jaunākajā versijā nedaudz mainīta datubāzes shēma – kādas tabulas vienam no teksta laukiem noņemts garuma ierobežojums. Uzstādot programmatūras jauninājumu, datu piekļuves komponentei vajadzētu veikt pārbaudi, vai pirms tam ir uzstādītas datubāzes shēmas izmaiņas. Lai lauka tipa pārbaudi aprakstītu kā prasību komponentes deskriptorā:

- sākotnēji tabulas lauks jādefinē kā izpildes vides resurss (līdzīgi kā ***2.8***. *att.* tiek definēta failu sistēma) pakotnes topoloģijas sadaļā,

- instalācijas profilā jābūt definētam atbilstošam resursa tipam (instalācijas profils jāpapildina, ieviešot visus nepieciešamos datubāzu jēdzienus, piemēram, tā kā no *CIM* modeļa [**57**] atvasinātā *XSD* shēma, nedefinē tik detalizētus *SQL* jēdzienus, tie jāievieš – jēdziens "lauks", tā atribūts „lauka garums ir ierobežots", bet tā pieļaujamās vērtības - „jā" vai „nē"),
- aprakstot meklēto lauku pakotnes topoloģijā, tas jānorāda identificējamā veidā (norādot hierarhiju „serveris/ datubāze / tabula/ lauks"),
- pakotnes prasību sarakstā jāpievieno atribūta vērtības ierobežojums – attiecīgajam laukam atribūta „garums ir ierobežots" vērtība nedrīkst būt „jā".

Uzdevums sarežģījas gadījumos, kad nepieciešams definēt prasības, kas gan pieprasa kādu noteiktu programmatūras izpildes vides īpašību, taču tā nav viegli aprakstāma, izmantojot tikai *SDD* piedāvātos attiecību veidus.

Šeit minētais ļauj spriest, ka *SDD* deskriptora standarts var tikt pielietots programmatūras izpildes prasību aprakstīšanai, taču tas tiešā veidā nesniedz pilnīgu šajā darbā aprakstītās problēmas risinājumu. Deskriptorā ir iespējams fiksēt dažādu veidu prasības, taču tas nepiedāvā veidu, kā prasībām piemeklēt atbilstošus pārbaudes rīkus, turklāt pilnībā distancējas no pārbaužu izpildes aprakstīšanas. Turpmākajās darba nodaļās apskatīti gan risinājumi, kuru pamatā izmantots *SDD* deskriptors, gan arī risinājumi, kas izmanto speciāli definētu prasību aprakstīšanas valodu.

3. PIEDĀVĀTAIS VIDES TESTĒŠANAS PROBLĒMAS RISINĀJUMS

3.1. Vispārīgie principi

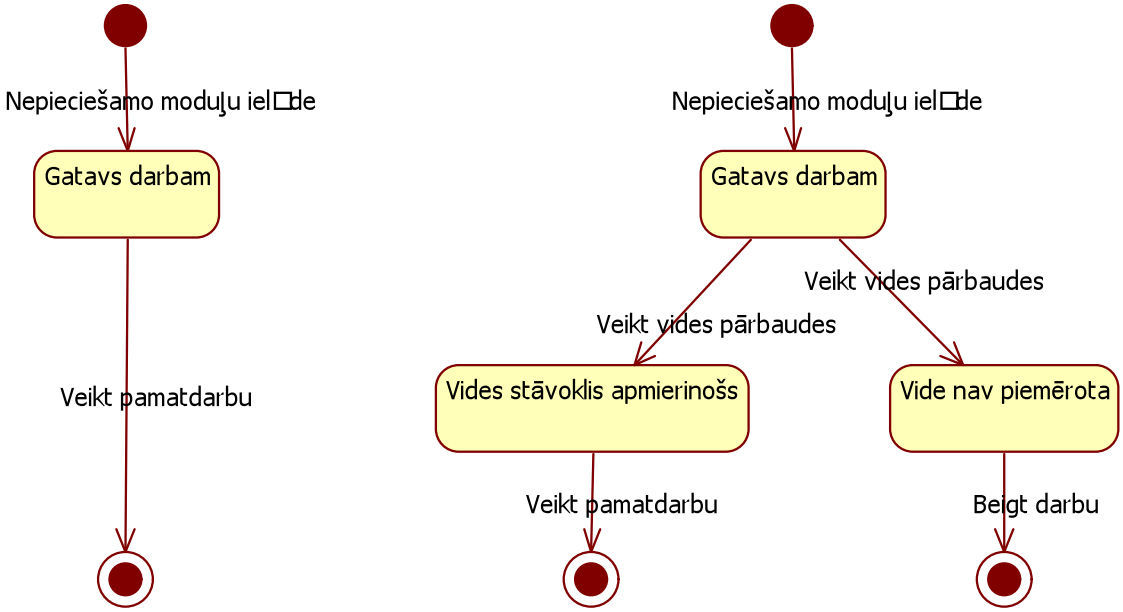
Programmatūras izpildes vides testēšanas ideja balstīta uz pieņēmuma, ka programmatūra ir piemērota darbam noteiktā programmatūras izpildes vides konfigurācijā. Kā norāda atsevišķi autori, programmatūra bieži vien tiek izstrādāta, balstoties uz pieņēmumiem par citu komponenšu dabu, nevis vadoties pēc to specifikācijas ([**63**,**64**]). Līdzīgs efekts novērojams arī attiecībā uz programmatūrā iekļautiem pieņēmumiem attiecībā uz programmatūras izpildes vidi – tiek pieņemts, ka programmatūra, kas sekmīgi darbojas izstrādes vidē, tikpat labi darbosies arī pēc uzstādīšanas citur. Šādu pieņēmumu rezultātā, pārnesot programmatūru citādā, no sākotnējās izstrādes atšķirīgā izpildes vidē[1], programmatūra var nedarboties nemaz vai darboties daļēji pareizi.

Pamatnostādne šajā darbā aprakstītajā pētījumā ir – programmatūrai patstāvīgi jāveic tās izpildes vides analīze, lai noskaidrotu, vai vide ir piemērota normālam darbam. Šādas pārbaudes veicamas regulāri, ne tikai pēc programmatūras instalācijas, bet arī vēlāk programmatūras uzturēšanas fāzes laikā. Informācija par to, kādas pārbaudes tiek veiktas, nedrīkstētu glabāties pašā programmas kodā, tai jābūt pieejamai ārpus programmatūras koda, lai datorsistēmas uzturētāji varētu analizēt veicamo pārbaužu sarakstu un nepieciešamības gadījumā to papildināt vai mainīt. Šī nostādne ir atšķirīga no citām praksē lietotām pieejām – gan no tām, kas tiecas programmatūras izpildes vidi pārbaudīt tūlīt pēc instalēšanas vai jauninājumu uzstādīšanas, gan arī no tām, kas veic vides pārbaudes, bet tās „paslēpj" programmatūras kodā, tādēļ šo pieeju rezultāti nav tiešā veidā pielietojami tādai vides testēšanas metodikai, kāda apskatīta šajā darbā. Tomēr, lai izprastu pasaules praksi un citu pētnieku idejas, šī darba turpmākajās nodaļās tiks apskatītas arī līdzīgas metodes, kuru pielietojuma mērķi atšķiras no šeit apskatītajiem, taču atsevišķas risinājuma idejas var būt viegli pārņemamas vai adaptējamas.

Būtiska šajā darbā apskatītās *viedās* programmatūras atšķirība no „klasiskas" programmatūras ir spēja katrā darba sesijā pirms darba uzsākšanas veikt darba spēju *pārbaudes*, lai noskaidrotu, vai pašreizējā vidē tā spēs pilnvērtīgi strādāt, līdzīgi kā tas tiek darīts datorsistēmu aparatūrā. (Skat. ***3.1*** *att.*) Lietojot šādu pieeju, programmatūra speciāli

[1] Šeit ar jēdzienu „vide" tiek saprasta loģiski nodalīta datorsistēmu infrastruktūra. Tā var būt ģeogrāfiski nodalīta (piemēram, ja izstrādātājs un pasūtītājs ir dažādas institūcijas) vai ģeogrāfiski vienota, taču loģiski šķirta (ja izstrāde notiek organizācijas iekšienē)

jāsagatavo tam, ka tiks veiktas šādas pārbaudes. Tā, piemēram, tādas programmatūras gadījumā, kad tiek izmantota grafiskā lietotāja saskarne, programmatūra būtu jāveido tā, lai jebkādu ekrāna formu parādīšana tiktu aizturēta līdz brīdim, kad ir veiktas vides pārbaudes. Līdzīga pieeja vērojama dažādos plaša lietojuma rīkos, kuru ielādei nepieciešams ilgs laiks, piemēram, *Adobe Photoshop*, kas ielādes laikā uz ekrāna rāda tikai uzplaiksnījuma ekrānu, bet nerāda galveno saskarni.



3.1 att. **Klasiska programmatūra, kas pēc ielādes uzreiz uzsāk pamatdarba veikšanu un viedā programmatūra, kas pirms darba veikšanas pārbauda izpildes vidi**

Ja vides pārbaužu rezultāti ir apmierinoši, programma uzskatāma par sagatavotu darbam dotajā izpildes vidē, pretējā gadījumā tā sniedz pēc iespējas daudz informācijas par novērojumiem un pārtrauc darbu, bet problēmas risināšanā tiek iesaistīts cilvēks – sistēmu administrators vai uzturētājs.

Šī darba izpratnē *pārbaude* ir darbību kopums, ko programmatūra var veikt patstāvīgi, nepieprasot lietotāja iejaukšanos šajā procesā. Tas ir, tiek apskatītas tikai tādas pārbaudes, kuru veikšanai ir zināms determinēts algoritms un ko iespējams aprakstīt kā datorprogrammas kodu. Tiek izšķirtas divu veidu pārbaudes – „*pilnā pārbaude*", kas tiek veikta pēc programmatūras sākotnējas instalācijas, jauninājumu uzstādīšanas vai problēmu pieteikuma gadījumā un „*ātrā pārbaude*", ko iespējams izpildīt katrā programmas darbināšanas sesijā. Šāds dalījums nepieciešams tādēļ, ka atsevišķu pārbaužu veikšana var būt laikietilpīga, turklāt relatīvi statiskā izpildes vidē visu pārbaužu atkārtota veikšana nav racionāla. Piemēram, programma, kas datu glabāšanai izmanto *SQL* datubāzi, varētu veikt *SQL* servera pārbaudes– gan pārbaudīt, vai *SQL* serveris ir pieejams, gan arī vai datubāzes struktūra atbilst

nepieciešamajai (piemēram, tajā jābūt tabulai „Klients", kas satur kolonnu „Nosaukums"). Šādā gadījumā pirmo pārbaudi – *SQL* servera pieejamību – ir racionāli veikt gan instalēšanas laikā, gan katrā programmas izpildes sesijā, jo *SQL* serveris var pēkšņi būt kļuvis nepieejams arī lietošanas laikā. Savukārt otro – *SQL* struktūras pārbaudi var veikt arī tikai instalācijas laikā, jo ir maza ticamība, ka datubāzes struktūra sistēmas lietošanas laikā tiks mainīta.

Atšķirībā no klasiskiem programmatūrā iebūvētajiem testiem, kuru galvenais uzdevums ir pārbaudīt, vai programmatūra darbojas „pareizi" (t.i., atbilstoši tās specifikācijai), šajā darbā apskatītās pārbaudes paredzētas tam, lai noskaidrotu vai pašreiz pieejamā *izpildes vide* atbilst tai, kāda programmai nepieciešama; t.i. pārbaudes paredzētas *ārējo apstākļu* mērīšanai un salīdzināšanai ar etalonu. Ar jēdzienu „izpildes vide" šeit jāsaprot viss ārējo apstākļu kopums, kas mijiedarbojas ar testējamo programmatūru. Tas nozīmē, ka programmatūras izpildes vides sastāvdaļas ir ne tikai operētājsistēma, kas izpilda programmu, virtuālā mašīna (tādas kā Java virtuālā mašīna vai *Microsoft .Net Framework*) vai skriptu interpretators (tādi kā *PHP* vai *Python*), bet arī dažādi citi tehnoloģiskie servisi, kuru klātesamība programmai nepieciešama, piemēram, interneta tīkla savienojums, datubāzu vai e-pasta serveris, arī ražotājam specifiskas tehnoloģijas, piemēram, *Microsoft DCOM*, *Java Hibernate* un citas. Tāpat pie programmatūras izpildes vides var pieskaitīt tādas tehnoloģijas, kam nav tiešas mijiedarbības ar testējamo programmatūru, bet kas maina citu izpildes vides artefaktu darbību tā, ka tā atšķiras no sagaidītās. Piemēram, ja novērots, ka kāda specifiska pretvīrusu programmas versija bloķē testējamās programmas darbību, arī tā uzskatāma par izpildes vides sastāvdaļu.

Lai programmatūra varētu patstāvīgi izpildīt pārbaudes, tai „jāzina", ko nepieciešams pārbaudīt un kāds ir pārbaužu mērķis. Zināšanas par veicamajām pārbaudēm aprakstot kādai citai pusei saprotamā veidā, tiek faktiski formulētas programmatūras *prasības* pret vidi, kādā tā ir izpildāma. Šī darba ietvaros tiek pieņemts, ka *prasība* apraksta programmatūrai nepieciešamos apstākļus, pēc iespējas izvairoties no algoritmiska apraksta, kā pārbaudīt, vai prasība ir izpildīta, savukārt *pārbaude* paredzēta, lai pārliecinātos, vai ir sastopama kāda pazīme, kas apliecina, ka prasība ir izpildīta. Piemēram, „jābūt instalētai *ActiveX* komponentei *MSREdit*", ir prasība, savukārt to, vai tā ir izpildīta, var noskaidrot ar pārbaudēm „vai Windows reģistrēto *ActiveX* komponenšu sarakstā atrodama klase ar identifikatoru {2F7FC18D-292B-11D2-A795-DFAA798E9148}" vai arī „vai izdodas izveidot jaunu *MSREdit* objektu?". Skaitliska attiecība starp prasībām un veicamajām pārbaudēm nav strikti definējama, taču jāuzskata, ka katras prasības izpildījuma noskaidrošanai jāveic viena vai vairākas pārbaudes.

Tehnoloģijas pamatā ir ideja, ka programmatūras dažādās prasības ir iespējams savākt vienkopus, izveidojot programmatūras izpildes profilu – dokumentu, ko pievieno programmatūras nodošanai jau ierastajiem artefaktiem – izpildāmajam kodam un dokumentācijai. Šāds dokuments satur datoram lasāmu būtisku dokumentāciju, kas apraksta programmatūras īpatnības un izmantojams dažādiem mērķiem programmatūras dzīves cikla tālākajos posmos. Dokumenta galvenais, taču nebūt ne vienīgais, lietojums ir jau minētās izpildes vides pārbaudes programmatūras lietošanas laikā. Veicot izpildes profilu apkopojošu analīzi kādā noteiktā izpildes vidē, iespējams, piemēram, identificēt programmu pārus, kam nepieciešama savstarpēji nesaderīga konfigurācija, vai sastādīt tādu programmu sarakstu, kas izmanto noteiktu vides resursus, pat – identificēt resursus, kurus potenciāli neizmanto neviena programma.

3.2. Izpildes vides testēšanas arhitektūra

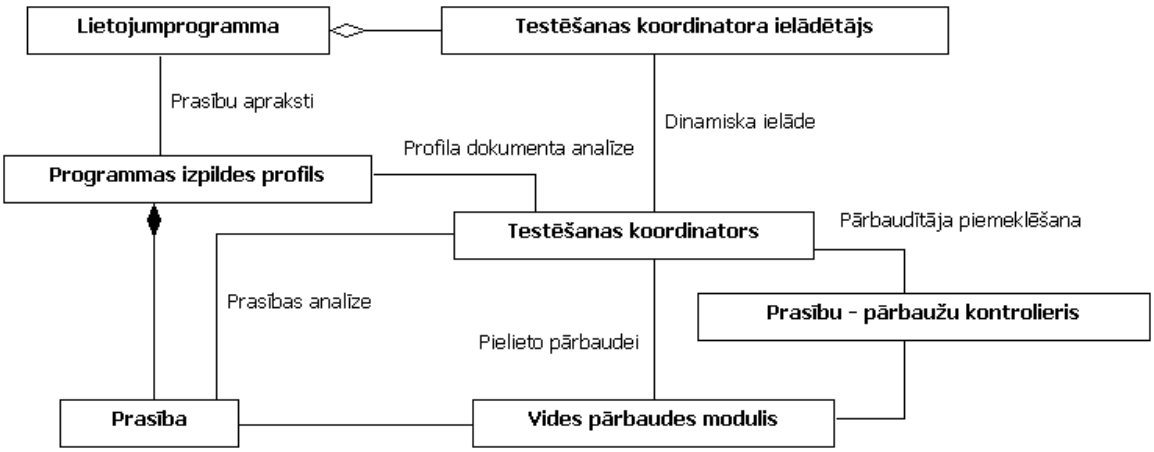
Realizējot vides testēšanas infrastruktūru, ir vēlams, lai vides testēšana netraucētu pamatdarbu, bet veicamie testi būtu brīvi konfigurējami. Nepieciešamo ierobežojumu saraksts ir šāds:

- algoritmiski realizētās programmatūras izpildes vides pārbaudes nav jāveido katrai darījumu sistēmai no jauna, tās ir atkārtoti izmantojamas;
- programmatūras izpildes vides pārbaužu veidu skaits nav ierobežots, ir iespējams realizēt jaunus pārbaudes mehānismus, kas aizstāj iepriekšējos;
- ir iespējams aprakstīt jaunas prasības pret programmatūras izpildes vidi un attiecīgi – veidot jaunas pārbaudes, kas atbilst šādām prasībām;
- pārbaužu veikšanai nepieciešamais laiks (veiksmīgas izpildes gadījumā) ir relatīvi neliels, var uzskatīt, ka tas netraucē programmas ielādi (reālais pieļaujamais izpildes laika maksimums atkarīgs no darījumu programmas veida, piemēram, ilgstoši darbināmām servisa tipa programmām pieļaujamais laiks ir ilgāks nekā programmām, kas ik dienu jāiedarbina vairākas reizes);
- izņēmumsituācijas testu veikšanas laikā nedrīkst ietekmēt darījumu programmatūras izpildi.

Lai realizētu šeit minētos nosacījumus, apkārtējās vides testēšanas mehānismu tiek piedāvāts veidot kā sadalītu koda bibliotēku kopumu, kas ar dinamiskas ielādes palīdzību veido konkrētajā vidē nepieciešamo izpildāmo lietotņu kompozīciju.

Galvenie jēdzieni apkārtējās vides testēšanas risinājumam ir (ilustrācijai skat. ***3.2****. att.* parādīto infrastruktūras diagrammu):

- *darījumu programma* – lietojumprogramma, kas paredzēta kāda darījumu procesa atbalstam un kuras pārvaldības uzlabošanai nepieciešams veikt programmatūras izpildes vides testus;
- *programmas izpildes profils* – formāli definētā gramatikā pierakstīts darījumu lietojumprogrammas izpildei būtisko izpildes prasību saraksts; šāds profils piesaistīts noteiktai darījumu lietojumprogrammai, detalizētāku aprakstu skat. 3.3.1. nodaļā;
- *vides pārbaudes modulis* – koda modulis (funkcija, objektorientētas programmatūras klase vai neliela komponente), kas atbildīgs par kāda noteikta prasību veida pārbaudīšanu; detalizētāku aprakstu skat. 3.3.1 nodaļā;
- *programmatūras izpildes vides testēšanas koordinators* – koda modulis, kas atbildīgs par lietojumprogrammas izpildes profila interpretēšanu, atbilstošu pārbaudes moduļu piemeklēšanu un iedarbināšanu, pārbaužu rezultātu apkopošanu un vispārējā novērtējuma sniegšanu;
- *prasību-pārbaužu kontrolleris* – programmatiski vai deklaratīvi definēts pārbaudes vides modulis, kas noteiktam prasību veidam spēj piemeklēt vienu vai vairākus atbilstošus pārbaudes moduļus
- *testēšanas koordinatora ielādētājs*. Kaut arī galveno vides pārbaudes darbu veic vides testēšanas koordinators, lietojumprogrammā jābūt iekļautam kodam, kas nodrošina paša koordinatora ielādi.



3.2. att. **Vides testēšanas infrastruktūras piekārtošana lietojumprogrammai**

Kā minēts 3.1 nodaļā, apkārtējās vides testēšanas funkcionalitāte pēc iespējas nodalāma no lietojumprogrammas galvenajiem moduļiem, kas realizē darījumu loģiku. Šī iemesla dēļ pašā lietojumprogrammā paredzēts iestrādāt tikai testēšanas koordinatora ielādētāju (***3.2****. att.* augšdaļā), bet pārējo ar vides testēšanu saistīto infrastruktūru ielādēt dinamiski programmas izpildes laikā. Testēšanas koordinatora ielādētājs ir relatīvi statiska komponente, ko pēc sākotnējās izstrādes un testēšanas iespējams pielietot dažādās lietojumprogrammās bez koda izmaiņām.

Lai izvairītos no koda bibliotēku savstarpējām atkarībām, kā arī lai izvairītos no t.s. „*DLL hell*" (arī „*JAR hell*" vai vispārīgāk - „*dependency hell*") problēmas [**65**], vides testēšanas infrastruktūrā paredzēts pielietot koda dinamiskās ielādes principus. Tas sniegs iespēju papildināt testēšanas infrastruktūras komponenšu kodu, izvairoties no lietojumprogrammatūras koda atkārtotas kompilēšanas. Atkarībā no platformas, kurā testēšanas infrastruktūra tiek realizēta, risinājumi var būt dažādi – *Microsoft .Net* gadījumā lietojama dinamiskā koda asambleju ielāde, izmantojot koda refleksijas paradigmu. Līdzīgi, *Java* platforma šādu risinājumu pielieto kā standartpakalpojumu, kas pieejams, izmantojot *ClassLoader* infrastruktūru. Vēl vienkāršāk dinamiskā koda ielāde realizējama interpretējamās skriptu valodās kā *Python* vai *PHP*. Līdzvērtīgs risinājums realizējams arī *C++* programmās, neizmantojot specifiskus izpildes vides servisus. [**66**].

Tādējādi, uzsākot lietojumprogrammas darbu, faktiski tiek realizēta šāda notikumu virkne:

- lietojumprogrammā iekļautais testēšanas koordinatora ielādētājs dinamiski ielādē testēšanas koordinatoru un nodod tam programmas vadību;
- testēšanas koordinators:
    - atrod un atver attiecīgās lietojumprogrammas izpildes profilu;
    - analizē izpildes profilā esošo prasību sarakstu;
    - katrai prasībai, izmantojot prasību-pārbaužu koordinatoru, piemeklē atbilstošu vides pārbaudes moduli (vai vairākus);
- vides pārbaudes modulis analizē apkārtējo vidi un rezultātus paziņo testēšanas koordinatoram;
- kad visas pārbaudes pabeigtas, testēšanas koordinators atdod programmas vadību lietojumprogrammai, kas pieņem lēmumu, vai turpināt tālāko izpildi.

3.3. Metodes izklāsts

3.3.1. Prasību aprakstīšanas valoda

Lai būtu iespējams aprakstīt programmatūras izpildei nepieciešamos nosacījumus, ir nepieciešama formāla valoda to pierakstīšanai. Valoda ir šīs tehnoloģijas pamats – visu pārējo vides pārbaudes infrastruktūras sastāvdaļu būve ir pakārtota tam, kā attiecīgā valoda apraksta prasības. Prasību aprakstīšanas valodā veidotam dokumentam jānodrošina divi mērķi – jāidentificē programmatūras vienība, uz kuru attiecas dotais apraksts un jāparāda visas zināmās prasības, kādas šai programmatūras vienībai ir pret izpildes vidi.

Definējot prasību aprakstīšanas valodu, jāņem vērā, ka tā var kļūt par ilgtermiņa „standartu" (piemēram, programmatūras izstrādes uzņēmuma iekšienē), kas savukārt liek rūpīgi pārdomāt ar ilglaicību saistītus aspektus. Pirmkārt, valoda nedrīkst būt statiska attiecībā uz aprakstāmo prasību veidiem, tajā jābūt iekļautai iespējai dinamiski papildināt valodas konstrukcijas. Kā redzams 2.6 nodaļā parādītajā piemērā ar „*SQL 2005*", jauni prasību veidi programmai var parādīties pat gadījumā, ja tā jau ilgstoši lietota un pati programma nav tikusi mainīta. Otrkārt, ņemot vērā to, ka programmas izpildes profils faktiski kļūst par būtisku programmatūras dokumentācijas daļu, kas pierakstīta specifiskā formātā, ir būtiski, ka dokumenta formāts ir viegli uztverams arī ilgākā laika periodā. Viens no risinājumiem ir veidot aprakstīšanas valodu tā, lai tā būtu pašdokumentējoša (piemēram, katras prasības apraksts satur arī cilvēkam lasāmu skaidrojumu, kas tieši tiek pieprasīts) vai arī valodas konstrukcijas aprakstītas viegli pārskatāmā dokumentācijā.

Prasību aprakstīšanas valodu racionāli veidot kā *XML* [**49**] atvasinātu valodu, tādējādi padarot prasību dokumentus viegli rediģējamus ar standarta *XML* apstrādes rīkiem un iegūstot iespēju šajā valodā rakstītos dokumentus ātri pārveidot nepieciešamajā prezentācijas izskatā, lietojot *XSLT* transformācijas.

Tālākajās apakšnodaļās aplūkoti vairāki risinājumi, ko iespējams pielietot programmatūras prasību aprakstīšanai – naivā pieeja, kas ļauj ātri un viegli realizēt prasību apraksta valodu, taču var izrādīties ilgtermiņā grūti uzturama, uz *XML Schema* balstīta pieeja, kas definē valodas struktūru strikti modulārā veidā, *RDF* un citu jau pastāvošu pieeju adaptācija kā risinājums, kas ļauj pārņemt citu pētnieku identificētās labākās prakses.

3.3.1.1. Naivā pieeja

Visvienkāršākais prasību aprakstīšanas valodas definēšanas veids ir t.s. naivā pieeja – definēt minimālu, *XML* bāzētu valodu, kas jebkurā brīdī paplašināma. Šādas *.Net* lietojumprogrammu saimei paredzētas valodas piemērs parādīts ***3.3***. *att.*, kur viss izpildes profils iekļauts *XML* elementā *SWProfile*, bet tajā kā bērnu elementi iekļautas divas galvenās

sadaļas – *subject*, kura mērķis ir identificēt aprakstāmo lietojumprogrammu (pēc tās „*fullname*" atribūta, kas .Net vidē unikāli identificē koda asambleju) un *requirements*, kas satur visu izpildes prasību sarakstu. Prasību sarakstā parādītas trīs atsevišķas prasības – *assemblyDependency*, kas apzīmē prasību, ka izpildes vidē jābūt pieejamai kādai citai koda bibliotēkai, *directoryDependency*, kas apraksta prasību, ka failu sistēmā jābūt pieejamam noteiktam katalogam un *localeDependency*, kas pieprasa operētājsistēmas reģionālos iestatījumus atbilstoši kādai specifiskai lokālei.

```
<SWProfile >
  <subject type="assembly"
    fullname="TestApp,
              Version=1.0.0.0, Culture=neutral,
              PublicKeyToken= 28f514e8eeca027c" />
  <requirements>
    <assemblyDependency
      fullname="Microsoft.Office.Interop.Excel,
                Version=11.0.0.0, Culture=neutral,
                PublicKeyToken=71e9bce111e9429c" />
    <directoryDependency
      path="C:\Program Files\Common Files\Microsoft Shared\
            Web server extensions\12\bin" requirement="existence" />
    <localeDependency localeName="lv-LV"/>
  </requirements>
</SWProfile>
```

3.3. att. **Programmatūras prasību pieraksts, izmantojot naivo pieeju**

Šīs pieejas vienkāršums ļauj nepieciešamības gadījumā ātri un vienkārši papildināt aprakstīšanas valodu. Piemēram, ja līdz šim nav bijis nepieciešams aprakstīt prasību „MS Windows reģionālajiem iestatījumiem jāatbilst noteiktai valodu kopai (lokālei)", valodu var papildināt, tajā ieviešot jaunu elementu, piemēram, „*localeDependency*" ar atribūtu „*localeName"*, kura vērtību var izvēlēties prasību dokumenta autors.

Naivās pieejas galvenais trūkums ir tas, ka, kaut arī piemērā parādītais dokuments ir sintaktiski korekts XML, tā semantika ir tikai nojaušama, precīzāk – tā netiek stingri definēta. Tādējādi, ja vien šāda valoda nav labi dokumentēta kādā citā vidē, izstrādātājiem var rasties grūtības ar šajā valodā rakstītu dokumentu izstrādi, turklāt tiek liegta iespēja automātiski pārbaudīt, vai attiecīgais dokuments ir pareizi sastādīts.

3.3.1.2. Uz XML shēmām balstītā pieeja

Nedaudz uzlabojot 3.3.1.1 nodaļā aprakstīto risinājumu, iespējams iegūt valodu, kas vieglāk pakļaujas rediģēšanai ar universālo *XML* rīku palīdzību un izstrādāto dokumentu pareizības pārbaudei. Tas darāms, visus nepieciešamos valodas elementus aprakstot ar *XML* shēmas (*XML Schema*, [**67**]) palīdzību un pielietojot t.s. vārdkopas jēdzienu. Tādējādi iespējams sākotnējās valodas konstrukcijas aprakstīt pamata vārdkopā un katram

nepieciešamajam papildinājumam veidot jaunu vārdkopu, kas apraksta jaunās valodas iespējas. Šādas pieejas lietojums demonstrēts [**17**], kur nepilnas 200 rindas garā *XSD* dokumentā aprakstīta semantika prasībām dokumentā, kas analoģisks ***3.3****. att.* parādītajam koda piemēram. Līdzīgs, taču daudz detalizētāk aprakstīts, risinājums izmantots *SDD* shēmā [**53**].

Šī darba 1.2. pielikumā parādīta un skaidrota vides testēšanas pieeja, kas izmanto uz *XML* shēmām balstītu valodu prasību aprakstīšanai.

3.3.1.3. RDF bāzētā pieeja

Cits risinājums, kas var tikt pielietots programmas izpildes prasību aprakstīšanai, ir *RDF* (angl. *resource description framework*) jeb resursu aprakstīšanas satvars [**68**], kas modelējamo informāciju apraksta trioļu formā subjekts-predikāts-objekts. Pielietojot *RDF*, daļu jēdzienu nebūtu jāievieš patstāvīgi, jo iespējams izmantot jau citu pušu definētas ontoloģijas (kā, piemēram, *FOAF – angl.* Friend of a Friend, ontoloģija, kas definē cilvēku savstarpējo attiecību jēdzienus). Tomēr jāņem vērā, ka *RDF* un tā atvasinājumi kā, piemēram, *OWL* paredzēti strikti korektai jēdzienu semantikas un dažādu jēdzienu attiecību aprakstīšanai, tādēļ šī pieeja var izrādīties tehniski sarežģīta. Ja informācijas aprakstīšanai, līdzīgi kā *SDD* shēmā, paredzēts aprakstīt daudzu dažādu jēdzienu atribūtus un to mērvienības, jārēķinās ar sarežģījumiem, definējot piemērotu *OWL* shēmu. Piemēram, [**69**] apraksta mēģinājumu pārveidot *CIM* shēmas nelielu apakškopu uz *OWL* standartu un parāda, ka jau pat relatīvi vienkāršu jēdzienu pārcelšana var būt sarežģīta – rodas problēmas ar dažādu modeļa jēdzienu atribūtu nosaukumu unikalitātes ierobežojumiem un citi sarežģījumi.

