LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

IVO ODĪTIS

BIZNESA PROCESU IZPILDES LAIKA VERIFICĒŠANA

PROMOCIJAS DARBS

Doktora grāda iegūšanai datorzinātnes nozarē
Apakšnozare: datu apstrādes sistēmas un datortīkli

ANOTĀCIJA

Promocijas darbā ir risināta biznesa procesu izpildes laika verificēšanas problēma. Autors piedāvā ārēju verificēšanas risinājumu, kas ļauj verificēt procesu izpildi, neiejaucoties procesu darbībā un neiekļaujot sistēmu papildus instrumentāciju.

Darbā piedāvātais risinājums paredz verificēšanas mehānisma izveidi, kas darbojas paralēli verificējamajam procesam. Verificēšana tiek veikta atbilstoši katra procesa verificēšanas aprakstam. Autors piedāvā lietot domēnspecifisku verificēšanas apraksta valodu. Aprakstā tiek definēti notikumi, kas apliecina korektu procesa izpildi, to iestāšanās secība un izpildes laika ierobežojumi. Attiecīgi verificēšanas mehānisms novēro vides notikumus un verificē to iestāšanās secības un laika atbilstību verificēšanas aprakstam.

Piedāvātajā verificēšanas mehānismā tiek lietotas divu tipu komponentes: centralizēts kontrolieris un vairāki aģenti notikumu reģistrēšanai. Aģentu uzdevums ir fiksēt verificēšanai nepieciešamos notikumus. Savukārt, kontrolieris, lietojot verificēšanas aprakstus, pieprasa aģentiem novērot nepieciešamos notikumus un, saņemot atbildes par fiksētajiem notikumiem, konstatē procesa instanču izpildes korektumu. Tādējādi autors piedāvā veidot asinhronu multiaģentu procesu verificēšanas mehānismu.

Verificēšanas mehānisma prototips tika aprobēts ar reālas sistēmas palīdzību, darbinot to testēšanas vidē. Lai arī prototips bija vienkārši veidots, tas sniedza pozitīvu rezultātu un parādīja, ka mehānisms nodrošina intensīvu procesu verificēšanu, kuru izpildes gaita ir daudz straujāka par tipiskiem biznesa procesiem. Aprobācijas laikā tika izvērtēta notikumu un taimeru bāzētu aģentu radīta virsslodze: tā bija novērojama, tomēr nozīmīgu iespaidu uz novērojamiem procesiem neatstāja. Ar prototipu tika apstiprināta piedāvātā risinājuma dzīvotspēja un lietojamība.

ABSTRACT

Business process run-time verification problem is the main objective of this thesis. The author suggests developing external multi-agent solution for business process run-time verification. It would run as parallel process and would not require modifying process under verification.

According to the author's idea, verification could be done by verification mechanism that uses process verification description. For this purpose domain specific process verification description language was developed by the author. The language includes concepts helping define events that confirm process step execution, event execution order and process timing restrictions. Respectively, the proposed verification mechanism verifies order and timing of the process execution.

There are two types of components used in the verification mechanism: centralized controller and decentralized agents. Agents are built to notice the events occurred during process execution. While the controller is the main verification component, which analyze process verification descriptions, requests agents to check the process events and verifies process execution correctness upon reception of event notices from agents.

A prototype including process verification controller and two agents was developed so as to prove the concept and specify the details of the proposed model. Though the prototype was rather simple, it provided positive results and confirmed the author's idea that the proposed verification mechanism can be broadly applied. Actually even more dynamic processes can be verified using the author’s solution. Likewise the prototype showed that the process description language can be improved. However, the prototype validated the usability of the proposed solution.

SATURS

Ievads ... 5

1. Problēmas identifikācija ... 18

1.1. Programmatūras kvalitātes nodrošināšanas veidi ... 18

1.1.1. Statiskā testēšana ... 19

1.1.2. Dinamiskā testēšana ... 20

1.1.3. Izpildes laika verificēšana ... 20

1.2. Biznesa procesu verificēšana ... 22

1.3. Autonomiskās sistēmas un viedtehnoloģijas ... 23

1.4. Biznesa procesu verificēšanas problēma ... 25

1.4.1. Verificēšanas sarežģītība ... 26

1.4.2. Verificēšanas atgriezeniskā saite ... 26

1.4.3. Verificēšanas varbūtiskais raksturs ... 28

1.4.4. Prasības procesu verificēšanai ... 28

2. Saistītie pētījumi ... 30

2.1. Iebūvēti izpildes laika verifikācijas risinājumi ... 30

2.1.1. Dokumentu plūsmu verificēšana ... 31

2.1.2. Paštestēšana ... 32

2.2. Ārēji izpildes laika verificēšanas risinājumi ... 36

2.2.1. Programmatūras izpildes vides testēšana ... 37

2.2.2. SOA risinājums WSCoL ... 40

2.3. Kombinēti izpildes laika verificēšanas risinājumi ... 43

2.3.1. Trases analizatori ... 45

2.3.2. Universāls analizators - ProM ietvars ... 46

2.3.3. Verificēšanas rīku ģenerēšana ... 48

2.4. Esošo risinājumu sniegtā pieredze ... 50

3. Asinhrons multiaģentu verificēšanas mehānisms ... 52

3.1. Verificējamie kritēriji ... 52

3.2. Mehānisma apraksts ... 53

3.3. Programmatūras komponentes ... 55
3.3.1. Kontrolieris ... 55
3.3.2. Aģenti ... 59
3.4. Verificēšanas apraksta valoda ... 63
3.4.1. Verificēšanas apraksta valodas elementi ... 65
3.4.2. Verificēšanas apraksta valodas sintakse ... 67
3.4.3. Verificēšanas apraksta valodas piemērs ... 71
3.5. Verificēšanas mehānisma darbības ierobežojumi ... 73
3.5.1. Kontroliera ierobežojumi ... 74
3.5.2. Aģentu ierobežojumi ... 75
4. Aprobācija ... 77
4.1. Implementācijas apraksts ... 78
4.1.1. Kontroliera implementācija ... 78
4.1.2. Aģentu implementācija ... 79
4.2. Reālu sistēmu verificēšana ... 79
4.2.1. Piemēra process ... 79
4.2.2. Iegūtie rezultāti ... 83
4.3. Aprobēšana simulētā vidē ... 85
4.3.1. Apskatītie piemēra procesi ... 86
4.3.2. Iegūtie rezultāti ... 87
4.4. Aprobācijas secinājumi ... 88
Secinājumi ... 90
Literatūras saraksts ... 95
Pielikums ... 100

IEVADS

Darbs ir veltīts biznesa procesu izpildes laika verificēšanai. Šī darba ietvaros ar izpildes laika verificēšanu tiek saprasta biznesa procesu analīze un darbības korektuma novērtēšana to izpildes laikā lietošanas vidē. Korektuma novērtēšanai var tikt lietota gan sistēmā iebūvēta instrumentācija, gan ārēja sistēmas darbības notikumu novērošana. Lai arī parasti izpildes laika verificēšana tiek attiecināta uz programmatūru un iekārtām, tomēr autors to piedāvā attiecināt arī uz biznesa procesiem - arī tie var tikt verificēti to izpildes laikā, tādējādi novērtējot to izpildes pareizību.

Sākot ar pirmo datoru izveidošanu, programmatūras izstrāde ir kļuvusi par nozīmīgu tehnoloģisku industriju. Šobrīd tai jau ir 70 gadu ilga vēsture, un šajos gados informācijas tehnoloģiju un telekomunikāciju industrija (turpmāk tekstā - IT) ir kļuvusi plaša, ietekmējot gandrīz visas modernās pasaules dzīves jomas. Īpaši strauja IT attīstība sākās pagājušā gadsimta 80-to gadu vidū, sākoties personālo datoru ērai. Pateicoties skaitļošanas tehnikas izmēru un cenas straujam samazinājumam, datori un attiecīgi arī programmatūra ir sastopama ik uz soļa. Turklāt patlaban pat ļoti ikdienišķas lietas ir datorizētas. Pirms 30 gadiem cilvēki pat nevarēja iedomāties, ka tajās ir nepieciešams iebūvēt skaitļošanas ierīces. Piemēram, auto riteņos tiek uzstādīti ventiļi ar raidītājiem, kuri nosūta informāciju automašīnas galvenajam datoram. Šobrīd šādas ierīces pieejamas arī velosipēdiem [COX12]: velobraucēja mobilajā telefonā tiek ziņots par gaisa spiedienu velosipēda riepās. Mūsdienās gandrīz vairs nepastāv veļas mazgājamās mašīnas vai elektriskās plītis, kurās nebūtu iebūvēts dators. Tomēr datorsistēmas dažādās ikdienišķās ierīcēs ir tikai aisberga virsotne. Lielākā daļa dažādu ražošanas uzņēmumu, sabiedriskās pārvaldes organizāciju u.c. institūciju balstās uz datorsistēmu darbību. Turklāt tās parasti nav tikai viena vai divas sistēmas, bet gan vairāku sistēmu kopums, kas savstarpēji apmainās ar informāciju.

Par mūsdienu mainīgo programmatūras izpildes vidi runā daudzi autori [HORN01]. Ņemot vērā, cik daudzi sabiedrības procesi tiek balstīti uz datorsistēmu darbību, to kļūdaina darbība potenciāli var un mēdz radīt nopietnas problēmas. Kāds no autoriem sniedz novērtējumu:

[WU13]: "Programmatūras sistēmu korektuma verificēšana uzstāda daudzus nozīmīgus izaicinājumus, kurus nākas risināt. Salīdzinoši nesen[1] Nacionālais standartu un tehnoloģiju institūts (*National Institute of Standards and Technology* jeb NIST) ziņoja, ka saskaņā ar aplēsēm katru gadu programmatūras radītās kļūdas rada 59.6 miljardus USD lielus

[1] ziņojums sagatavots 2002. gadā (promocijas darba autora piezīme)

zaudējumus [PROJ02][2]. Jaunākā rakstā ir norādīts, ka daži kļūdainas programmatūras darbības incidenti ir radījuši nozīmīgus monetārus zaudējumus [CHAR05]. Piemēram, Lielbritānijas nodokļu dienests 2004. un 2005. gadā neieņēma 3.45 miljardus USD programmatūras kļūdu dēļ. Turklāt programmatūras kļūdas rada ne tikai naudas zaudējumus."

Pat pieņemot, ka tehniskā un operētājsistēmu bāze ir pietiekami stabila, servisu orientētu sistēmu (servisu orientēta arhitektūra jeb SOA) veidošana daudzviet ir likusi saskarties ar programmatūras mainību un nozīmīgu ārējās vides mainības ietekmi uz izstrādāto programmatūru. Piemēram, pirms 30 gadiem izveidota programmatūra reti sadarbojās ar citām informācijas sistēmām, un, ja sadarbojās, tad šīs sistēmas parasti tika uzturētas viena uzņēmuma ietvaros un kvalitātes nodrošināšanai pilnībā pietika ar vienreizēju šo sistēmnu savstarpējās sadarbības testēšanu. Attīstoties SOA un "mākoņskaitļošanai", liela daļa no informācijas sistēmām izmanto citu sistēmu piedāvātus servisus, kuri tiek uzturēti ārpus sistēmu uzturošā uzņēmuma. Attiecīgi ir gandrīz neiespējami pārbaudīt uzturamo sistēmu sadarbību ar ārējiem servisiem, un bieži vien pat nav pieejamas ziņas par šo servisu pārmaiņām. Līdz ar to lietotāji vairs nevar paļauties tikai uz sistēmu izstrādes un ieviešanas laika testēšanu, bet sistēmu darbība ir jāverificē visā to lietošanas laikā, lai savlaicīgi pamanītu kādas sistēmas vai pat veselu procesu izpildes anomālijas.