3.3.1.4. Eksistējošu valodu pielāgošana

Prasību aprakstīšanas valodu iespējams izstrādāt pilnībā no jauna, izmantojot kādu no iepriekšējās nodaļās aprakstītajām pieejām, taču risinājumu var meklēt arī kādā no jau pastāvošiem standartiem.

Kā minēts 2.6 nodaļā, *IUDD* standartā piedāvātā valoda nav piemērota šī darba mērķiem – tā „nestandarta" prasību aprakstīšanai ļauj izmantot lietotāja definētu pārbaužu aprakstus (tieši pārbaužu, nevis prasību), taču pārbaužu aprakstos neiekļauj jēdzienu semantisko jēgu. Tā, piemēram, lai aprakstītu ***3.3****. att.* parādīto prasību „operētājsistēmas reģionālajos iestatījumos lokālei jābūt lv-LV", *IUDD* shēmā būtu jālieto konstrukcija, kas līdzīga ***3.4*** *att.* parādītajai. Kā redzams, šāds risinājums ļauj *IUDD* sintaksē pierakstīta dokumenta lasītājam saprast, kāda pārbaude jāveic, taču prasības semantiskā jēga daļēji zūd pārlieku tehniskajā sintaksē.

```
<custom artifactIdRef="winLocaleCheck"
        checkId="required_Win_Locale_Check">
     <parameter variableNameRef="Windows_Locale_ID">lv-LV</parameter>
 </custom>
```

3.4 att. **Specifiskas prasības kodēšana, izmantojot *IUDD* pieļauto sintaksi**

Analizējot iespējas šī darba mērķiem pielietot *SDD* shēmā definēto valodu prasību aprakstīšanai [**53**], var novērot, ka tā, salīdzinot ar *IUDD*, ir labāk piemērota šī darba mērķiem, taču jāievēro *SDD* ideoloģija attiecībā uz resursiem un to ierobežojumiem.

Pirmkārt, tā kā *SDD* shēmā definētā valoda ir resursu bāzēta, jebkurš programmas darbības ierobežojums jāattiecina uz kādu resursu, kas programmai pieejams izpildes laikā. Tā kā valoda ļauj definēt resursu hierarhiju, ir iespējams detalizēti precizēt, uz kuru resursu ierobežojums attiecināms. Tā, piemēram, resursu „instalācijas direktorija" iespējams aprakstīt hierarhijā „operētājsistēma -> failu sistēma -> instalācijas direktorija". Katram resursam iespējams definēt pēc patikas daudz atribūtu, kuru vērtības iespējams ierobežot, izmantojot valodā esošās konstrukcijas.

Otrkārt, lai būtu iespējams precīzi definēt, kāds ir resursam noteiktais ierobežojums, visiem izteikumiem attiecībā uz resursu vērtībām jābūt aprakstāmiem, izmantojot dažas pamatkonstrukcijas:

- *propertyConstraint* – atribūta vērtība sakrīt ar prasīto,
- *versionConstraint* – attiecīgā resursa versija ir norādītajās robežās,
- *relationshipConstraint* – prasību dokumenta subjektam ir noteikta attiecība ar aprakstīto resursu (attiecības tips ir aprakstāms atsevišķi, tipisks lietojuma piemērs: „*connectedTo*" – subjekts ir savienots ar aprakstīto resursu,
- *capacityConstraint* – aprakstītajam resursam piemīt prasītā ietilpība (iespējams norādīt augšējo vai apakšējo robežu),
- *consumptionConsrtaint* – aprakstītajam resursam ir brīvs prasītās ietilpības daudzums.

Piemēram, prasība par operētājsistēmas reģionālo iestatījumu atbilstību *lv-LV* lokālei ir aprakstāma, ja resursam „operētājsistēma" pakārto jaunu resursu „reģionālie iestatījumi", kuram definē atribūtu „lokāle" (līdzīgi, šim resursam būtu jādefinē arī atribūti „decimālais atdalītājs", „datuma formāts", „laika formāts' u.c.). Pēc tam, lietojot *propertyConstraint* ierobežojumu, lokāles atribūta vērtība tiktu ierobežota ar konstanti *lv-LV* (skat. ***3.5*** *att.*).

```
<SDD-dd:Topology>
  <SDD-dd:Resource id="os" type="sp:CIM_OperatingSystem">
    <SDD-dd:HostedResource id="regionalsettings"
            type="c1:RegionalSettings" />
  </SDD-dd:Resource>
</SDD-dd:Topology>
...
<SDD-dd:Requirements>
  <SDD-dd:Requirement id="locale.reqt" operation="install use">
      <SDD-dd:Description>Tikai lv-LV</SDD-dd:Description>
      <SDD-dd:ResourceConstraint id="localeConstraint"
              resourceRef="regionalsettings">
          <SDD-dd:PropertyConstraint>
              <SDD-dd:PropertyName>
                      c1:RegionalSettings.Locale</SDD-dd:PropertyName>
              <SDD-dd:Value>lv-LV</SDD-dd:Value>
          </SDD-dd:PropertyConstraint>
      </SDD-dd:ResourceConstraint>
  </SDD-dd:Requirement>
</SDD-dd:Requirements>
```

3.5 att. ***SDD*** **sintaksē pierakstīta reģionālo iestatījumu prasība**

Kā redzams koda piemērā, lai aprakstītu resursu „*regionalSettings*" un tā atribūtu „*Locale*", lietots prefikss „*c1*". Tā kā *SDD* tehnikas pamatā ir *XSD* shēmu lietošana, katrs datu tips jādefinē atsevišķi. Šajā gadījumā vārdkopa „*sp*" pielieto *SDD* „*starter profile*" piedāvāto shēmu [**56**], bet pārējie jēdzieni – tātad arī „*regionalSettings*" – jādefinē atsevišķā shēmā, kas šajā piemērā parādīta ar nosaukumu „*c1*".

Arī pārējās ***3.3****. att.* piemērā parādītās prasības iespējams formulēt *SDD* shēmā, ieviešot atbilstošus resursus – *.Net* asamblejas gadījumā jāveido resursu topoloģijas hierarhija operētājsistēma > izpildes vide > asambleja, failu sistēmas direktorijas gadījumā jāveido hierarhija operētājsistēma > failu sistēma > direktorija – un, pielietojot „*resourceConstraint*" ierobežojumu, jāpieprasa paša resursa eksistence testējamajā vidē.

Tomēr jāņem vērā, ka *SDD* aprakstītā valoda ierobežo to prasību klāstu, ko iespējams aprakstīt. Tā, piemēram, šī valoda nesniedz iespēju aprakstīt izteikumus formātā „resursa A atribūta B vērtība nav vienāda ar X" (valodā nav nolieguma izteikuma). Šāda prasība ir, piemēram, „*MS Windows* servisam, kura kontekstā tiek darbināta šī lietojumprogramma, nedrīkst būt iestatīts noklusētais lietotāja konts „*Local System Account*" (t.i., der jebkurš cits konts, izņemot norādīto)". Attiecīgo resursu – *MS Windows* servisu – iespējams aprakstīt kā izvietojuma topoloģijas resursu, lietotāja kontu definējot kā servisam piemītošu atribūtu, taču nepastāv ierobežojuma veids, kas pieprasītu atribūta vērtības atšķiršanos no norādītās.

Tādējādi iespējams, ka, izstrādājot pilnvērtīgu prasību aprakstīšanas valodu, *SDD* shēma jāpapildina ar patstāvīgi definētām konstrukcijām, kas paredzētas iepriekšminētajam mērķim. Šāda rīcība gan ietekmētu valodas saderīub ar citiem *SDD* atbalstošiem rīkiem, taču

jāatceras, ka šajā darbā valoda tiek izmantota citiem mērķiem, nekā tā sākotnēji tikusi paredzēta.

Šādi veidota deskriptora piemērs parādīts un skaidrots šī darba 1.1. pielikumā. Pielikumā izskaidrotas arī situācijas, kad *SDD* bāzētā pieeja nav piemērota izpildes prasību aprakstīšanai un, salīdzinot ar 1.2. pielikumā demonstrēto uz *XML* shēmām balstīto pieeju, tā kļūst šķietami „neveikla".

3.3.2. Izpildes prasības un prasību pārbaudes moduļi

Apkārtējās vides testēšanas praktisko darba daļu realizē atsevišķie prasību pārbaudes moduļi, kur katrs pārbaudes modulis atbildīgs par kāda noteikta prasību tipa testēšanu. Pārbaudes modulis, mijiedarbojoties ar programmatūras izpildes vidi, nolasa no tās informāciju, kas nepieciešama, lai pārbaudītu vides atbilstību prasībai. Veicot pārbaudi, pārbaudes modulis izmanto tikai resursu pārvaldības metodes, lai novērotu dažādas pazīmes, kas liecina par prasības izpildīšanos, taču tas „izvairās" no tādu darbību veikšanas, kam varētu būt blakusefekti izpildes vidē. Tā, piemēram, lai noskaidrotu, vai kāda failu sistēmas direktorija ir izdzēšama, vienkāršākais pārbaudes veids būtu mēģināt to izdzēst, taču šāds risinājums nav uzskatāms par korektu pārbaudi – to būtu jāaizstāj ar atbilstošiem operētājsistēmas servisu izsaukumiem, kas nolasa pašreizējā lietotāja tiesības uz šo failu sistēmas objektu.

Ja izpildes vidē ir iebūvēta pārvaldības funkcionalitāte, tā ir labi piemērota vides pārbaudes moduļu būvei – šīs bibliotēkas izstrādātas ar mērķi atvieglot vides pārvaldību un tādēļ var tikt pielietotas arī vides atribūtu nolasīšanai. Kā vienu no šādiem pārvaldības rīkiem var minēt *Microsoft Windows Management Instrumentation* [**70**] funkciju bibliotēku saimi, kas dažādus vides atribūtus padara pieejamus rīku izstrādātājiem kā programmatūras API. Atribūtu hierarhija *WMI* strukturēta, izmantojot *CIM* definēto nomenklatūru un papildinot *CIM* objektu kopu ar *MS Windows* specifiskiem jēdzieniem. Tādējādi, lai *WMI* modelī apstrādātu iepriekšminēto prasību „*Windows* reģionālajiem iestatījumiem jāatbilst *lv-LV* lokālei", jāizmanto *Win32_OperatingSystem* tipa objekts un jānolasa tā atribūts *Locale*.

Tā kā vides pārbaudes moduļi ir universāli cilvēka veiktu pārbaužu aizstājēji, tie jāuztver kā uzņēmuma zināšanu bāzes sastāvdaļa un tādēļ jārūpējas par pēc iespējas racionālu šo moduļu pārvaldību. Būtiski ir veidot pārbaudes moduļu bibliotēku, kas koplietojama visām uzņēmumā izstrādātajām vai izmantotajām informācijas sistēmām. Šādas bibliotēkas izveidošana nodrošinātu gan to, ka ir zināms, kuras pārbaudes ir iespējams veikt automātiski un kuras – tikai manuāli, gan arī ļautu viegli atrast nepieciešamo pārbaudes moduli.

3.3.2.1. Pārbaudes moduļa iedarbināšana

Prasību pārbaudes moduļus iedarbina vides testēšanas koordinators. Veids, kā testēšanas koordinators ielādē un/vai iedarbina testēšanas moduli, atkarīgs gan no izpildes vides, kurā vides testēšanas ideoloģija realizēta, gan no konkrētās realizācijas. Var apskatīt vairākus testēšanas moduļu izsaukšanas veidus – statisku sasaisti, testu izpildi atsevišķā procesā un testa moduļu dinamisku ielādi.

Statiska sasaiste, kas izveidota testēšanas koordinatora kompilācijas laikā. Šis veids nav saderīgs ar testēšanas infrastruktūras ideoloģiju, proti, katras sastāvdaļas atsevišķu versiju uzturēšanu un dinamisku sasaisti.

Testēšanas moduļu izpilde atsevišķā procesā. Šāds risinājums ir iespējams, ja katrs vides testu modulis izstrādāts kā atsevišķi izpildāma lietojumprogramma, ko testu koordinators iedarbina, tiešā vai netiešā veidā izveidojot jaunu operētājsistēmas līmeņa procesu. Testu koordinators un vides pārbaudes modelis var savstarpēji sazināties, lietojot standarta ieejas un izejas plūsmas (*stdin, stdout, stderr*), kā arī operētājsistēmas piedāvāto parametru nodošanas mehānismu. Šāda pieeja nodrošina ļoti brīvu sasaisti starp testu koordinatoru un testēšanas moduļiem, taču tā var radīt blakusefektu uz veiktajiem testiem. Tā, piemēram, ja prasību pārbaudes modulis veic datortīkla pārbaudes, pastāv iespēja, ka ugunsmūrī piekļuves likumi programmām aprakstīti pēc operētājsistēmas procesa nosaukuma [**71**]. Līdz ar to pārbaudes moduļa veiktās pārbaudes attieksies uz paša pārbaudes nosaukumu, nevis uz testējamo lietojumprogrammu.

Testēšanas moduļa koda dinamiska ielāde testējamās lietojumprogrammas procesā. Kaut tehniski sarežģītākā, šī metode ir vispiemērotākā programmatūras izpildes vides automātiskai testēšanai. Šajā gadījumā testēšanas koordinators, kas pats jau ir ielādēts darījumu lietojumprogrammas procesā (skat. 3.2. nod. definētos principus), šajā procesā dinamiski ielādē arī nepieciešamā vides testēšanas moduļa kodu. Tādējādi testēšanas moduļa veiktās pārbaudes pilnībā tiek veiktas lietojumprogrammas kontekstā. Ja izmantota šī pieeja, testēšanas koordinators sazinās ar testēšanas moduli, izmantot koda funkciju izsaukumus.

Interesantu testēšanas moduļu ielādēšanas pieeju apraksta [**18**], kur viens un tas pats testu koordinēšanas modulis spēj ielādēt vides pārbaudes moduļus trīs dažādos veidos – ielādējot moduli atmiņā dinamiski, izpildot ārējā procesā vai izsaucot attālinātas komponentes pārbaudi ar tīmekļa pakalpes palīdzību. Veids, kādā modulis ielādējams, norādīts moduli aprakstošajos metadatos, tādējādi, izvēloties konkrēto moduli kādas pārbaudes veikšanai, testu koordinators vispirms noskaidro veidu, kā modulis ielādējams, un tikai tad veic tā dinamisku ielādi.

3.3.2.2. Atbilstoša pārbaudes moduļa piemeklēšana

Kā minēts 3.2. nodaļā, vides testēšanas koordinators ir tā testēšanas infrastruktūras daļa, kas interpretē lietojumprogrammas izpildes profila dokumentu. Sastopot dokumentā kādu izpildes prasību, testēšanas koordinators „izdomā", kādus pārbaudes moduļus pielietot, lai attiecīgo prasību testētu. Šim mērķim tiek izmantota atsevišķa komponente – prasību-pārbaužu kontrolleris, kuras uzdevums ir pārzināt pieejamo vides pārbaudes moduļu klāstu un to, kuri moduļi spēj testēt kuru prasību. Testēšanas kontrolleris izmanto prasību-pārbaužu kontroliera sniegto informāciju tālākajam darbam – pēc kārtas atrod visu pārbaudes moduļu izpildāmos failus, tos ielādē atmiņā un izpilda pārbaudes.

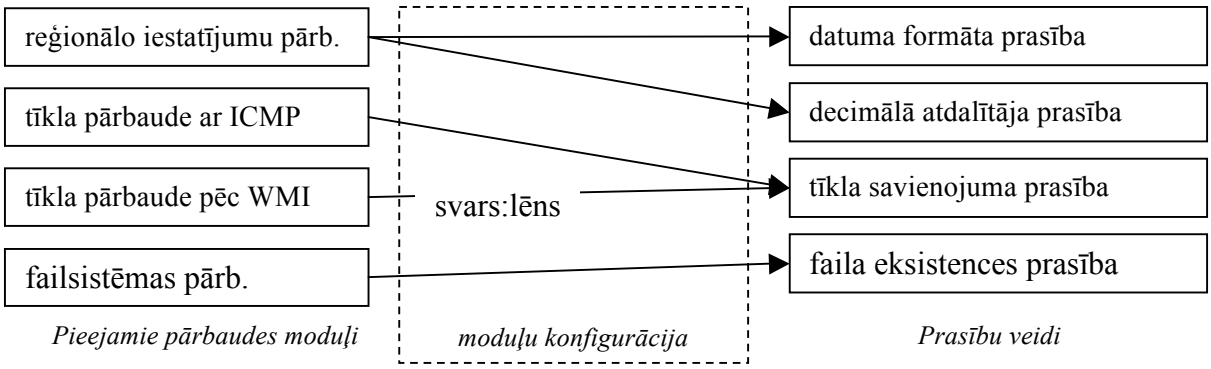
Katrā izpildes vidē varētu tikt izmantoti citi pārbaudes moduļi – piemēram, testēšanas vidē iespējams izmantot pārbaudes moduļus, kas veic intensīvākas pārbaudes (piemēram, tādas, kam iespējami blakusefekti) vai kas rada papildus pārskatus par savu izpildi, savukārt lietošanas vidē pieejamie pārbaudes moduļi nedrīkst būt lēndarbīgi vai darboties ar blakusefektiem, tādēļ tur lietojamas pilnīgi citas pārbaudes.

Lai prasību-pārbaužu kontrolieris varētu darboties, tam jāzina attiecīgajā izpildes vidē pieejamo un atļauto pārbaudes moduļu saraksts. Sarakstu ir iespējams izveidot dinamiski, piemēram, skenējot kādu ar iekšēju konvenciju noteiktu taku failu sistēmā. Piemēram, konvencija varētu noteikt, ka visi pārbaudes moduļi novietoti direktorija */etc/lib/environmenttests/* apakšdirektorijos. Konvencijai būtu arī jāparedz, kā jānosauc pārbaudes moduļa izpildāmais fails un kā jāsaglabā šo pārbaudes moduli aprakstošie dati. Uzsākot darbu, prasību-pārbaužu kontrolieris skenētu failu sistēmu un saglabātu atmiņā visu atrasto pārbaudes moduļu atrašanās vietas. Tomēr jāņem vērā, ka šāds risinājums var būt diezgan laikietilpīgs un, veidojot šādu realizāciju, jāpārdomā, vai failu sistēmas skenēšana katrā lietojumprogrammas izpildes reizē ir efektīva.

Kā alternatīvu šim risinājumam iespējams veidot lokālu datubāzi, kas satur informāciju par visiem attiecīgajā vidē pieejamajiem moduļiem. Šādā gadījumā prasību-pārbaužu kontrolierim būtu jānolasa tikai šī datubāze – piemēram, viens *XML* fails – kas apraksta visus moduļus uzreiz. Turklāt šādā veidā tiek rasta iespēja fiziski izvietot visās vidēs vienādu vides pārbaudes moduļu komplektu, bet veidot dažādas datubāzes katrai videi - testēšanas videi pilnīgu, kurā iekļautas arī „agresīvās" pārbaudes, lietošanas videi – tikai daļu pārbaužu.

Neatkarīgi no tā, kā prasību-pārbaužu kontrolieris atrod pārbaudes moduļus, jābūt definētam arī veidam, kā atsevišķai prasībai tiek piekārtota pārbaude. Kā viens no vienkāršākajiem risinājumiem šai problēmai ir veidot t.s. „abonēšanas" (angl. *subscription*) mehānismu, proti, katrs pārbaudes modulis „pierakstās" uz noteikta veida pārbaužu veikšanu. Šāds „pieraksts" arī veido atbilstību starp prasību un pārbaudes moduļu kopām (skat. piemēru

***3.6** att.* – katru moduli var piesaistīt vairākiem prasību veidiem, katram prasību veidam var būt vairāki pārbaudes moduļi).



3.6 att. **Vides moduļu „pierakstīšanās" uz atsevišķu prasību veidu pārbaudi**

Šādā gadījumā, meklējot pārbaudes moduļus, kas spēj pārbaudīt kādu noteiktu prasību veidu, testu koordinators pārskatītu „pierakstus" uz attiecīgo veidu un izvēlētos pašreizējam izpildes kontekstam piemērotākos vides pārbaudes moduļus.

Vides pārbaudes ideoloģijas sistēma paredz vismaz divus prasību pārbaudīšanas režīmus – „pilno" un „ātro". Šāds mehānisms var tikt izveidots, katram pārbaudes modulim piekārtojot „svaru", kas parāda moduļa izpildei nepieciešamo sistēmas resursu daudzumu. Meklējot piemērotu pārbaudes moduli, testu koordinators var ņemt vērā katras metodes svaru un tādējādi komplektēt pieļaujamas sarežģītības pārbaužu virkni. Vienlaicīgi, šāda metode palīdz piemeklēt labāko alternatīvu gadījumos, kad vienu un to pašu pārbaudi spēj veikt vairāki vides pārbaudes moduļi. ***3.6** att.* redzamajā piemērā – tīkla savienojuma prasību var pārbaudīt gan ar metodi, kas pārbauda kāda noteikta galamērķa sasniedzamību ar *ICMP* pakešu sūtīšanu, gan ar metodi, kas, izmantojot *WMI* infrastruktūru, noskaidro, vai operētājsistēmas līmenī pašlaik ir pieejams datortīkls. Otrā metode marķēta ar svaru „lēns", tādēļ prasību-pārbaužu kontrollerim vajadzētu izvairīties no tās lietošanas gadījumos, kad nepieciešams atlasīt tikai ātrākos pieejamos pārbaudes moduļus.

Pierakstīšanās forma ir atkarīga no veida, kā tiek aprakstītas programmatūras izpildes prasības, proti, no prasību aprakstīšanas valodas. Piemēram, ja valodas veidošanai izvēlēta naivā pieeja vai ar shēmām strukturētā *XML* pieeja (skat. 3.3.1.1 un 3.3.1.2 punktus), pierakstīšanās sistēmu var veidot vienkārši, katram vides pārbaudes modulim norādot *XML* elementu prasību dokumentā, kuru šis pārbaudes modulis spēj apstrādāt.

Tādējādi, ja pārbaudes moduļu datubāze veidota kā *XML* fails, pārbaudes moduli iespējams aprakstīt vienkopus ar tā „pierakstiem". ***3.7.** att.* parādīts *XML* fragments, kas apraksta direktoriju pieejamības pārbaudes moduli „*directoryDependencyChecker*". Piemērā redzamais apraksts paredzēts *Microsoft .Net* platformai, kur koda asambleju iespējams pilnībā

identificēt pēc tās „*strong name*” nosaukuma. Tādējādi, zinot taku, kur meklējamas koda asamblejas un asamblejas pilno nosaukumu, testēšanas koordinatoram būs iespējams atrast un ielādēt nepieciešamo kodu. Piemērā redzamie „*subscription*” elementi, kas iekļauti „*validator*” elementā, apraksta attiecīgā vides pārbaudes moduļa „parakstīšanos” uz divām diviem prasību veidiem prasību apraksta dokumentā – „*directoryDependency*” (atkarība no lokālajā failu sistēmā izvietota direktorija) un „*fileShareDependency*” (atkarība no *SMB* protokolā pieejama tīkla resursa).

```
<validator
    id="directoryDependencyChecker"
    assembly="ExtensionChapters, Version=1.0.0.0,
              Culture=neutral, PublicKeyToken=28f514e8eeca027c"
    assemblypath=".\bin"
    class="KR.SelfTest.DirectoryDependency">
         <subscription requirementNodeName="directoryDependency" />
         <subscription requirementNodeName="fileShareDependency" />
</validator>
```

3.7. att. **Vides pārbaudītāja apraksts XML valodā lietošanai .net platformā**

Ja prasību aprakstīšanas valoda apraksta prasības kā ar atsevišķiem resursiem saistītus ierobežojumus (kā tas tiek darīts *SDD* shēmā un no tās atvasinātās valodās, skat. 3.3.1.4. punktu), pierakstīšanās uz noteiktu prasību veidu nav tik vienkārša. Šādā gadījumā prasību pārbaudes modulis acīmredzami ir pielietojams noteiktām resursu/atribūtu kombinācijām, t.i. piemēram, reģionālo iestatījumu pārbaudītājs pielietojams tad, ja tiek izvirzīta prasība pret resursu „*operatingSystem*” un kādu no tā atribūtiem „*locale*”, „*shortDateFormat*”, „*decimalSeparator*” (u.c.). Ņemot vērā, ka šajā gadījumā prasību aprakstīšanas valoda ir *XML* bāzēta, viens no ērtākajiem veidiem, kā fiksēt šāda tipa „pierakstīšanās” uz prasībām, ir aprakstīt *XPath* [**72**] izteiksmi, kurai atbilstošas prasības attiecīgais pārbaudes modulis spēj apstrādāt. Šādā gadījumā ***3.7. att.*** redzamie „*subscription*” elementi būtu jāpapildina, aizstājot „*requirementNodeName*” ar atbilstošu *XPath* selektoru.

Šī darba 1.4. pielikumā demonstrēta prasību pārbaudes moduļu aprakstīšanas realizācija, kas izmanto „pierakstīšanās” mehānismu.

3.3.2.3. Vides informācijas nodošana pārbaudes modulim

Sagaidāms, ka praksē jaunus prasību pārbaudes moduļus varētu būt nepieciešams izstrādāt brīdī, kad programmatūras lietojuma vidē fiksēts jauns prasību veids, ko pašlaik iespējams pārbaudīt, tikai pielietojot cilvēka darbu.

Piemēram, var iedomāties gadījumu, kad novērots, ka apskatītā programma nespēj kvalitatīvi strādāt, ja datorā instalēta *OpenOffice 3* versija, kas jaunāka par 3.0.0 būvējumu.

Pārbaudi, vai dators atbilst šīm prasībām, cilvēks var veikt ārkārtīgi viegli (iedarbinot *OpenOffice* un izvēloties „Palīdzība > Par *OpenOffice*"), taču gadījumā, ja apskatītā lietojumprogramma izplatīta daudziem lietotājiem, visu datoru inspicēšana var aizņemt daudz laika. Lai šo uzdevumu automatizētu, ir iespējams izstrādāt nelielu programmu, kas būvējuma numuru noskaidro pēc datorā atrodamām pazīmēm – failu sistēmā atrodot failu *C:\Program Files (x86)\OpenOffice.org 3\program\version.ini* un tajā sameklējot nepieciešamo atslēgu „*ProductMinor*". Savukārt, ja līdzīgu pārbaudi būtu jāveic „*Microsoft Office*" instalācijas versijas noskaidrošanai, pārbaudes algoritms ir pavisam citāds un visdrošāk versijas numurs noskaidrojams, izmantojot sistēmas reģistru (*Windows registry*).

Var secināt, ka programmatūras izpildes vides automātiskas testēšanas uzdevums ir grūti unificējams — kaut arī prasības pēc satura ir līdzīgas (noskaidrot, vai sistēmā instalēta pareizā biroja programmatūras versija), to pārbaudīšanas algoritmi var būt būtiski atšķirīgi. Līdzīgi novērojumi izdarāmi arī attiecībā uz vides pārbaudītājam nepieciešamās informācijas (parametru) daudzumu: atsevišķos gadījumos pārbaudītājam nav nepieciešama papildu informācija par tā izpildes kontekstu (piemēram, pārbaude, kas noskaidro, vai dators ir pieslēgts datortīklam), bet citiem pārbaudes veidiem var būt nepieciešami daudzi programmatūras izpildes vidi raksturojoši parametri (piemēram, lai noskaidrotu, vai ir pieejama sistēmas datubāze, jāzina visa pieslēgšanās informācija – datubāzu servera nosaukums, datubāzes vārds u.c.) Tomēr, lai visus vides pārbaudes moduļus varētu pielietot vienotā arhitektūrā, kur vides testēšanu kopumā pārvalda testēšanas koordinators, ir nepieciešama formāla pārbaudes moduļu saskarnes unifikācija.

Atkarībā no izvēlētā vides testēšanas moduļu ielādes veida, mainās arī parametru nodošanas principi. Turpmāk analizētas tikai parametru nodošanas iespējas risinājumā, kad testu koordinators dinamiski ielādē testēšanas moduļus lietojumprogrammas procesā.

Relatīvi vienkārši ir iespējams unificēt vides testēšanas moduļu rezultātu atgriešanas mehānismu – testēšanas arhitektūra paredz, ka pārbaudes rezultāts ir vai nu „pārbaude veiksmīga" vai „pārbaude nav veiksmīga" ([**17**] gan piedāvā mehānismu, kurā definēts rezultāts „veiksmīga pārbaude ar brīdinājumu"). Tas ļauj testēšanas moduli uztvert kā funkciju, kuras atgrieztās vērtības tips ir patiesumvērtība (*boolean*).

Taču, ja pārbaudes modulis apskatīts kā funkcija, vides informācija tam jānodod kā funkcijas parametri. Parametru skaits, tips un struktūra ir iepriekš nezināmi – lai vides pārbaudītājam sniegtu pietiekamas ziņas par izpildes kontekstu, var būt pietiekami, ja tam tiek nodots vienkāršs pāru „atslēga-vērtība" kortežs, taču viegli iedomāties situācijas, kurās kā parametrs būtu jānodod arī sarežģītāka datu struktūra. Piemēram, ja izveidots modulis, kas pārbauda, vai failu sistēmā atrodams „kaut viens no minētajiem failiem" – šādam pārbaudes

modulim kā ieejas dati nepieciešams failu nosaukumu saraksts. Līdzīgi iespējams izveidot vides pārbaudes moduli, kas noskaidro, vai sistēmas uzstādīšanas laikā izveidota pareiza failu struktūra failu sistēmā. Šim mērķim kā parametrs jānodod kokveida datu struktūra.

Viens no vienkāršiem parametru nodošanas problēmas risinājumiem ir serializēt nododamos datus simbolu virknes veidā un ļaut vides pārbaudes modulim patstāvīgi atjaunot datu struktūru atmiņā. Serializēšanai iespējams izmantot gan jau par industrijas standartu kļuvušo *XML* [**49**], gan arī vienkāršākus datu aprakstīšanas formātus, piemēram, *JSON* [**73**], kas ir sintaktiski vienkāršāks un cilvēkam vieglāk lasāms formāts ar atbalstu daudzās programmatūras izstrādes platformās.

Parametru serializēšanas paņēmiens ļauj definēt vides pārbaudi kā funkciju ar vienu parametru (kur parametrs satur visus pārbaudes modulim nododamos datus simbolu virknes formā), un kur funkcijas rezultāta tips ir patiesumvērtība.