[LEUC09]: "Milzīgās sistēmās dažas to komponentes (aparatūra) regulāri var "iziet no ierindas". Lietojot izpildes laika monitorēšanu vēlamās darbības uzraudzīšanai, šādas problēmas var tikt pamanītas un savlaicīgi paziņotas. Papildus kods šādā situācijā var reaģēt uz kļūdu, piemēram, parādot atbilstošu kļūdas paziņojumu. Kaut kādā nozīmē šī ir kļūdu apstrādes koncepcijai atbilstoša ideja, tomēr šajā pieejā kļūdu pārtraukumi tiek definēti kā sistēmu izpildes korektuma kritēriju pārkāpumi."

Iepriekšējā citāta autors norāda uz kļūdu apstrādes un izpildes laika verificēšanas mehānismu atšķirību. Kļūdu apstrāde tiek iekļauta jebkurā programmatūrā un tā nodrošina sistēmā konstatētas kļūdas (parasti uzsāktās funkcijas izpildes pārtraukšanās) paziņošanu lietotājam un, iespējams, arī izstrādātājiem un pēc iespējas detalizētu reģistrēšanu kļūdu žurnālā. Savukārt, izpildes laika verificēšanā konstatētās neatbilstības var arī neizraisīt programmatūras darbības pārtraukumus. Programmatūra turpina darboties, bet izpildes rezultāts nav korekts - tas neatbilst korektuma kritērijiem.

Kā jau autors norādīja iepriekš, mūsdienās lielākajā daļā uzņēmumu un organizāciju uzdevumu veikšanai tiek lietoti datori un programmatūra. Ņemot vērā iespējamās programmatūras darbības problēmas un neatbilstības, tās potenciāli var atstāt negatīvu ietekmi uz uzņēmumu biznesa procesu izpildi. Turklāt nepareizu biznesa procesu izpildi bieži vien var

[2] zaudējumu novērtējums aptver tikai ASV (promocijas darba autora piezīme)

izraisīt ne tikai programmatūras un tās darbības vides problēmas, bet arī biznesa procesu izpildītāju, cilvēku, pieļautās kļūdas. Piemēram, laikā nenosūtīta atbildes vēstule vai neapstiprināts dokuments var radīt ne tikai materiāla, bet arī juridiska rakstura problēmas. Tādējādi no uzņēmumu viedokļa biznesa procesu izpildes uzraudzība (verificēšana) ir ne mazāk nozīmīga kā tieša programmatūras darbības verificēšana.

Biznesa procesu izpildes laika verificēšana ir šā darba autora zinātniskā novitāte. Līdz šim apskatītajā literatūrā nav sastapts analogs risinājums, kurš nodrošinātu ārēju biznesa procesu izpildes laika verificēšanu. Darba autora pētījuma starprezultāti ir aprakstīti vairākās publikācijās [ODIT10, ODIT14, ODIT15, ODIT16]. Tāpat darba rezultāti ir apskatīti viedtehnoloģiju kopējā kontekstā [BICE15A, BICE15B, BICE16].

Darba mērķi un uzdevumi

Darba vispārīgais mērķis ir izstrādāt metodi (praktiski implementējamu mehānismu) biznesa procesu verificēšanai, lai nodrošinātu korektu pilnībā vai tikai daļēji automatizētu biznesa procesu izpildi. Promocijas darba konkrētais mērķis ir izveidot praktiski lietojamu biznesa procesu izpildes laika verificēšanas mehānismu, kas ļautu verificēt biznesa procesu korektu izpildi.

Promocijas darbam izvirzīti šādi uzdevumi:

- izstrādāt procesu verifikācijas mehānisma arhitektūru;
- definēt biznesa procesu verifikācijas mehānismu;
- izveidot procesu izpildes apraksta valodu vai arī pielāgot kādu no esošajām procesu apraksta valodām;
- izstrādāt verificēšanas mehānisma prototipu;
- aprobēt izstrādāto prototipu reālos praktiskās lietošanas projektos;
- novērtēt izveidotā mehānisma pielietojamības robežas.

Darbā izvirzītās tēzes

Promocijas darba ietvaros ir izvirzītas šādas tēzes:

- biznesa procesiem, līdzīgi kā informācijas sistēmām, kā kvalitātes nodrošināšanas risinājums ir pielietojama izpildes laika verificēšana;
- biznesa procesu verificēšanu var veikt procesu izpildes laikā līdzīgi kā tiek verificēta programmatūras darbība, turklāt ir iespējams izveidot mehānismu,

kas būtu pielietojams lielai biznesa procesu daļai neatkarīgi no to implementācijas;

- biznesa procesu izpildes laika verificēšanas risinājums neprasa lielus skaitļošanas resursus un ir salīdzinoši vienkārši implementējams un paplašināms;
- biznesa procesu verificēšanas mehānisms ir paplašināms uz dažādām biznesa procesus realizējošām platformām.

Darba struktūra

Darba autors savu pētījumu izklāsta četrās nodaļās. **Pirmajā** autors iepazīstina ar problēmas apgabalu, sākot ar programmatūras kvalitātes nodrošināšanas tehniskajiem risinājumiem un beidzot ar biznesa procesu verificēšanai izvirzāmajām prasībām. Darba autors šo jautājumu skata nedaudz netipiski, iedalot visas tehniskās kvalitātes nodrošināšanas pieejas trīs lielās grupās: statiskā testēšana, dinamiskā testēšana un izpildes laika verificēšana. Tās varētu definēt šādi:

- *statiskā testēšana* paredz testējamās programmatūras korektuma pārbaudi, programmatūru nedarbinot (piemēram, šādi to veic programmatūras kompilatori);
- *dinamiskā testēšana* pārbauda programmatūras korektumu, to darbinot testa vidē saskaņā ar testēšanas scenārijiem, padodot programmatūrai testa datus un pārliecinoties par saņemto rezultātu atbilstību prasībām;
- *izpildes laika verificēšana* ir programmatūras darbības uzraudzība lietošanas vidē tās izpildes laikā, darbinot to uz lietošanas vides datiem un identificējot programmatūras darbības anomālijas.

Atkarībā no lietotā testēšanas veida un izvirzītajiem korektuma kritērijiem, var tikt atpazītas atšķirīga līmeņa programmatūras darbības neatbilstības.

Autors norāda, ka šāda veida kvalitātes verificēšanas pieejas ir attiecināmas ne tikai uz programmatūru, bet arī biznesa procesiem, kas parasti tiek implementēti ar informācijas sistēmu palīdzību. Lai arī biznesa procesus var verificēt, lietojot visus trīs veidus, tomēr izpildes laika verificēšana ir visnozīmīgākā, jo biznesa procesu izpildē daudzviet ir iesaistīts cilvēks.

Analizējot biznesa procesus, var pamanīt, ka ikdienā cilvēki darbojas saskaņā ar procesiem, kas tikai daļēji ir nodrošināti ar informācijas sistēmu atbalstu vai tiek nodrošināti ar vairāku savstarpēji nesaistītu informācijas sistēmu atbalstu. Tādējādi biznesa procesu izpildes laika verificēšanai varētu būt nepieciešamas cita veida verificēšanas metodes, kā tas ir

pierasts programmatūras verificēšanas gadījumā. Tas arī ir pamats darba pētījumam: piedāvāt biznesa procesiem piemērotu izpildes laika verificēšanas mehānismu.

Tā kā pētījums ir attiecināms uz viedtehnoloģiju ietvaru, tad paralēli kvalitātes nodrošināšanas veidiem autors norāda uz autonomiskās skaitļošanas un viedtehnoloģiju saistību ar izpildes laika verificēšanu. Autonomiskās skaitļošanas pamatideja ir veidot sistēmas, kuru uzraudzīšana un uzturēšana prasītu minimālu cilvēka iejaukšanos. Autonomiskās skaitļošanas idejas autori, salīdzinot sistēmas ar dzīviem organismiem, sākotnēji izvirzīja četras autonomisku sistēmu īpašības [HORN01]:

- *paškonfigurācija* - autonomiskām sistēmām jāspēj pašām sevi nokonfigurēt un pārkonfigurēt mainīgā vidē;
- *pašoptimizācija* - komponentes un sistēmas pašas pastāvīgi meklē iespēju uzlabot savu veiktspēju un efektivitāti;
- *pašatjaunošanās* - sistēma automātiski atklāj, diagnosticē un labo atklātās vides problēmas;
- *pašaizsardzība* - sistēmas automātiski aizsargājas pret ļaunprātīgiem uzbrukumiem.

Kopš 2001. gada šis īpašību saraksts ir papildināts ar jaunām īpašībām vai, precīzāk sakot, īpašības ir precizētas, tās nošķirot un izveidojot precīzāku katras īpašības aprakstu. Pēdējos pētījumos izvirzīto īpašību skaits tuvojas 20 [PHIL13]. Lai arī šeit norādītās īpašības reizēm tiek implementētas (parasti robotikā un autonomi strādājošās sistēmās), tomēr vispārējā gadījumā programmatūrā tās ir sastopamas reti, jo to realizēšana ir sarežģīta. Lai sniegtu atbalstu programmatūras izstrādātājiem, tiek veidots viedtehnoloģiju ietvars, kurā dažādi autori [BICE10A, RAUH10, DIEB12] piedāvā tehnoloģijas un konkrētus risinājumus autonomisko īpašību implementēšanai. Arī šā darba autora izpildes laika verificēšanas risinājums ir veidots kā viena no viedtehnoloģiju ietvara komponentēm.

Nodaļas beigās autors izvirza vairākas prasības biznesa procesu verificēšanas risinājumam:

- risinājumam ir jābūt pilnībā ārējam, nevis iebūvētam;
- risinājums nedrīkst pieprasīt esošu sistēmu modifikācijas;
- mehānismam jānodrošina vienkārši papildināms un maināms verificējamo procesu apraksts;
- par konstatētajām problēmām risinājumam ir jāziņo pēc iespējas ātri;
- risinājumam ir jābūt ērti paplašināmam dažādām vidēm, lai nodrošinātu pēc iespējas plašu sistēmu / procesu verificēšanu.

Otrajā nodaļā autors apskata citu autoru piedāvātus izpildes laika verificēšanas risinājumus. Tie tiek apskatīti pēc to implementēšanas veida:

- verificējamajā procesā iebūvēti risinājumi;
- ārēji verificēšanas risinājumi, t.i., tādi, kas neprasa verificējamā procesa modificēšanu;
- jaukti risinājumi, kas paredz gan ārējas komponentes, gan iebūvētas (parasti iebūvētas tiek instrumentācijas komponentes, bet ārējas ir kontroles vai monitorēšanas komponentes).

Iebūvēto risinājumu apraksta sadaļā autors min dokumentu pārvaldības sistēmas un paštestēšanu. Dokumentu pārvaldības sistēmās parasti tiek nodrošināta dokumentu darba plūsmu definēšanas un izpildes funkcija. Tāpat šī funkcionalitāte parasti paredz dažāda veida atgādinājumu vai brīdinājumu nosūtīšanu dokumentu apstrādē iesaistītajiem lietotājiem. Šādi darbu plūsmās iestrādāti atgādinājumi ir uzskatāmi par dokumentu apstrādes procesu verificēšanas līdzekļiem: tie vai nu savlaicīgi atgādina par nepieciešamību veikt kādu uzdevumu, vai arī norāda uz procesa aizkavēšanos, t.i., laikā nepaveiktu uzdevumu. Tādējādi tiek verificēta noteiktu darba plūsmu instanču izpilde. Tomēr šādu atgādinājumu funkciju iekļaušana dokumentu darba plūsmās ir to izstrādātāju uzdevums. Attiecīgi, apskatot dokumentu pārvaldības sistēmas, var secināt, ka to platforma ļauj nodrošināt dokumentu plūsmu izpildes laika verificēšanu (piemēram, [NOVE14] un [SPPS14]).

Otrs apskatītais iebūvētas verificēšanas risinājums ir sistēmu paštestēšana. Viens no autoriem viedtehnoloģiju ietvarā piedāvā lietot paštestēšanu kā papildus kvalitātes nodrošināšanas mehānismu [DIEB12]. Paštestēšana ļauj pārbaudīt programmatūras darbību lietošanas vidē, lietojot lietošanas vides datus. Lai nodrošinātu sistēmas paštestēšanu, tā jau izstrādes sākumā ir jāieprojektē un jāimplementē izstrādājamajā programmatūrā. Paštestēšanas risinājuma autors norāda, ka programmatūrā ir jāiestrādā testēšanas punkti, kuros programmatūra sniedz kādu informāciju par tās stāvokli, un jānodrošina sistēmas komponenšu darbība testa režīmā (dati tiek glabāti reālas datu bāzes kopijā). Tāpat ir jāveido programmatūra, kas izsauktu testējamos programmatūras moduļus, padotu testa datus un pārliecinātos par rezultātu pareizību. Lai arī paštestēšana rada papildus izstrādes uzdevumus, tomēr ir efektīvs programmatūras kvalitātes nodrošināšanas un verificēšanas līdzeklis:

- paštestēšana var tikt lietota testēšanas laikā;
- tā nodrošina testu atkārtojamību un automātisku izpildi;
- paštestēšana ir izpildāma reālajā lietošanas vidē un ar lietošanas vides datiem.

Tādējādi, lai arī paštestēšana ir izpildāma tikai noteiktos programmatūras darbības brīžos (tā nav nepārtraukta programmatūras darbības uzraudzīšana), tā implementē svarīgu izpildes laika verificēšanas prasību - programmatūra tiek verificēta lietošanas vidē ar lietošanas vides datiem.

Ārējo verificēšanas risinājumu ideja paredz, ka programmatūru vai kādas tās īpašības ir iespējams verificēt, programmatūru īpaši nepielāgojot verificēšanai. Šajā sadaļā autors iepazīstina ar vēl vienu viedtehnoloģiju risinājumu - programmatūras izpildes vides testēšanu [RAUH10]. Kā jau liecina piedāvātā risinājuma nosaukums, tas nenodrošina pašas programmatūras pārbaudi jeb verificēšanu, bet programmatūras izpildes vides verificēšanu, t.i., pārbaudi, vai programmatūras izpildes vide ir atbilstoša programmatūras prasībām. Ņemot vērā, ka visbiežāk mūsdienu sarežģītajā programmatūras izpildes vidē tieši nesaskaņotas vides pārmaiņas izraisa programmatūras darbības pārtraukumus, šis ir nozīmīgs risinājums stabilas procesu izpildes nodrošināšanai. Piedāvātais risinājums paredz katrai verificējamai sistēmai izveidot profilu, kurā ir norādītas visas programmatūras izpildei nepieciešamās un nozīmīgās vides atribūtu un uzstādījumu vērtības. Izstrādātāji, izstrādājot sistēmu, sagatavo minēto profilu, savukārt, konkrētās profila vērtības (piemēram, servera nosaukums, dažādas tīkla adreses u.c.) tiek norādītas, uzstādot sistēmu lietošanas vidē. Programmatūras profilu pārbaudi veic īpaša programmatūra, testēšanas koordinators, kas uztur pārbaudes moduļu datu bāzi. Pārbaudes moduļi ir testēšanas programmatūras komponentes, kas nodrošina noteiktu sistēmas profilā norādītu īpašību pārbaudi (piemēram, viens modulis varētu nodrošināt reģistra atslēgu pārbaudi, cits - pieejamos datortīkla resursus). Testēšanas koordinators, lietojot pārbaudes moduļus, salīdzina profila uzstādījumus ar lietošanas vidē esošajiem un nosaka to korektumu. Vispārējā gadījumā testēšanas koordinators ir atsevišķa programmatūras komponente, kuru var arī iebūvēt sistēmā.

Apskatot ārējas verificēšanas risinājumus, darba autors norāda uz diviem risinājumiem, kuri ir attīstījušies, veidojoties SOA. Tā paredz, ka programmatūra tiek veidota kā atsevišķi servisi, kuri viens otram var sniegt noteiktus pakalpojumus. Servisu izstrādātāji publicē izveidoto servisu saskarnes, kuras izmanto citi programmatūras izstrādātāji, veidojot savas sistēmas. Ņemot vērā, ka daudzos servisus var uzturēt atšķirīgi uzturētāji, kopējā izpildes vide var radīt un rada nozīmīgas problēmas: pat nemainot servisa saskarni, tā implementācija var tikt mainīta, tādējādi izmainot arī saistīto procesu izpildi. Attiecīgi SOA gadījumā programmatūras izpildes laika verificēšana ir īpaši nozīmīga, jo ir svarīgi savlaicīgi pamanīt servisu pārmaiņas un to izpildes problēmas.

Pirmais no SOA risinājumiem tiek piedāvāts kā WS-BPEL platformas [OASIS07] papildinājums. Tā autori [BARE05] ir izveidojuši valodu WSCoL, kurā ir iespējams definēt

WS-BPEL platformā aprakstītu procesu (webservisu izsaukumu virknes) verificēšanu: tiek aprakstīti atsevišķu servisu izsaukumu pirmsizpildes vai pēcizpildes nosacījumi. Ja verificēšanas mehānisms ir iespējots, tad pirms izpildes verificējamais WS-BPEL process tiek aizstāts ar instrumentētu procesu, kurā verificējamo servisu izsaukumi ir aizstāti ar verificēšanas starpniekservera (*monitorēšanas pārvaldnieks*) izsaukumu. Starpniekserveris veic verificējamā servisa izsaukumu, nepieciešamības gadījumā pārbaudot pirmsizpildes vai pēcizpildes nosacījumus. Vēlāk risinājuma autori valodu WSCoL ir papildinājuši ar laika kontrolēm, izveidojot valodas paplašinājumu *Timed WSCoL* [BARE07]. Analoģiski ir izmainīts arī verificēšanas mehānisms, lai tas pārbaudītu arī noteiktos procesu laika ierobežojumus.

Otrs risinājums SOA ir LT_FLO+. Līdzīgi kā WSCoL, tas paredz webservisu korektuma definēšanu un pārbaudi izpildes laikā. Arī šis risinājums paredz starpniekservera lietošanu un servisu izsaukumu pārbaudi.

Trešā apskatītā verificēšanas risinājumu grupa ir *jaukti risinājumi*, t.i., daļa no verificēšanas komponentēm ir iebūvētas verificējamajā programmatūrā, savukārt, daļa ir ārējas komponentes. Parasti šādi risinājumi paredz instrumentēšanas komponentes iebūvēšanu verificējamajā programmatūrā, savukārt, analīzes rīki ir ārējas komponentes. Pirmie no diviem apskatītajiem jauktajiem risinājumiem ir saistīti ar programmatūras izpildes trases analizēšanu: programmatūrā tiek iebūvēta trasēšanas funkcionalitāte, savukārt, ar ārējiem rīkiem tiek analizēts trases saturs un novērtēts tās korektums.

Pirmais no risinājumiem piedāvā bibliotēku valodām C/C++. Tajā ir ietvertas vairākas metodes, kuras ļauj programmatūrā iekļaut trasēšanas funkcionalitāti. Trasēšanu ir iespējams konfigurēt ar ārējiem līdzekļiem, norādot gan trases glabāšanas vietu, gan trases pieraksta veidu. Nozīmīgākais uzdevums ir trases analīzes rīku izveide. Tas ļauj definēt noteiktus trases notikumu šablonus, lietojot divus primitīvus - notikumi un intervāli - un dažādas loģiskas operācijas. Šabloni ļauj analizēt trasi, lietojot sarežģītus pieprasījumus. Lai arī konkrētais risinājums nav paredzēts programmatūras darbības analīzei tās izpildes laikā, tomēr tas ļauj detalizēti analizēt tās izpildes gaitu lietošanas vidē un pamanīt darbības problēmas.

Otrs trases analīzes risinājums (ProM ietvars) savā ziņā ir ļoti līdzīgs: tas paredz, ka verificējamā programmatūra veido trases failus XML formātā. Lai nodrošinātu pēc iespējas plašu lietojumu, ProM ietvarā ir izveidota universāla formāta datu bāze un datu analīzes platforma. Trases datu ielādei, datu analīzei, transformācijai u.c. tiek veidoti spraudņi. Tā kā šī ir atvērta platforma, tad visām funkcijām ir pieejams ļoti plašs ProM spraudņu klāsts. Ar ProM ietvara palīdzību ir iespējams identificēt dažādas programmatūras darbības "šaurās

vietas" un analizēt pārmaiņas procesu izpildē. Tā kā datu analīzei tiek lietotas ProM ietvara datu bāzē ieimportētas trases, tad analīzes laikā ir iespējams sasaistīt vairāku sistēmu notikumus. Tādējādi var nodrošināt kopskatu par procesiem, kurus implementē vairākas sistēmas. Tomēr līdzīgi kā iepriekš minētais arī šis risinājums nodrošina procesa verificēšanu, kad izpilde ir pabeigta, bet ne izpildes laikā.

Visbeidzot, trešais šajā grupā apskatītais verificēšanas risinājums ir veidots robotu darbības verificēšanai. Tas paredz automātisku verificēšanas monitora jeb "novērotāja" ģenerēšanu, lietojot verificējamā procesa modeli. Robotu gadījumā modelis atbilst darbības plānam, saskaņā ar kuru notiek robota darbības. Darbinot robotu, tā plānošanas slānis dod komandas izpildes slānim jeb izpildes platformai. Tādējādi plānošanas slānis mēģina realizēt plānā uzdoto uzdevumu. Lai nodrošinātu "novērotājam" nepieciešamās informācijas sniegšanu, izpildes platformā tiek iebūvēta instrumentācija. Risinājuma autori norāda, ka vienkāršākajos gadījumos instrumentācija var tikt ģenerēta automātiski, tomēr parasti instrumentācija ir jāiekļauj manuāli. Izpildes platforma tās darbības laikā ģenerē izpildes ziņojumus, kuri tiek nosūtīti "novērotājam". Savukārt, "novērotājs" pārliecinās, vai izpildes ziņojumi atbilst ierīcei uzticētajam plānam. Autori norāda, ka šādas pārbaudes ir jāveic vairākkārt un katra pozitīva izpilde ar lielāku varbūtību apstiprinās ierīces korektu darbību.