Šī pētījuma sākotnējās praktiskajās realizācijās *Microsoft .Net* videi katrs pārbaudes modulis veidots kā atsevišķa koda klase, kas realizē „*IEnvironmentValidator*" interfeisu. Šis interfeiss definē divas metodes „*SetEnvironmentData(string)*" un „*DoTests():boolean*". Pirmo no šīm metodēm testu koordinators izmanto, lai nodotu pārbaudes modulim darbam nepieciešamos datus, otro – lai izsauktu veicamās pārbaudes un saņemtu pārbaudes rezultātus. Šajās realizācijās pārbaudītājam kā parametrs tika nodots izpildes profila dokumenta fragments, kas satur prasību, kas izraisījusi attiecīgā moduļa izsaukšanu. Prasībai atbilstošais *XML* (tika izmantota 3.3.1.1 aprakstītā naivā pieeja) fragments saturēja visu nepieciešamo informāciju, lai pārbaudītājs varētu pilnvērtīgi strādāt. Šāds risinājums ļāva vienkārši realizēt testu koordinatoru, kuram, lai nodotu informāciju pārbaudes modulim, nebija dziļāk jāanalizē prasības saturs. Šī pieeja nav tiešā veidā izmantojama gadījumā, ja prasību aprakstīšanai lietota *SDD* bāzēta valoda – tajā resursu topoloģija aprakstīta atsevišķi no prasībām, kas uz tiem attiecināma (prasība atsaucas uz resursa identifikatoru, bet pats resurss aprakstīts atsevišķi). Tas nozīmē, ka testēšanas koordinatoram, pirms tas nodod informāciju tālāk pārbaudes modulim, jātransformē prasības apraksts tā, lai informācija par resursu un tā atribūtiem būtu iekļauta prasības aprakstā.

3.3.3. Izpildes parametru pielietošana prasību aprakstīšanai

3.3.3.1. Kāpēc programmatūras izpildes profilā nepieciešami parametri?

Pētot iespējas aprakstīt sistēmu izpildes prasības, viegli pamanīt, ka praksē izpildes prasību formulējums ne vienmēr ir aprakstāms, lietojot konstantes, kuru vērtība būtu zināma jau prasību aprakstīšanas laikā – bieži vien, formulējot prasību, tās aprakstā jāietver arī dažādas atsauces uz vidi, kurā programmatūra tiks darbināta. Piemēram, ja *MS Windows* vides

lietojumprogrammā paredzēts lejupielādēt failus no datortīkla un tos nepieciešams īslaicīgi saglabāt uz datora cietā diska, šim mērķim programma varētu izmantot operētājsistēmas piedāvāto īslaicīgās glabāšanas direktoriju. Šīs direktorijas atrašanās vieta nav vienāda visos datoros, to nosaka sistēmas mainīgais „%TEMP%". Tādējādi, lai aprakstītu izpildes prasību „lietotājam jābūt tiesībām veidot failus īslaicīgās glabāšanas direktorijā", nepieciešams veids, kā aprakstīt nepieciešamo direktoriju.

Līdzīgi, saskaņā ar labo programmatūras izstrādes praksi, informācijas sistēmu izstrādātāji izvairās dažādu resursu nosaukumus vai atrašanās vietas aprakstīt programmatūras kodā. Vidi raksturojošā informācija tiek apkopota atsevišķā, ērti pieejamā konfigurācijas datubāzē (piemēram, strukturētā teksta failā, tādos kā Windows INI faili vai dažādi XML bāzēti faili), no kuras katrā lietošanas sesijā nolasa nepieciešamo informāciju par vides konfigurāciju. Šādas prakses popularitātes dēļ programmatūras izstrādes platformās tiek iekļautas konfigurācijas nolasīšanas komponentes, kas tādējādi ļauj abstrahēties no konfigurācijas datubāzes formāta (piemēram, Java vides *„java.util.pref.Preferences"* klase, *Microsoft .Net* vides *„System.Configuration"* klase). Tā rezultātā, piemēram, lai pieslēgtos datubāzu serverim, lietojumprogramma vispirms noskaidro datubāzu servera nosaukumu no konfigurācijas faila un tikai tad uzsāk pieslēgšanos.

Lai aprakstītu atbilstošu izpildes vides prasību – „jābūt pieejamam lietojumprogrammai nepieciešamajam datubāzu serverim" – ir jāzina attiecīgā servera nosaukums. Nosaukumu ir iespējams norādīt programmas izpildes prasību profilā, taču tas izraisa informācijas dublēšanos (servera nosaukumam jābūt gan konfigurācijas failā, gan izpildes profilā), tādējādi paverot jaunas kļūdīšanās iespējas, manuāli veicot sistēmu uzturēšanas darbu. Šī iemesla dēļ ir būtiski, ka apkārtējās vides testēšanas infrastruktūra vides konfigurāciju noskaidro no tiem pašiem informācijas avotiem, no kuriem to nolasa pati lietojumprogramma.

3.3.3.2. Parametru aprakstīšanas risinājums citās vidēs

Vienkāršākais risinājums šai problēmai ir – papildināt programmatūras prasību aprakstīšanas valodu ar mainīgo jēdzienu un ļaut sistēmu izstrādātājiem lietot vidi aprakstošos mainīgos situācijās, kad prasība apraksta konkrētai izpildes videi specifisku konfigurācijas vērtību. Tādējādi pēc programmas izpildes prasību profila nolasīšanas vides testēšanas koordinators varētu aizstāt visas mainīgo vērtības ar attiecīgajā vidē definētajām vērtībām un tālāk turpināt darbu kā iepriekš aprakstīts – katrai prasībai piemeklējot vides testēšanas moduli un to izpildot.

Līdzīgs risinājums piedāvāts *SDD* deskriptorā [**53**] – atsevišķās deskriptora dokumenta daļās, to skaitā arī topoloģijas apraksta un prasību apraksta daļā paredzēta iespēja iekļaut

mainīgo nosaukumus, tos speciāli marķējot pierakstā „`{$MainīgāVārds}`". Deskriptora dokumenta apstrādes laikā tie tiek aizstāti ar tobrīd aktuālajām vērtībām. *SDD* iedala mainīgos trīs veidos:

- no izpildes vides nolasāmi mainīgie – *SDD* izpratnē programmatūra saskaras ar apkārtējo vidi kā ar atsevišķiem resursiem (kur katru resursu raksturo tā atribūti, kuru var būt pēc patikas daudz) - jebkurš šāds resursa atribūts izmantojams *SDD* deskriptora shēmā kā mainīgais;
- lietotāja doti procesa parametri – tā kā *SDD* deskriptori paredzēti programmatūras instalācijai, tie paredz, ka daļu konfigurācijas sniedz lietotājs, kurš instalē programmatūru - piemēram, instalācijas mērķa direktorija taka;
- atvasināmi mainīgie – to vērtību nosaka, balstoties uz citu atribūtu vērtībām. *SDD* lietošanas piemēros [**58**] demonstrēta pieeja, kur mainīgā „*InstallRoot*" (taka, kurā programmatūra tiks instalēta) vērtība tiek aprēķināta atkarībā no cita mainīgā „*OSType*" (operētājsistēmas veids) vērtības.

SDD piedāvātais risinājums sniedz parocīgu veidu, kā sintaktiski aprakstīt mainīgos, taču tikai daļēji izmantojams šajā darbā formulētajai problēmai. Viena no būtiskākajām neatbilstībām ir tā, ka *SDD* standartā, lai izvairītos no piesaistes kādai konkrētai platformai, ne tikai netiek definēts tehniskais process, kā nolasīt vides atribūtu vērtības, bet deskriptora izstrādātājam pat netiek dota iespēja šo niansi specificēt. Kaut arī dažādi vides resursi deskriptora dokumentā aprakstīti vienotā sintaksē, to atribūtu nolasīšanai jālieto ļoti atšķirīgas metodes, tādēļ ļoti būtiski ir definēt, kā attiecīgais atribūts nolasāms. Piemēram, resursam „operētājsistēma" ir atribūts „īslaicīgās glabāšanas direktorija taka" un atribūts „datuma pieraksta formāts", bet katrs no tiem nolasāms pilnīgi citā veidā – pirmais kā sistēmas mainīgais, bet otrs glabājas MS Windows sistēmas reģistrā.

3.3.3.3. Programmatūras izpildes vides testēšanas infrastruktūrai piemērots algoritms

Lai būtu iespējams pilnvērtīgi aprakstīt izpildes prasību profila dokumentā pieminēto mainīgo vērtības, tiek piedāvāts no vairākiem soļiem sastāvošs risinājums.

1. Izstrādāt nelielu funkciju bibliotēku, kas satur dažādu vides konfigurāciju klašu nolasīšanas funkcijas, piemēram, „Windows .ini failu lasītājs", „Windows reģistra atslēgu lasītājs" u.c. Sākotnējie testi parāda, ka šādas funkcijas bieži vien triviāli vienkāršas un to izstrāde nav resursietilpīga.
2. Pieejamās konfigurācijas vērtību nolasīšanas funkcijas aprakstīt programmatūras izpildes prasību profilā (tās saturīgi nozīmēs, piemēram, šādu aprakstu: „ja

nepieciešams nolasīt *.ini* failā ierakstītu atslēgu, jāizmanto koda klase A, kas atrodama koda bibliotēkā B")

3. Katram profilā izmantojamajam mainīgajam piekārtot formulu, kā tas aprēķināms. Formula var saturēt atsauces uz citiem mainīgajiem, izmantot iepriekšdefinētās atribūtu nolasīšanas funkcijas vai saturēt konstantas vērtības.
4. Programmas izpildes prasību profilu papildināt ar jaunu sadaļu, kas satur visas mainīgo vērtību nolasīšanas formulas

Tādējādi, interpretējot programmas izpildes profila tekstu, ja tajā sastopama atsauce uz kāda mainīgā vērtību, testēšanas koordinatoram jārīkojas pēc sekojoša algoritma:

- mainīgo definīciju sarakstā jāatrod nepieciešamais mainīgais (identificējot to pēc nosaukuma) un jānoskaidro tā aprēķināšanas formula,
- jāinterpretē atrastā formula:
    - visas formulas definīcijā sastaptās atsauces uz citiem mainīgajiem jāaizstāj ar to vērtībām, rekursīvi pielietojot šo pašu algoritmu,
    - formulā sastaptās atsauces uz vides atribūta nolasīšanas funkcijām jāapstrādā pēc šāda algoritma,
        - funkciju sarakstā jāatrod nepieciešamā funkcija (identificējot to pēc nosaukuma) un jānoskaidro, kurš koda artefakts realizē šo funkciju,
        - jāielādē nepieciešamais koda artefakts un jāizsauc atribūta nolasīšanas metode, tai kā parametrus nododot mainīgā aprēķina formulā esošo informāciju.

Lietojot šādu pieeju, viegli iespējams aprakstīt pat relatīvi sarežģītas vides mainīgo lielumu noskaidrošanas ķēdes. ***3.8****. att.* demonstrē vairāku savstarpēji saistītu mainīgo aprakstīšanas shēmu. Sākotnēji tiek definēts mainīgais *programFilesPath*, kura vērtības noskaidrošanai pielietota viena no vides vērtību nolasīšanas funkcijām – *resolve-WindowsVariable* (kas tipiskas *MS Windows* instalācijas gadījumā atgriež vērtību „*c:\program files*"). Mainīgo *rootPath* un *iniFilePath* definīcijas parāda atvasinātu mainīgo veidošanu, atsaucoties uz agrāk definētu mainīgo vērtību un to savienojot ar konstantu simbolu virkni. Savukārt mainīgā *dbServer* definīcijā atkal izmantota viena no vides atribūtu nolasīšanas funkcijām *resolve-IniParameter*. Viegli pamanīt, ka divu piemērā izmantoto nolasīšanas funkciju: *resolve-WindowsVariable* un *resolve-IniParameter* parametru skaits ir atšķirīgs. Piemērā parādītais kods aprakstīts pseidokodā, taču praktiskās apkārtējās vides infrastruktūras testēšanas realizācijās tas jāpielāgo atbilstoši izvēlētajai apraksta valodai.

```
$programFilesPath=resolve-WindowsVariable("programfiles")
$rootPath={$programFilesPath}\testsys\
$iniFilePath={$root'Path}\preferences.ini
$dbServer=resolve-IniParameter($iniFilePath, "TestSection", "KeyName")
```

3.8. att. **Vides parametru nolasīšanas formulu pieraksts pseidokodā**

Šī darba 1.3. pielikumā parādīts un izskaidrots uz *XML* bāzētas valodas papildinājums, kas vienkopus apraksta gan vides mainīgo definēšanas, gan resursu atribūtu nolasīšanas funkciju sarakstu.

3.3.4. Izpildes profila versiju veidošana

Uzņēmumā ieviešot programmatūras izpildes prasību automatizētu pārbaudi kā standarta praksi, šī metode prasa papildu darba ieguldījumu – gan vienreiz veicamu darbu izpildē kā vides testēšanas koordinatora izstrāde, gan biežāk veicamos darbos, piemēram, izstrādājot jaunus vides pārbaudes moduļus (detalizētāk skat. nodaļu 3.4 „Programmatūras izpildes profila veidošana").

Viens no būtiskiem veicamajiem papildu darbiem ir programmas izpildes prasību dokumenta uzturēšana un aktualizēšana, lai tas atbilstu aktuālajai darījumu lietojumprogrammas versijai.

Lai noskaidrotu darījumu lietojumprogrammas mainības dinamiku, tika veikta 4.1 nodaļā aprakstītās organizācijas atsevišķas programmatūras sistēmas versiju vēstures analīze. Šīs sistēmas izstrāde veikta iteratīvi, tādēļ versiju skaits ir salīdzinoši liels – 59 versijas pusotra gada laikā, t.i. gandrīz 40 versijas gadā. ***3.1***. tabulā parādīts šo versiju kalendārais sadalījums, ar jēdzienu „versiju intervāls" saprotot dienu skaitu starp kādas versijas uzstādīšanu un datumu, kad uzstādīta iepriekšējā programmatūras versija. Kopsavilkuma dati parāda, ka iteratīvajai pieejai piemērots [**74**] versiju nodošanas intervāls (7 un vairāk dienas) tiek realizēts tikai apmēram 42% gadījumu, savukārt pārējos gadījumos jaunas versijas tiek nodotas biežāk, bieži vien – pat vairākas reizes dienā. Biežās izmaiņas pārsvarā grupējas ap pēc būtības jaunu izmaiņu ieviešanu izstrādājamajā sistēmā un to pamatā ir nepilnīgi vai nepareizi interpretētas darījumu loģikas prasības.

3.1. tabula

Programmatūras versiju uzstādīšanas biežuma sadalījums iteratīvas programmatūras izstrādē

Intervāla nosaukums	*Versiju skaits ar šādu intervālu*	*Šādu versiju īpatsvars*
Tajā pašā dienā	1	2%
Viena vai divas dienas	12	20%
3-7 dienas pēc iepriekšējās versijas	20	34%

Vairāk nekā 7 dienas	24	41%
Kopā	59	

Gadījumos, kad jauna programmatūras versija tiek uzstādīta gandrīz ik dienas, versijas sagatavošana aizņem būtisku laika daļu, tādēļ pilnvērtīga programmatūras izpildes profila dokumenta sagatavošana var kļūt apgrūtinoša, sevišķi tad, ja process nav pilnībā automatizēts. Praktiskie pielietojumi (skat. 4. nodaļu „Praktiskie metodes pielietojumi") parāda, ka pat relatīvi vienkārša projekta gadījumā programmatūras izpildes profils sastāv no daudzām prasībām. Ņemot vērā aprakstāmo prasību skaitu, pastāv risks, ka izpildes prasību profila dokuments netiks sagatavots vienlaicīgi ar programmatūras versiju un tādējādi vides pārbaudes nebūs iespējamas. Šādas rīcības rezultātā savukārt pieaug risks, ka uzstādītā programmatūra nebūs pilnībā darboties spējīga nepareizas konfigurācijas dēļ.

Lai vienkāršotu programmatūras izpildes profila sagatavošanu, arī šim programminženierijas artefaktam var piemērot versiju veidošanas mehānismu. Tas ir, katru programmatūras versiju apraksta atsevišķs programmatūras prasību dokuments. Ja specifiska prasību dokumenta nav, tiek pieņemts, ka sistēmai saglabājas iepriekšējas versijas prasības. Šāda pieeja vienkāršo prasību failu sagatavošanu, turklāt vienlaicīgi var pastāvēt dažādas prasību dokumenta versijas (ticams, ka pēc programmatūras ieviešanas ekspluatācijas vidē veidojas atšķirīgas versijas katrai videi – testa vidē ir jaunāks kods nekā ekspluatācijā).

Izmantojot versiju mehānismu, ir iespējams izpildes vidē saglabāt visu instalēto versiju prasību vēsturi (tehniskā realizācija nav būtiska - atsevišķos failos, direktorijās vai datubāzē). Šāda pieeja ļauj definēt sākotnējās prasības kā bāzi un, gatavojot nākamo programmatūras versiju, atsaukties uz bāzes prasību sarakstu, to mainot – definējot operācijas "jauna prasība" un "atcelta prasība". Lai realizētu šādu versiju mehānismu, nepieciešams unikāli identificēt katru prasību.

Tādējādi programmatūras profila vēsture būtu kā parādīts koda piemērā ***3.9***. *att.* – sākotnējā versija 1.0.0.1 definē atsevišķas prasības, katrai no tām piešķirot unikālu identifikatoru. Nākamā izpildes prasību profila versija 1.0.0.2 *papildina* iepriekšējo versiju, pievienojot jaunu prasību, tas ir, ir spēkā gan iepriekš definētās prasības gan jaunajā versijā papildu pieliktās. 1.0.0.3 versija papildina iepriekšējo versiju, tādējādi izveidojot versiju ķēdi. Tā definē jaunu prasību P5 attiecībā uz failu „f1.html", bet tā ir pretrunā ar iepriekš definēto prasību P1, tādēļ agrākā prasība tiek atcelta.

```
Versija 1.0.0.1
     P1:  Jābūt pieejamam failam F1.html,
          tā baitu kontrolsummai jābūt c321e6f26b072035d1d281d2be5935e6
     P2:  Jābūt pieejamam failam F2.html
     P3:  Jābūt pieejamai bibliotēkai
          OurCommonLibrary,version=1.0.0.0,publicKeyToken=e3a944d90afb52,
          culture=neutral
Versija 1.0.0.2, papildina versiju 1.0.0.1
     P4:  Jābūt pieejama failam F3.aspx,
          tā baitu kontrolsummai jābūt 755f465059799be2e7561a11a78548ef
Versija 1.0.0.3, papildina versiju: 1.0.0.2
     P1:  atcelts
     P5:  Jābūt pieejamam failam F1.html,
          tā baitu kontrolsummai jābūt ade01cd5229bbaf2ba0c5d9ba9f6dcfb
     P3:  atcelts
     P6:  Jābūt pieejamai bibliotēkai
          OurCommonLibrary,version=1.0.3.0,publicKeyToken=e3a944d90afb52,
          culture=neutral
```

3.9. att. **Programmatūras prasību faila versiju saistīšana**

Veidojot jaunu programmatūras laidienu vai arī gadījumos, kad ķēdē savienoto izpildes profilu versiju skaits kļūst grūti aptverams, visas veiktās izmaiņas iespējams apvienot vienā izpildes prasību profilā, kuru turpmāk izmanto kā bāzes prasību dokumentu.

3.4. Programmatūras izpildes profila veidošana

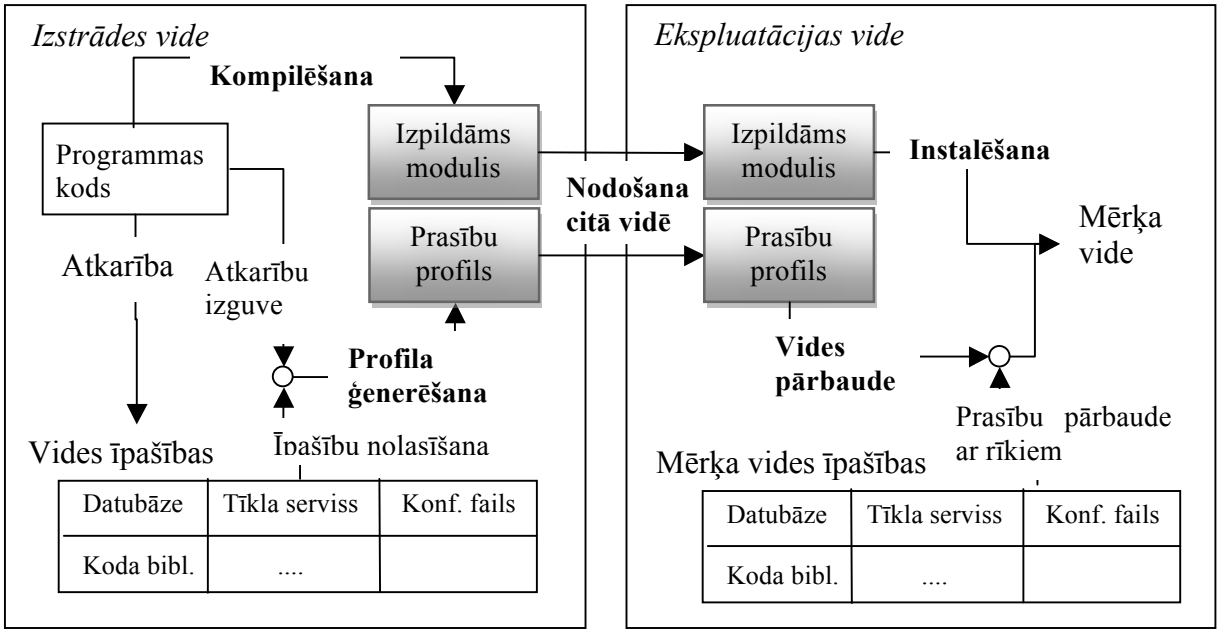
Viena no būtiskākajām izpildes vides testēšanas tehnoloģijas sastāvdaļām ir programmatūras izpildes profils – dokuments, kas apraksta programmatūras prasības pret izpildes vidi, tādēļ būtisks ir jautājums, kurā programmatūras dzīves cikla fāzē šis dokuments tiek veidots un kam tas jādara.

Ziņas par programmatūras specifiskām prasībām attiecībā uz izpildes vidi parādās jau ļoti agrīnā programmatūras dzīves cikla posmā, kad tiek specificētas tehnoloģijas, kas tiks pielietotas aprakstāmās programmatūras būvei. Tā, piemēram, ja zināms, ka klienta-servera arhitektūrā būvētas sistēmas komponenšu savstarpējā saziņa tiks realizēta, izmantojot SOAP tīmekļa servisus, tas ļauj izdarīt vairākus secinājumus – ka datoriem, uz kuriem tiek darbināta sistēma, jābūt pieslēgtiem datortīklam, ka tiem jābūt savstarpēji savienotiem (t.sk., ka servera puses datora ugunsmūrī jābūt atvērtiem nepieciešamajiem *TCP* portiem). Viegli saprast, ka šādi tiek formulētas tikai viegli saprotamas un iepriekš prognozējamas prasības, taču projektēšanas fāzē esošā detalizācijas pakāpe nav pietiekama precīzu prasību noteikšanai. Ja programmatūras izstrādei tiek izmantota *MDE* (angl. *model-driven engineering*) pieeja [**75**], daļu programmatūras profila ir iespējams ģenerēt, lietojot piemērotus *MDA* rīkus. Šāds transformācijas papildinājums būtu iespējams, piemēram, *UML* modelī lietojot specifiskus stereotipus, ka apzīmē noteiktu prasību veidu. Modeļa transformācijas laikā, analizējot modeļa artefaktiem piemērotos stereotipus, tiktu veidota atbilstoša dokumentācija vai ģenerēts

specifisks kods. Šāds risinājums ir ārpus pašlaik apskatāmā darba robežām un tādēļ detalizētāk netiks apskatīts.

Daudz detalizētākas un precīzākas programmatūras izpildes prasības kļūst zināmas praktiskās izstrādes jeb kodēšanas laikā, kad projektējums tiek transformēts platformai specifiskā kodā. Šajā fāzē tiek radīta visa programmatūras sistēmai nepieciešamā infrastruktūra, turklāt tā ir korekti konfigurēta, tas ir, sistēma spēj attiecīgajā vidē veiksmīgi darboties. Tas nozīmē, ka programmatūras izstrādes vide var kalpot par labu paraugu programmatūras prasību apkopošanai, jo tajā ir apzināti vai neapzināti izveidota tās darbam ideāla konfigurācija. Uzsākot programmatūras nodošanu citā vidē (piemēram, sistēmas nodošanu testā), šo zināšanu nodošana tālāk ir ļoti būtiska, tādēļ labs risinājums prasību profila automātisku sagatavošanu ieviest kā vienu no programmatūras versijas komplektēšanas sastāvdaļām, izgūstot ziņas par programmatūras prasībām pret izpildes vidi no programmatūras un analizējot īpašības, lai noteiktu vēlamo konfigurāciju.

Ideāls programmatūras prasību apraksta sagatavošanas scenārijs būtu tāds, kurā prasības tiktu automātiski noteiktas, analizējot kodu un, balstoties uz atrastajām prasībām, tiktu piemeklēta tām vēlamā vides konfigurācija. ***3.10****. att.* parādīts vēlamais programmatūras izpildes profila sagatavošanas risinājums – vienlaikus ar programmatūras koda kompilāciju tiek apzinātas visas kodā iekļautās atkarības no izpildes vides. Pēc tam, vadoties pēc izstrādes vides īpašībām (jo zināms, ka izstādes vidē programmatūra spēj labi darboties), tiek sagatavots programmatūras prasību profils. Profils tiek nodots uzstādīšanai citā izpildes vidē, piemēram, ekspluatācijas vidē, kopā ar citiem nododamajiem artefaktiem. Tālākajā izpildes vidē profilu izmanto, lai salīdzinātu vides īpašības ar tām prasībām, kas minētas prasību profilā.



3.10. att. **Programmatūras izpildes profila sagatavošana un nodošana citā vidē**

Darba nākamajās nodaļās apskatīti risinājumi, kas ļautu automātiski identificēt programmatūras prasības un nolasīt vides īpašības izstrādes vidē.

3.4.1.1. Pilnībā automatizēts atkarību meklēšanas rīks

Viens no risinājumiem būtu realizējams kā programmatūras pirmkoda (vai baitkoda) analīzes rīks, kas kodā mēģina identificēt „jutīgus" fragmentu šablonus. Piemēram, analizators varētu identificēt kādas funkcijas izsaukumu, kas simbolu virkni cenšas atpazīt kā datuma tipa vērtību un no tā secināt, ka programma ir atkarīga no reģionālajiem iestatījumiem, konkrēti – datuma formāta iestatījuma operētājsistēmā. Pēc tam analizators varētu noskaidrot, kāds ir datuma formāts izstrādes vides datorā, un izvirzīt programmatūras izpildes prasību – "lai programma sekmīgi darbotos, datuma formātam jābūt (ievietota vērtība no izstrādes vides)". Tomēr jāņem vērā, ka šādu „jutīgo" šablonu aprakstīšana var būt sarežģīta. Piemēram, varētu tikt definēts *PHP* valodā rakstīta koda šablons „izsaukums *fopen($filename, $mode)* [1] izraisa failu sistēmas prasību". Atkarībā no tā, kāda ir otrā šīs funkcijas parametra vērtība, mainītos arī failu sistēmas prasības tips – ja režīms ir „atvērt pastāvošu failu lasīšanai", attiecīgā prasība ir „failam ar nosaukumu *$filename* jābūt izveidotam", savukārt, ja režīms ir „rakstīšanai atvērt jaunu failu", tad jāveido tieši pretēja prasība –„fails ar nosaukumu *$filename* nedrīkst būt izveidots"

[1] fopen – P*HP* valodas funkcija failu atvēršanai. Atkarībā no otrā parametra vērtības, fails var tikt atvērts dažādos piekļuves režīmos – lasīšanai vai rakstīšanai, pieprasot izslēdzošu piekļuvi failam vai pieļaujot paralēlas faila lasīšanas sesijas

Šādu lietojuma šablonu izveide prasītu veidot pilnu apskatāmās programmas izpildes grafu, pēc tam ļaujot relatīvi viegli identificēt jutīgo fragmentu šablonus. Taču šādai pieejai ir vairāki negatīvie aspekti – pirmkārt, pilnā programmas izpildes grafa būve tiek uzskatīta par vispārīgi sarežģītu problēmu [**76**], otrkārt – ne visi programmā identificējamie jutīgie šabloni patiešām apzīmē prasības pret izpildes vidi. Piemēram, ja programmas kodā jau apstrādāta situācija, kad meklētais fails var nebūt pieejams, nav nepieciešams veikt automātiskus testus. Programmatūras izpildes vides testēšana drīzāk nepieciešama gadījumos, kad pieņēmumi par vidi izdarīti netieši un parādās kā programmatūras izpildes blakusefekts.

3.4.1.2. Daļēji automatizēta prasību identificēšana

Kā alternatīva iepriekšminētajai pilnībā automatizētās prasību identificēšanas metodei iespējams realizēt daļēji automatizētu prasību identificēšanu, šajā procesā daļu darba uzticot sistēmu izstrādātājiem. Šī pieeja paredz, ka programmatūras koda izstrādes laikā programmētāji izmanto papildus koda marķējumus, lai aprakstītu programmatūras koda atkarības no apkārtējās vides. Nodotamās koda versijas sagatavošanas laikā tiek veikta koda skenēšana, veidojot tikai daļēji precīzu programmatūras izpildes grafu, un tajā identificējot programmētāju norādītos marķējumus. Šādā veidā tiktu identificēti tikai tie koda punkti, kas tiešām izraisa programmatūras prasības pret izpildes vidi.

Piemēram, ja aprakstītā sistēma tiek izstrādāta kādā no *Microsoft .Net* saimes valodām, tajās ir iespējams izmantot t.s. koda deklaratīvos atribūtus [**77**] – klasi vai funkciju papildinošu metadatu informācijas vienību, kas pieejama ar koda refleksijas palīdzību (tādējādi vienkāršojot tieši baitkoda analīzi). Piemēram, varētu tikt ieviesta koda atribūtu klase „*NetworkDependency*", kas apzīmē programmatūras prasību pēc *TCP* savienojuma ar kādu citu tīkla resursu. Šai atribūtu klasei būtu vairāki parametri – protokols, virziens, ports). Izstrādājot darījumu koda funkciju, kas izmanto tīkla savienojumu, attiecīgā funkcija tiktu papildināta ar šādu deklaratīvo atribūtu, skat. ***3.11****. att.*

```
<TcpDependency(Protocol.TCP, Direction.Out, 80)> _
Public Function ReadData()
```

3.11. att. **Koda funkcijai ReadData norādīts papildu deklaratīvais atribūts**

Pamanot pazīmi „*TCPDependency*", koda skenēšanas rīks izpildes profilā veidotu jaunu programmatūras prasību, pievienojot deklaratīvā atribūtā parametrus. Viegli pamanīt, ka šāda prasību aprakstīšana kļūst sarežģīta, ja kodā jāiekļauj vairāk informācijas par prasību, kas ir specifiska videi (piemēram, ja ***3.11****. att.* paraugā būtu jāpievieno arī *TCP* savienojuma galamērķa IP adrese, ne tikai virziens un ports). Viens no risinājumiem šādās situācijās ir izmantot izpildes vides mainīgos, līdzīgi, kā tas aprakstīts 3.3.3.3 nodaļā.

3.5. Metodes adaptācijas principi

Lai programmatūras izstrādes uzņēmumā pilnībā adaptētu šajā darbā aprakstīto tehnoloģiju, jāveic daudzi projektēšanas un plānošanas soļi, kā arī nepieciešams izstrādāt relatīvi lielu papildus koda kvantumu – gan jārealizē pati testēšanas infrastruktūra, gan jāveido standarta pārbaudes moduļu bibliotēka, kuras tālāk papildināšana ir relatīvi vienkārša. Galvenie šī procesa soļi parādīti ***3.2***. tabulā.