Sava darba **trešajā** nodaļā autors iepazīstina ar piedāvāto izpildes laika verificēšanas risinājumu. Tas tiek piedāvāts kā asinhrons multiaģentu mehānisms:

asinhrons - tas darbojas paralēli verificējamajam procesam, verificēšanas procesu sinhronizējot tikai pēc ārējiem novērojamā procesa radītiem notikumiem;

multiaģentu - informāciju par novērojamajiem procesiem sniedz aģenti, kuri katrs spēj nodrošināt noteikta tipa notikumu fiksēšanu.

Piedāvātā risinājuma nolūks ir "izsekot" verificējamos procesus pēc to izpildes "atstātajām pēdām". Piemēram, sistēma, eksportējot datus, atstāj savas darbības "pēdu nospiedumus" failu sistēmā: tiek izveidots jauns fails. No verificēšanas viedokļa svarīgi ir konstatēt faila izveidošanas faktu, t.i., pamanīt failsistēmas notikumu "izveidots jauns fails". Līdzīgi var norādīt arī citus notikumus, kurus verificēšanas nolūkā būtu svarīgi pamanīt, piemēram:

- datu bāzes tabulā izveidots jauns ieraksts;
- fails ir izdzēsts;
- fails modificēts;
- e-pasts nosūtīts;
- izmainīts html lapas saturs.

Šādus notikumu piemērus varētu minēt daudz, un tie visi var liecināt par kādu verificējamā procesa soļu izpildi. Tādējādi, sekojot līdzi ārējiem notikumiem, verificēšanas mehānisms mēģina secināt, vai verificējamais process izpildās korekti. Lai varētu izsekot verificējamā procesa instances izpildei, katrs no notikumiem satur zināmus to identificējošus parametrus, piemēram, faila nosaukumu, kura modificēšana ir jāpamana, vai tabulas ieraksta identifikatoru ieraksta dzēšanas gadījumā.

Piedāvātais verificēšanas mehānisms sastāv no divām pamatkomponentēm: kontroliera un aģentiem:

- *Kontrolieris*. Šī komponente ir centralizēta un nodrošina verificēšanas pamatuzdevumu - salīdzina konstatētos procesu izpildes notikumus ar sagaidāmajiem, t.i., kādiem vajadzētu notikt procesa instances izpildes laikā. Kontrolieris, zinot kādi ir sagaidāmie procesu instanču izpildes radītie notikumi, pieprasa aģentiem novērot vidi un fiksēt atbilstošos notikumus.
- *Aģenti* ir verificēšanas mehānisma komponentes, kuras pēc kontroliera pieprasījuma novēro vides notikumus un ziņo par to iestāšanos. Notikumiem bagātā izpildes vidē (piemēram, failu serveris vai datu bāzu serveris) asinhrona aģenta un kontroliera sadarbība (kontrolieris pieprasa notikumu, un aģents nosūta atbildi, konstatējot notikumu) ir efektīvs risinājums. Tomēr autors piedāvā izmantot arī "sinhronus" jeb vienkāršus aģentus. Ja ir zināms, ka visi konkrētās vides notikumi ir nepieciešami procesu verificēšanai, aģents var sūtīt visus konstatētos vides notikumus kontrolierim, negaidot to pieprasījumu. Šādu aģentu veidošana ir vienkāršāka, jo aģentā nav jāimplementē divvirziena komunikācija un citi sarežģīti mehānismi. Attiecīgi aģenti ir iedalāmi kompleksajos un vienkāršajos.

Verificēšanas mehānisma ideja paredz, ka aģentu klāsts var tikt brīvi paplašināts, implementējot tos pēc nepieciešamības dažādām vidēm, nodrošinot noteikto saskarni un vispārīgo darbības algoritmu.

Lai kontrolieris varētu veikt procesu verificēšanu un indentificēt verificējamos notikumus, autors piedāvā verificējamos procesus aprakstīt ar domēnspecifiskas valodas palīdzību. Apraksta veidošanas pamatideja paredz katram verificējamajam procesam piekārtot procesa verificēšanas aprakstu, kurā tiek norādīti verificējamie notikumi, to secība un izpildes laika ierobežojumi. Darbā ir aprakstīta gan domēnspecifiskās valodas sintakse, gan semantika. Valodā ir trīs pamatelementi: aktivitāte, aktivitāti raksturojoši notikumi un saites starp aktivitātēm. Katra aktivitāte satur vismaz vienu notikumu, kurš apliecina aktivitātes izpildes. Šis notikums tiek saukts par *galveno notikumu*. Katrai aktivitātei vēl var būt piekārtoti papildu

notikumi jeb apakšnotikumi, kuru iestāšanās ir jāpārbauda, ja ir konstatēts galvenais notikums. Aktivitātēm var tikt norādīti laika ierobežojumi, kuri var būt:

- absolūts laiks, piemēram, 14:15;
- relatīvs laiks attiecībā pret iepriekš notikušām aktivitātēm, piemēram, "faila dzēšanas" aktivitātei ir jānotiek 23 sekunžu laikā pēc "faila izveidošanas" aktivitātes.

Lai aktivitātes un tās identificējošos notikumus varētu sasaistīt ar konkrētu procesa instanci, valodā ir ieviesti mainīgie. To vērtības ļauj izsekot konkrētai procesa instancei. Piemēram, notikums "jauns fails" atgriež izveidotā faila nosaukumu, kas tiek saglabāts procesa verificēšanas apraksta mainīgajā. Savukārt, pieprasot failu sistēmas aģentam konstatēt notikumu "fails dzēsts", tiek padota šī mainīgā vērtība. Tādējādi ir iespējams izsekot konkrēta faila apstrādes procesa instancei.

Katram procesa verificēšanas aprakstam viena no aktivitātēm ir sākuma aktivitāte, bet beigu aktivitātes var būt vairākas. Tāpat katrā aprakstā norāda, vai:

- procesa verificēšanas instanci uzsāk sākuma aktivitāti apliecinošs notikums vai cits verificēšanas process;
- var būt viena vai vairākas verificējamā procesa instances.

Darba **ceturtajā** nodaļā autors izklāsta piedāvātā asinhronā multiaģentu procesu verificēšanas risinājuma aprobāciju. Nozīmīgākais aprobācijas mērķis bija pārliecināties par risinājuma darbību intensīvu procesu gadījumā. Acīmredzot, procesu, kuri izpildās salīdzinoši lēni, verificēšana nozīmīgu virsslodzi neradīs. Tādēļ šis bija mēģinājums novērtēt, vai risinājums spēj verificēt procesus, kuriem ir daudzas instances un kuri izpildās samērā ātri.

Paralēli iespējamās ātrdarbības novērtēšanai, aprobācijai tika izvirzīts papildu mērķis:

- izveidot verifikācijas mehānisma prototipu, kas varētu strādāt ar vairāku procesu daudzām instancēm vienlaicīgi;
- pārliecināties par kontroliera un aģentu saskarņu korektumu: kontrolieris spēj strādāt ar aģentiem, zinot tikai aģentu saskarni un to adresi;
- procesu verifikācijas apraksta valodas pietiekamība (valodā ir iekļauti visi procesu verificēšanai nepieciešamie jēdzieni).

Aprobācijai tika izvēlēta neto norēķinu sistēma EKS [EKS13], kuras viens no procesiem ir maksājuma failu apstrāde. Šis process tika izvēlēts kā piemērots, jo tas izpildās ātri un visi procesa izpildes soļi atstāj sistēmā "pēdas", kuras ir iespējams izsekot. Tāpat šai sistēmai ir pieejama ērta testa vide, kurā ir iespējama dažāda apjoma testa datu ģenerēšana.

Ņemot vērā izvēlēto piemēra procesu, darba autors aprobācijai implemetēja divus aģentus: failsistēmas notikumu un datu bāzes notikumu apstrādes aģentus. Failsistēmas notikumu aģents tika veidots, balstoties uz operētājsistēmas nodrošināto notikumu apstrādi, savukārt, datu bāzes aģents - balstoties uz taimeriem (t.i., pieprasītā notikuma verificēšana notika ik pēc zināma laika perioda). Aprobācija apstiprināja sākotnējo vērtējumu, ka uz notikumiem balstītu aģentu darbība rada mazāku virsslodzi nekā uz taimeriem balstīta: pirmajā gadījumā aģentu izpilde notiek tikai notikuma iestāšanās brīdī, bet otrajā gadījumā aģents rada slodzi katrā taimera izsaukuma brīdī, pat ja notikums nav iestājies, t.i., ar zināmu regularitāti.

Testa vidē darbinot sistēmu, tika apstrādāti daudzi faili, kurus bija sagatavojuši vairāki sistēmas "dalībnieki" (testa piemēri). Katram no sistēmas dalībniekiem tika sagatavots savs failu apstrādes procesa verificēšanas apraksts. Tādējādi verificēšanas mehānisma prototips vienlaicīgi strādāja ar vairākiem verificēšanas aprakstiem. Verificēšanas rezultātā tika pārbaudīta piedāvātā mehānisma spēja pamanīt aizkavētus procesus, kā arī tika novērtēts, ka aģentu radītā sistēmas virsslodze ir nenozīmīga un nerada pamanāmu ietekmi uz sistēmas darbību.

Aprobācijas rezultātā tika identificētas divas verifikācijas apraksta valodā trūkstošas konstrukcijas:

- nosacījuma zarošanās;
- paralēla procesa zaru izpilde.

Darba **secinājumos** autors apkopo darbā paveikto. Autors norāda, ka ir piedāvāts biznesa procesu verifikācijas mehānisms, kas ir piemērojams biznesa procesiem un lietojams arī plašam citu problēmu lokam. Kā piemērus autors min četrus verifikācijas lietojumus, kas iekļaujas piedāvātajā risinājumā.

Autors norāda vairākas pozitīvas piedāvātā risinājuma iezīmes:

- risinājums ir samērā vienkārši implementējams (prototipa implementēšanai tika patērētas 2 cilvēknedēļas);
- risinājums ir vienkārši paplašināms;
- risinājums ir plašāk pielietojams, kā tas sākotnēji tika plānots;
- procesu verifikācijas apraksti ir veidojami salīdzinoši vienkārši;
- nav nepieciešamas verificējamo procesu izmaiņas.

Tāpat autors iezīmē turpmākos pētījuma attīstības virzienus:

- verifikācijas apraksta valodas paplašināšana;
- iespējamo aģentu izpēte;

- risinājuma asinhronitātes potenciāli radīto problēmu novērtēšana un risināšana;
- procesu apraksta valodu un procesa verifikācijas apraksta valodu sasaiste.

Tādējādi darba autoram ir izdevies izveidot verifikācijas mehānismu, kas būtu pielietojams ne tikai biznesa, bet arī notikumu ziņā intensīvākiem procesiem. Verifikācijas mehānisms implementācijas ziņā ir pietiekami vienkāršs, praktiski pielietojams un efektīvs.