3.2. tabula

Metodes adaptācijas laikā veicamie uzdevumi un to izpildītāji

Nr.	*Uzdevums*	*Veicējs*	*Rezultāti*
1	Prasību aprakstīšanas valodas definēšana Valodā izmantojamo mainīgo izteiksmju aprakstīšana	Valodas autori	Visu valodas sadaļu sintakses apraksti Piemēri
2	Koda dinamiskās ielādes principu specificēšana Testēšanas infrastruktūras galveno moduļu izstrāde	Valodas autori	Koda bibliotēka, API
3	Uzņēmumā izstrādātās programmatūras tipiskā prasību komplekta noteikšana	Sistēmu arhitekti, vadošie izstrādātāji, projektu vadītāji	Automatizējamo pārbaužu saraksts
4	Stratēģijas noteikšana tehnoloģijas ieviešanai uzņēmumā	Sistēmu arhitekti, projektu vadītāji	Dokumentētas vadlīnijas
5	Tipiski lietojamo vides pārbaudes moduļu izstrāde	Vadošie izstrādātāji, izstrādātāji	Dokumentēta koda bibliotēka
6	Jaunas programmatūras versijas būvēšana un nodošana Atbilstoša izpildes profila izveide	Izstrādātāji, kas atbildīgi par konfigurācijas vadību	Izpildāmais kods, izpildes profila dokuments
7	Jaunu prasību aprakstīšana, Papildus pārbaudes moduļu izstrāde	Valodas autori, Attiecīgās darījumu sistēmas izstrādātāji	Dokumentēts pārbaudes moduļa kods Informācija citiem izstrādātājiem
8	Prasību pievienošana esošam izpildes profila dokumentam	Testētāji, uzturētāji	Papildināts profila dokuments, informācija sistēmu izstrādātājiem

3.2. tabulas 1. uzdevums detalizētāk aprakstīts 3.3.1. nodaļā „Prasību aprakstīšanas valoda". Kā darba veicēji tabulā norādīta darbinieku grupa „valodas autori", kas praksē var būt daži (1-3) sistēmu arhitekti, kuri piedalās arī turpmākajos tehnoloģijas ieviešanas posmos. Šajā solī ir būtiski vienoties par izvēlēto stratēģiju – esošas sintakses adaptācija vai jaunas sintakses izstrādāšana un diezgan precīzi specificēt izvēlēto valodu. Visu tālāko posmu izpildē nepieciešams šī soļa kvalitatīvs rezultāts.

Otrais uzdevums, testēšanas infrastruktūras galveno moduļu izstrāde, paredz praktiski realizēt testēšanas koordinatoru, kas spēj apstrādāt 1. solī definētajā valodā rakstītus dokumentus, kā arī spēj dinamiski ielādēt vides testēšanas moduļus atbilstoši specifikācijai, kas rodas šī soļa realizēšanas laikā. Arī šo soli realizē grupa „valodas autori" – šeit būtiski, ka izstrādājamais programmatūras kods ir sarežģīts, jo darbojas ar koda refleksiju, dinamisku funkciju izsaukšanu utt., tādējādi tas var būt grūti deleģējams mazāk kvalificētiem darbiniekiem.

Pēc 2. uzdevuma izpildes ir izstrādāts funkcionējošs apkārtējās vides testēšanas infrastruktūras paraugs, kas vēl praktiski nav pielietojams, jo nav realizēts neviens pārbaudes modulis. 3. uzdevuma mērķis ir izskaidrot jaunā vides testēšanas modeļa darbības principus plašākam darbinieku lokam un saņemt atgriezenisko saiti par tipiskākajiem lietojumiem, kādi ir paredzami vides testēšanas rīkiem, tas ir, apkopot biežāk lietojamo vides pārbaudes moduļu sarakstu. Lai noskaidrotu, kādas ir tipiskākās aprakstāmās prasības, var lietot dažādus informācijas avotus:

- vēsturiskos datus par sistēmu atteicēm, ieskaitot notikumu žurnālus un atskaites,
- sistēmu uzstādīšanas aprakstus, uzstādīšanas priekšnoteikumu aprakstus, kas attiecas uz uzstādīšanas vidi,
- sistēmu uzturētāju mācību materiālus,
- jauna datora instalēšanas, sagatavošanas un konfigurēšanas instrukcijas, u.c.

4. soļa uzdevums ir dažādām iesaistītajām pusēm savstarpēji vienoties par tehnoloģijas tālāku ieviešanu uzņēmumā. Tas ietver vienošanos par šādiem jautājumiem:

- kurš būs atbildīgs par prasību dokumentu izstrādi un uzturēšanu? (jāatceras, ka, ieviešot jaunas programmatūras versijas, profils var tikt papildināts ar jaunām prasībām vai mainīt iepriekšējās prasības)
- kā tiks definēti jaunu prasību apraksti (paredzams, ka pēc tehnoloģijas ieviešanas var parādīties jauni izpildes prasību veidi, kam būs nepieciešams prasības aprakstīšanas veids un kam tiks izstrādāti atbilstoši vides pārbaudes moduļi; tāpat jāformulē uzņēmuma politika, kā izstrādātāji tiek informēti par iespējām aprakstīt jaunus prasību veidus)
- kā tiks dokumentēti prasību pārbaudes moduļi, kur glabāsies dokumentācija?

5. solis – bieži lietojamo vides pārbaudes moduļu izstrāde – turpina iepriekš uzsākto praktisko disciplīnu, radot pārbaužu „džentlmeņa komplektu", ka var tikt pielietots gandrīz jebkurā sistēmu izstrādes projektā. Būtiski ir šo kodu izstrādāt pēc iespējas kvalitatīvi un atbilstoši iepriekš noteiktajām vadlīnijām, jo tas tiks uztverts kā piemērs tālākajos tehnoloģijas ieviešanas posmos.

6. solis apskata tehnoloģijas lietošanu jau pēc tās adaptācijas uzņēmumā, proti, to pielietojot praktiskā sistēmu izstrādē, kad nododamo artefaktu komplekts jāpapildina ar programmatūras prasību dokumentu. Atkarībā no uzņēmuma vai uzņēmumā pielietotās dzīves cikla pieejas, šo darbu mēdz realizēt dažādas dzīves ciklā iesaistītās puses. Piemēram, saskaņā ar MSF (angl. *Microsoft Solutions Foundation,* [**78**]), šo darbu būtu loģiski realizēt izstrādes lomas dalībnieki kā vienu no infrastruktūras izstrādes funkcionālās sfēras elementiem.

Ja, veidojot programmatūras prasību aprakstu, tiek secināts, ka programmatūrai ir kāda prasība, kas nav aprakstāma ar pašlaik definēto prasību valodu, attiecīgās darījumu sistēmas izstrādātāji realizē automatizēto pārbaudes moduli un sadarbībā ar aprakstīšanas valodas autoriem vienojas par valodas papildinājumu, kas ļautu prasību aprakstīt.

Būtiski arī, ka programmatūras izpildes profila sagatavošana nebeidzas ar 6. solī aprakstīto programmatūras versijas komplektēšanu – ja tālākās izpildes vidēs (piemēram, testēšanas vidē) tiek novērotas iepriekš nedokumentētas programmatūras prasības pret izpildes vidi, arī sistēmu uzturētāji vai testētāji var papildināt izpildes profilu ar jaunām prasībām. Šeit gan svarīgi, ka par veiktajām izmaiņām tiek informēti arī tie darbinieki, kuri sagatavo prasību apraksta dokumentu.

4. PRAKTISKIE METODES PIELIETOJUMI

Šajā darbā aprakstītā tehnoloģija praksē aprobēta divās dažāda profila organizācijās ar atšķirīgiem programmatūras izstrādes un lietošanas principiem.

Sākotnējā aprobācija notikusi organizācijā, kurā lielāko daļu darījumu sistēmu izstrādā uzņēmuma iekšienē un šo risinājumu uzstādīšanu un dažādus citus programmatūras uzturēšanas uzdevumus veic manuāli. Šajā organizācijā ieviešot vides pārbaudes mehānismus, galvenais mērķis bija atbrīvoties no nehomogēnas vides un sistēmu savstarpējo atkarību radītajām programmatūras atteicēm. Pieredze, šajā organizācijā ieviešot automatizētās vides testēšanas metodes, aprakstīta nodaļās 4.1 – „Pielietojums organizācijā, kurā tiek veikta iekšēja IS izstrāde" un 4.2 – „Metodes pielietošana sadalītu sistēmu integritātes pārbaudei"

Otra organizācija, kurā tikusi aprobēta vides testēšanas tehnoloģija, ir datorsistēmu ražotājs, kas izstrādā specializētu programmatūru ārējiem pasūtītājiem. Šīs organizācijas īpatnība ir tā, ka tās izstrādātas datorsistēmas tiek vienlaikus izmantotas pie daudziem pasūtītājiem, taču gandrīz katra pasūtītāja pieejamā datorsistēmu infrastruktūra ir dažāda, turklāt ne katrs pasūtītājs ir gatavs savlaicīgi sekot dažādiem programmatūras laidieniem. Tādējādi šai programmatūras izstrādes organizācijai vienlaikus jāuztur ārkārtīgi daudz dažādu produktu variāciju. Šīs organizācijas pieredze apkopota 4.3 nodaļā „Vides testu pielietošana problēmu ziņotāja programmatūrā"

4.1. Pielietojums organizācijā, kurā tiek veikta iekšēja IS izstrāde

4.1.1. Par organizāciju

Lai nodrošinātu sev uzticēto funkciju kvalitatīvu, bet vienlaikus arī ātru izpildi, Latvijas Bankai (*LB*) nepieciešamas dažādas informācijas sistēmas (IS). Katra sistēma palīdz automatizēt kādu noteiktu darījumu sfēru. Tā, piemēram, *EKS* jeb elektroniskā klīringa sistēma paredzēta starpbanku norēķinu nodrošināšanai. Gada laikā sistēmā apstrādāto maksājumu skaits pārsniedz 30'000'000 [**79**]. Kredītu reģistra darbību nodrošina KREG sistēma, kas jau 2008. gada laikā apkalpojusi vairāk nekā 2 500 000 pieprasījumus par reģistra datiem[**80**], taču šī darba tapšanas laikā tās noslodze ir augusi sakarā ar funkcionalitātes paplašināšanu. *LB* ikdienā izmanto vēl daudzas citas IS, kas nodrošina *LB* pamatfunkciju izpildi.

Daudzo IS darbināšanai nepieciešama augstas veiktspējas aparatūra, kas praksē tiek uzturēta kā vairāku atsevišķu serveru klasteris. Dažādu netriviālas funkcionalitātes dalītu sistēmu uzturēšana šādā serveru infrastruktūrā – starp sistēmām vēsturiski izveidojušās

dažādas savstarpējās saites, kas jāsaglabā, mainot programmatūras vai aparatūras konfigurāciju.

Lai pēc iespējas samazinātu uzturēšanai nepieciešamo darbu, izstrādājot organizācijas iekšienē lietošanai paredzētas sistēmas, *LB* tiek izmantota vienota un standartizēta sistēmu arhitektūra. Lielākā daļa *LB* izmantoto darījumu sistēmu ir izstrādātas *LB* iekšienē un tās atbilst pašu definētajiem standartiem un procedūrām.

4.1.2. Pašlaik izmantotie uzturēšanas procesu uzlabojumi

Lai pēc iespējas vienkāršotu uzturēšanas darbu, *LB* pēdējos gados ir papildinājusi sistēmu kvalitātes prasības, dabiskā veidā tiecoties *IBM* autonomo sistēmu [**2**] iniciatīvas definēto mērķu virzienā, tas ir, cenšoties samazināt cilvēka darba ieguldījumu sistēmu uzturēšanā un padarot pašas IS pēc iespējas intelektuālas un spējīgas identificēt izpildes problēmas.

4.1.2.1. Dinamiska izpildes trasēšana

Kā standarta rīks *LB* sistēmās tiek izmantota programmatūras izpildes dinamiskā trasēšana. *LB* realizācijā par trasēšanu sauc programmatūras „ziņošanu" par faktu, ka izpilde nonākusi noteiktā koda vietā (piemēram, funkcijā „Faila parakstīšana ar elektronisko parakstu"). To, vai ziņojums tiek ierakstīts t.s. „trases" failā, nosaka programmatūras izpildes konfigurācijas fails, kurā norādāms, kāda veida un cik detalizētu informāciju ierakstīt.

Darbības izpilde tiek trasēta programmas, komponentes vai koda klases līmenī. Detalizētākais trasēšanas līmenis ir koda funkciju līmenis, kur pieņemts, ka katrai netriviālai koda funkcijai uzsākot un nobeidzot darbu jāveic ieraksti izpildes trasē. Trasē tiek rakstīti arī citi notikumi un fakti, kam ir būtiska jēga programmatūras izpildes izprašanā – piemēram, faila apstrādes sākšana, faila statusa vai nosaukuma maiņa, pieslēgšanās datubāzei u.c. Protams, trasē rakstāmajai informācijai jābūt samērojamai ar *LB* informācijas pieejamības noteikumiem.

Dinamiskas trasēšanas lietošana apstākļos, kad pieejams programmatūras kods, dod iespēju programmatūras atteices gadījumā ātri identificēt „vainīgo" komponenti un veikt problēmas cēloņu diagnostiku.

4.1.2.2. Iebūvētie vides testi

Lai sistēmiski panāktu nepieciešamo datu drošības līmeni un nodrošinātu aprēķinu veikšanas pareizu izpildi, *LB* arvien biežāk izstrādā IS, kur darbs ar datiem faktiski tiek veikts centralizēti uz servera, bet klienta komponentes paredzētas tikai statusa novērošanai un biznesa lēmumu pieņemšanai. Šādā sistēmas arhitektūras modelī lielu daļu sistēmas darba

veic t.s. servisa komponentes – programmatūra, kurai nav lietotāja saskarnes un kuras darbināšanai nav nepieciešama cilvēka līdzdalība (piemēram, nav jābūt aktīvai *MS Windows* sesijai). Kā negatīvas sekas tam, ka servisa komponentes savu darbu veic neatkarīgi no lietotāja klātbūtnes, var minēt faktu, ka programmatūras atteices gadījumā sistēmu uzturētājiem nav viegli pamanīt incidenta iestāšanos.

Praksē novērots, ka servisa komponenšu darbību var traucēt dažādi programmatūras izpildes vides apstākļi, piemēram, neadekvāti reģionālie iestatījumi. Sevišķi nepatīkamas sekas šāda iestatījumu neatbilstība var radīt tad, ja programmatūras kods, kuram neatbilstība rada problēmas, netiek izpildīts uzreiz pēc komponentes iedarbināšanas. Šādā gadījumā ir pilnīgi iespējams, ka „nepieskatītā" komponente ilgstoši nespēj veikt savas funkcijas, bet sistēmu uzturētāji par to nav informēti.

Vadoties no augstākminētajiem apsvērumiem *LB* programmatūras kvalitātes standarts paredz, ka servisa tipa komponentei nevajadzētu uzsākt darbu, ja tā tiek darbināta neadekvātā izpildes vidē. Tādējādi jebkura servisa komponente, uzsākot darbu, izpilda nelielu skaitu pārbaužu, vai vide ir pieņemama. Sarakstā iekļautas atsevišķas tipiskākās reģionālo iestatījumu pārbaudes kā decimālatdalītājs, īsais datuma formāts, valoda un sistēmas kodu tabula. Vides pārbaudes realizētas kā programmatūrā iebūvētie testi, kā tos apraksta [**42**], kā uzturēšanas režīmu izmantojot programmatūras iedarbināšanas brīdi.

Šī darba tapšanas laikā programmatūrā iebūvētās vides pārbaudes tiek izmantotas vairākās *LB* izstrādātajās servisa komponentēs, tomēr novērojams, ka tehnoloģija nav pilnīga – lai papildinātu veicamo pārbaužu sarakstu, jāmaina kods, atkārtoti jākompilē un no jauna uz datoriem jāuzstāda visas sistēmas, kurās nepieciešams izmantot jaunās pārbaudes.

4.1.3. Pārmaiņu nepieciešamība

Iepriekšējā nodaļā minētie risinājumi – dinamiskā programmas izpildes trasēšana un programmatūrā iebūvētie testi – labi veic savas funkcijas un palīdz identificēt problēmu cēloņus brīdī, kad, darbinot programmatūru, problēma jau ir parādījusies. Tomēr, ja nepieciešama augstāka sistēmu uzticamība un paredzamība, sistēmu uzturētājiem pieejamie līdzekļi ne vienmēr ir pietiekami.

To, ka sistēmu darbs ir atkarīgs no to izpildes vides izmaiņām, labi ilustrē sistēmu atteiču analīze. Šādu analīzi veikt iespējams relatīvi viegli, jo visas *LB* sistēmas tiek būvētas tā, lai atteices gadījumā tās apkopotu pēc iespējas detalizētu informāciju par notikušo kļūdu un iegūto aprakstu reģistrētu žurnālā – parasti, vienkārša teksta failā.

Analīzei tika izvēlēta kāda *LB* lietota starpbanku maksājumu apstrādes sistēma, kas darba gaitā apstrādā failus – veic failu atšifrēšanu un šifrēšanu, failu satura saglabāšanu

datubāzē vai nolasīšanu no tās. Sistēmā galveno darbu veic programma, kas realizēta kā *MS Windows* serviss - programma bez grafiskas lietotāja saskarnes, kas tiek darbināta serverī, nevis lietotāja darbstacijā. Lielāko daļu sistēmas veiktā darba serviss kontrolē automātiski, taču atsevišķas operācijas, piemēram, norēķina uzsākšana, ir veicama manuāli. Šādas operācijas bankas darbinieks izpilda, uz sava datora iedarbinot attiecīgu lietojumprogrammu. Tomēr arī šajā gadījumā lietojumprogramma faktiski tikai nedaudz izmaina norēķina stāvokli datubāzē un visu tālāko darbu veic serviss.

Pētot šīs sistēmas failu apstrādes servisa reģistrēto kļūdu žurnālu, tika konstatēts, ka divu darbināšanas gadu laikā žurnālā reģistrēti 486 ieraksti. Žurnalēšanas sistēmas tehnisku īpatnību dēļ viens un tas pats ieraksts var tikt reģistrēts vairākas reizes, tādēļ pēc dublikātu atfiltrēšanas atliek 245 izņēmumsituācijas, ko sistēma reģistrējusi. Lielākā daļa no reģistrētajām izņēmumsituācijām nav unikālas, t.i., tās ar dažādiem laika intervāliem atkārtojas, veidojot piecas atsevišķas izņēmumsituāciju grupas, piemēram – datubāzu servera nepieejamība, failu sistēmas nepieejamība, datu šifrēšanas infrastruktūras nepieejamība utt.

Kaut arī vairākus gadus pēc žurnāla ieraksta izveides ir grūti identificēt katras atteices patiesos iemeslus, kļūdu apraksti labi parāda tās situācijas, kad atteice notikusi programmatūras vai to atbalstošās infrastruktūras nepareizas konfigurācijas dēļ. Tā, piemēram, 2007. gada jūnijā reģistrēta kļūda ar aprakstu '*Cannot generate SSPI context*' (kļūdas iemesls parasti ir nepareiza autentifikācijas sistēmas konfigurācija [**81**]). Šāda kļūda reģistrēta divas dienas pēc kārtas, katru dienu tai atkārtojoties vairākas reizes ar dažu minūšu intervālu – tas liecina, ka sistēmu administratori centušies atrast risinājumu, līdz beidzot otrajā piegājienā tāds ir atradies (šo hipotēzi daļēji apliecina arī sistēmas izpildes trases fails, kurā redzams, ka laikā, kad žurnālā fiksētas kļūdas, attiecīgais *MS Windows* serviss vairākas reizes apturēts un tā darbs atsākts no jauna – šāda rīcība tipiska gadījumos, kad uzturētāji cenšas novērst kļūdas, mainot vides konfigurāciju). Aptuveni gadu vēlāk, 2008. gada aprīlī, atkal novērota šī pati kļūda, taču šajā gadījumā atkārtotu ierakstu par kļūdu nav, bet izpildes trasē fiksēts tikai viens servisa pārstartēšanas piegājiens. Tas ļauj secināt, ka šajā gadījumā sistēmu administratori jau zinājuši, kā novēršama konfigurācijas kļūda.

Līdzīgi iespējams identificēt arī citu apskatītās sistēmas atteiču iemeslus – piemēram, gan 2008. gada augustā, gan 2009. gada janvārī sistēma reģistrējusi problēmu – nav pieejama failu sistēmas taka, kurā tiek meklēti šifrētie faili. Kļūdas iemesls acīmredzami nav tas, ka attiecīgā taka būtu izdzēsta, bet gan tīkla infrastruktūras (piemēram, ugunsmūra konfigurācijas) izmaiņas, kuru rezultātā attālinātais serveris, kurā tiek meklēti faili, nav bijis pieejams.

Kļūdu analīze ļauj secināt – tāpat kā citiem atteiču cēloņiem, arī cilvēka apzināti vai neapzināti izveidotajai sistēmai nepiemērotajai vides konfigurācijai ir tendence ar zināmu laika intervālu atkārtoties. Ja atbildīgo darbinieku zināšanas ir pietiekamas, atkārtotās kļūdas tiek novērstas daudz ātrāk nekā sākotnējās atteices gadījumā, taču būtiski, ka konfigurācijas nepilnības tiek atklātas tikai tad, kad notiek kļūda sistēmā.

Ja datorsistēmā būtu definēts veids, kā tā var patstāvīgi pārbaudīt, vai vide konfigurēta pareizi (vai pieejama sistēmas datubāze, vai pieejamas nepieciešamās takas failu sistēmā u.c.), no šeit aprakstītajām atteicēm būtu bijis iespējams izvairīties. Lai rastu preventīvu risinājumu, kas ļauj nepilnīgo konfigurāciju pamanīt pirms tam, kad iedarbināta darījumu programmatūra, *LB* uzsākta iniciatīva, kas paredz datorsistēmu komponentes papildināt ar dinamiski definētu apkārtējās vides testu veikšanas iespējām, kā tās definētas šī darba iepriekšējās nodaļās. Tas ļaus gan servisa tipa, gan citu veidu programmatūrai pirms darba uzsākšanas veikt vides pārbaudes, turklāt veicamo pārbaužu saraksts būs laika gaitā papildināms.

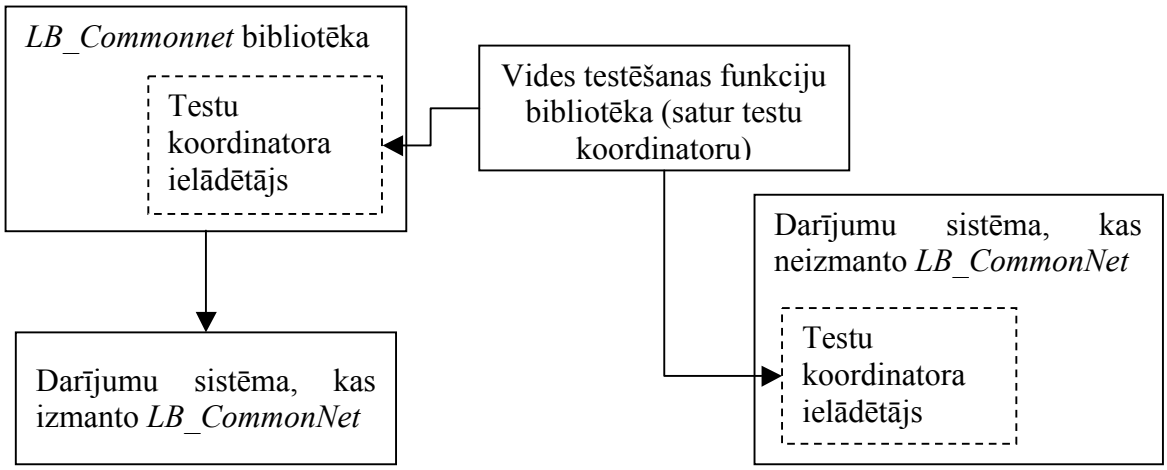
Tādējādi gadījumos, kad sistēmas ekspluatācijā tiek fiksēts jauns veids, kā vides konfigurācija ietekmē programmatūras darbu un ir noformulējams algoritms, kā pārbaudīt šādu programmatūras atkarību, ir jāpapildina veicamo vides pārbaužu saraksts. Praktiski tas nozīmētu, ka, piemēram, brīdī, kad sistēmu administratori pirmo reizi sastopas ar „*SSPI* konteksta" izveides problēmu un secina, ka „vainīgs" ir sistēmas datubāzu serveris, kurā nekorekti konfigurēta lietotāju autentifikācija, datorsistēmu izstrādātāji izveidotu vides pārbaudes moduli, kas programmatiski pārbauda šo konfigurācijas slēdzi. Pēc vides prasību profila papildināšanas, darījumu programmas serviss turpmāk katras ielādes sesijas sākumā vienlaikus ar citām pārbaudēm veiktu arī pārbaudi, vai datubāzu serveris pareizi konfigurēts un vai nepieciešamās datubāzes pieejamas, izmantojot izstrādātāju izveidoto jauno pārbaudes moduli.

4.1.4. Izvēlētais uzlabojumu ieviešanas modelis

Ieviešot vides pārbaudes infrastruktūru, jāņem vērā organizācijā izstrādāto sistēmu īpatnības un jāizvēlas racionālākais metodes ieviešanas veids. *LB* gadījumā šāda „īpatnība" ir programmatūras funkciju bibliotēka *LB_CommonNet*, ko izmanto visas pēdējos gados izstrādātās datorsistēmas. Papildinot šādu koplietošanas bibliotēku ar vides testēšanas koordinatora funkcionalitāti, visas darījumu sistēmas, kas izmanto šo bibliotēku, var sākt veikt vides pārbaudes, izmantojot pieejamo testēšanas koordinatoru.

Praksē tika izvēlēts nedaudz sarežģītāks risinājums, proti, koplietošanas bibliotēkā tika iekļauts tikai vides testēšanas koordinatora ielādei nepieciešamais kods, bet pati testēšanas infrastruktūra izvietota atsevišķā koda bibliotēkā. Šādā veidā tika rasts risinājums, kas ļauj

vides pārbaudes veikt gan tām sistēmām, kas izmanto *LB_CommonNet*, gan tām, kas darbu veic bez šīs koplietošanas bibliotēkas (kā redzams ***4.1****. att.*, tām bibliotēkām, kas izmanto *LB_CommoNet*, automātiski būs pieejama vides testēšanas infrastruktūra un testu koordinators, savukārt pārējās sistēmās jāiekļauj testu koordinatora ielādētājs, kas identisks *LB_CommoNet* iekļautajam)



4.1. att. **Testu koordinatora ielādes veidi dažādās LB izstrādātās sistēmās**

Tādējādi, lai kāda darījumu programma, kas izmanto koplietošanas bibliotēku *LB_CommonNet*, varētu veikt automātiskās vides pārbaudes, tās kods tikai nedaudz jāpapildina, papildus pievienojot vien vides pārbaudes funkcionalitātes izsaukšanu un rezultātu apstrādi (vides testēšanas infrastruktūra tikai veic pārbaudes, lēmumu par to, vai turpināt darbu šādā konfigurācijā, jāpieņem pašai darījumu programma).

```
Dim oTester As New LB.Runtime.SelfTestProxy()
oTester.TextListener = AddressOf Console.WriteLine
oTester.LogLevel = TraceLevel.Verbose
If Not oTester.DoTests() Then
    Application.Exit()
End If
```

4.2. att. **Darījumu programmā iekļaujamā koda piemērs vides testu izpildei**

4.2*. att.* redzams ļoti vienkāršs, taču pilnībā funkcionējošs vides testu ielādes un testu rezultātu apstrādes piemērs. Tā pirmās rindas demonstrē testu ielādes koordinatora ielādētāja objekta izveidi un „konfigurēšanu" (*LB* realizācijā katrs vides testēšanas modulis var sniegt detalizētu informāciju par saviem novērojumiem par apkārtējo vidi, savukārt izsaucēja komponente var izvēlēties, vai un kā saņemt apkopoto informāciju; šo īpašību raksturo *TextListener* un *LogLevel* atribūti). Savukārt koda metode *DoTests()* veic praktiskos vides testus – atrod un ielādē programmatūras izpildes profilu, identificē nepieciešamos vides pārbaudes moduļus, tos iedarbina, saņem apkopo moduļu sniegtās ziņas. *DoTests()* metode

rezultātā sniedz vienkāršu „jā” vai „nē” tipa atbildi, pēc kuras vadoties darījumu programma var izvēlēties, vai turpināt darbu šajā vidē.

Ja darījumu sistēmu izstrādātājiem pieejama šādi sagatavota koplietošanas bibliotēka, tad var uzskatīt, ka galvenais veicamais uzdevums ir korekti sagatavot programmatūras izpildes profila dokumentu un tā versijas, kā arī realizēt vides pārbaudes moduļus tām pārbaudēm, kas atšķiras no iepriekš realizētajām. Lai vēl vairāk vienkāršotu datorsistēmu izstrādātāju darbu, tehnoloģijas aprobācijas laikā tika noskaidroti daži populārākie programmatūras atteiču iemesli un vides testēšanas funkciju bibliotēkā iekļauti pārbaudes moduļi, kas spēj testēt ar šīm atteicēm saistītās izpildes vides nepilnības. Kā tipiskākās vides nepilnības var minēt – konfigurācijas failu nepieejamība, konfigurācijas failos norādīto vērtību kļūdainība, t.sk. failu sistēmas objektu trūkums, reģionālo iestatījumu kļūda.

Šī darba tapšanas laikā vides testēšanas infrastruktūra ir pilnībā iestrādāta koplietošanas bibliotēkā *LB_CommonNet*, un atsevišķi sistēmu projektu izstrādātāji jau uzsākuši pārbaužu iekļaušanu savos projektos, tomēr tehnoloģija vēl nav uzskatāma par pilnībā ieviestu, tādēļ rezultativitātes analīze tiks veikta nākamo pētījumu ietvaros.

4.2. Metodes pielietošana sadalītu sistēmu integritātes pārbaudei

Definējot programmatūras izpildes profilu kā atsevišķu prasību kopumu, kas pārbaudāms, katrai prasībai lietojot vienu vai vairākus pārbaudes rīkus, savulaik ticis uzsvērts [**8**], ka šādu metodi būtu *ļoti noderīgi* pielietot sadalītu sistēmu gadījumā. Apgalvojums izteikts galvenokārt tādēļ, ka sadalītu sistēmu arhitektūra bieži vien ir krietni sarežģītāka nekā identisku “monolītu” sistēmu gadījumā, kas attiecīgi iespaido kļūdu identificēšanas un novēršanas sarežģītību.

Šajā nodaļā aprakstīta metode, kā praksē pielietot programmatūras izpildes profila jēdzienu sadalītas sistēmas gadījumā. Apskatāmā sistēma ir uz *Windows SharePoint Services* (turpmāk tekstā – *SharePoint*) platformas būvēts lietvedības rīks.

Lai būtu vieglāk izprotama vides testēšanas ideoloģijas pielietošana šajā sistēmā, nedaudz plašāk jāapraksta *SharePoint* arhitektūra.

4.2.1. SharePoint platforma

SharePoint ir programmatūra, kas nodrošina tabulāru datu (ar lietotāja definētu struktūru) saglabāšanas iespēju, t.sk. katram tabulas ierakstam piesaistot vienu vai vairākus failus. Lietotājam nodefinējot vēlamo tabulas struktūru (tas ietver kolonnu nosaukumu un tipu noteikšanu, datu lauku izmēra noteikšanu), *SharePoint* automātiski sagatavo arī nepieciešamās ekrāna formas datu ievadei, rediģēšanai, parādīšanai.