1. PROBLĒMAS IDENTIFIKĀCIJA

Izstrādājot programmatūru, ir būtiski pārliecināties, ka tā ir uzbūvēta korekti un strādā pareizi. Pieņemot, ka programmatūra ir uzbūvēta atbilstoši lietotāja prasībām, ir nepieciešams pārliecināties, ka tā strādā bez kļūdām. Par programmatūras kvalitāti pārliecinās, to testējot, turklāt to ir iespējams veikt vairākos *"līmeņos"*: var veikt programmatūras statisko analīzi, var testēt atsevišķas programmatūras koda daļas, var izpildīt visu programmatūru uz testa datiem utt. "Programmatūras testēšana ir tās izmeklēšana, ar mērķi sniegt ieinteresētajām pusēm informāciju par testējamo produktu vai servisu" [KANE06].

Acīmredzami, ka šādā izpratnē testēšana ir attiecināma ne tikai uz programmatūru, bet arī uz biznesa procesiem, kurus, iespējams, daļēji implementē programmatūra. Arī biznesa procesi var tikt testēti un pārbaudīti, turklāt tas pat būtu jādara, jo programmatūras korektība negarantē arī saistītā biznesa procesa korektu izpildi.

1.1. Programmatūras kvalitātes nodrošināšanas veidi

Programmatūras dzīves ciklā nozīmīgu lomu spēlē programmatūras kvalitātes nodrošināšana. Datorspeciālisti ir izstrādājuši daudzas tehnikas un pieejas, kuras nodrošina augstus programmatūras kvalitātes rādītājus jau kodēšanas laikā (daļēju statisko testēšanu mūsdienās nodrošina izstrādes vide, bet dinamisko testēšanu, piemēram, vienībtestēšanu, veic programmētāji), pēc iespējas samazinot iespējamās kļūdas programmatūras testēšanas laikā. Tomēr kvalitatīvas programmatūras izstrāde nav iespējama bez testēšanas, kas atdalīta no kodēšanas , jo daudzas no kļūdām ar kodēšanas laikā pieejamām metodēm praktiski nav iespējams konstatēt. Attiecīgi testēšana jeb kvalitātes pārbaude tiek veikta visā programmatūras izstrādes laikā.

Tomēr, lai arī cik kvalitatīva būtu programmatūras testēšana, ikdienā lietotāji saskaras ar programmatūras kļūdām vai nekorektu darbību. Ne par velti viens no "Merfija datoru likumiem" saka "Katrā netriviālā programmā ir vismaz viena kļūda" [MURP14]. Vēl vairāk, pat ja programmatūra ir kvalitatīva un ar lielu varbūtību tās kļūdas ir novērstas, programmatūra tiek izpildīta vidē, kura var būt nestabila un izraisīt nekorektu prorammatūras darbību.

Mūsdienās strauji pieaug programmatūras sarežģītība, un bieži vien dažādus savstarpēji saistītus servisus uztur savstarpēji nesaistīti uzturētāji. Attiecīgi šie servisi izpildes laikā nesaistīti var tikt mainīti. Pat, ja korekti tiek uzturēta servisu saskarne, nekas negarantē, ka iepriekš pārbaudītā servisa funkcionalitāte pēc pārmaiņām būs tieši tāda pati. Iespējams, ka

izmaiņas ir skārušas servisa izpildes laiku, kas, savukārt, izraisītu izpildes noildzi (no angļu valodas *timeout*) vai vienkārši izpildes paildzināšanos sistēmām, kurām izpildes laiks ir nozīmīgs rādījums. Tādējādi varam teikt, ka pat nelielas izmaiņas mūsdienu kompleksajā programmatūras vidē var radīt nevēlamas pārmaiņas sistēmu darbībā, un tās testēšana nevarētu novērtēt.

Iepriekš minētās problēmas visos gadījumos nav iespējams novērst, t.i., šādus kompleksus risinājumus, kuri darbojas heterogēnā vidē, praktiski nav iespējams pilnībā notestēt un pārtestēt pie katrām vides izmaiņām, par kurām, iespējams, sistēmas turētājs pat nenojauš. Lai minimizētu šādu problēmu ietekmi uz procesiem, kurus nodrošina programmatūra, tiek veikta programmatūras izpildes laika verificēšana - programmatūra tiek verificēta nepārtraukti vai pēc noteikta grafika tās izpildes laikā.

Turpmākajās trijās nodaļās tiks apskatīti trīs iepriekš norādītie testēšanas veidi:

- statiskā testēšana - programmatūra netiek darbināta;
- dinamiskā testēsana - programmatūra tiek darbināta;
- izpildes laika verificēšana - programmatūra tiek darbināta lietošanas vidē.

1.1.1. Statiskā testēšana

"Statiskā testēšana ir programmatūras testēšanas veids, kad programmatūra faktiski netiek lietota" [STAT14]. Vienkāršākais statiskās testēšanas veids ir programmatūras koda sintaktiskā analīze, piemēram, pārliecinoties, ka visi lietotie mainīgie ir definēti un visos izsaukumos sasniedzami.

Statiskai koda testēšanai (analīze) ir teorētiski un praktiski ierobežojumi. Nozīmīgākie teorētiskie ierobežojumi rodas no lēmumu pieņemšanas - bieži vien, neizpildot programmatūru, nav iespējams noteikt tās izpildes ceļu. Teorētiski patvaļīgam programmatūras kodam eksistē dati, par kuriem šādu statisku analīzi nav iespējams veikt. Pat vairāk, ir iespējams noteikt, kurām programmām šāda izpildes analīze ir iespējama un kurām nav. Pretējā gadījumā apstāšanās problēma Tjūringa mašīnām būtu atrisināma [HALT14].

Nozīmīgi praktiski statiskās testēšanas ierobežojumi attiecināmi uz masīvu indeksu un rādītāja (no angļu valodas - *pointer*) tipa mainīgo pārbaudēm. Masīva indeksi un rādītāji ir programmēšanas līdzekļi dinamiskai datu vienumu adresēšanai tās izpildes laikā. Tie parasti tiek veidoti, balstoties uz iepriekš veikto skaitļošanu. Tādējādi statiskā analīze nevar izvērtēt masīva indeksu vai rādītāju pareizību. Daļēji šādu programmēšanas līdzekļu lietojumu varētu pārbaudīt, lietojot simboliskās izpildes tehnikas, tomēr parasti šādus koda risinājumus ievērojami vieglāk ir pārbaudīt, tos izpildot jeb testējot dinamiski. Tādēļ dinamiskā testēšana

parasti tiek lietota situācijās, kurās programmatūru nav iespējams pārbaudīt vai to ir ļoti grūti izdarīt ar statisko testēšanu jeb analīzi [FAIR78].

1.1.2. Dinamiskā testēšana

"Dinamiskā testēšana (jeb dinamiskā analīze) ir datorinženierijas termins, ar kuru apzīmē koda dinamiskās uzvedības testēšanu" [DYNA14], t.i., dinamiskā analīze norāda uz sistēmas reakciju uz laikā mainīgiem datiem, kuri nav konstantes. Dinamiskai testēšanai kodam ir jābūt nokompilētam un izpildāmam. Šajā procesā tiek testēta programmatūra, padodot tai ieejas datus un pārbaudot, vai izejas dati atbilst attiecīgā testa scenārijā specificētajiem. Dinamiskā testēšana var tikt veikta manuāli vai arī automatizēti, lietojot atbilstošus testēšanas atbalsta rīkus.

Tipiski dinamiskās testēšanas veidi ir vienībtestēšana, integrācijas testēšana un sistēmtesti: tie visi paredz visas vai daļējas izstrādātās programmatūras darbināšanu ar testa vai akcepttesta datiem saskaņā ar testa scenāriju, pārbaudot saņemto izejas datu korektumu.

1.1.3. Izpildes laika verificēšana

Kā pēdējo ir jāmin programmatūras izpildes laika verificēšanu. Ar šo metodi programmatūras darbība tiek pārbaudīta tās izpildes laikā. "Izpildes laika verificēšana ir datorsistēmu analīze, apkopojot informāciju par sistēmu tās izpildes laikā. Tās izpildes laikā iespēju robežās tiek reaģēts uz novēroto pareizo vai noteiktiem kritērijiem neatbilstošo izpildi" [RUNV14].

Autori [LEUC09] norāda, ka viens no būtiskiem izpildes laika verificēšanas ieguvumiem, salīdzinot ar citiem verificēšanas veidiem, ir uzraudzīšana tiešsaistē. Tas ļauj nekavējoties reaģēt uz korektības kritēriju pārkāpumiem. Tas ļauj reaģēt uz sistēmu kļūmēm, pirms tās kļūst par sistēmu darbības atteicēm (no angļu valodas *failure*). Izpildes laika verificēšana var nodrošināt informācijas sniegšanu arī uzraugāmajai programmatūrai, palīdzot novērst problēmu tās izpildes laikā. Citas verificēšanas tehnikas šādas iespējas nesniedz.

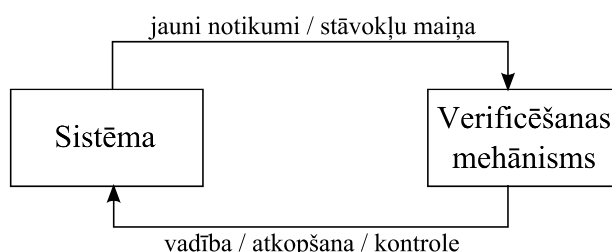
Daži autori [LEUC09] norāda, ka izpildes laika verificēšana ir "vieglā" verificēšanas tehnika, kura ir vērtējama kā zināms kompromiss starp testēšanu un modeļu verificēšanu. "Viena no svarīgākajām izpildes laika verificēšanas atšķirībām ir tās būtība tikt izpildītai programmatūras izpildes laikā. Tas paver plašas iespējas reaģēt uz programmatūras nekorektu darbību nekavējoties."

Izpildes laika verificēšanas attīstību nozīmīgi ir ietekmējusi SOA attīstība: programmatūras un procesu darbību nodrošina daudzi savstarpēji nesaistīti servisi, kurus uztur atšķirīgi uzturētāji. Ir izveidoti piedāvājumi servisorientētas vides verificēšanai

[BARE05, BARE07] un adaptīvu sistēmu veidošanas ietvari [CALI11], kuri balstās uz izpildāmo sistēmu verificēšanu un tālāku to konfigurēšanu.

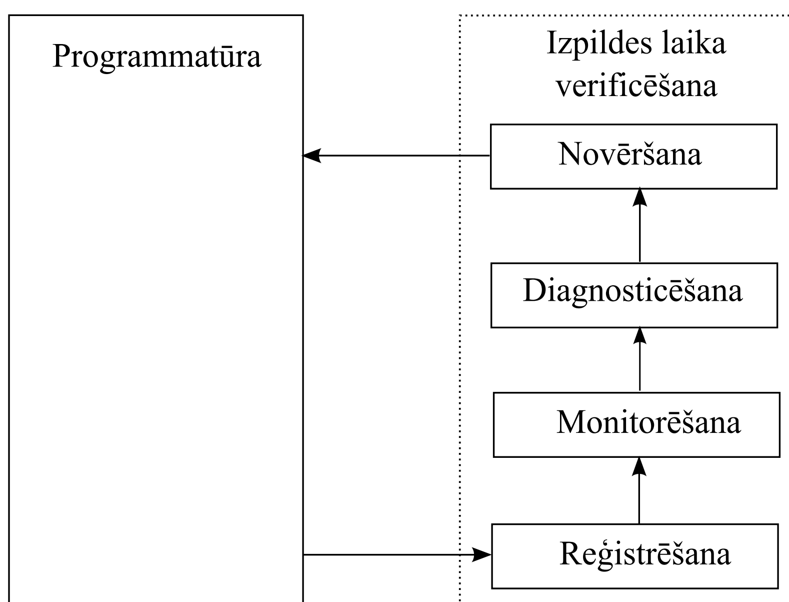
Daudzviet autori izpildes laika verificēšanu skaidro kā notikumu iniciētu monitorēšanas tehniku [WU13]. Šādā notikumu iniciētā izpildes laika verificēšanā katru reizi, kad notiek kāds sistēmai nozīmīgs notikums (piemēram, stāvokļu maiņa vai kritiska kļūda), pārbaudāmā sistēma izraisa monitorēšanu. 1. attēls parāda, kā sistēma un verificēšanas mehānisms vispārīgā gadījumā sadarbojas:

- sistēma ziņo par tās notikumiem;
- verifikācijas mehānisms, reaģējot uz notikumiem un izvērtējot to izpildes korektību, var veikt uzraudzības, ierobežojošas vai pat atkopjošas darbības.