Katru lietotāja definēto tabulu jeb sarakstu identificē *URI* shēmā, piemēram, http://serveris/pieteikumi/, bet atsevišķas tabulas rindas tiek identificētas pēc *ID* atribūta, kas ir unikāls atsevišķā saraksta ietvaros.

Ja visi saraksti serverī tiktu definēti vienā līmenī - kā "plakana" struktūra, tie drīz kļūtu nepārskatāmi un grūti izmantojami. Šī iemesla dēļ *SharePoint* sarakstus strukturē atsevišķās *vietnēs*, tādējādi nodrošinot hierarhisku navigējamību (katra vietne var saturēt pēc patikas daudz citas vietnes vai sarakstus). Arī vietni identificē URI shēmā, piemēram "http://serveris/e-rekini/" varētu būt rēķinu apstrādes vietnes nosaukums, savukārt "http://serveris/e-rekini/sanemtie" jau būtu saraksta nosaukums, kur tiek glabāti saņemtie rēķini.

Papildus iespējām konfigurēt datu struktūru, piegādes veidus un formātu (platforma piedāvā dažādus saskarnes risinājumus – tīmekļa lietojumprogramma, tīmekļa servisi, e-pasts, *WebDAV*), *SharePoint* piedāvā arī lietojumprogrammu saskarni (API) papildus funkcionalitātes izstrādei. Pati *SharePoint* programmatūra veidota *Microsoft .Net* vidē un arī izstrādātie papildinājumi jāveido šajā vidē. Piemēram, lai izstrādātu datu ievades ekrāna formu, kas atšķirīga no *SharePoint* automātiski piedāvātās, izstrādātājam jāveido sava *ASPX* lapa, kas nodrošina vēlamo funkcionalitāti.

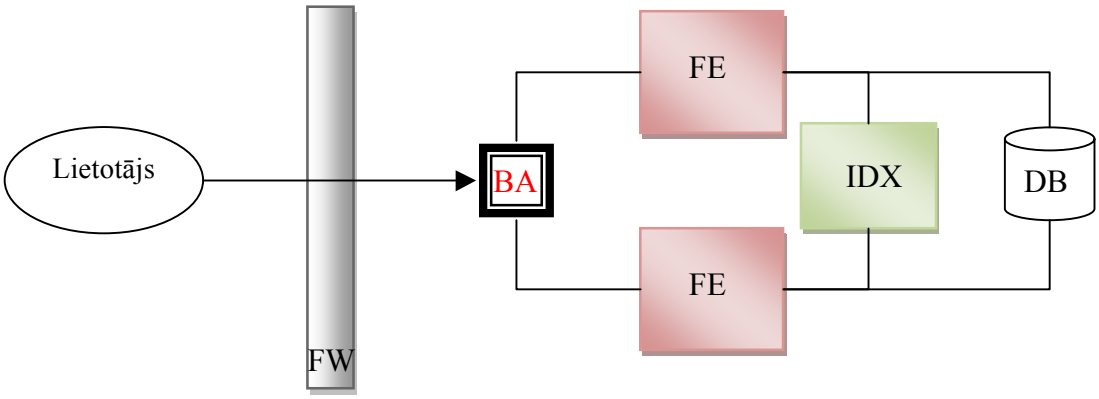
Tāpat platformā iebūvētas iespējas papildināt izpildi ar savu kodu:

- pārķerot dažādus notikumus (sarakstam pievienots jauns ieraksts, saraksta ieraksts mainīts, lietotājs mēģina dzēst ierakstu),
- izmantojot specifisku *XML* shēmu, definēt sarakstu šablonus (piemēram, definēt šablonu "Lietvedības dokuments", kas jau pēc definīcijas satur kolonnu "lietvedības numurs". Izveidojot sarakstu pēc šī šablona, šāda kolonna izveidosies automātiski),
- definējot tīmekļa vietnes lapu izkārtojumu, izmantojot savas *ASPX* lapas un kontroles,
- izmantojot *Windows Workflow Foundation*, definēt darba plūsmas, kas attiecināmas uz SharePoint saraksta elementiem.

4.2.2. SharePoint infrastruktūra

SharePoint platforma paredzēta izmantošanai daudzlietotāju vidē, turklāt viens no tās mērķiem ir panākt relatīvi vieglu mērogojamību. Tādēļ *SharePoint* veidota kā sadalītu lietojumprogrammu platforma. Vienkāršākās instalācijas gadījumā (piemēram, uz izstrādes vides datora) viss darbs tiek veikts uz viena un tā paša servera, taču, ja nepieciešams, jau esošo *SharePoint* instalāciju iespējams pārveidot, darbu sadalot pa vairākiem serveriem, kuri katrs pilda savu funkcionālo lomu. Šādu infrastruktūras konfigurāciju dēvē par serveru *fermu*.

4.3. att. parādīta tipiskas *SharePoint* serveru fermas konfigurācija. Ferma parasti izvietota aiz ugunsmūra (attēlā – *FW*). Saņemto *HTTP* datu plūsmu pārvalda tīkla noslodzes balansa klastera pārvaldnieks, kas tālāk plūsmu novirza uz vienu no „priekšgala" (angl. *front-end*) tīmekļa serveriem (attēlā – *FE*). Lai nodrošinātu iespēju turpināt uzsākto sesiju citā priekšgala serverī, nekā tā uzsākta, visi dati tiek izvietoti centralizēta datubāzu serverī (attēlā - DB). Dažādu papilddarbu (piemēram, meklēšanas servisa indeksa atjaunošanai) veikšanai tiek izmantots atsevišķs serveris (attēlā – *IDX*), kas saņem datus gan no *FE*, gan no *DB* serveriem [**82**].



4.3. att. **Tipiskas SharePoint serveru fermas komponentes**

Mazākā iespējamā *SharePoint* serveru ferma sastāv no četriem serveriem – datubāzu servera, diviem tīmekļa serveriem un fona darbu servera, taču katra tipa serveru skaitu iespējams pēc patikas palielināt.

4.2.3. SharePoint vidē darbināmu lietotņu izstrāde

Jebkura SharePoint videi izstrādāta darījumu sistēma faktiski sastāv no:

- atsevišķām *ASP.Net* formām,
- *DLL* bibliotēkām, kas izmanto *SharePoint API* definēto objektmodeli,
- specifiskā sintaksē aprakstītiem *XML* failiem, kas apraksta risinājumam nepieciešamās datu struktūras,
- citiem izstrādātajam risinājumam nepieciešamiem failiem.

Lai vienkāršotu programmatūras uzstādīšanu uz visiem *SharePoint* fermas serveriem, platforma piedāvā īpašu "instalācijas ietvaru" (angl. *installation framework*). Tā pamatideja ir – visus darījumu sistēmai nepieciešamos failus iekļaut vienā instalācijas pakotnē (arhīva failā) ar strikti definētu struktūru. Šādu pakotni identificē pēc tai piešķirtā *GUID* identifikatora, kas jāsaglabā visā programmatūras dzīves cikla laikā.

Pakotni uzstāda SharePoint vidē, izmantojot speciālu uzturēšanas rīku. Saņemot pakotni, SharePoint instalāciju ietvars to atarhivē un izplata programmatūras jauninājumu uz visiem fermas serveriem – kopē failus, veic nepieciešamās izmaiņas datubāzē un konfigurācijas failos. Tādējādi tiek vienkāršots uzturēšanas darbs, jo sistēmu administratoriem nav jāveic failu kopēšana uz visiem *SharePoint* fermas serveriem, manuāla *DLL* failu pārreģistrēšana, *IIS* konfigurācijas izmaiņas, turklāt *SharePoint* ņem vērā katra servera lomu serveru fermā un uz servera uzstāda tikai tās komponentes, kas tajā nepieciešamas. Piemēram, uz t.s. "fona darbu servera" netiek uzstādītas *ASPX* lapas, jo šis serveris neveic tīmekļa vietņu apkalpošanu.

Uzstādīšanas ietvars paredzēts arī gadījumiem, kad nepieciešams uzstādīt jau agrāk instalētas programmatūras jauninājumus. Lai izmantotu šo iespēju, atkārtoti jāsagatavo šīs pašas programmas uzstādīšanas pakotne, tajā iekļaujot jauno programmas versiju (visus nepieciešamos failus). Saņemot šādu pakotni, *SharePoint* to salīdzina ar pašlaik uzstādīto pakotnes versiju un veic nepieciešamās izmaiņas – atjaunina uz visiem serveriem tos failus, kuru saturs ir mainījies, kopē jaunos failus, dzēš tos failus, kas atrodami iepriekšējā pakotnes versijā, bet nav atrodami jaunajā.

4.2.4. Programmatūras uzstādīšanas principi apskatāmajā uzņēmumā

Jebkura programmatūra, t.sk. *SharePoint* videi paredzētās programmas, Latvijas Bankā tiek izstrādāti izstrādes vidē, kas ir pilnībā nodalīta no ekspluatācijas vides. Izstrādes gaitā visas programmatūras koda versijas tiek glabātas *Microsoft Visual SourceSafe* repozitorija atsevišķā zarā.

Lai uzstādītu programmatūru testēšanas vidē, tā tiek manuāli kopēta uz citu *Visual SourceSafe* repozitorija zaru – testēšanas zaru. Lai programmatūru uzstādītu, kvalitātes *kontrolieris* sagatavoto koda versiju lejupielādē no koda repozitorija, veic programmatūras kompilēšanu ar izstrādātāju sagatavotu kompilācijas skriptu un iegūto kodu (*SharePoint* gadījumā – *WSP* pakotni) uzstāda testēšanas vidē.

Līdzīgi, kad iegūta stabila versija, kas pārbaudīta testēšanas vidē, kods no *SourceSafe* repozitorija testēšanas zara tiek manuāli pārcelts uz ekspluatācijas vides zaru. Šajā gadījumā programmatūru uz ekspluatācijas serveriem uzstāda sistēmu administratori, bet sistēmu kvalitātes kontrolieriem vai datorsistēmu izstrādātājiem nav tiešas piekļuves ekspluatācijas vides serveros esošajiem datiem.

4.2.5. SharePoint bāzētu sistēmu uzstādīšanas problēmas

SharePoint izmantotā programmatūras uzstādīšanas shēma sākotnēji šķiet ļoti vienkārša, sevišķi, ja ņem vērā, ka koda uzstādīšana uz visiem fermas serveriem tiek veikta

automātiski, izmantojot platformā iekļauto rīku. Tomēr prakse parāda, ka uzstādīšanas procesā mēdz notikt kļūdas, kuru iemesls var būt gan cilvēciskais faktors, gan uzstādīšanas procesā izmantotās programmatūras īpatnības. Nākamajās rindkopās apskatītas populārākās identificētās uzstādīšanas kļūdas.

Faila kodējums. Pārceļot *ASPX* lapu no viena *SourceSafe* repozitorija zara uz citu zaru, ir "sabojājies" lapas kodējums. Sākotnēji visas lapas tiek sagatavotas *UTF-8* kodējumā, taču *SourceSafe* to reizēm interpretē kā *ASCII*, tādējādi tas tiek uz diska saglabāts citādi nekā bija sākumā.

„Read-only" atribūts. Paņemot *ASPX* lapu no *SourceSafe* repozitorija, tai nav noņemts "*Read Only*" atribūts. Uzstādot programmatūru pirmo reizi, viss tiek uzstādīts bez problēmām. Nākamajā reizē uzstādot programmatūras jauninājumus, SharePoint vairs nespēj pārrakstīt ar „*Read Only*" aizsargāto failu.

Trūkstoši faili. Kāda no *ASPX* lapām nav pārkopēta uz ekspluatācijas vides zaru *SourceSafe* repozitorijā. Tās trūkums nav pamanāms kompilācijas laikā, jo *ASP.Net* lapas netiek iekļautas koda asamblejas failā. Funkcionalitāte, ko pilda attiecīgā lapa, nav pieejama pēc versijas uzstādīšanas;

Novecojušas DLL versijas, Gadījumos, kad instalēšanas pakotne sastāv no vairākām savstarpēji saistītām *DLL* bibliotēkām (piemēram, bibliotēka "billing.dll" izmanto bibliotēku "common.dll"), kāda no bibliotēkām jaunās versijas instalācijas pakotnē iekļauta no iepriekšējās versijas. Ticams, ka kompilēšanas laikā bibliotēku atkarības tikušas apstrādātas korekti, taču, veidojot instalācija pakotni, tajā iekļauta agrāk kompilēta versija, kas saglabājusies uz datora. Tādējādi programmatūras jaunā versija joprojām izmanto daļu koda no iepriekšējās koda bibliotēkas, kura, iespējams, nesatur kādus būtiskus labojumus.

Versiju numerācija. *SharePoint* iebūvētie darbplūsmu pārvaldības risinājumi, lai ilglaicīgi saglabātu ziņas par darba plūsmas stāvokli, izmanto t.s. *objektu serializēšanas* metodi. Proti, pēc tam, kad notikusi lietotāja veikta mijiedarbība ar sistēmu un lietojuma sesija beigusies, sistēmas atmiņā esošie darbplūsmas stāvokli aprakstošie objekti tiek pārveidoti bināru datu masīvā un saglabāti datubāzē. Brīdī, kad lietotājs vēlas uzsākt nākamo mijiedarbības sesiju, darba plūsmu aprakstošie dati tiek *deserializēti*, atmiņā no datubāzes atjaunojot to objektu stāvokli, kāds bijis iepriekšējā lietojuma sesijā. Šāda pieeja sniedz iespēju vienas un tās pašas ilglaicīgas darba plūsmas atsevišķas sesijas apstrādāt uz dažādiem *SharePoint* fermas serveriem.

Stāvokļa atjaunošanai *SharePoint* vide izmanto *.Net* koda refleksiju, katru koda asambleju identificējot pēc tās pilnā nosaukuma, bet objektu klases – pēc to vārdkopas un nosaukuma. Lai pilnvērtīgi atjaunotu objektus, *SharePoint* videi nepieciešama tieši tās

versijas bibliotēka, kāda bijusi objekta serializācijas brīdī. Viegli noprotams, ka rezultātā jebkurai bibliotēkai, kas izmantota *SharePoint* vidē, nedrīkst mainīt versijas numuru laikā, kad tā jau tiek izmantota ekspluatācijā, jo šādas darbības rezultātā varētu panākt, ka datubāzē saglabātās darba plūsmas nav iespējams turpināt.

Tā rezultātā šādās *SharePoint* paredzētās lietotnēs nav iespējams lietot *Microsoft*.Net standarta versiju numurēšanas nomenklatūru un tādējādi kļūst sarežģīti noskaidrot, vai kādā no serveriem nav nonākusi novecojusi bibliotēkas versija – faktiski atšķiras tikai failu izveides vai modifikācijas laiks.

Lai risinātu augstākminēto problēmu, aprakstāmajā uzņēmumā pieņemts *SharePoint* paredzētu bibliotēku versijas numuru nemainīt (visā dzīves laikā izmanto versiju 1.0.0.0), bet tā vietā izmantot citu .Net asamblejām pielietojamu atribūtu – *FileVersion*. Šis atribūts sintaktiski ir identisks *AssemblyVersion* atribūtam, taču tam ir tikai informatīva nozīme un tas neietekmē *Mictrosoft .Net Framework* bibliotēku ielādes mehānismu.

Bibliotēkas atsauces noņemšana. Sagatavojot instalācijas pakotnes sākotnējo versiju kādai darījumu sistēmai, piemēram, Sys1, tajā bijusi iekļauta atsauce uz koda bibliotēku bibliotēka Bib1. Nākamajā programmatūras versijā pamanīts, ka atsauce uz šo bibliotēku ir lieka un tā no pakotnes izņemta. Tā kā SharePoint instalēšanas ietvars salīdzina pašreizējo versiju un no jauna uzstādāmo, tas "pamanīs", ka bibliotēka Bib1 vairs nav nepieciešama un to noņems no visiem fermas serveriem.

Rezultāts – ja šī pati bibliotēka Bib1 bijusi nepieciešama ne tikai sistēmai Sys1, bet arī kādai citai darījumu sistēmai Sys2, tad pēc Sys1 jauninājumu uzstādīšanas Sys2 nespēs pilnvērtīgi darboties (nebūs pieejama nepieciešamā bibliotēka).

SQL izmaiņas. Atjaunotā koda versija modificē datubāzes saturu vai struktūru, piemēram, pievienojot kādu jaunu *SQL* saglabāto procedūru. Dažādu iemeslu dēļ attiecīgā *SQL* procedūra nav uzstādīta mērķa vides *SQL* serverī, veicot SharePoint programmatūras atjaunināšanu (SQL izmaiņas veicamas atsevišķi, neizmantojot SharePoint instalēšanas ietvaru).

4.2.6. Programmatūras izpildes profila lietošana aprakstīto problēmu risināšanai

Pielietojot programmatūras izpildes profila dokumentācijas ideoloģiju, iespējams automātiski "pamanīt" visas iepriekšējā nodaļā minētās problēmas.

Failu kodējuma neatbilstība. Ja, failu paņemot no SourceSafe repozitorija, tam ticis mainīts kodējums, fails nebūs identisks oriģinālam, t.i., salīdzinot tos kā baitu virknes, būs novērojamas atšķirības.

Šī tipa problēmu risināšanai populāri ir izmantot kriptogrāfiskās jaucējfunkcijas kā,

piemēram, MD5. Programmatūras izpildes profilā norādot, kādai MD5 vērtībai fails atbilst, to var viegli pārbaudīt mērķa vidē.

"*Read-only*" atribūta pārbaudi iespējams viegli veikt ar failu sistēmas API līdzekļiem. Pēc versijas uzstādīšanas pārbaudot, vai nepieciešamajiem failiem noņemts "*read-only*" atribūts, iespējams nodrošināties pret kopēšanas problēmām nākamajā koda uzstādīšanas reizē. Protams, pārbaudi iespējams veikt arī pirms jaunā koda kopēšanas.

Līdzīgi kā "*read-only*" problēmu, arī **failu trūkumu** iespējams pārbaudīt ar failu sistēmas API palīdzību. Pēc programmatūras versijas uzstādīšanas jāveic pārbaudes, vai nepieciešamie faili ir parādījušies failu sistēmas takās, kuras ir iepriekš zināmas.

Novecojušas DLL versijas iespējams pārbaudīt, aprakstot apkārtējās vides prasību, ka jābūt pieejamai bibliotēkai ar noteiktu identitāti, kurai turklāt norādīts "*FileVersion*" atribūts atbilstoši prasītajam.

Līdzīgi kā novecojušo DLL versiju pārbaude, arī **DLL bibliotēku pieejamību** iespējams aprakstīt kā prasību attiecībā pret izpildes vidi – servera GAC mapē jābūt reģistrētai noteiktai DLL bibliotēkai.

SQL izmaiņas iespējams kontrolēt, izmantojot SQL prasību, kas, izmantojot datu definēšanas valodas vaicājumus, pārbauda, vai datubāzē ir noteikts objekts (piemēram, *select * from sysobjects where ...*).

Tādējādi prasības, kas iekļaujamas SharePoint vidē darbināmas programmatūras izpildes prasību profilā, nav būtiski sarežģītākas par vienkāršu sistēmu izpildes prasībām. Iepriekšminētie problēmu tipi apkopoti ***4.1***. tabulā.

4.1. tabula

SharePoint instalācijās novēroto problēmu apkopojums

Prasība vārdiem	Prasība / pārbaudes modulis	Pārbaudes moduļa parametri
Jābūt pieejamai bibliotēkai Commn.DLL, tai jābūt instalētai Windows GAC	AssemblyDependency	**FullyQualifiedName***:* OurCommonLibrary,version=1.0.0.0, publicKeyToken=e3a944d90afb52, culture=neutral **DeployedInGAC** true **FileVersion** 1.0.2.18

Failam "ContactPicker.aspx" jābūt pieejamam	FileDependency	**Path** \program files\common files*Microsoft* shared\web server extensions\12\template\layouts\contactpicker.aspx
Failam "ContactPicker.aspx" jāsakrīt ar izstrādes vidē esošo	FileHashDependency	**Path** \program files\common files*Microsoft* shared\web server extensions\12\template\layouts\contactpicker.aspx **MD5hash** 098eb8ba2cc924fad0ec05acd869a4eb
Failam "ContactPicker.aspx" nedrīkst būt uzstādīts "ReadOnly" atribūts	FileAttributeDependency	**Attribute** ReadOnly **Value** False **Path** \program files\common files*Microsoft* shared\web server extensions\12\template\layouts\contactpicker.aspx
Sistēmas datubāzē jābūt SQL procedūrai GetCurrencyRateForDate	SQLDependency	**Server** localhost **Database** Northwind **RequestedObjectType** Procedure **RequestedObjectName** GetCurrencyRateForDate

4.2.7. Vides pārbaužu pielietošana serveru fermā

Programmatūra, kas instalēta serveru fermā, atšķirībā no monolītās programmatūras nav tieši identificējama ar atsevišķu datoru, uz kura tā tiek izpildīta – atsevišķas programmas daļas tiek instalētas katrā serveru fermas serverī. Šādā konfigurācijā identificējamas divas nosacīti nodalītas izpildes vides – katra fermas servera lokālā izpildes vide un kopīgā uzņēmuma datortīkla infrastruktūra. Uz lokālo vidi attiecināma tā prasību daļa, kas attiecas uz lokāliem resursiem – piemēram, vai fails ir iekopēts tur, kur nepieciešams, vai bibliotēka piereģistrēta serverī. Savukārt uz uzņēmuma infrastruktūru attiecas tās prasības, kas nav konkrētam serverim specifiskas, ārpus serveru fermas jēdzieniem pastāvošas. Piemēram, visi serveri lieto vienu un to pašu *SQL* datubāzi, vienu un to pašu *SMTP* serveri vai kopīgu tīkla mapi, kurā

iekopēt failus. Tādējādi prasības attiecībā uz koplietošanas resursiem ne vienmēr nepieciešams pārbaudīt uz katra no serveriem.

Nodaļā "4.2.2 SharePoint infrastruktūra" aprakstītā fermas konfigurācija paredz vismaz divus dažādus *SharePoint* fermas serveru veidus jeb serveru *lomas* – tīmekļa serverus un fona darbu serverus.

Tīmekļa serveris. Nelielās serveru fermās ir vismaz divi tīmekļa serveri (piemēram, FE1 un FE2), kas tiek darbināti t.s. "tīkla noslodzes balansēšanas" režīmā. Tas ir, šo serveru konfigurācija ir identiska, tie tīkla līmenī specifiski konfigurēti, lai abi apkalpotu to pašu IP adresi un uz tiem iedarbojas viens un tas pats DNS ieraksts (piemēram, "portal"). Tādējādi, no klienta darbstacijas veicot pieprasījumu uz šo tīkla adresi (attiecīgi - http://portal), nav iespējams paredzēt, kurš no serveriem izpildīs pieprasījumu. Katrā no tīmekļa serveriem instalētas ari apskatāmajā organizācijā izstrādātas komponentes, turklāt ir būtiski, ka komponenšu konfigurācija ir identiska. Ja šis nosacījums netiktu ievērots, pastāvētu iespēja, ka, lietotājam veicot nākamo tīmekļa lapas pieprasījumu, to izpildītu otrs tīmekļa serveris, kas varētu nekorekti interpretēt pašreizējo programmatūras stāvokli;

Fona darbu serveris paredzēts dažādu "fona darbu" izpildīšanai – meklēšanas dzinēja indeksu atjaunošanai, e-pasta atgādinājumu izsūtīšanai u.c. Uz šī servera instalētas dažas apskatāmajā uzņēmumā izstrādātas komponentes, taču tā darbība nav saistīta ar tīmekļa pieprasījumu apkalpošanu.

Lai novērtētu kopējo programmatūras stāvokli, tiek piedāvāts risinājums, ka pārbaudes darbu uzņemas un koordinē viens no fermas serveriem. Ņemot vērā, ka fona darbu serveris ir unikāls fermā, koordinēšanai tas ir vispiemērotākais. Tādējādi, uz šī servera izpildot pārbaudes, kas attiecas uz kopīgo uzņēmuma infrastruktūru, tās tiktu izpildītas tikai vienu reizi.

Koordinējošajā serverī programmatūras izpildes profils arī definē specifisku, līdz šim neapskatītu prasību – atkarību no programmatūras izpildes vides pārbaudes rezultātiem citos serveros. Tas ir, šī prasība būtu formulējama kā izteikums "Serveru ferma spēj korekti darboties tikai tad, ja serveri FE1 un FE2 darbojas korekti".

Lai noskaidrotu, vai uz serveriem FE1 un FE2 prasības ir apmierinātas, jāpanāk, ka arī uz tiem tiek izpildītas prasību pārbaudes. Šī darba tapšanas laikā kā piemērotākais un SharePoint infrastruktūrā iederīgākais risinājums izvēlētas *ASP.Net SOAP* tīmekļa pakalpes, kas uzstādītas uz katra tīmekļa servera atsevišķi. Piekļuve šai tīmekļa pakalpei realizēta nevis pa tīkla noslodzes balansēšanai paredzēto adresi http://portal, bet katram serverim specifiskā adresē, piemēram, http://fe1/dependencyservice.asmx Izmantojot *SOAP* pieprasījumus šai

tīmekļa pakalpei, ir iespējams pieprasīt veikt vides pārbaudi attiecīgajā serverī. Izpildes gaitā tiek pārbaudīta attiecīgā tīmekļa servera lokālo resursu konfigurācija un rezultāti (pārbaude veiksmīga vai nav veiksmīga) tiek nodoti pakalpes izsaucējam.

Rezultātā, uz fermas serveriem veidojas šādas prasību failu konfigurācijas (piemērs):

- Fona darbu serveris
    - Uzņēmuma infrastruktūras resursu pārbaudes
        - vai uzstādīta SQL procedūra X,
        - vai ir pieejama tīkla mape Y;
    - Fona darbu servera lokālās pārbaudes
        - vai ir instalēts *Windows* serviss, kas izsūta atgādinājuma e-pasta vēstules;
    - Pārējo serveru pārbaudes
        - Vai serveris FE1 apstiprina saderību ar versiju 1.0.0.3,
        - Vai serveris FE2 apstiprina saderību ar versiju 1.0.0.3;
- Tīmekļa serveri FE1, FE2
    - Vai ir pieejama ASPX lapa A,
    - Vai ir reģistrēta DLL bibliotēkas B versija C ar FileVersion D.

Izmantojot pieeju, ka fona darbu serveris ierosina pārbaudes uz pārējiem serveru fermas serveriem, iespējams iegūt kopējo programmatūras izpildes vides novērtējumu, kas sastāv gan no infrastruktūras resursu, gan katra atsevišķa servera vides novērtējumiem.

4.2.8. Turpmākā izpēte

Attīstot šeit aprakstīto tehnoloģiju, jāizstrādā praktiski risinājumi, kas vienkāršotu programmatūras izpildes vides prasību dokumenta sagatavošanu versijas komplektēšanas laikā. Risinājumam jānodrošina arī dažādu izpildes profilu veidošana dažādām serveru lomām.

4.3. Vides testu pielietošana problēmu ziņotāja programmatūrā

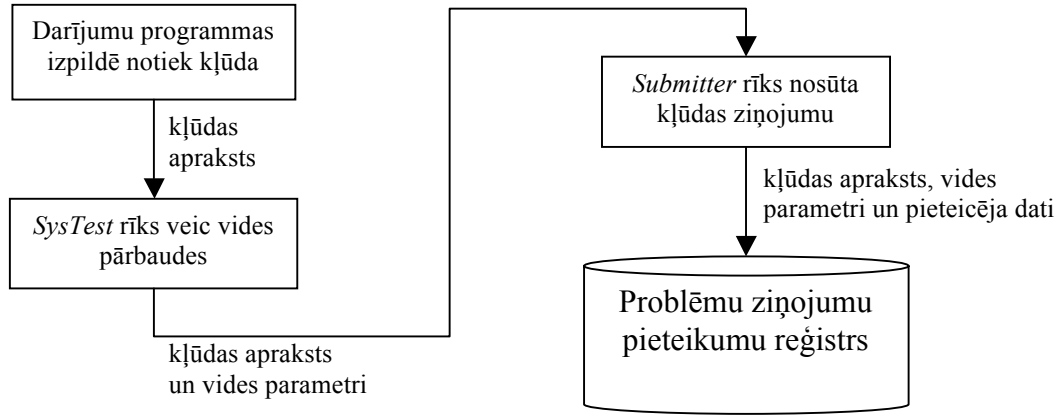
Programmatūras izpildes vides testēšanas metodes praksē pielietotas arī Latvijas programmatūras izstrādes organizācijā „Datorikas institūts DIVI" (*DIVI*). Šajā gadījumā tehnoloģija ieviesta nedaudz atšķirīgi no iepriekšējās nodaļās aprakstītā lietojuma – pati darījumu programmatūra netiek izstrādāta citādi, nekā pirms metodes aprobācijas un vides pārbaudes tā neveic, taču vides pārbaudes funkcionalitāte iekļauta papildu rīkā *SysTest*, kas tiek pievienots darījumu sistēmām.

DIVI izstrādātāji piedāvā rīku izmantot divos dažādos scenārijos, kas parādīti ***4.4****. att.* un ***4.5*** *att.* un detalizētāk aprakstīti nākamajās nodaļās.

4.3.1. SysTest rīka lietošanas scenāriji

4.3.1.1. Vides informācijas apkopošana kļūdu notikumu apstrādē

Pirmais lietojums (***4.4****. att.*) paredz darījumu programmas darbināšanu, ikdienas darbā neveicot nekādas vides pārbaudes. Tikai gadījumos, kad notikusi kļūda darījumu programmas izpildē, tā automātiski ielādē *SysTest* rīku, kas veic detalizētas apkārtējās vides pārbaudes un tad nodod apkopotos datus citam rīkam – *Submitter*, kas paredzēts kļūdas apraksta reģistrēšanai centralizētā kļūdu pieteikumu reģistrā. Pirms kļūdas apraksta reģistrēšanas *Submitter* rīks to papildina ar *SysTest* iegūtajiem izpildes vides konfigurācijas aprakstiem.



4.4. att. **SysTest lietošanas scenārijs, lai papildinātu kļūdu pieteikumu aprakstus**

Šāds *SysTest* pielietojuma veids radies vēsturiski, organizācijas iekšienē tiecoties uzlabot lietotāju atbalsta procesus. Agrāk ticis novērots, ka atsevišķu darījuma sistēmu lietotāju pieteiktās kļūdas bieži vien ir identiskas un to cēloņi ne obligāti ir saistīti ar kļūdu pašā darījumu sistēmā, bet gan ar izpildes vides nepilnīgu komplektāciju vai nepareizu konfigurāciju (piemēram, datorā vairs nav instalēta *MS Excel* programma, kaut arī sistēmas instalācijas laikā pārbaudīts, ka nepieciešamā programmatūra ir uzstādīta).

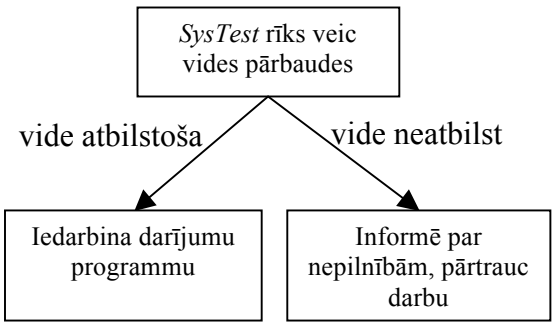
Ja, lietotājam uzsākot problēmu ziņojuma pieteikšanu, tiek veiktas papildu vides pārbaudes, ir iespējams izvairīties no šāda veida ziņojumu liekas reģistrēšanas pieteikumu reģistrā un problēmu atrisināt, pirms par to ziņots izstrādātājiem. *SysTest* rīks viegli uztveramā veidā parāda, kādas pārbaudes vidē tikušas veiktas, kādi bijuši to sagaidāmie rezultāti un kāds ir faktiskais stāvoklis (piemēram – veikta pārbaude „*Microsoft Excel* versija", vēlamais rezultāts: *Excel 2000* vai augstāka, faktiskais rezultāts – programma nav instalēta), turklāt rezultātu parādīšanai lietotas grafiskas līmeņa indikācijas zīmes – „viss kārtībā", „brīdinājums", „uzmanību!".

Lietojot šādu tehnoloģiju, tiek sasniegti divi mērķi – problēmu ziņojumu pieteikumu reģistrā tiek reģistrētas tikai tās problēmas, ko nav iespējams risināt ar lokāli pieejamajiem

resursiem, savukārt tie problēmu pieteikumi, kas nokļūst problēmu ziņojumu reģistrā, satur detalizētu informāciju par datoru, kurā kļūda notikusi.

4.3.1.2. Vides pārbaude pirms darījumu programmas iedarbināšanas

Savukārt *SysTest* rīka autoru piedāvātais risinājums piedāvā alternatīvu *SysTest* rīka lietojumu – vides pārbaudes rīks tiek iedarbināts pirms darījumu programmas palaišanas un tikai tajā gadījumā, ja nav novērojams, ka vide ir nepilnīgi komplektēta vai neatbilstoši konfigurēta, tiek iedarbināta darījumu programmatūra (skat. ***4.5*** *att.*).



4.5 att. **SysTest rīka lietošana vides pārbaudei pirms programmatūras iedarbināšanas**

Šajā risinājumā rīka autori paredz, ka *SysTest* programmas grafiskā lietotāja saskarne tiek izmantota tikai gadījumā, ja novērotas vides neatbilstība, bet normālajā lietojuma gadījumā rīks tiek izpildīts pilnībā bez grafiskas saskarnes parādīšanas. Tādējādi *SysTest* tiek izmantots kā universāls programmu „palaidējs" – kā ieejas parametru tas saņem nepieciešamās darījumu programmas nosaukumu un var aizturēt tās izpildi gadījumā, ja programmas izpilde potenciāli apdraudēta.

Šāda „palaidēja" ideoloģija labi sader arī ar [**35**] aprakstītā *IS Technology* rīka principiem, kur tā kļūst par vienu no faktoriem, kādēļ programmatūru ir vai nav pieļaujams iedarbināt attiecīgā laika momentā.

4.3.2. SysTest rīks kā vides pārbaudes tehnoloģijas realizācija

Atšķirībā no šī darba agrākajās nodaļās paustās nostādnes, ka darījumu programmatūrai izpildes vides testi jāveic patstāvīgi, *SysTest* rīks ir universāls rīks, kas pārbaudes veic ārpus darījumu sistēmas. Šādā veidā tiek panākts mērķa organizācijai būtisks efekts – ir iegūts programmatūras modulis, kas derīgs kā universāls papildinājums lielākajai daļai organizācijā izstrādāto darījumu sistēmu un kura darbība ir pilnībā vadāma, mainot rīka konfigurāciju. Pat gadījumā, ja vienas un tās pašas darījumu sistēmas dažādas variācijas tiek vienlaikus darbinātas dažādu pasūtītāju vidēs, nepieciešams tikai katrai variācijai izstrādāt atsevišķu programmatūras izpildes profilu, ko *SysTest* izmanto vides pārbaudei.

Jāatzīmē, ka šeit aprakstītais risinājums, kaut arī tas padara vides pārbaudes par vienkāršu, jau gatavai sistēmai pielāgojamu sistēmas darbināšanas aspektu, zaudē daļu vides testēšanas tehnoloģijas iespēju ar to, ka vides pārbaudes tiek izpildītas pilnībā nodalīti no pašas darījumu sistēmas, nevis darījumu programmatūras procesā (t.i., *SysTest* tiek izpildīts kā atsevišķs operētājsistēmas process):

- lietojot *SysTest* rīku kā universālo programmatūras „palaidēju", tiek ierobežots tās programmatūras loks, kam rīks izmantojams, proti, to nav iespējams lietot gadījumos, kad izpildāmā programmatūra netiek iedarbināta ar tieša veida komandrindas izsaukumu palīdzību, tādējādi šo metodi nav iespējams pielietot, piemēram, ilglaicīgi darbināmām servisa tipa lietotnēm, tīmekļa lapām vai tīmekļa pakalpēm;
- nav garantēta to prasību adekvāta pārbaude, kur ar pārbaudāmo resursu saistītās politikas var būt balstītas uz izpildāmā moduļa nosaukuma (piemēram, pārbaudot tīkla savienojumu, var gadīties, ka ugunsmūrī tīkla piekļuve organizēta, tiesības piešķirot *.exe failam, kas atrodas noteiktā failu sistēmas takā).

Šeit aprakstītie ierobežojumi tomēr nav uztverami par būtiskiem tajos lietojuma scenārijos, kādu paredz rīka autori. Decentralizētās sistēmās, kas arī ir galvenais šī rīka paredzētais lietojums, lielākā daļa lietotāju darbina darījumu programmatūru uz savām darbstacijām, tādēļ tieši šo datoru konfigurāciju arī ir būtiski pārbaudīt. Savukārt serveru skaits sistēmā, uz kuriem tiek darbināti ilglaicīgi darbināmi servisi vai tīmekļa lapas, ir relatīvi neliels, bet to konfigurācija – relatīvi statiska.

Tādējādi, darījumu sistēmai pievienojot *SysTest* rīku vides informācijas apkopošanai un *Submitter* rīku problēmu ziņojumu pieteikšanai, tiek izveidota daļa no [**5**] aprakstītā viedās programmatūras karkasa, kur darījumu programmatūra tiek atslogota no nefunkcionālo lomu izpildes, šīs īpašības saņemot kā viedās programmatūras karkasa automātiski sniegtus servisus.

4.3.3. Vides testētāja ieviešana uzņēmumā

Rīka ieviešana „Datorikas institūtā DIVI" notikusi pakāpeniski – sākotnēji izstrādājot *SysTest* rīka bāzes prototipu [**17**], lai pārliecinātos par tehnoloģijas iespējamo pielietojumu vides pārbaudēm. Izstrādātais prototips nav bijis uzreiz ieviešams ražošanā, jo vides pārbaužu rīkam bijusi nepieciešama instalācija – šāda prasība nav saderīga ar "vieglā rīka" pieeju.

Tomēr jau prototipa fāzē *SysTest* rīks arhitektūra veidota pēc principiem, kas atbilst šī darba 3.2. nodaļā aprakstītajiem, atšķiroties vien ar to, ka lietojumprogrammas vietā darbojas *SysTest* rīks, kas arī uzņemas testu koordinatora lomu. Izstrādājot šādu prototipu, rīka autori faktiski paveikuši būtisku daļu no ***3.2***. tabulā minētajiem darba uzdevumiem – definēta

prasību aprakstīšanas valoda, specificēti moduļu dinamiskās ielādes principi, izstrādāti testēšanas infrastruktūras galvenie moduļi.

Turpinot *SysTest* rīka ieviešanu, rīka autori veikuši anketēšanu un pārrunas ar citiem vadošajiem sistēmu izstrādātājiem [**18**], lai identificētu tālākās ieviešanas stratēģiju un noskaidrotu, kādas ir visbiežāk nepieciešamās vides pārbaudes. Anketēšanā noskaidrojies, ka *SysTest* rīkam jāatbalsta daudz plašāks operētājsistēmu loks (dažādas MS Windows versijas, sākot ar *Windows 98* līdz *Windows Vista*), nekā sākotnēji paredzēts, savukārt nav pieļaujams izmantot sākotnēji iecerēto platformu – *Microsoft .Net 2.0*, jo viena no nepieciešamajām veicamajām pārbaudēm ir – pārliecināties, vai datorā pieejama nepieciešamā *Microsoft .Net* versija.

Tālākā *SysTest* ieviešana atkarīga no attiecīgo darījumu sistēmu projektu vadītāju iniciatīvas, sagatavojot attiecīgajai sistēmai nepieciešamo prasību dokumentu profilus. Šī darba tapšanas laikā *SysTest* rīks pielietots vienai darījumu sistēmai – finanšu un budžeta pārskatu sistēmai FIBU, kas uzstādīta daudzu dažādu pasūtītāju datoros un kurā vidēji mēneša laikā tiek reģistrēti gandrīz 70 problēmu ziņojumu pieteikumi.

REZULTĀTI

Promocijas darbā ir definēta jauna programmatūras būves tehnoloģija, kas ļauj vieglāk veidot pašaizsargātu programmatūru, kas aizsardzību pret izpildes vides neadekvātu konfigurāciju saņem kā vienu no *viedo tehnoloģiju* ietvara servisiem. Izmantojot šo pieeju, pirms katras programmatūras darba sesijas iespējams veikt pārbaudi, vai pašreizējā izpildes vides konfigurācija ir piemērota normālam darbam un, ja rezultāti nav apmierinoši, programmatūra darba sesiju nemaz neuzsāk. Vides pārbaudes tiek veiktas, izmantojot testēšanas rīkus, kas par kādu izpildes vides raksturlielumu empīriski noskaidro, vai tas atbilst sagaidāmajam. Šādu testēšanas rīku izstrāde tiek veikta neatkarīgi no darījumu programmatūras izstrādes.

Izpildes vides testēšana tiek vadīta, izmantojot izpildes prasību aprakstīšanas valodu. Prasību valodā izstrādāts dokuments – programmatūras izpildes profils – tiek pievienots programmatūras izpildāmajiem moduļiem un seko tiem visā programmatūras dzīves cikla laikā. Šāds profila dokuments uztverams kā datoram lasāma vides pārbaudes instrukcija.

Galvenās šīs metodes priekšrocības, salīdzinot to ar programmatūrā iebūvētu testu metodi, ir:

- vides testus iespējams veikt neatkarīgi no pašas programmatūras, neskarot darījumu funkcionalitāti,
- izpildes vidi iespējams testēt jebkurā programmatūras dzīves cikla brīdī, piemēram, katru reizi, kad tiek uzsākta jauna darba sesija,
- pamanot līdz šim nezināmas prasības, tās viegli pievienot aprakstam un izstrādāt pārbaudes rīkus, nemainot pašu darījumu programmatūru,
- jaunu prasību atklāšana un automātisku testu izstrāde iespējama jebkurā programmatūras dzīves cikla fāzē,
- nav nepieciešams vides pārbaudes funkcionalitāti iestrādāt darījumu programmatūrā,
- apraksts precīzi dokumentē programmatūras nefunkcionālos aspektus, kas citos gadījumos var nebūt aprakstīti, tas noderīgs arī datorsistēmu uzturētājiem konfigurācijas maiņas gadījumos,
- izstrādātos vides pārbaudes rīkus iespējams atkārtoti izmantot citu programmatūras sistēmu būvē.

Darbā definētās metodes pielietošana vienkāršo sarežģītu datorsistēmu ilglaicīgu uzturēšanu nehomogēnā vidē, kur datorsistēmu infrastruktūras resursus koplieto vairākas informācijas sistēmas vai starp informācijas sistēmām pastāv savstarpējas atkarības. Promocijas darbā aprakstītas metodes pielietošana praksē ļauj samazināt atkarību no personāla zināšanām par sistēmu uzbūvi.

Promocijas darbā definēti izpildes prasību valodas izstrādes principi un doti konkrēti valodas realizāciju piemēri. Darba gaitā izstrādāti vairāki programmatūras izpildes vides testēšanas moduļi, kas ļauj automātiski pārbaudīt atsevišķas sevišķi populāras sistēmu prasības pret izpildes vidi.

Promocijas darbā izstrādātā metode aprobēta divās atšķirīgās programmatūras izstrādes vidēs.

Pirmajā organizācijā tiek veikta iekšēja programmatūras izstrāde, tas ir, izstrādātā programmatūra paredzēta tikai izmantošanai šīs organizācijas iekšienē. Pēdējo 20 gadu laikā izstrādāti dažādi procesu automatizācijas risinājumi un daudzi no tiem joprojām tiek lietoti. Laika gaitā attīstoties palīgtehnoloģijām, daudzas no tām tikušas pielietotas jaunāko datorsistēmu izstrādē, taču esošā programmatūra joprojām izmanto iepriekšējās pieejas. Tādējādi kopējā uzturamo sistēmu saime pieprasa ārkārtīgi plaša tehnoloģiju spektra pārzināšanu. Šajā organizācijā ieviešot automatizēto vides testēšanu, pielietoto tehnoloģiju specifiku būtu iespējams iekļaut vides pārbaudēs, tādējādi mazinot uzturēšanas personālam nepieciešamo zināšanu daudzumu. Ir realizēts pilotprojekts, vides testus iekļaujot vienā no organizācijā izmantotajām datorsistēmām. Iegūtie rezultāti apliecina piedāvātās tehnoloģijas lietderību.

Otra pilotorganizācija ir individuālās programmatūras izstrādes organizācija, kas izstrādāto programmatūru nodod lietošanā citām organizācijām. Vides testēšanas tehnoloģijas aprobācijai tika izvēlēts ilglaicīgs projekts, kur datorsistēma tiek izplatīta datoros ģeogrāfiski plašā teritorijā, turklāt datoru konfigurācija nav standartizēta un aparatūras uzturēšanas darbinieku skaits ir neliels (tādējādi problēmu gadījumos kļūdu cēloņu identificēšana ir sarežģīta). Pārveidojot programmatūrā līdz šim jau iekļautās izpildes vides pārbaudes par deklaratīvi vadāmām prasībām pret izpildes vidi, tika panākts, ka, identificējot jaunas, specifiskas prasības pret izpildes vidi, nav nepieciešama programmatūras atkārtota kompilēšana un izvietošana visos lietotāju datoros. Šajā gadījumā var uzskatīt, ka metodes sasniegtie rezultāti ir ļoti labi.

Rezumējot darbā paveikto, var secināt, ka:

- ir izstrādāta metode programmatūras izpildei nepieciešamo prasību semantikas un sintakses nodalīšanai,
- ir izstrādāta valoda prasību aprakstīšanai un definēts vairākas iespējamās pieejas citu valodu radīšanai,
- ir formulēta metodika šīs programmatūras būves tehnoloģijas ieviešanai uzņēmumā,
- izstrādātā programmatūras būves tehnoloģija aprobēta praksē,
- ir identificēti arī citi iespējamie programmatūras izpildes prasību aprakstu lietojumi, kas atšķirīgi no sākotnējiem mērķiem, piemēram, programmatūras iznīcināšana un programmatūras sistēmu savstarpējo saišu identificēšana,
- ir izstrādāta koda funkciju bibliotēka *.Net* vidē, kas realizē galvenās vides testēšanas infrastruktūras sastāvdaļas.

Promocijas darba galvenie rezultāti ir atspoguļoti 4 publikācijās [**8**] [**9**] [**10**] [**11**], par pētījuma rezultātiem ziņots divās starptautiskās konferencēs. Par promocijas darba tēmu izstrādāts viens kursa darbs [17] un viens bakalaura darbs [18], šo ir darbu vadītājs ir promocijas darba autors. Abi darbi veiksmīgi aizstāvēti Latvijas Universitātes Datorikas Fakultātē.

Kā turpmākie pētījumu virzieni izstrādātās tehnoloģijas attīstīšanai minami:
- programmatūras prasību dokumenta automātiska ģenerēšana, izmantojot izstrādes vidē esošo informāciju un programmatūras kodu,
- pārbaudes laikā atklāto problēmu automātiska detalizētāka izpēte, salīdzinot problēmas apstākļus ar vēsturiskajiem datiem un/vai citās vidēs iegūtajiem mērījumiem, lai tādējādi lietotājam piedāvātu iespējamos risinājumus problēmas novēršanai.

DARBĀ LIETOTIE APZĪMĒJUMI UN SAĪSINĀJUMI

Saīsinājumi

CMMI (Capability Maturity Model) - programminženierijas un organizāciju attīstības modelis, kas definē uzņēmuma procesu attīstības nosacījumus un to izpildes metrikas.

CIM (Common Information Model) – atvērts standarts, kas definē IT infrastruktūras pārvaldāmo elementu reprezentāciju kā vienotu objektu un to savstarpējo attiecību kopu.

COM+ (Component Services) - *MS Windows* komponenšu serveris, kas nodrošina transakciju atbalstu, notikumu apstrādes rindas, lietotāju tiesību pārvaldību.

CORBA (Common Object Request Broker Architecture) - starpprogrammatūras standarts, kas ļauj dažādās valodās un platformās rakstītiem objektiem sazināties savā starpā.

EJB (Enterprise Java Beans) - vienošanās jeb interfeisu kopums Java platformā, kas ļauj jebkuram lietojumprogrammu serverim apkalpot jebkuru komponenti (avots: "EJB ievads", skat. http://www.ltn.lv/~apsitis/java-eim/de/intro_java_mod81_ejb_intro.html).

IUDD (Installable Unit Deployment Descriptor) – programmatūras izvietošanas aprakstīšanas valoda, kas programmatūras pakotni apskata kā atsevišķu uzstādāmu vienumu kopu un apraksta attiecības starp tiem.

MD5 - kriptogrāfiska funkcija, kas nodrošina teksta vienvirziena kripogrāfisku apstrādi (jaucējfunkcija).

MDE (*Model-Driven Engineering)* – programminženierijas pieeja, kas balstīta uz iteratīvas arvien detalizētākas sistēmas modelēšanas.

MSF (Microsoft Solutions Foundation) - kompānijas *Microsoft* piedāvātais programmatūras dzīves cikla modelis iteratīvai programmatūras izstrādei.

QoS (Quality of Service) - servisa kvalitātes mērīšana – ideoloģija, kas programmatūras vai aparatūras darba kvalitāti saista ar apkārtējās vides raksturlielumu kvalitāti (piemēram, mobilā tālruņa zvana kvalitātes saistība ar joslas platumu).

RAID (Redundand Array of Independent Disks) - divu vai vairāku diskdziņu izmantošana vienu un to pašu datu glabāšanai dažādās vietās uz vairākiem cietajiem diskiem. (avots: LZA TK Informācijas tehnoloģijas un telekomunikācijas terminoloģijas apakškomisijas apstiprinātie termini).

SOA (Service- Oriented Architecture) – programmatūras arhitektūru pieeja, kas kā būtisku prioritāti nosaka izstrādāto komponenšu savstarpējo sadarbību un katras komponentes „atbildību" par tās definētajiem mērķiem.

SOAP (Simple Object Access Protocol) - attālinātu izsaukumu veikšanas metodika, visbiežāk realizēta kā HTTP protokola virsbūve.

STSW (Smart Technology Compatible Software) - programmatūra, kas izstrādāta saskaņā ar viedās programmatūras izstrādes vadlīnijām. Tiek uzskatīts, ka šāda datorsistēmu arhitektūra vienkāršo sistēmu uzturēšanu, samazinot cilvēka lomu uzturēšanas procesos.

TCO (Total Cost of Ownership) - īpašuma kopējās izmaksas. Faktiskās personālā datora ekspluatācijas izmaksas. Tajās ietvertas: - datora un programmatūras sākotnējā cena; - aparatūras un programmatūras jauninājumi; - uzturēšana; - tehniskais atbalsts; - datorizētās mācības. (avots: LZA TK Informācijas tehnoloģijas un telekomunikācijas terminoloģijas apakškomisijas apstiprinātie termini).

TCP (Transport Control Protocol) - transporta vadības protokols, relatīvi zema līmeņa tīkla protokols, ka nodrošina datu pakešu pārsūtīšanu datortīklā.

UML - Unified Modelling Language – grafiska modelēšanas valoda, kas paredzēta jēdzienu savstarpējo attiecību strukturētai aprakstīšanai.

Termini

.Net Remoting - .Net platformā iekļauts starpprogrammatūras standarts, kas ļauj .Net programmu procesiem sazināties savā starpā. *.Net Remoting* var izmantot gan bināru transporta slāņa protokolu, gan HTTP datu kanālu.

autonomā sistēma - IBM autonomo sistēmu iniciatīvas izpratnē - datorsistēma, kas, līdzīgi dzīvu radību autonomajām nervu sistēmām, patstāvīgi nodrošina sistēmas darba turpināšanai nepieciešamos apstākļus.

baitkods - izpildīšanai paredzēts, kompilēts programmatūras kods.

izmitināšanas vide - vide, kurā tiek darbināta programmatūra. Šī darba ietvaros ar jēdzienu "izmitināšanas vide" tiek saprasta gan vide, kas darbojas viena fiziska datora ietvaros, gan - dalītu sistēmu gadījumā - visu to atdalīto infrastruktūras fragmentu kopums, kas nodrošina datorsistēmas pilnvērtīgu funkcionēšanu.

lokāle - Jēdziens, kas raksturo noteiktai valodai vai valodas lokalizācijai specifiskus reģionālos iestatījumus. Parasti lokāle ietver tādus konfigurējamus vienumus kā datuma un laika formāts, mēnešu nosaukumi, nedēļas pirmā diena u.c.

programmatūras izpildes profils – dokuments, kurā aprakstītas programmatūras prasības pret izpildes vidi. Tas tiek izmantots par pamatu programmatūras automātiski veiktajām vides pārbaudēm.

refleksija - datorzinātnē par refleksiju (angl. *reflection*) sauc procesu, ar kura palīdzību datora programma var aplūkot un modificēt pati savu struktūru un uzvedību (avots: *Wikipedia* tiešsaistes enciklopēdija skat http://en.wikipedia.org).

replikācija – SQL datubāzēm bieži pielietota uzturēšanas metode, kas sistēmas ātrdarbības uzlabošanas nolūkā veido datu dublikātus. Replicēšana parasti ir datubāzu vadības sistēmā iekļauts serviss.

serializācija - Datora operatīvajā atmiņā glabāta objekta pārveidošana binārā simbolu virknē ar mērķi saglabāt objekta stāvokli ilglaicīgā atmiņā.

serveru ferma - Divu un vairāk serveru veidota infrastruktūra, kas loģiski apvienota kāda servisa sniegšanai.

XML Schema - Valoda, kas ļauj definēt XML dokumentu struktūras likumus.

XPath - Valoda, kas ļauj definēt datu atlases vaicājumus XML datu struktūrām.

BIBLIOGRĀFIJA

1. **Gielens, P.**.*.NET Configuration Hell - A discussion.* [tiešsaiste.]: Paul Gielens: Thoughts Service, 2007 [atsauce: 03.2009]. Pieejams: http://weblogs.asp.net/pgielens/archive/2007/04/05/net-configuration-hell.aspx

2. **Ganek, A., Corbi, T.** The dawning of the autonomic computing era. *IBM Systems Journal*, 2003, vol. 42, pp.5-18

3. **Patterson, D.** *Availability and Maintainability >> Performance:New Focus for aNew Century.* [tiešsaiste.]: USENIX Conference on File and Storage Technologies (FAST '02), 2002 [atsauce: 05.2009]. Pieejams: http://roc.cs.berkeley.edu/talks/Fastkeynote5.ppt

4. **Bičevska, Z., Bičevskis, J.** Smart Technologies in Software Life Cycle. ***In:*** *Proceedings of Product-Focused Software Process Improvement. 8th International Conference, PROFES 2007, July 2-4, 2007* (Münch, J., Abrahamsson, P., eds.), Riga, Latvia, vol. 4589/2007, 2007. pp.262-272

5. **Bičevska, Z., Bičevskis, J.** Application of Smart Technologies in Software Development: Automated Version Updating. ***In:*** *Scientific papers, vol. 733* (Bārzdiņš, J., Freivalds, R.-M., Bičevskis, J., eds.) University of Latvia, 2008, pp.24 - 37

6. **Bicevska, Z., Bicevskis, J.** Applying Self-Testing: Advantages and Limitations. ***In:*** *Databases and Information Systems V - Selected Papers from the Eighth International Baltic Conference, DB&IS 2008, June 2-5, 2008, Tallinn, Estonia* (Haav, H.-M., Kalja, A., eds.), vol. 187, 2008. pp.192-202

7. **International Business Machines Corporation** *AUTONOMIC VISION & MANIFESTO.* [tiešsaiste.]: IBM web site, 2001 [atsauce: 04.2009]. Pieejams: http://www.research.ibm.com/autonomic/manifesto/

8. **Rauhvargers, K., Bičevskis, J.** Towards a semantic execution environment testing model. ***In:*** *LU scientific Papers, vol. 733* (Bārzdiņš, J., Freivalds, R.-M., Bičevskis, J., eds.) University of Latvia, 2008, pp.38-52

9. **Rauhvargers, K., Bičevskis, J.** Automating the Software Environment Testing Process. ***In:*** *Proceedings of the Eighth International Baltic Conference Baltic DB&IS 2008 Tallinn, June 2–5, 2008* (Haav, H.-M., Kalja, A., eds.), Tallinn, 2008. p.155–166

10. **Rauhvargers, K.** On the Implementation of a Meta-data Driven Self Testing Model. ***In:*** *Software Engineering Techniques in Progress* (Hruška, T., Madeyski, L., Ochodek, M., eds.), Brno, Czech Republic, 2008. pp.153-166

11. **Rauhvargers, K., Bičevskis, J.** Environment Testing Enabled Software – a Step Towards

Execution Context Awareness. ***In:*** *Databases and Information Systems V - Selected Papers from the Eighth International Baltic Conference, DB&IS 2008* (Haav, H.-M., Kalja, A., eds.), 2009. pp.169-179

12. **Winter, R., Schelp, J.** Enterprise architecture governance: the need for a business-to-IT approach. ***In:*** *Proceedings of the 2008 ACM symposium on Applied computing* , 2008. pp.548-552

13. **Dearle, A.** Software Deployment, Past, Present and Future. ***In:*** *FOSE '07: 2007 Future of Software Engineering*, 2007. pp.269--284

14. **DocLogix UAB** *DocLogix – Informācijas un uzņēmuma darbības procesa pārvaldības sistēma.* [tiešsaiste.] [atsauce: 05.2009]. Pieejams: http://www.doclogix.lt/

15. **Microsoft, Corp.** *Microsoft Office SharePoint Server - Connecting People, Process, and Information.* [tiešsaiste.] [atsauce: 05.2009]. Pieejams: http://sharepoint.microsoft.com/Pages/Default.aspx

16. **Bisbal, J., Lawless, D., Wu, B., Grimson, J.** Legacy Information Systems: Issues and Directions. *IEEE Software*, 1999, vol. 16, pp.103-111

17. **Ivanovs, J.** *Automātiska programmatūras apkārtējās vides pārbaude.* kursa darbs, LU, FMF Datorikas nodaļa, Rīga (2008)

18. **Ivanovs, J.** *Vides pārbaudītāja integrēšana ar dažādām programmatūras izstrādes platformām.* Bakalaura darbs, Latvijas Universitāte, Rīga (2009)

19. **Keller, A., Brown, A., Hellerstein, J.** A Configuration Complexity Model and Its Application to a Change Management System. *IEEE Transactions on Network and Service Management*, 2007, vol. 4, pp.13-27

20. **Brown, A., Hellerstein,.** An approach to benchmarking configuration complexity. ***In:*** *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, Leuven, Belgium , 2004. p.18

21. **Brown, A., Patterson, D.** To err is human. ***In:*** *Proceedings of the First Workshop on Evaluating and Architecting System dependabilitY (EASY '01)*, 2001.

22. **Chellaiah, S.** *Computer, heal thyself.* [tiešsaiste.]: e-World (The Hindu Business Line), 2006 [atsauce: 04.2009]. Pieejams: http://www.thehindubusinessline.com/ew/2006/03/27/stories/2006032700170300.htm

23. **Lipner, S.** The trustworthy computing security development lifecycle. ***In:*** *Computer Security Applications Conference, 2004. 20th Annual*, 2004. pp.2-13

24. **Heiser, G., Elphinstone, K., Kuz, I., Klein, G., Petters, S.** Towards trustworthy

computing systems: taking microkernels to the next level. *SIGOPS Oper. Syst. Rev.*, 2007, vol. 41, pp.3--11

25. **Sterritt, R., Bustard, D.** Towards an autonomic computing environment. ***In:*** *Proceedings of 14th International Workshop on Database and Expert Systems Applications* (Marík, V., Retschitzegger, W., Stepánková, O., eds.), Prague, Czech Republic, 2003. pp.694 - 698

26. **Lightstone, S** Foundations of Autonomic Computing Development. ***In:*** *Proceedings of the Fourth IEEE international Workshop on Engineering of Autonomic and Autonomous Systems, March 26 – 29*, Tucson, USA, 2007. pp.163-171

27. **Nami, M., K., Bertels.** A Survey of Autonomic Computing Systems. ***In:*** *ICAS '07: Proceedings of the Third International Conference on Autonomic and Autonomous Systems*, 2007. p.26

28. **Tosi, D** *Research Perspectives in Self-Healing Systems.* Technical report LTA:2004:06, University of Milano-Bieocca, Milano (2004)

29. **Orso, A., Harrold, M., Rosenblum, D.** Component Metadata for Software Engineering Tasks. ***In:*** *EDO '00: Revised Papers from the Second International Workshop on Engineering Distributed Objects*, London, vol. 1999, 2001. pp.129-144

30. **Kephart, J.,** Research challenges of autonomic computing. ***In:*** *ICSE '05: Proceedings of the 27th international conference on Software engineering*, 2005. pp.15-22

31. **Brodie, M., Ma, Sheng, Lohman, G., Mignet, L., Wilding, M., Champlin, J., Sohn, P.** Quickly Finding Known Software Problems via Automated Symptom Matching. ***In:*** *Second International Conference on Autonomic Computing, 2005. ICAC 2005. Proceedings.*, 2005. pp.101-110

32. **Arnautovic, E., Kaindl, H., Falb, J., Popp, R., Szep, A.** Gradual transition towards autonomic software systems based on high-level communication specification. ***In:*** *Proceedings of the 2007 ACM symposium on Applied computing*, 2007. pp.84-89

33. **Herrmann, K., Muhl, G., Geihs, K.** Self management: the solution to complexity or just another problem? *Distributed Systems Online*, 2005, 1, vol. 6

34. **Bichevskii, Y.,Borzov,** Development of Symbolic-testing Methods for Computer-programs. *Automation and Remote Control*, 1982, 8, vol. 43, pp.1054-1061

35. **Iljins, J.** Metamodel Based Approach to IS Development and Lessons Learned. ***In:*** *Proceedings of the Eighth International Baltic Conference Baltic DB&IS 2008* (Haav, H.-M., Kalja, A., eds.), Tallinn, vol. 167-178, 2008.