1. attēls. Izpildes laika verificēšanas process

Izpildes laika verificēšanas mehānisms var būt dažādas sarežģītības un implementēt dažādas komponentes (skat. 2. attēls) [LEUC09].



2. attēls. Izpildes laika verificēšanas komponentes

Jebkurš izpildes laika verificēšanas mehānisms satur vismaz divas no shēmā redzamajām komponentēm: reģistrēšana un monitorēšana. "Reģistrēšana" paredz, ka verificēšanas mehānisms saņem informāciju par programmatūras izpildi. Tas parasti tiek

nodrošināts ar instrumentēšanas palīdzību - programmatūrā tiek iebūvēts mehānisms, kas sniedz informāciju par tā izpildes gaitu, t.i., kuru soli programmatūra izpilda. "Monitorēšana", analizē saņemto informāciju un "attēlo" to turpmākajai "lietošanai". Tas var attiekties tikpat labi uz programmatūras atbalsta personālu kā uz kādu verificēšanas sistēmu. Piemēram, sniedzot ziņas personālam, monitorēšanas rīki izdalītu maksimālās un minimālās vērtības un izmaiņu tendences.

"Diagnosticēšanas" mērķis ir novērtēt atbilstoši saņemtajai informācijai, vai programmatūra izpildās korekti. Lai to veiktu, ir jābūt izvirzītiem kādiem korektuma kritērijiem, kuriem jāizpildās programmatūras izpildes laikā.

Autori norāda, ka ir iespējama arī ceturtā, "novēršanas", komponente. Tās uzdevums ir novērst vai vismaz mazināt konstatēto problēmu. Dažreiz tas var būt paveicams, piemēram, mainot programmatūras konfigurāciju. Tomēr, acīmredzot, daudzos gadījumos programmatūras darbības problēmu novēršana var būt pietiekami komplicēta.

1.2. Biznesa procesu verificēšana

Tuvāk iepazīstoties ar biznesa procesiem, to izveidi un ieviešanu, tos var salīdzināt ar programmatūru un tās izstrādi. Biznesa procesus ir iespējams aprakstīt (izveidot modeļus), tos ir iespējams darbināt testējot un tie darbojas lietošanas vidē. Analoģiski var skatīties uz biznesa procesu kvalitātes nodrošināšanu. Tiem var veikt statisku testēšanu, pārbaudot procesa aprakstu un pārliecinoties, ka tajā nav loģisku kļūdu: ir noteikti visi izpildītāji, visi saistītie datu avoti ir pieejami, ir skaidri definēta katra procesa aktivitāte utt. Tāpat procesiem var veikt dinamisku testēšanu, t.i., mēģināt procesus izpildīt un pārbaudīt tos darbībā. Procesiem var veikt arī izpildes laika verificēšanu: sekot līdzi to izpildei ikdienā. Biznesa procesiem izpildes laika verificēšana ir viens no nozīmīgākajiem testēšanas/pārbaudes veidiem, jo parasti biznesa procesu izpildē cieši ir iesaistīti cilvēki. Piemēram, dažāda veida dokumentu plūsmu izpilde ir atkarīga no tā, kad atbildīgie cilvēki veikts viņiem uzticētās darbības. Vienlaicīgi cilvēki parasti ir biznesa procesu neuzticamākie posmi. Šī iemesla dēļ dokumentu pārvaldības sistēmās bieži vien tiek iebūvēta automātiska atgādinājumu sūtīšana, ja kāds no soļiem nav savlaicīgi izpildīts. Šādi mehānismi ir izpildes laika verifikācija, kura ļauj verificēt konkrētā biznesa procesa nevis dokumentu pārvaldības sistēmas darbību.

Lai arī gandrīz visās dokumentu un darba plūsmu pārvaldības sistēmās ir iebūvēti zināmi izpildes laika verifikāciju atbalstoši mehānismi, tie tomēr nenodrošina visaptverošu biznesa procesu izpildes laika verificēšanu. Turklāt dokumentu un darba plūsmu izpildes sistēmas ir tikai neliela daļa no informācijas sistēmām, kuras tiek lietotas biznesa procesu

ietvaros. Ir vairāki iemesli, kāpēc nav pieejams visaptverošs biznesa procesu izpildes laika verificēšanas mehānisms:

- Viena biznesa procesa izpildi var nodrošināt vairāk kā viena informācijas sistēma. Pat ja katra no saistītajām sistēmām nodrošina savu verificēšanas mehānismu, kopīgu priekšstatu par procesa izpildi būs grūti iegūt. Sliktāk ir gadījumos, ja šīs sistēmas neapmainās ar datiem automātiski, bet datu apmaiņa tiek veikta manuāli. Piemēram, dati tiek ievadīti vienā sistēmā un pēc tam otrā. Šādā gadījumā sistēmas pat "nenojauš" par otras esamību un nezina par procesa uzsākšanu citā sistēmā.
- Par biznesa procesu izpildes korektību satraucas pietiekami lieli uzņēmumi. Šādos uzņēmumos parasti ir pietiekami daudz dažāda veida mantotu sistēmu (no angļu valodas - *legacy software*), kurās vai nu nav iebūvēta verificēšana, vai arī to iebūvēt vairs nav iespējams.

1.3. Autonomiskās sistēmas un viedtehnoloģijas

Jau iepriekšējās nodaļās tika minēts, ka straujā tehnoloģiju attīstība nozīmīgi palielina informācijas sistēmu sarežģītību. Pirms 10 gadiem autori norādīja, ka "skaitļošanas sistēmu sarežģītība tiecas uz cilvēka iespēju robežām" [KEPH03]. Citi autori norāda uz paradoksu: "lai palīdzētu risināt problēmu - veidot tās vienkāršāk administrējamas un vienkāršāk lietojamas IT personālam - *ir jāveido vēl sarežģītākas sistēmas*" [HORN01]. Savā ziņā mēs to varam novērot jau šobrīd: ir tik daudz ērti lietojamas mobilās lietotnes, kuras ir "gudras" un prot pielāgoties, un sistēmu lietotājiem rodas nepamatots priekšstats par "triviālu" IT pasauli. Tikai sistēmu izstrādātāji apzinās, ka katras lietotāja ērtības ieviešana sistēmas padara sarežģītākas.

Aprakstot vienu no iespējamajiem problēmas risinājumiem, 2001. gadā IBM piedāvāja autonomiskās skaitļošanas manifestu [HORN01]. Tā galvenais mērķis ir aprakstīt tādu informācijas sistēmu izveidi, kas ir spējīgas pašas sevi vadīt, tādējādi pašas spējot pielāgoties sarežģītajai datorsistēmu videi un palīdzot lietotājiem pārvarēt tehnoloģiju barjeru. Šajā manifestā tiek piedāvātas četras galvenās īpašības, kas raksturo autonomisku skaitļošanu:

- **Paškonfigurācija** (self-configuration) - autonomiskām sistēmām jāspēj pašām sevi nokonfigurēt vai pārkofigurēt mainīgā un neparedzamā vidē.
- **Pašoptimizācija** (self-optimizaton) - komponentes un sistēmas pastāvīgi meklē iespējas uzlabot savu veiktspēju un efektivitāti.
- **Pašatjaunošanās** (self-healing) - sistēma automātiski atklāj, diagnosticē un labo atklātās programmatūras un aparatūras problēmas.

- **Pašaizsardzība** (self-protection) - sistēma automātiski aizsargājas pret ļaunprātīgiem uzbrukumiem vai kaskādes neveiksmēm. Tā izmanto agrīnu brīdināšanu, lai paredzētu un novērstu sistēmas nepilnības.

Vēlāk 2003. gadā šis saraksts tika papildināts [KEPH03] ar vēl četrām īpašībām:

- spēju sistēmai pašai pazīt sevi un vadīt savus resursus;
- pazīt savu darbības vidi un spēt tajā adaptēties;
- darboties heterogēnā pasaulē;
- apslēpt risinājuma heterogēno sarežģītību.

Šobrīd izvirzīto īpašību skaits jau ir sasniedzis gandrīz 20 [PHIL13]. Visu šo īpašību kopējais mērķis ir padarīt informācijas sistēmas paš-vadošas, tādējādi samazinot to uzturēšanas sarežģītību.

Pirmajos desmit gados sasniegtie autonomiskās skaitļošanas rezultāti ir apkopoti [KEPH11] 2011. gadā. Jāatzīst, ka šobrīt programmatūras izstrāde ir tālu no šo mērķu sasniegšanas, un pagaidām, galvenokārt, notiek autonomisko sistēmu teorētiska izpēte un daļēja praktiska realizācija.

2007. gadā tika piedāvāta viedtehnoloģiju pieeja [BICE07], kura ir praktisks atbalsts ceļā uz autonomisku sistēmu ieviešanu. Viedtehnoloģijas piedāvā virkni praktiski realizējamus uzlabojumus informācijas sistēmu funkcionalitātē, kas ļauj vienkāršot to uzturēšanu un lietošanu, tādējādi tuvojoties autonomisko sistēmu galvenajam mērķim.

Kā galvenais veidtehnoloģiju paņēmiens tika piedāvāta nevis universālu komponenšu izveide, bet iekļaut viedtehnoloģiju iespējas konkrētu sistēmu programmatūrā tieši. Pētījumi arī parāda, ka viedtehnoloģiju lietošana, veidojot programmatūru, kas nebūtu uzskatāma par autonomisku, uzlabo programmatūras uzturamību [BICE10A].

Iepriekš minētajā viedtehnoloģiju ietvarā eksperimentāli ir izveidoti vairāki viedtehnoloģiju veidi:

- **dinamisks biznesa modelis** - mainoties biznesa procesiem, informācijas sistēmas funkcionalitāte, pateicoties sistēmā iekļautajam biznesa modelim, mainās automātiski [BICE10B];
- **iebūvēta versiju pārvaldība un datu sinhronizācija** - automātiska informācijas sistēmas komponentu (koda, datu struktūru, ekrānskatu, atskaišu formu u.c.) versiju izplatīšana no centrālās glabātuves uz lokālām darbstacijām un serveriem [BICE08];

- **ārējās vides testēšana** - automatizēta ārējās vides parametru pārbaude un atkarībā no konstatētajiem rezultātiem iespēja informēt lietotājus par neatbilstībām un nepieciešamajām pārmaiņām [RAUH09];
- **paštestēšana** - programmatūras spēja pārbaudīt pašas integritāti un darbspējas, izpildot iepriekš uzkrātus testpiemērus ar programmatūrā iebūvētām testēšanas atbalsta iespējām [DIEB09].