36. **American National Standards Institute** *ANS X3.159-1989: Programming Languages---*

C. standard (1989)

37. **Satpathy, M., Siebel, N., Rodriguez, D.** Assertions in object oriented software maintenance: analysis and case study. ***In:*** *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, 2004. pp.124-133

38. **IEEE/IEEE Std** *IEEE standard for software unit testing.* standard 1008-1987 (1986)

39. **NUnit.org** *NUnint - unit testing framework.* [tiešsaiste.] [atsauce: 04.2009]. Pieejams: http://nunit.org

40. **Janzen, D., Saiedian, H.** Test-driven development concepts, taxonomy, and future direction. *Computer*, 2005, vol. 38, pp.43-50

41. **Kim, T., Park, C., Wu, C.** Mock Object Models for Test Driven Development. ***In:*** *SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, 2006. pp.221-228

42. **Wang, Y., King, G., Wickburg, H.** A Method for Built-in Tests in Component-based Software Maintenance. ***In:*** *CSMR '99: Proceedings of the Third European Conference on Software Maintenance and Reengineering*, 1999. p.186

43. **Barbier, F., Belloir, N.** Component Behavior Prediction and Monitoring through Built-In Test. *IEEE International Conference on the Engineering of Computer-Based Systems*, 2003, , p.17

44. **Atkinson, C., Gross, H.** Built-in Contract Testing in Model-driven, Component-Based Development. ***In:*** *First International Working Conference on Component Deployment*, Austin, 2002.

45. **Belli, F., Budnik, C.** Towards Self-Testing of Component-Based Software. ***In:*** *Proceedings of the 29th Annual International Computer Software and Applications Conference*, vol. 2, 2005. pp.205-210

46. **Beydeda, S.** Research in testing COTS components - built-in testing approaches. ***In:*** *The 3rd ACS/IEEE International Conference on Computer Systems and Applications, 2005.*, 2005. pp.101-vii

47. **Suliman, D., Paech, B., Borner, L., Atkinson, C., Brenner, D., Merdes, M., Malaka, R.** The MORABIT Approach to Runtime Component Testing. *Computer Software and Applications Conference, Annual International*, 2006, vol. 2, pp.171-176

48. **van Hoff, A., Partovi, H.** *The Open Software Description Format (OSD).* [tiešsaiste.]: World Wide Web Consortium, 1997 [atsauce: 04.2009]. Pieejams: http://www.w3.org/TR/NOTE-OSD.html

49. *Extensible Markup Language (XML) 1.0.* [tiešsaiste.]: World Wide Web Consortium, 1998 [atsauce: 04.2009]. Pieejams: http://www.w3.org/TR/1998/REC-xml-19980210

50. **Vitaletti, M., Draper, C., George, R., McCarthy, J., Poolman, D., Miller, T., Middlekauff, A., Montero-Luque, C.** *Installable Unit Deployment Descriptor Specification Version 1.0.* [tiešsaiste.]: World Wide Web Consortium, 2004 [atsauce: 04.2009]. Pieejams: http://www.w3.org/Submission/InstallableUnit-DD/

51. **Jacob, B., Basu, S., Tuli, A., Witten, P.** *A First Look at Solution Installation for Autonomic Computing*. International Business Machines corp., 2004. p.194

52. **Draper, C., George, R., Vitaletti, M.** Installable Unit Deployment Descriptor for Autonomic Solution Management. ***In:*** *DEXA '04: Proceedings of the Database and Expert Systems Applications, 15th International Workshop*, 2004. pp.742-746

53. **OASIS Solution Deployment Descriptor (SDD) TC** *Solution Deployment Descriptor Specification 1.0.* [tiešsaiste.]: Organization for the Advancement of Structured Information Standards, 2008 [atsauce: 04.2009]. Pieejams: http://docs.oasis-open.org/sdd/v1.0/os/sdd-spec-v1.0-os.html

54. **OASIS**[tiešsaiste.]: Organization for the Advancement of Structured Information Standards (OASIS) [atsauce: 04.2009]. Pieejams: http://www.oasis-open.org/

55. **McCarthy, J., Miller, B.** *Solution Deployment Descriptor (SDD), Part 1: An emerging standard for deployment artifacts.* [tiešsaiste.]: IBM - Tivoli Technical Library, 2008 [atsauce: 04.2009]. Pieejams: http://www.ibm.com/developerworks/library/ac-artifacts/index.html

56. **OASIS Solution Deployment Descriptor (SDD) TC** *Solution Deployment Descriptor Starter Profile Version 1.0.* [tiešsaiste.]: Organization for the Advancement of Structured Information Standards, 2008 [atsauce: 04.2009]. Pieejams: http://docs.oasis-open.org/sdd/v1.0/cd01/sdd-starter-profile-v1.0-cd01.html

57. **Distributed Management Task Force** *CIM Schema: Version 2.21.0.* [tiešsaiste.]: Distributed Management Task Force [atsauce: 04.2009]. Pieejams: http://www.dmtf.org/standards/cim/cim_schema_v2210

58. **OASIS Solution Deployment Descriptor (SDD) TC** *Example SDDs showing the use of the schema.* [tiešsaiste.]: Organization for the Advancement of Structured Information Standards, 2008 [atsauce: 04.2009]. Pieejams: http://www.oasis-open.org/committees/download.php/31437/v1.1_examples.zip

59. **Weis, T., Ulbrich, A., Geihs, K., Becker, C.** Quality of Service in Middleware and Applications: A Model-Driven Approach. ***In:*** *Proceedings of the Eighth Enterprise Distributed Object Computing IEEE International Conference*, 2004. pp.160-171

60. **Ansaloni, S., Sztajnberg, A., Cerqueira, C., Loques, O.** Deploying QoS Contracts in the Architectural Level. ***In:*** *IFIP International Federation for Information Processing, vol. Volume 176/2005* 2005, pp.19-34

61. **Cardoso, L., Sztajnberg, A., Loques, O.** Self-Adaptive Applications Using ADL Contracts. ***In:*** *Self-Managed Networks, Systems, and Services, Second IEEE International Workshop* (Keller, A., Martin-Flatin, J.-P., eds.), vol. 3996, 2006. pp.87-101

62. **McDovwell, D., Veerman, E., Otey., M.** *SQL Server 2005 Upgrade Handbook.* [tiešsaiste.]: Microsoft TechNet internet site, 2005 [atsauce: 04.2009]. Pieejams: http://www.microsoft.com/technet/prodtechnol/sql/2005/sqlupgrd.mspx

63. **Brenner, D., Atkinson, C., Malaka, R., Merdes, M., Paech, B., Suliman, D.** Reducing verification effort in component-based software engineering through built-in testing. *Information Systems Frontiers*, 2007, vol. 9, pp.151-162

64. **Strunk, W., Lilienthal, C.** Tool Support for Testing and Documenting Framework-Based Software. ***In:*** *Technology of Object-Oriented Languages and Systems, 1999. TOOLS 30. Proceedings*, 1999. pp.237-246

65. **Anderson, R.** *The End of DLL Hell.* [tiešsaiste.]: MSDN Library, 2000 [atsauce: 04.2009]. Pieejams: http://msdn.microsoft.com/en-us/library/ms811694.aspx

66. **Hjálmtýsson, G.,** Dynamic C++ classes. A lightweight mechanism to update code in a running program. ***In:*** *USENIX Annual Technical Conference (NO 98), Proceedings*, 1998. pp.65-76

67. **Gao, S., Sperberg-McQueen, C., Thompson, S.** *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures.* [tiešsaiste.]: W3C Consortium, 2009 [atsauce: 05.2009]. Pieejams: http://www.w3.org/TR/xmlschema11-1/

68. **Decker, S., Mitra, P., Melnik, S.** Framework for the semantic Web: an RDF tutorial. *Internet Computing*, 2000, vol. 4, pp.68-73

69. **Heimbigner, D.** DMTF - CIM to OWL: A Case Study in Ontology Conversion. ***In:*** *Ontology in Action Workshop in conjunction with the 2004 Conference on Software Engineering and Knowledge Engineering (SEKE'04)*, Alberta, Canada, 2004. pp.470-474

70. **Lavy, M., Meggitt, A.** *Windows Management Instrumentation (WMI)*. Sams Publishing, 2001.

71. **Microsoft, Corp.** *Add a Program to the Exceptions List.* [tiešsaiste.]: TechNet Library, 2005 [atsauce: 04.2009]. Pieejams: http://technet.microsoft.com/en-us/library/cc775783.aspx

72. **World Wide Web Consortium** *XML Path Language (XPath).* [tiešsaiste.]: World Wide

Web Consortium [atsauce: 04.2009]. Pieejams: http://www.w3.org/TR/xpath

73. **Crockford, D.** *The application/json Media Type for JavaScript Object Notation (JSON).* Request for Comments RFC 4627, The Internet Society (2006)

74. **Abrahamsson, P., Salo, O., Ronkainen, J., Warsta, J.** *Agile Software Development Methods: Review and Analysis*. Espoo: VTT, 2002.

75. **Kleppe, A., Warmer, J., Bast, W.** *MDA Explained: The Model Driven Architecture-Practice and Promise*. Addison Wesley, 2003.

76. **Gao, D., Reiter, M., Song, D.** Gray-box extraction of execution graphs for anomaly detection. ***In:*** *Proceedings of the 11th ACM Conference on Computer and Communications Security*, Washington DC, USA, 2004. pp.318-329

77. **Box, D., Pattison, T.** *Essential.Net: the Common Language Runtime*. Addison-Wesley Longman Publishing Co., Inc., p.432

78. **Microsoft, Corp.** *MSF Team Model v. 3.1.* [tiešsaiste.]: Microsoft Solutions Framework Core Whitepapers, 2002 [atsauce: 04.2009]. Pieejams: http://www.microsoft.com/downloads/details.aspx?FamilyID=c54114a3-7cc6-4fa7-ab09-2083c768e9ab&displaylang=en

79. **Latvijas Banka** *Maksājumu sistēmu statistika. Svarīgākie maksājumu sistēmu rādītāji.* [tiešsaiste.]: Latvijas Banka, 2008 [atsauce: 08.2008]. Pieejams: http://www.bank.lv/lat/main/fa/statistika/ms-stat

80. **Latvijas Banka** *Kredītu reģistra statistika 2009. gada 1. janvārī.* [tiešsaiste.]: Latvijas Banka, 2009 [atsauce: 05.2009]. Pieejams: http://www.bank.lv/lat/main/all/lvbank/registrs/kreditu_registra_statistika/

81. **Microsoft, Corp.** *How to troubleshoot the "Cannot generate SSPI context" error message.* [tiešsaiste.]: Microsoft atbalsta centrs, 2007 [atsauce: 05.2009]. Pieejams: http://support.microsoft.com/kb/811889

82. **Microsoft, Corp** *Server farms and topologies.* [tiešsaiste.]: TechNet Library, 2009 [atsauce: 04.2009]. Pieejams: http://technet.microsoft.com/en-us/library/cc263440.aspx

83. **Taliver, H., Centeno, A., George, P., Ramos, L., Jaluria, Y., Bianchini, R.** Mercury and Freon: Temperature Emulation and Management for Server Systems. ***In:*** *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, San Jose, USA, 2006. pp.106-116

PIELIKUMI

Turpmākajās nodaļās iekļautie promocijas darba pielikumi atspoguļo daļu darbā veiktā teorētiskā pētījuma praktisko rezultātu. To mērķis ir vienkāršot darbā izstrādātās metodikas uztveri, taču tos apjoma dēļ nav iespējams tiešā veidā iekļaut darba pamattekstā.

1. un 2. pielikumā demonstrēta programmatūras izpildes prasību dokumenta instance, kas veidot ar mērķi ir aprakstīt programmatūras komponentes *eDMIS_Correspondence* prasības pret izpildes vidi. 1. pielikumā parādīta šāda instances dokumenta veidošana saskaņā ar *SDD* deskriptorā piedāvāto shēmu, savukārt 2. pielikumā – izmantojot vienkāršotu un tieši vides prasību aprakstīšanai konstruētu prasību aprakstīšanas valodu.

Izpildes profilā iekļautas šādas prasības:

- **reģionālie iestatījumi**
  - vai datora lokāle atbilst Latvijas (latviešu) standartlokālei lv-LV,
  - vai datuma formāts ir „dd.mm.gggg" (organizācijas iekšienē lietots standarts, kas atšķirīgs no ISO noteiktā),
  - vai decimālatdalītāja simbols (simbols, kas atdala skaitļa veselo daļu no daļskaitļa) datorā ir norādīts kā komats (UNICODE: U+002C),
  - vai sarakstu atdalītājs datorā ir noteikts kā semikols (UNICODE: U+003B),
- izpildei nepieciešamo **bibliotēku pieejamība**: vai datorā instalētas trīs dažādas bibliotēkas. Tās visas atrodamas pēc „*fullName*" atribūta, taču atsevišķām bibliotēkām būtisks arī papildus atribūts – „*fileVersion*", kas ļauj identificēt īsto bibliotēkas versiju;
- failu sistēmas objektu pieejamība
  - vai pieejams direktorijs, kurā tiek veikta failu apmaiņa ar citām sistēmām. Šim direktorijam jābūt ne tikvien lasāmam, bet arī rakstāmam,
  - vai pieejams direktorijs, kurā glabājas sistēmā izmantotās tīmekļa lapas (sistēmai ir tīmekļa bāzēta lietotāja saskarne),
  - vai tīmekļa lapu direktorijā ir atrodami nepieciešamie faili (pārbaude pēc MD5 jaucējsummas),
- datubāzes pieejamība
  - vai pieejama sistēmas datubāze,
  - vai tajā eksistē nepieciešamā tabula.

Piemērā apskatītais programmatūras profils sarežģītības ziņā ir gandrīz triviāls (piemēram, pārbauda tikai divu failu sistēmas elementu pieejamību, kaut praksē sastopamās sistēmās šādu objektu skaits var pārsniegt vairākus desmitus un pat simtus atkarībā no lietojuma), taču tas labi demonstrē atšķirības starp *SDD* pieeju un pielāgotas valodas lietošanu.

1. pielikums

Programmatūras prasību profila realizācija, izmantojot *SDD* sintaksi

Aprakstot prasības ar *SDD* shēmu, sākotnēji nepieciešams identificēt resursus, uz kuriem prasības attiecināmas. Resursus vēlams aprakstīt to topoloģiskā hierarhijā (piemēram, failu sistēma ir operētājsistēmas sastāvdaļa, tabula ir datubāzes sastāvdaļa utt.), taču šajā lietojumā topoloģija nav kritiska. Topoloģijas apraksts parādīts piemēra 15 – 61. rindiņā.

Kā redzams, šajā risinājumā katrs resurss tiek aprakstīts atsevišķi, norādot arī resursu identificējošus atribūtus (piemēram, 40.-46. rindiņā aprakstīts fails „*default.aspx*", kas uztverams kā resursa *WebRootDir* sastāvdaļa), taču prasības, kas attiecināmas uz pārskaitītajiem resursiem, aprakstītas atsevišķi no paša resursa.

Aprakstot resursu, tiek norādīts tā tips, kam jāatbilst kādam no iepriekš zināmiem resursu tipiem. Gadījumā, ja visi nepieciešamie resursu tipi jau aprakstīti kādā *SDD* resursu deskriptorā, šos aprakstus var atkalizmantot, lietojot *XML Schema*. Šajā piemērā izmantoti „*CIM Starter profile*" [**56**] piedāvātie jēdzieni, piesaiste *CIM* shēmai redzama koda piemēra 5. rindiņā. Ja resursi līdz šim nav aprakstīti, jāveido savs resursu deskriptors (izmantojot *XML Schema* līdzekļus), tajā iekļaujot visus nepieciešamos jēdzienus un to atribūtus. Piemērā izmantots arī šāds resursu deskriptors (skat. pielikuma nodaļu „Koda piemērs 1.2. Pielāgots resursu deskriptors"), kas apraksta jēdzienus kā:

- *RegionalSettings* – datora reģionālo iestatījumu kopums, ko raksturo tādi atribūti kā lokāle, decimālatdalītājs, sarakstu atdalītājs, datuma formāts u.c.,
- *CodeAssembly – Microsoft .Net* koda asambleja,
- *GAC* (*MS Windows* operētājsistēmās – speciāls direktorijs, kurā tiek vienkopus glabātas dažādas sistēmā izmantojamas *Microsoft .Net* asamblejas),
- *SqlTable* – relāciju datubāzu tabula (*CIM Starter Profile* apraksta datubāzu jēdzienus tikai līdz datubāzes līmenim)

Piesaiste šādi veidotajam deskriptoram realizējama, izmantojot *XML Namespace* jēdzienus piemērā tā redzama 8. koda rindā.

Kad visi nepieciešamie resursi ir definēti, tiek aprakstīta programmatūras komponente, kura izvirza noteiktas prasības pret šo resursu. Šāda komponentes definīcija redzama 64. – 179. rindiņā, aprakstot „*eDMIS_Correspondence*" prasības pret vidi. Viegli pamanīt, ka prasības, kas patiešām attiecas uz kāda resursa atsevišķa atribūta vērtību, aprakstāmas diezgan viegli un loģiski. Piemēram, 77. – 80. piemēra rindiņā aprakstītā prasība „īsais datuma formāts atbilst *mm.dd.gggg*". Gadījumos, kad tiek aprakstīta nevis īpašības vērtība, bet kāds cits uz resursu attiecināms predikāts, *SDD* realizācijā veidojas neveikla konstrukcija. Piemēram, prasība, ka failu apmaiņas direktorijam jābūt lietotājam pieejamam gan rakstīšanai, gan lasīšanai, aprakstīta 134. – 144. rindiņā, pieprasot divu direktorijam piemītošu atribūtu „*readable*" un „*writable*" vērtību „*true*". Diskutabli, vai šādi atribūti patiešām ir direktorija īpašības, vai uztverami tikai kontekstā ar lietotāju, kurš pārbaudi veic. Līdzīgi, 156. – 163. rindiņā pieprasīts, ka resursa

„*Files_Default_aspx*” atribūtam „*Exists*” jābūt vērtībai „*true*”, taču faktiski tas izmantots kā veids, lai pārbaudītu, vai failu sistēmā ir izveidots fails ar šādu nosaukumu.

Koda piemērs 1.1. Prasību faila instance

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <SDD-dd:DeploymentDescriptor
3.         xmlns:SDD-dd="http://docs.oasis-open.org/SDD/ns/deploymentDescriptor"
4.         xmlns:SDD-common="http://docs.oasis-open.org/SDD/ns/common"
5.         xmlns:sp="http://docs.oasis-open.org/SDD/ns/starterProfile"
6.         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
7.         xmlns:cim="http://docs.oasis-open.org/SDD/1/0/cim-Profile"
8.         xmlns:lb="http://bank.lv/softwareProfile"
9.         xsi:schemaLocation="http://docs.oasis-open.org/SDD/ns/deploymentDescriptor
10.        http://docs.oasis-open.org/SDD/v1.0/os/FullSchema/
11.             SDD-deploymentDescriptor-1.0.xsd"
12.         schemaVersion="1.0"
13.         lastModified="2009-05-01T12:00:00.0Z"
14.         descriptorID="51EC96D28ECC40448E8324A709C9ADDA">
15.    <SDD-dd:Topology>
16.      <SDD-dd:Resource id="os" type="sp:CIM_OperatingSystem">
17.        <SDD-dd:HostedResource id="regionalsettings"
18.                               type="lb:LB_RegionalSettings" />
19.        <SDD-dd:HostedResource id="GAC" type="lb:LB_GlobalAssemblyCache">
20.          <SDD-dd:HostedResource id="LB_CommonNetAssembly"
21.                                 type="lb:LB_CodeAssembly" />
22.          <SDD-dd:HostedResource id="LB_eDMISCommonAssembly"
23.                                 type="lb:LB_CodeAssembly" />
24.          <SDD-dd:HostedResource id="LB_eCorrespondenceWorkFlowsAssembly"
25.                                 type="lb:LB_CodeAssembly" />
26.        </SDD-dd:HostedResource>
27.        <SDD-dd:HostedResource id="filesystem" type="sp:CIM_FileSystem">
28.          <SDD-dd:HostedResource id="FileExchangeDir" type="sp:CIM_Directory">
29.            <SDD-dd:Property>
30.              <SDD-dd:PropertyName>Path</SDD-dd:PropertyName>
31.              <SDD-dd:Value>${FileExchangeDir}</SDD-dd:Value>
32.            </SDD-dd:Property>
33.          </SDD-dd:HostedResource>
34.          <SDD-dd:HostedResource id="WebRootDir" type="sp:CIM_Directory">
35.            <SDD-dd:Property>
36.              <SDD-dd:PropertyName>Path</SDD-dd:PropertyName>
37.              <SDD-dd:Value>${AspxPagesDir}</SDD-dd:Value>
38.            </SDD-dd:Property>
39.          </SDD-dd:HostedResource>
40.          <SDD-dd:HostedResource id="Files_Default_aspx"
41.                                 type="sp:CIM_LogicalFile">
42.            <SDD-dd:Property>
43.              <SDD-dd:PropertyName>Path</SDD-dd:PropertyName>
44.              <SDD-dd:Value>${AspxPagesDir}\default.aspx</SDD-dd:Value>
45.            </SDD-dd:Property>
46.          </SDD-dd:HostedResource>
47.        </SDD-dd:HostedResource>
48.      </SDD-dd:Resource>
49.      <SDD-dd:Resource id="edmisDB" type="sp:CIM_DatabaseSystem">
50.        <SDD-dd:Property>
51.          <SDD-dd:PropertyName>ConnectionString</SDD-dd:PropertyName>
52.          <SDD-dd:Value>${DBServerConnectionString}</SDD-dd:Value>
53.        </SDD-dd:Property>
54.        <SDD-dd:HostedResource id="UsersTable" type="lb:LB_SqlTable">
55.          <SDD-dd:Property>
56.            <SDD-dd:PropertyName>Name</SDD-dd:PropertyName>
57.            <SDD-dd:Value>Users</SDD-dd:Value>
58.          </SDD-dd:Property>
59.        </SDD-dd:HostedResource>
60.      </SDD-dd:Resource>
```

```
61.    </SDD-dd:Topology>
62.
63.
64.    <SDD-dd:CompositeInstallable id="eDMIS_Correspondence" operation="use" >
65.      <SDD-dd:Requirements>
66.        <!--Pārbaudes, kas attiecas uz mērķa
67.            vides reģionālo iestatījumu konfigurāciju-->
68.        <SDD-dd:Requirement id="locale.reqt" operation="install use">
69.          <SDD-dd:Description>Reģionālo iestatījumu nosacījumi
70.          </SDD-dd:Description>
71.          <SDD-dd:ResourceConstraint id="localeConstraint"
72.                                     resourceRef="regionalsettings">
73.            <SDD-dd:PropertyConstraint>
74.              <SDD-dd:PropertyName>lb:locale</SDD-dd:PropertyName>
75.              <SDD-dd:Value>lv-LV</SDD-dd:Value>
76.            </SDD-dd:PropertyConstraint>
77.            <SDD-dd:PropertyConstraint>
78.              <SDD-dd:PropertyName>lb:ShortDateFormat</SDD-dd:PropertyName>
79.              <SDD-dd:Value>dd.MM.yyyy</SDD-dd:Value>
80.            </SDD-dd:PropertyConstraint>
81.            <SDD-dd:PropertyConstraint>
82.              <SDD-dd:PropertyName>lb:DecimalSymbol</SDD-dd:PropertyName>
83.              <SDD-dd:Value>,</SDD-dd:Value>
84.            </SDD-dd:PropertyConstraint>
85.            <SDD-dd:PropertyConstraint>
86.              <SDD-dd:PropertyName>lb:ListSeparator</SDD-dd:PropertyName>
87.              <SDD-dd:Value>;</SDD-dd:Value>
88.            </SDD-dd:PropertyConstraint>
89.          </SDD-dd:ResourceConstraint>
90.        </SDD-dd:Requirement>
91.        <!--Pārbaudes, vai visas nepieciešamās koda bibliotēkas ir pieejamas-->
92.        <SDD-dd:Requirement id="assemblies.reqt" operation="install use">
93.          <SDD-dd:Description>Vai Windows GAC ir
94.              izvietotas nepieciešamās bibliotēkas</SDD-dd:Description>
95.          <SDD-dd:ResourceConstraint id="LBCommonNetAvailable"
96.                                     resourceRef="LB_CommonNetAssembly">
97.            <SDD-dd:PropertyConstraint>
98.              <SDD-dd:PropertyName>lb:fullName</SDD-dd:PropertyName>
99.              <SDD-dd:Value>LB_CommonNet, Version=2.1.2.0,
100.                 Culture=neutral, PublicKeyToken=1b7e4d24feeb9a6c
101.               </SDD-dd:Value>
102.           </SDD-dd:PropertyConstraint>
103.         </SDD-dd:ResourceConstraint>
104.         <SDD-dd:ResourceConstraint id="eDMISCommonAvailable"
105.                                    resourceRef="LB_eDMISCommonAssembly">
106.           <SDD-dd:PropertyConstraint>
107.             <SDD-dd:PropertyName>lb:fullName</SDD-dd:PropertyName>
108.             <SDD-dd:Value>eDMIS_Common, Version=1.0.0.1,
109.                 Culture=neutral, PublicKeyToken=28f514e8eeca027c
110.               </SDD-dd:Value>
111.           </SDD-dd:PropertyConstraint>
112.           <SDD-dd:PropertyConstraint>
113.             <SDD-dd:PropertyName>lb:fileVersion</SDD-dd:PropertyName>
114.             <SDD-dd:Value>1.2.1.0</SDD-dd:Value>
115.           </SDD-dd:PropertyConstraint>
116.         </SDD-dd:ResourceConstraint>
117.         <SDD-dd:ResourceConstraint id="eDMISWorkflowsAvailable"
118.                          resourceRef="LB_eCorrespondenceWorkFlowsAssembly">
119.           <SDD-dd:PropertyConstraint>
120.             <SDD-dd:PropertyName>lb:fullName</SDD-dd:PropertyName>
121.             <SDD-dd:Value>eDMIS_CorrespondenceWF, Version=1.0.0.1,
122.                 Culture=neutral, PublicKeyToken=28f514e8eeca027c
123.               </SDD-dd:Value>
124.           </SDD-dd:PropertyConstraint>
125.           <SDD-dd:PropertyConstraint>
126.             <SDD-dd:PropertyName>lb:fileVersion</SDD-dd:PropertyName>
```

```
127.            <SDD-dd:Value>1.0.3.0</SDD-dd:Value>
128.          </SDD-dd:PropertyConstraint>
129.        </SDD-dd:ResourceConstraint>
130.      </SDD-dd:Requirement>
131.      <!--Failu sistēmas objektu pārbaudes-->
132.      <SDD-dd:Requirement id="FSObjectsReqt" operation="install use">
133.        <!-- Direktoriji -->
134.        <SDD-dd:ResourceConstraint id="FileExchangeDirAccessible"
135.                                   resourceRef="FileExchangeDir">
136.          <SDD-dd:PropertyConstraint>
137.            <SDD-dd:PropertyName>sp:Writable</SDD-dd:PropertyName>
138.            <SDD-dd:Value>True</SDD-dd:Value>
139.          </SDD-dd:PropertyConstraint>
140.          <SDD-dd:PropertyConstraint>
141.            <SDD-dd:PropertyName>sp:Readable</SDD-dd:PropertyName>
142.            <SDD-dd:Value>True</SDD-dd:Value>
143.          </SDD-dd:PropertyConstraint>
144.        </SDD-dd:ResourceConstraint>
145.        <SDD-dd:ResourceConstraint id="WebRootDirAccessible"
146.                                   resourceRef="WebRootDir">
147.          <SDD-dd:PropertyConstraint>
148.            <SDD-dd:PropertyName>sp:Readable</SDD-dd:PropertyName>
149.            <SDD-dd:Value>True</SDD-dd:Value>
150.          </SDD-dd:PropertyConstraint>
151.        </SDD-dd:ResourceConstraint>
152.        <!--Faili-->
153.        <SDD-dd:ResourceConstraint id="Default_aspx_accessible"
154.                                   resourceRef="Files_Default_aspx">
155.          <SDD-dd:PropertyConstraint>
156.            <SDD-dd:PropertyName>sp:Exists</SDD-dd:PropertyName>
157.            <SDD-dd:Value>True</SDD-dd:Value>
158.          </SDD-dd:PropertyConstraint>
159.          <SDD-dd:PropertyConstraint>
160.            <SDD-dd:PropertyName>lb:fileHashSum</SDD-dd:PropertyName>
161.            <SDD-dd:Value>5fabeceb6b1f958617eebe98bd058a14</SDD-dd:Value>
162.          </SDD-dd:PropertyConstraint>
163.        </SDD-dd:ResourceConstraint>
164.      </SDD-dd:Requirement>
165.      <!--datubāze: vai var pieslēgties, vai tabula eksistē -->
166.      <SDD-dd:Requirement id="db_requirements" operation="install use">
167.        <SDD-dd:ResourceConstraint id="db_connectivity" resourceRef="edmisDB">
168.          <SDD-dd:RelationshipConstraint type="sp:connects" />
169.        </SDD-dd:ResourceConstraint>
170.        <SDD-dd:ResourceConstraint id="db_table_exists"
171.                                   resourceRef="UsersTable">
172.          <SDD-dd:PropertyConstraint>
173.            <SDD-dd:PropertyName>Exists</SDD-dd:PropertyName>
174.            <SDD-dd:Value>True</SDD-dd:Value>
175.          </SDD-dd:PropertyConstraint>
176.        </SDD-dd:ResourceConstraint>
177.      </SDD-dd:Requirement>
178.    </SDD-dd:Requirements>
179.  </SDD-dd:CompositeInstallable>
180.</SDD-dd:DeploymentDescriptor>
```

Koda piemērs 1.2. Pielāgots resursu deskriptors

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <xsd:schema targetNamespace="http://bank.lv/softwareProfile"
3.     xmlns="http://bank.lv/softwareProfile"
4.     xmlns:sp="http://docs.oasis-open.org/SDD/ns/starterProfile"
5.     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
6.     elementFormDefault="qualified"
7.     attributeFormDefault="unqualified"
8.     version="1.0">
9.   <xsd:complexType name="LB_RegionalSettings">
```

```
10.     <xsd:sequence>
11.       <xsd:element name="Locale" type="xsd:string"/>
12.       <xsd:element name="DateSeparator" type="xsd:string"/>
13.       <xsd:element name="ListSeparator" type="xsd:string"/>
14.       <xsd:element name="ShortDateFormat" type="xsd:string"/>
15.       <xsd:element name="LongTimeFormat" type="xsd:string"/>
16.       <xsd:element name="LongDateFormat" type="xsd:string"/>
17.       <xsd:element name="Currency" type="xsd:string"/>
18.       <xsd:any namespace="##other" processContents="lax"
19.       minOccurs="0" maxOccurs="unbounded"/>
20.     </xsd:sequence>
21.     <xsd:anyAttribute namespace="##other" processContents="lax"/>
22.   </xsd:complexType>
23.   <xsd:complexType name="LB_GlobalAssemblyCache">
24.     <xsd:sequence>
25.       <xsd:any namespace="##other"
26.       processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
27.     </xsd:sequence>
28.     <xsd:anyAttribute namespace="##other" processContents="lax"/>
29.   </xsd:complexType>
30.   <xsd:complexType name="LB_CodeAssembly">
31.     <xsd:sequence>
32.       <xsd:element name="FullName" type="xsd:string"/>
33.       <xsd:element name="Name" type="xsd:string"/>
34.       <xsd:element name="Version" type="xsd:string"/>
35.       <xsd:element name="Culture" type="xsd:string"/>
36.       <xsd:element name="PublicKeyToken" type="xsd:string"/>
37.       <xsd:element name="LastModified" type="xsd:dateTime"/>
38.       <xsd:element name="FileVersion" type="xsd:string"/>
39.       <xsd:any namespace="##other"
40.       processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
41.     </xsd:sequence>
42.     <xsd:anyAttribute namespace="##other" processContents="lax"/>
43.   </xsd:complexType>
44.   <xsd:complexType name="LB_SqlTable">
45.     <xsd:sequence>
46.       <xsd:element name="Name" type="xsd:string"/>
47.       <xsd:element name="Exists" type="xsd:string" />
48.       <xsd:any namespace="##other"
49.       processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
50.     </xsd:sequence>
51.   </xsd:complexType>
52. </xsd:schema>
```

2. pielikums

Programmatūras prasību profila realizācija, izmantojot speciāli definētu valodu

Šajā nodaļā iekļautajā piemērā redzamais dokuments apraksta tieši to pašu prasību kopu, ka 1. pielikumā, taču domēnspecifiskā valoda izstrādāta speciāli šim mērķim promocijas darba ietvaros.