Kā redzams, tad visas no iepriekš minētajām tehnoloģijām ir lietojamas tikai programmatūras izstrādē, tās iekļaujot pašā programmatūrā. Atsevišķi gala lietotājiem tās nav piegādājamas. Iespējams, ka sākotnējā posmā viedtehnoloģiju lietošana izstrādi pat padara sarežģītāku (uz to savā pētījumā norāda arī autore [BICE10A]), tomēr vēlāk tās ievērojami var atvieglināt programmatūras uzturēšanu un lietošanas stabilitāti, arvien tuvāk pietuvinoties galamērķim - autonomiskām sistēmām.

Arī šā darba autors savus pētījumus biznesa procesu verificēšanā uzsāka viedtehnoloģiju ietvarā [ODIT10]. Sākotnējais mērķis bija izveidot tehnoloģiju, kura nodrošinātu automātisku dokumentu darba plūsmu izpildes kontroli, t.i., iespēju izsekot un pārbaudīt biznesa procesu izpildes korektību. Tikai vēlākā darba rezultātā darbs iekļāvās izpildes laika verificēšanā.

Tomēr arī šādi darbs ir attiecināms uz autonomisko sistēmu un viedtehnoloģiju ietvaru:

- Izpildes laika verificēšana spēlē būtisku lomu autonomisko sistēmu kontekstā. Jebkurš lēmums, kuru sistēma pieņem par tās tālāko darbību, tiek balstīts uz izpildes laika verificēšanas novērtējumiem: vai izpildes laikā konstatētie novērojumi atbilst noteiktiem korektības kritērijiem. Tādējādi izpildes laika verificēšana ir uzskatāma par autonomisko sistēmu lēmumu pieņemšanas iniciētāju.
- Darbā piedāvātais verificēšanas mehānisms paredz ārēju verificēšanas komponenšu lietošanu, šādi nodrošinot arī iepriekš izstrādātu sistēmu verificēšanu. Tomēr, ņemot vērā mehānisma (tehnoloģijas) piedāvātās iespējas, tas ievērojami atvieglinātu verificēšanas iestrādāšanu jaunās sistēmās.

1.4. Biznesa procesu verificēšanas problēma

Sākotnējais autora pētījumu objekts bija dokumentu darba plūsmu kontrole un uzraudzība. Tas tika veidots viedtehnoloģiju ietvarā, mēģinot piedāvāt tehnoloģijas, kuras laika gaitā ļautu veidot autonomiskas sistēmas (skat. 1.3. nodaļā) [ODIT10]. Tikai tālākajos

pētījuma soļos tika apzināts, ka šāda veida uzraudzība un kontrole faktiski attiecas uz izpildes laika verificēšanas sfēru. Iespējams tāpēc daži no izvirzītajiem pētījuma mērķiem nedaudz atšķiras no klasiskas izpildes laika verificēšanas (skat. 2. nodaļā).

Turpmākajās apakšnodaļās autors apskata dažādas verificēšanas mehānismu īpašības un apkopo biznesa procesu verificēšanai nepieciešamās.

1.4.1. Verificēšanas sarežģītība

Apskatot izpildes laika verificēšanas pētījumu un rīku klāstu, var izdalīt divus robežgadījumus:

- noteiktu sistēmu vai iekārtu rādītāju monitorēšana;
- pilna funkciju izpildes verificēšana (izpildes laikā verificē, vai izsaukumu rezultāts atbilst nodotajiem parametriem, t.i., diagnosticē sistēmas darbību).

Pirmā no pieejām praktiski neveic verificēšanu (maksimums - ziņo, ja kāda no rādītāju vērtībām pārsniedz vai neatbilst kādai no robežām). Tā nodrošina noteiktu rādītāju mērīšanu un lietotāja informēšanu par konstatējumiem. Visus lēmumus par rādījumu pareizību pieņem attiecīgā verificēšanas rīka lietotājs. Piemēram, procesora noslodzes mērījumi vai nu tiek attēloti nepārtraukti, vai arī lietotājam tiek sniegta informācija, ja mērījums pārsniedz noteiktu brīdinājuma noslodzi. Lietotāja ziņā tiek atstāta lēmuma pieņemšana (verificēšana), vai konkrētā iekārta vai kaut kādi procesi darbojas nekorekti.

Savukārt, otrs robežgadījumus paredz iebūvētu testēšanas mehānismu. Lai precīzi verificētu sistēmas darbību izpildes laikā, ir nepieciešams ļoti detalizēts tās izpildes apraksts: ieejas dati un sagaidāmie izejas dati. Acīmredzot, šādu izpildes laika verificēšanas aprakstu sagatavošana nav atdalāma no programmatūras izstrādes, t.i., to ir jādara ar attiecīgās sistēmas izstrādi saistītam personālam. Tāpat parasti piedāvātie risinājumi paredz iebūvēt verificējamajā programmatūrā kādas komponentes vai arī darbināt programmatūru ierobežotā jeb izmēģinājuma vidē (no angļu valodas - *sandbox*). Savukārt, šāda vide vai papildus kods rada virsslodzi programmatūras izpildes laikā, tādēļ vairāki autori ir ieteikuši šādas verificēšanas metodes lietot selektīvi, t.i., pamatā programmatūru darbināt bez verificēšanas, to ieslēdzot tikai dažreiz [CALI11]. Attiecīgi, lai arī sākotnējais uzstādījums ir precīza programmtūras izpildes verificēšana, to pielietojot selektīvi, iegūtais verificēšanas rezultāts tikai nosacīti apliecina programmatūras darbības korektību.

1.4.2. Verificēšanas atgriezeniskā saite

Cita nozīmīga izpildes laika verificēšanas iezīme ir atgriezeniskās saites sniegšana un reakcija uz verificēšanas konstatējumiem, t.i., vai un kā tiek reaģēts uz izpildes laika verificēšanas novērojumiem. Bieži vien lēmumu pieņemšanas un reakcijas jautājumi tiek

atstāti cilvēku ziņu, tomēr daudzviet lēmums tiek pieņemts automātiski. Piemēram, automašīnās vadības datori, konstatējot kāda dzinēja cilindra darbības problēmas, var atslēgt degvielas padevi attiecīgajam motora cilindram. Šādā veidā, balstoties uz verificēšanas rezultātu, tiek pieņemts lēmums koriģēt automašīnas dzinēja darbību, nodrošinot vismaz daļēju turpmāko sistēmas darbību. Līdzīgus risinājumus varam redzēt arī programmatūras jomā - operētājsistēmas, lai nodrošinātu kopējo darbības stabilitāti, apstādina procesus, kuri darbojas kļūdaini, piemēram, mēģinot nolasīt vai ierakstīt datus ārpus tiem atvēlētā atmiņas apgabala. Teorētiski var tikt apskatīti arī sarežģītāki reakcijas modeļi, piemēram, atkarībā no verificēšanas rezultātiem tiek mainīta programmatūras konfigurācija, mēģinot novērst konstatēto problēmu (šis būtu autonomisku sistēmu uzstādījums – 1.3. nodaļa), tomēr šie risinājumi būs cieši saistīti ar konkrēto verificējamo sistēmu un, visticamāk, nebūs universāli.

Sākotnējā autora izvirzītā problēmas nostādne bija nodrošināt verificēšanu dokumentu pārvaldības sistēmas darba plūsmām, kā vissvarīgāko faktoru nodrošinot izpildes laika kontroli. Svarīgi bija atrast risinājumu, kas ļautu verificēt nevis konkrētu darbu plūsmu vai būtu salāgojams ar noteiktu dokumentu pārvaldības sistēmu, bet kas būtu pietiekami universāls, lai varētu darboties ar dažādiem risinājumiem. Ņemot vērā universālu pieeju, kā pietiekama tika uzskatīta iespēja savlaicīgi identificēt verificējamā procesa problēmu un par to ziņot sistēmas uzturētājiem. Vēlāk, pētot problēmas apgabalu, kļuva skaidrs, ka problēmas nostādne (piemērota verifikācijas veida meklēšana) ir attiecināma ne tikai uz biznesa procesiem dokumentu pārvaldības sistēmās, bet arī uz atsevišķiem nelieliem procesiem vienas informācijas sistēmas ietvaros, piemēram, failu apstrādes mehānismu maksājumu apstrādes sistēmā (skat. 4.2.1. nodaļā). Tomēr arī programmatūrā lietojamas saskarnes gadījumā būtu lietojama tikai un vienīgi informācijas sniegšana par verificējamo procesu, t.i., neiejaucoties verificējamā procesa izpildē. Verificējamā procesa ierobežošanu var atstāt:

- cilvēka pārziņā, ja cilvēks ir verificēšanas informācijas saņēmējs;
- programmatūras pārziņā, kas saņem verificēšanas rezultātu.

Šī darba galvenais mērķis ir izstrādāt metodiku, lai savlaicīgi pamanītu procesu izpildes problēmu, ziņošanu atstājot otrā plānā, jo tā var tikt implementēta dažādi.

Lai arī darbā tas nav iekļauts, tika apskatīta iespēja paplašināt verificēšanas mehānisma sniegto atgriezenisko saiti: sākotnējo uzstādījumu par ziņošanu tikai cilvēkiem papildināt ar ziņošanas saskarni citām informācijas sistēmām, tādējādi vēl vairāk iekļaujot meklējamo risinājumu viedtehnoloģiju un autonomo sistēmu ietvarā.

1.4.3. Verificēšanas varbūtiskais raksturs

Ikdienā cilvēki ir pieraduši dzīvot varbūtiskā pasaulē. Lai arī arvien vairāk pierodam pie stingri noteiktām lietām, kuras notiek noteiktos laikos un notiek precīzi atbilstoši plāniem, tomēr vēl aizvien daudziem mūsu dzīves notikumiem ir varbūtisks raksturs. Piemēram, pilsētas autobusam var būt pārplīsusi riepa un tas neierodas, lai mūs aizvestu uz pilsētu. Mēdz izrādīties, ka pasūtītā prece mums netiek piegādāta, jo tā noliktavā nav pieejam vai arī to vienlaicīgi ir vēlējušies saņemt divi pircēji. Analoģiskas situācijas mēdz gadīties arī datorizētās vidēs, kur tiek uzskatīts, ka viss ir precīzs un nekļūdīgs: nosūtot e-pastu, mēs nevaram būt pilnīgi droši, ka tas sasniegs adresātu. Turklāt, ja tas sasniegs adresātu, tad saņēmējs to var neizlasīt, jo kādu iemeslu dēļ vēstuli varētu būt pārtvēris surogātpasta filtrs. Lai arī cik varbūtiska ir pasaule, kurā mēs dzīvojam, mēs esam ar to ļoti labi iemācījušies sadzīvot un bieži vien pat nepamanām lietas, kuras nenotiek 100% gadījumu.