Izstrādājot domēnspecifisku valodu prasību aprakstīšanai, prasības iespējams aprakstīt daudz "koncentrētāk" nekā *SDD* realizācijā. Šai īpatnībai ir vairāki izskaidrojumi.

Pirmkārt, atšķirībā no *SDD*, valodai nav jābūt resursorientēta. Tajā tiek aprakstīti nevis resursi un ierobežojumi to atribūtu vērtībām, bet gan izteikumi par resursiem. Katra prasība ir atsevišķs predikāts, kura patiesums tiek pārbaudīts vides testēšanas laikā,

Otrkārt, tā kā valodai nav jābūt universālai, t.i. tādai, kas būtu piemērota jebkāda tipa programmatūras komponenšu aprakstīšanai, tajā drīkst aprakstīt tikai tos programmatūras aspektus, kas attiecas uz izpildes vidi un prasībām pret to. 2. un 3. pielikumos apskatītās valodas attiecināmas uz *Microsoft .Net Framework* asamblejām un tādējādi pielietojamas tikai *MS Windows* operētājsistēmā un atsevišķās brīvā koda platformās, kas atbalsta *Mono-project* realizēto *.Net* versiju.

Treškārt, rodoties jauniem programmatūras izpildes prasību veidiem, to sintakse ir viegli definējama, paplašinot aprakstiem izmantojamo valodu. Tam izmanto atsevišķu *XSD* (*XML Schema* instances) failu, kurā aprakstīta tieši nepieciešamā predikāta sintakse.

Koda piemērā 2.1. redzamajā prasību dokumenta piemērā parādīta realizācija, kuras vispārējo sintaksi apraksta ļoti vienkārša struktūra:

- *SWProfile*
    - *Subject* (identificē programmatūru, ko apraksta šis profila dokuments)
    - *Requirements* (prasību saraksts)

Šo struktūru apraksta *XML Schema* dokuments (koda piemērā 2.1. atsauce uz to redzama koda 2. rindiņā), kas tiek izstrādāts vienu reizi šādas realizācijas pastāvēšanas laikā un pēc tam netiek mainīts.

Sintakse tām prasībām, kas aprakstītas „*Requirements*" elementa iekšienē, nav strikti definēta valodas definēšanas laikā, tādēļ tā aprakstīta ar citu *XML Schema* dokumentu palīdzību. Šajā piemērā visas prasības apkopotas vienā XML vārdkopā „*CommonRequirements*", kas kā *XML Schema* dokuments piesaistīta prasību profila dokumentam (profila dokumenta piemērā piesaiste shēmai redzama 3. rindiņā, savukārt pats shēmas dokuments redzams pielikumā „Prasību aprakstīšanas valodas sintakse"). Šī shēma apraksta šādu atsevišķu prasību sintaksi:

- *regionalSettingsDependency* - prasība pret reģionālajiem iestatījumiem, t.sk. lokāles vārds, īsais datuma formāts, decimālatdalītājs, sarakstu atdalītājs.
    - Prasību aprakstīšanas sintakse definēta XML shēmas dokumenta 12-13. un 55-60. rindiņā,

- Prasību lietošanas piemērs parādīts prasību dokumenta 30.-35. rindiņā

- *assemblyDependency* - prasība pret *.Net* asamblejām, identificējot tās pēc *fullName*.
  - Prasību aprakstīšanas sintakse definēta XML shēmas dokumenta 9. un 19-53. rindiņā,
  - Prasību lietošanas piemērs parādīts prasību dokumenta 19.-21. rindiņā
- *directoryDepdencency* - prasība pret noteiktu failu sistēmas direktoriju, identificējot to pēc takas un pieprasot tā eksistenci, neeksistēšanu, kā arī norādot piekļuves līmeni)
  - Prasību aprakstīšanas sintakse definēta XML shēmas dokumenta 10. un 62-68. rindiņā,
  - Prasību lietošanas piemērs parādīts prasību dokumenta 30.- 35. rindiņā
- *fileDependency* - prasība pret noteiktu failu, identificējot to pēc takas un pieprasot tā eksistenci, neeksistēšanu un pieejamību, arī – faila satura pārbaude
  - Prasību aprakstīšanas sintakse definēta XML shēmas dokumenta 11. un 70.-77. rindiņā
  - Prasību lietošanas piemērs parādīts prasību dokumenta 39. – 43. rindiņā
- prasības pret dažādiem SQL objektiem, t.sk. SQL serveris, SQL datubāze, SQL tabula
  - Prasību aprakstīšanas sintakse definē XML shēmas dokumenta 14. -16. un 79.-100. rindiņas
  - Prasību lietošanas piemēri parādīti prasību dokumenta 44.-56. rindiņa

Praktiski realizējot šādus prasību aprakstus, katru no tām ir lietderīgi aprakstīt atsevišķā XML vārdkopā, strikti aprakstot katra atsevišķa atribūta semantiku, taču šajā piemērā tās vienkāršības dēļ apvienotas vienā vārdkopā *cmn*.

Koda piemērs 2.1. Prasību dokuments

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <SWProfile xmlns="http://bank.lv/requirementProfile"
3.             xmlns:cmn="http://www.bank.lv/commonrequirements"
4.             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.             xsi:schemaLocation="http://bank.lv/requirementProfile
6.             requirementProfile.xsd http://www.bank.lv/commonrequirements
7.             commonRequirements.xsd ">
8.    <Subject
9.         type="assembly"
10.         fullName="eDMIS_Correspondence, Version=1.0.0.1,
11.         Culture=neutral, PublicKeyToken=28f514e8eeca027c"
12.    />
13.    <Requirements>
14.      <cmn:regionalSettingsDependency
15.        localeName="lv-LV"
16.        shortDatePattern="dd.MM.yyyy"
17.        listSeparator=";"
18.        decimalSymbol="," />
19.      <cmn:assemblyDependency deploymentKind="globalAssemblyCache"
20.        fullName="LB_CommonNet, Version=2.1.2.0,
21.                  Culture=neutral, PublicKeyToken=1b7e4d24feeb9a6c" />
22.      <cmn:assemblyDependency deploymentKind="globalAssemblyCache"
23.        fullName="eDMIS_Common, Version=1.0.0.1,
24.                  Culture=neutral, PublicKeyToken=28f514e8eeca027c"
25.        fileVersion="1.2.1.0" />
26.      <cmn:assemblyDependency deploymentKind="globalAssemblyCache"
27.        fullName="eDMIS_CorrespondenceWF, Version=1.0.0.1,
28.                  Culture=neutral, PublicKeyToken=28f514e8eeca027c"
29.        fileVersion="1.0.3.0" />
30.      <cmn:directoryDependency path="{$FileExchangeDir}">
31.        <cmn:requirement>exists</cmn:requirement>
```

```
32.       <cmn:requirement>readable</cmn:requirement>
33.       <cmn:requirement>writable</cmn:requirement>
34.     </cmn:directoryDependency>
35.     <cmn:directoryDependency path="{$WebRoot}">
36.       <cmn:requirement>exists</cmn:requirement>
37.       <cmn:requirement>readable</cmn:requirement>
38.     </cmn:directoryDependency>
39.     <cmn:fileDependency path="${WebRoot}\default.aspx"
40.                          hash='5fabeceb6b1f958617eebe98bd058a14'>
41.       <cmn:requirement>exists</cmn:requirement>
42.       <cmn:requirement>readable</cmn:requirement>
43.     </cmn:fileDependency>
44.     <cmn:sqlServerDependency
45.        connectionString='{$SqlConnectionString}'>
46.       <cmn:requirement>reachable</cmn:requirement>
47.     </cmn:sqlServerDependency>
48.     <cmn:sqlDbDependency
49.        connectionString='{$SqlConnectionString}'>
50.       <cmn:requirement>reachable</cmn:requirement>
51.     </cmn:sqlDbDependency>
52.     <cmn:sqlObjectDependency
53.        connectionString='{$SqlConnectionString}'
54.       type='table' name='UsersTable'>
55.       <cmn:requirement>exists</cmn:requirement>
56.     </cmn:sqlObjectDependency>
57.   </Requirements>
58. </SWProfile>
```

Prasību aprakstīšanas valodas sintakse, izmantojot XML Schema

Definējot prasību aprakstīšanas valodas atsevišķu apgabalu, kāds redzams šajā piemērā, ir būtiski pēc iespējas pilnvērtīgi dokumentēt katru prasību veidu. Šim mērķim var izmantot *XML Schema* piedāvātos *annotation* un *documentation* elementus. Izmantojot pietiekami kvalitatīvus XML izstrādes rīkus, tie spēj izmantot *documentation* blokos sniegto informāciju, lai izstrādātāju informētu par iespējamajiem elementiem, to atribūtiem un to ietekmi uz vides pārbaudes procesu. Vietas taupīšanas nolūkos koda piemērā 2.3. iekļautajā dokumentā pilnvērtīga dokumentācija iekļauta tikai *assemblyDependency* tipa elementam, dokumentāciju pievienojot *assemblyDependencyType* elementam un tā atribūtiem.

Koda piemērs 2.3. XML shēma

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <schema
3.     targetNamespace="http://www.bank.lv/commonrequirements"
4.     elementFormDefault="qualified" xmlns="http://www.w3.org/2001/XMLSchema"
5.     xmlns:cmn="http://www.bank.lv/commonrequirements">
6.
7.   <!--definē XML dokumenta instancē pieejamos elementus,
8.       katrs no tiem atsaucas uz shēmā definētu saliktu datu tipu-->
9.   <element name="assemblyDependency" type="cmn:assemblyDependencyType"/>
10.   <element name="directoryDepdencency" type="cmn:directoryDependencyType"/>
11.   <element name="fileDependency" type="cmn:fileDependencyType" />
12.   <element name="regionalSettingsDependency"
13.            type="cmn:regionalSettingsDependencyType"/>
14.   <element name="sqlDbDependency" type="cmn:sqlDbDependencyType"/>
15.   <element name="sqlServerDependency" type="cmn:sqlServerDependencyType"/>
16.   <element name="sqlObjectDependency" type="cmn:sqlObjectDependencyType"/>
17.
18.   <!--saliktie datu tipi, apraksta pieejamo elementu struktūru-->
19.   <complexType name="assemblyDependencyType">
```

```
20.      <annotation>
21.        <documentation>
22.          Atkarība no noteiktas koda asamblejas pieejamības izpildes vidē.
23.        </documentation>
24.      </annotation>
25.      <attribute name="fullName" type="string">
26.        <annotation>
27.          <documentation>
28.            .Net asamblejas pilnais nosaukums formā
29.            '&lt;nosaukums&gt;, Version=&lt;versija (#.#.#.#)&gt;,
30.            Culture=&lt;lokāle, ja ir&gt;,
31.            PublicKeyToken=&lt;publiskās atslēgas jaucējsumma&gt;'
32.          </documentation>
33.        </annotation>
34.      </attribute>
35.      <attribute name="deploymentKind"
36.                 type="cmn:deploymentKindType">
37.        <annotation>
38.          <documentation>
39.            Veids, kā asamblejai jābūt izvietotai - globālajā asambleju
40.            glabātuvē (GAC), tajā pašā direktorijā, kur izvietota programma,
41.            bin direktorijā blakus programmai.
42.          </documentation>
43.        </annotation>
44.      </attribute>
45.      <attribute name="fileVersion" type="string">
46.        <annotation>
47.          <documentation>
48.            Ja šis atribūts norādīts, tiks veikta pārbaude, vai atrastās DLL
49.            bibliotēkas deklaratīvais atribūts fileVersion atbilst prasītajam.
50.          </documentation>
51.        </annotation>
52.      </attribute>
53.    </complexType>
54.
55.    <complexType name="regionalSettingsDependencyType">
56.      <attribute name="localeName" type="string" />
57.      <attribute name="shortDatePattern" type="string" />
58.      <attribute name="listSeparator" type="string" />
59.      <attribute name="decimalSymbol" type="string" />
60.    </complexType>
61.
62.    <complexType name="directoryDependencyType">
63.      <sequence>
64.        <element name="requirement" minOccurs="1"
65.                 type="cmn:fsObjectRquirement" maxOccurs="unbounded" />
66.      </sequence>
67.      <attribute name="path" type="string" />
68.    </complexType>
69.
70.    <complexType name="fileDependencyType">
71.      <sequence>
72.        <element name="requirement" type="cmn:fsObjectRquirement"
73.                 minOccurs="1" maxOccurs="unbounded" />
74.      </sequence>
75.      <attribute name="hash" type="string" />
76.      <attribute name="path" type="string" />
77.    </complexType>
78.
79.    <complexType name="sqlServerDependencyType">
80.      <sequence>
81.        <element name="requirement" type="cmn:dbObjectRequirement" />
82.      </sequence>
83.      <attribute name="connectionString" type="string" />
84.    </complexType>
85.
```

```
86.   <complexType name="sqlDbDependencyType">
87.     <sequence>
88.       <element name="requirement" type="cmn:dbObjectRequirement" />
89.     </sequence>
90.     <attribute name="connectionString" type="string" />
91.   </complexType>
92.
93.   <complexType name="sqlObjectDependencyType">
94.     <sequence>
95.       <element name="requirement" type="cmn:dbObjectRequirement" />
96.     </sequence>
97.     <attribute name="name" type="string" />
98.     <attribute name="type" type="cmn:sqlObjectType" />
99.     <attribute name="connectionString" type="string" />
100.  </complexType>
101.
102.  <!--Vienkāršie datu tipi, kas realizē saliktajiem datu tipiem
103.      nepieciešamos vērtību pārskaitījumus - tie kalpo kā izvēles
104.      salikto datu tipu atribūtu vērtībām.-->
105.  <simpleType name="fsObjectRquirement">
106.    <restriction base="string">
107.      <enumeration value="exists" />
108.      <enumeration value="readable" />
109.      <enumeration value="writable" />
110.      <enumeration value="executable" />
111.    </restriction>
112.  </simpleType>
113.
114.  <simpleType name="deploymentKindType">
115.    <restriction base="string">
116.      <enumeration value="globalAssemblyCache" />
117.      <enumeration value="runDirectory" />
118.      <enumeration value="binFolder" />
119.    </restriction>
120.  </simpleType>
121.
122.  <simpleType name="dbObjectRequirement">
123.    <restriction base="string">
124.      <enumeration value="exists" />
125.      <enumeration value="reachable" />
126.    </restriction>
127.  </simpleType>
128.
129.  <simpleType name="sqlObjectType">
130.    <restriction base="string">
131.      <enumeration value="table" />
132.      <enumeration value="storedProcedure" />
133.      <enumeration value="function" />
134.      <enumeration value="user" />
135.    </restriction>
136.  </simpleType>
137.</schema>
```

3. pielikums

Parametrizējamu konfigurācijas lielumu saistību aprakstīšana, izmantojot XML

Programmatūras prasību dokumenta ķermenī iespējams pielietot *parametrus* – mainīgos, kuru vērtība tiek noteikta vides pārbaudes laikā. Prasību dokumentā šie mainīgie tiek pierakstīti formātā `{$mainīgā_vārds}`. Parametra piemērs redzams 2.1. koda piemēra 49. rindā, kur minēta atsauce uz parametru *SqlConnectionString*).

Parametru vērtību noskaidrošana ir veicama pārbaudes veikšanas laikā, lai parametru vērtības tiktu nolasītas no tās vides, kurā tiek veiktas pārbaudes, nevis no datora, kurā ticis sagatavots programmatūras izpildes prasību profils. Atkarībā no parametra, kura vērtību nepieciešams noskaidrot, vides testēšanas koordinatoram jāspēj parametra vērtību lasīt ar dažādām metodēm. Realizējot vides testēšanas metodiku vienā no promocijas darba pētījumā iesaistītajām organizācijām, kā vieni no būtiskākajiem parametru nolasīšanas veidiem tika atzīti šādi mehānismi:

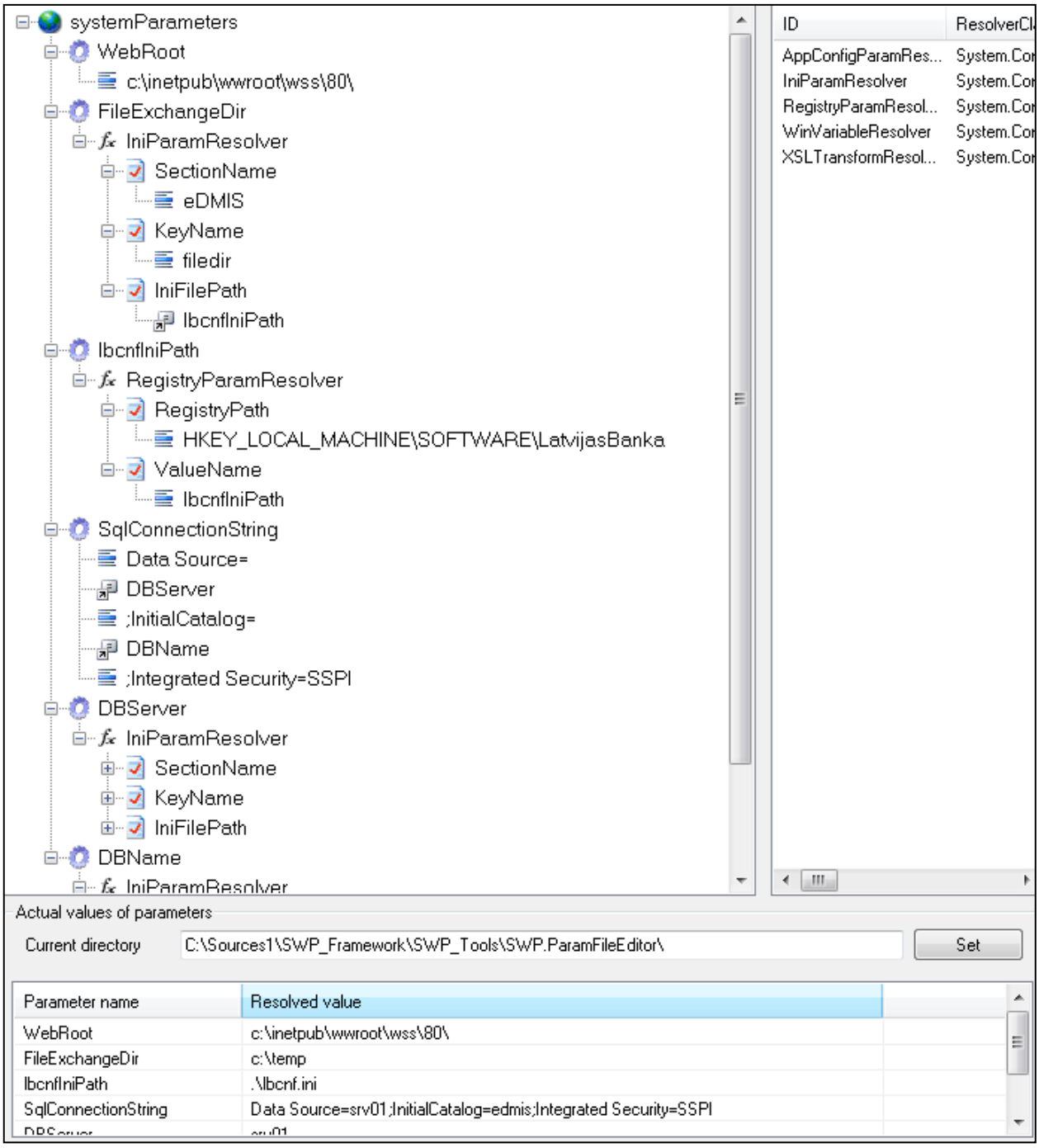
- vērtības nolasīšana no konfigurācijas *.ini* faila,
- vērtības nolasīšana no *MS Windows* reģistra,
- vērtību nolasīšana no sistēmas mainīgajiem (tādi kā, piemēram, %TEMP% u.c.),
- vērtības nolasīšana no brīva formāta XML faila (pielietojot XML transformāciju).

Praksē vērtību nolasīšanas mehānismi tikuši izstrādāti kā atsevišķas programmatūras koda klases, kas visas mantotas no vienas un tās pašas bāzes klases Koda piemērā 3.1 (tālāk šajā pielikumā) šie nolases mehānismi aprakstīti 108 – 123. rindiņā.

Sākotnējās realizācijās iekļauta valoda, kas pieļauj trīs parametru veidus (tā kā parametru faila struktūra ir sarežģīta, veidu demonstrēšanai izmantots šī darba izstrādes ietvaros veidotās programmas „*SWP Parameter File Editor*" ekrāna izskats):

- atomāri parametri, kuru vērtība ir konstanta teksta virkne (skat. p.3.1. att. mainīgo *WebRoot*); šādus parametrus izmanto, lai samazinātu iespēju kļūdīties, atkārtoti aprakstot garas un vienveidīgas simbolu virknes;
- aprēķināmi mainīgie, kuru vērtība noskaidrojama ar kādu no vērtību nolases mehānismiem (skat. p.1.3.1. att. mainīgo *lbcnfIniPath* – tā vērtību nolasa, izmantojot *MS Windows* reģistra nolasīšanas mehānismu); mehānismam nododamie parametri var saturēt atsauces uz parametru vērtībām (piemēram, „nolasīt konfigurācijas vērtību no XML faila, kura atrašanās vieta atrodama *MS Windows* reģistra zarā *HKEY_CURRENT_USER\Volatile Environment* atslēgā *Appdata*);
- simbolisku vērtību konkatenācija, kurā iekļaujamas:
    - atomāras simbolu virknes,

- atsauces uz citu mainīgo vērtībām (skat., p.3.1. att. mainīgo *SqlConnectionString*, kas atsaucas uz mainīgajiem *DBServer* un *DBName*),
- no vides nolasāmu vērtības.



p.3.1.att. **Parametru rediģēšanas programmas ekrāna izskats**

Valodā, kas apraksta vides parametrus, ieviesti šādi elementi:

- *systemParameter* – apraksta sistēmas parametru, tam ir divi būtiski apakšelementi (piemēram, 3.1. koda piemēra koda 5. rindiņā),
  - *name* - parametra nosaukums, atomāra simbolu virkne,
  - *value* - parametra vērtība, drīkst būt atomāra vai salikta virkne,

- *resolve-param* – iedarbina kādu no vērtību nolasīšanas mehānismiem, lai nolasītu parametra vērtību (piemēram, koda 12. rindiņā),
    - *resolverId* – vērtību nolasīšanas mehānisms, kas pielietojams šī parametra nolasīšanai. Šai vērtībai jāsakrīt ar kādu no to pārbaudes mehānismu identifikatoriem, kas pārskaitīti *parameterResolvers* sadaļā,
    - *propertyPack* – katram vērtību nolasīšanas mehānismam var būt nepieciešami dažādi argumenti, lai varētu nolasīt attiecīgo vides parametru (piemēram, *.ini* faila nolasīšanai būtiski, kur atrodams fails, kurā sekcijā jāmeklē vērtība un kāds ir vērtības nosaukums, savukārt citiem nolases mehānismiem argumentu komplekts ir pavisam citāds). Koda piemērā šāda konstrukcija redzama, piemēram, 64. rindiņā. *PropertyPack* elements satur vienu vai vairākus *Property* elementus, katram no tiem definējot *Name* un *Value* apakšelementus, kuru vērtības neprasa plašāku skaidrojumu,
- *param-ref* – atsauce uz kāda cita parametra pašreizējo vērtību. Labs piemērs redzams 54. – 56. rindiņā, kur tiek aprakstīta parametra *SqlConnectionString* vērtība – tā ir konkatenācija no atomārām simbolu virknēm un citu mainīgo vērtībām,
- *parameterResolvers* – pārskaita sistēmā pieejamos parametru nolasīšanas mehānismus, katram no tiem piešķirot identifikatoru, ko var izmantot *resolve-param* direktīvas. *ParameterResolvers* sekcija sastāv no viena vai vairākiem *parameterResolver* elementiem
    - *parameterResolver* – atsauce uz atsevišķu parametru nolasītāja mehānismu. Katram mehānismam piekārtota koda asambleja un klase, kura realizē vērtību nolasīšanas mehānismu.

Koda piemērs 3.1. Parametru aprakstīšana

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <parameterDescriptionFile
3.   xmlns="http://naivist.net/schemas/parameterDescription/">
4.   <systemParameters>
5.     <systemParameter>
6.       <name>WebRoot</name>
7.       <value>c:\inetpub\wwroot\wss\80\</value>
8.     </systemParameter>
9.     <systemParameter>
10.       <name>FileExchangeDir</name>
11.       <value>
12.         <resolve-param>
13.           <resolverId>IniParamResolver</resolverId>
14.           <propertyPack>
15.             <property>
16.               <name>SectionName</name>
17.               <value>eDMIS</value>
18.             </property>
19.             <property>
20.               <name>KeyName</name>
21.               <value>filedir</value>
22.             </property>
23.             <property>
24.               <name>IniFilePath</name>
25.               <value>
```

```
                <param-ref name="lbcnfIniPath" />
              </value>
            </property>
          </propertyPack>
        </resolve-param>
      </value>
    </systemParameter>
    <systemParameter>
      <name>lbcnfIniPath</name>
      <value>
        <resolve-param>
          <resolverId>RegistryParamResolver</resolverId>
          <propertyPack>
            <property>
              <name>RegistryPath</name>
              <value>HKEY_LOCAL_MACHINE\SOFTWARE\LatvijasBanka</value>
            </property>
            <property>
              <name>ValueName</name>
              <value>lbcnfIniPath</value>
            </property>
          </propertyPack>
        </resolve-param>
      </value>
    </systemParameter>
    <systemParameter>
      <name>SqlConnectionString</name>
      <value>
        Data Source=<param-ref name="DBServer" />;
        InitialCatalog=<param-ref name="DBName" />;
        Integrated Security=SSPI
      </value>
    </systemParameter>
    <systemParameter>
      <name>DBServer</name>
      <value>
        <resolve-param>
          <resolverId>IniParamResolver</resolverId>
          <propertyPack>
            <property>
              <name>SectionName</name>
              <value>eDMIS</value>
            </property>
            <property>
              <name>KeyName</name>
              <value>Server</value>
            </property>
            <property>
              <name>IniFilePath</name>
              <value>
                <param-ref name="lbcnfIniPath" />
              </value>
            </property>
          </propertyPack>
        </resolve-param>
      </value>
    </systemParameter>
    <systemParameter>
      <name>DBName</name>
      <value>
        <resolve-param>
          <resolverId>IniParamResolver</resolverId>
          <propertyPack>
            <property>
              <name>SectionName</name>
              <value>eDMIS</value>
```

```
92.             </property>
93.             <property>
94.               <name>KeyName</name>
95.               <value>Database</value>
96.             </property>
97.             <property>
98.               <name>IniFilePath</name>
99.               <value>
100.                <param-ref name="lbcnfIniPath" />
101.              </value>
102.            </property>
103.          </propertyPack>
104.        </resolve-param>
105.      </value>
106.    </systemParameter>
107.  </systemParameters>
108.  <parameterResolvers>
109.    <parameterResolver id="IniParamResolver">
110.      <resolverAssembly>SWP.SysParams, Version=1.0.0.0,
111.       Culture=neutral, PublicKeyToken=28f514e8eeca027c</resolverAssembly>
112.      <resolverClass>
113.        System.Configuration.SWP.SysParams.CoreResolvers.IniParamResolver
114.      </resolverClass>
115.    </parameterResolver>
116.    <parameterResolver id="RegistryParamResolver">
117.      <resolverAssembly>SWP.SysParams, Version=1.0.0.0,
118.       Culture=neutral, PublicKeyToken=28f514e8eeca027c</resolverAssembly>
119.      <resolverClass>
120.       System.Configuration.SWP.SysParams.CoreResolvers.RegistryParamResolver
121.      </resolverClass>
122.    </parameterResolver>
123.  </parameterResolvers>
124.</parameterDescriptionFile>
```

4. pielikums

Testu koordinatoram pieejamo vides pārbaudītāju apraksts ar XML

Aprakstot vides testēšanas koordinatoram pieejamos testēšanas moduļus, tiek izmantots „pierakstīšanās" princips – testēšanas moduļu sarakstā aprakstītas programmatūras klases, kas spēj veikt noteiktas vides pārbaudes, katra no tām definē atsevišķus prasību veidus, ko tā spēj pārbaudīt. Koda piemērs demonstrē 2. pielikumā demonstrētā prasību profila pārbaudei nepieciešamos rīkus. Katra rīka aprakstīšanai izmantots atsevišķs *validator* elements, kas satur vienu vai vairākus *subscripiton* apakšelementus. *Subscription* ir atsauce uz noteiktu prasību, tās pareiza darbība realizēta ar *XML* vārdkopu palīdzību (requirementNodeName satur atsauci uz vārdkopu un vārdkopā definētu elementu).

```
1.  <?xml version="1.0" encoding="utf-8" ?>
2.  <validatorDefinitions
3.    xmlns="http://www.bank.lv/validatordefinitions"
4.    xmlns:cmn="http://www.bank.lv/commonrequirements">
5.    <validator
6.      title="directoryDependency"
7.      assembly="Extensionvalidators, Version=1.0.0.0,
8.                Culture=neutral, PublicKeyToken=28f514e8eeca027c"
9.      class="LB.SelfTest.DirectoryDependency">
10.     <subscription requirementNodeName="cmn:directoryDependency" />
11.     <subscription requirementNodeName="cmn:fileShareDependency" />
12.   </validator>
13.
14.   <validator
15.     title="fileDependency"
16.     assembly="Extensionvalidators, Version=1.0.0.0,
17.               Culture=neutral, PublicKeyToken=28f514e8eeca027c"
18.     class="LB.SelfTest.FileDependency">
19.     <subscription requirementNodeName="cmn:fileDependency" />
20.   </validator>
21.
22.   <validator
23.     title="localeDependency"
24.     assembly="Extensionvalidators, Version=1.0.0.0,
25.               Culture=neutral, PublicKeyToken=28f514e8eeca027c"
26.     class="LB.SelfTest.LocaleDependency">
27.     <subscription requirementNodeName="cmn:localeDependency" />
28.   </validator>
29.
30.   <validator
31.     title="assemblyDependency"
32.     assembly="Extensionvalidators, Version=1.0.0.0,
33.               Culture=neutral, PublicKeyToken=28f514e8eeca027c"
34.     class="LB.SelfTest.AssemblyDependency">
35.     <subscription requirementNodeName="cmn:assemblyDependency" />
36.   </validator>
37.
38.   <validator
39.     title="sqlObjectDependency"
40.     assembly="Extensionvalidators, Version=1.0.0.0,
41.               Culture=neutral, PublicKeyToken=28f514e8eeca027c"
42.     class="LB.SelfTest.SmtpDependency">
43.     <subscription requirementNodeName="cmn:sqlServerDependency" />
44.     <subscription requirementNodeName="cmn:sqlDbDependency" />
```

```
45.      <subscription requirementNodeName="cmn:sqlObjectDependency" />
46.    </validator>
47. </validatorDefinitions>
```