Iepriekš minētais attiecas arī uz informācijas sistēmu verificēšanu. Visi izstrādātāji zina teicienu: "Katrā netriviālā programmā ir vismaz viena kļūda." Tomēr informācijas sistēmas tiek lietotas un nodrošina nozīmīgas mūsu ikdienas funkcijas. Ņemot vērā iepriekš minēto, varam pieņemt, ka ir pilnīgi pietiekami, ja procesu izpildes laika verificēšanas mehānisms nodrošinātu pietiekami uzticamu informācijas sniegšanu par procesu izpildes problēmām.

1.4.4. Prasības procesu verificēšanai

Apskatot iezīmes, kas jānodrošina biznesa procesu verificēšanai, pirmkārt, jāmin, ka piedāvātais mehānisms nevar būt vienkārša kādu rādījumu monitorēšana. Tam jābūt pietiekami sarežģītam, lai varētu verificēt izpildes procesa pareizību, kas sastāv no vairākiem soļiem. Ņemot vērā meklētā mehānisma universālo raksturu, mērķis nevar būt sistēmās iegults mehānisms. Tāpat verificējamā procesa definēšanai ir jābūt pietiekami vienkāršai, lai mehānisms būtu praktiski pielietojams.

Otrkārt, kā jau tika norādīts iepriekš (1.4.2. nodaļa), verificēšanas mehānismam ir jānodrošina atgriezeniskās saites sniegšana procesa lietotājiem. Ilgtermiņā tas var tikt papildināts ar ziņojumu sniegšanas saskarni informācijas sistēmām, tomēr nebūtu jāparedz iejaukšanās verificējamajā procesā.

Treškārt, ņemot vērā, ka tiek meklēts ārējs risinājumu, nevis iegults, tas, visticamāk, nesniegs 100% uzticamu verificēšanas rezultātu. Tomēr, lai mehānisms būtu lietojams, tam būtu jāsniedz pēc iespējas uzticams procesa izpildes korektuma vērtējums: būtu jāpanāk, ka ar lielu varbūtību tiek fiksētas visas kļūdainās situācijas un viltus pozitīvi (no angļu valodas - *false positive*) gadījumi būtu pēc iespējas reti.

Tādējādi, apkopojot visu iepriekš rakstīto, meklētajam biznesa procesu verificēšanas mehānismam tika izvirzītas šādas īpašības:

- ārējs, verificējamajā sistēmā neiegults risinājums, tādējādi nodrošinot procesu verificēšanu, kuri izpildās vairāku informācijas sistēmu ietvaros;
- lai nodrošinātu mantotu sistēmu (no angļu valodas - *legacy software*) verificēšanu, mehānisms neprasa verificējamo procesu modificēšanu;
- jaunu procesu iekļaušanai verificēšanai ir jābūt pēc iespējas vienkāršai;
- ir pietiekami, ja mehānisms strādā ar varbūtisku, tomēr pietiekami precīzu rezultātu;
- par konstatētajām problēmā ir jāziņo pēc iespējas ātri;
- risinājumam ir jābūt ērti paplašināmam dažādām vidēm, lai nodrošinātu pēc iespējas plašu sistēmu / procesu verificēšanu.

2. SAISTĪTIE PĒTĪJUMI

Vairāki autori ir norādījuši, ka statiskā un dinamiskā testēšana ir nepietiekami līdzekļi modernu biznesa procesu verificēšanai. Tas tiek pamatots ar izpildes vides heterogenitāti un komponenšu un servisu bāzētu sistēmu arhitektūru. Procesus nodrošina daudzas komponentes, kuras neatkarīgi viena no otras laika gaitā tiek mainītas, un bieži vien pārmaiņu laikā ir sarežģīti novērtēt to ietekmi uz visiem biznesa procesiem. Tas nozīmē, ka procesa izpildes laika verificēšana ir jāveic visā programmatūras dzīves cikla laikā [GHEZ07, BARE05].

Mūsdienās izpildes laika verificēšana kļūst arvien nozīmīgāka programmatūras uzturēšanas sastāvdaļa. Attiecīgi šai nozarei tiek veltīti arvien vairāk dažādu pētījumu.

Piedāvātos izpildes laika verificēšanas risinājumus var iedalīt trīs grupās, kas tiek apskatītas turpmākajās nodaļās:

- verificējamajā sistēmā iebūvēti risinājumi;
- no verificējamās sistēmas neatkarīgi jeb ārēji risinājumi;
- jaukti risinājumi, kas paredz gan iebūvētu, gan ārēju komponenšu izmantošanu.

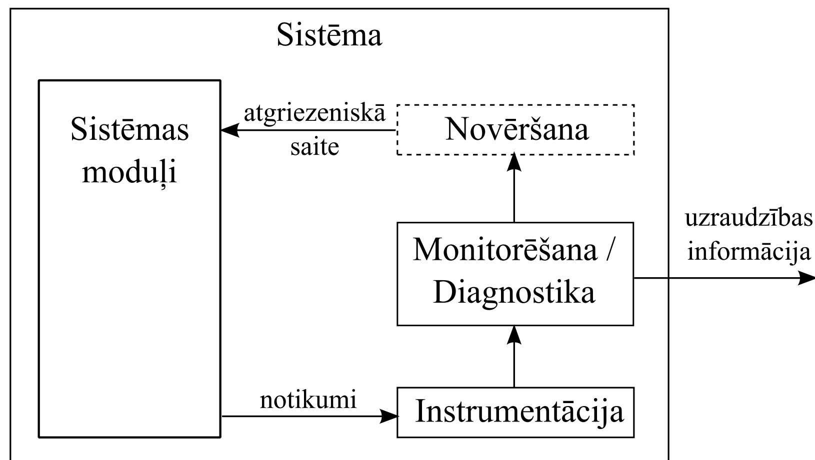
Turpmākajās nodaļās tiks apskatīti risinājumi katrai no šīm grupām.

2.1. Iebūvēti izpildes laika verifikācijas risinājumi

Daudzi autori (piemēram, [CIAN96] un [BERG98]) norāda, ka izpildes laika verificēšana var tikt efektīvi realizēta, iebūvējot to informācijas sistēmā. Tam ir vairāki iemesli:

- izstrādātājiem ir pieejams programmatūras kods, tādēļ testēšanas punktus var ievietot atbilstoši procesu izpildes nozīmībai, t.i., vietās, kur tas ir nepieciešams;
- ja sistēma jau sākotnēji tiek būvēta pašverificējoša, verificēšanas mehānismu var organiski integrēt sistēmā kā neatņemamu tās sastāvdaļa.

Tomēr praksē tikai ļoti nedaudzās sistēmās ir realizēta funkcionalitāte, kura ļauj verificēt informācijas sistēmu tās izpildes laikā. Liela daļa no sistēmām implementē tikai izpildes laika kļūdu reģistrēšanu. Tomēr, ja verificēšana ir implementēta, tā parasti ir vienkāršota, piemēram, nodrošinot kādu programmatūras rādītāju monitorēšanu ar operētājsistēmas līdzekļiem vai pārbaudot vides uzstādījumus programmatūras izpildes sākumā. Tomēr eksistē arī sistēmas, kurās izstrādes laikā verificēšana (piemēram, paštestēšana) ir implementēta.



3. attēls. Iebūvēta izpildes laika verificēšana

Iebūvētā izpildes laika verificēšanas mehānismā (skat. 3. attēls) visas verificēšanas komponentes ir iekļautas sistēmā. Tajā ir iekļauta gan instrumentācija, gan monitorēšana, gan diagnosticēšana. Instrumentācija konstatē notikumus, monitorēšana tos filtrē, diagnosticēšana identificē problēmas. Ja šāds mehānisms tiek iekļauts sistēmā, tad sistēmā ir iekļaujama arī informācijas sniegšana sistēmas lietotājiem vai tās uzturētājiem.

Šis ir veids, kurā nepieciešamības gadījumā visvienkāršāk var iekļaut arī novēršanas (skat. 2. attēls) komponentes. Sistēmas izstrādātājiem ir pieejama pilna informācija par sistēmas uzbūvi un darbību, tādēļ viņiem ir pieejama arī visa novēršanai vai atkopšanai nepieciešamā informācija un funkcionalitāte. Attiecīgi šādā risinājumā atkopšanu (ja to vispār ir iespējams veikt) ir iespējams implementēt visvienkāršāk.

Ideja par sistēmās iebūvētu paštestēšanas vai izpildes laika verificēšanas mehānismu nav jauna [BARB03]. Autori definē paštestēšanu: "Termins "iebūvēti testi" ir attiecināms uz programmatūras kodu, kas pārbauda tās darbību izpildes laikā" [BIND00]. Paštestēšanas ideja jau ilgu laiku plaši tiek lietota elektronikā, kur iekārtas pašas pārliecinās par to darbības korektību - tajās ir iebūvēta paštestēšanai nepieciešamā instrumentācija un programmatūra, lai novērtētu iekārtu pareizu darbību. Tikpat labi paštestēšana var tikt iebūvēta arī programmatūrā [DIEB12].

2.1.1. Dokumentu plūsmu verificēšana

Lielākā daļa dokumentu pārvaldības sistēmu zināmā veidā implementē dokumentu apstrādes plūsmu mehānismu. Lai arī tiešā veidā parasti verificēšanas mehānismi šajās sistēmās nav iebūvēti, tomēr liela daļa no šīm sistēmām nodrošina iespēju plūsmā iekļaut dažādus atgādinājumus un brīdinājumus par izpildes periodu pārsniegšanu. Tādējādi, veidojot procesus ar šādu dokumentu pārvaldības sistēmu starpniecību, ir iespējams realizēt iekļautu verificēšanas mehānismu: tas tiek iekļauts dokumentu plūsmā.

2.1.2. Paštestēšana

Viens no viedtehnoloģiju ietvarā piedāvātajiem risinājumiem ir paštestēšana [DIEB12]. Autors piedāvā veidot paštestēšanu, iebūvējot sistēmā testēšanas punktus, kas nodrošina lietotāju veikto darbību un ierakstīto testa piemēru izpildes reģistrēšanu testa piemēru failā. Testa punktu iebūvēšana nodrošina, ka sistēma tiek testēta no sistēmas "iekšpuses" nevis no "ārpuses" kā to parasti veic trešo pušu testēšanas rīki. Iebūvējot testēšanas darbības pašā sistēmā, ir iespējams notestēt sistēmas funkcionalitāti, ko nespēj notestēt rīki, kas veic sistēmas testēšanu no ārpuses.

Lai nonāktu pie praktiski lietojama rīka, šīs paštestēšanas idejas autors piedāvā veidot rīku, kurš sastāv no diviem moduļiem un pašā sistēmā iebūvētiem testa punktiem:

- **Paštestēšanas modulis** - bibliotēka (dll fails), kas satur paštestēšanas funkcijas, kuru izsaukumi tiek iekļauti testējamā sistēmā. Tas nozīmē, ka, testējot programmatūru, lietotājiem nav nepieciešams uzstādīt papildus testēšanas rīkus. Sistēmas testēšanu nodrošina paštestēšanas programmatūra, kas daļēji ir iebūvēta pašā sistēmā.
- **Paštestēšanas testu pārvaldes modulis** - lietojumprogramma, kas nodrošina jaunu testu veidošanu, esošu testu rediģēšanu, izpildi, rezultātu salīdzināšanu, rīka konfigurēšanu un citas iespējas. Šis modulis izmanto iepriekš aprakstīto paštestēšanas moduli. Tā pamatuzdevums ir nodrošināt testu automatizētu atspēlēšanu, izmantojot ierakstītos testa punktu XML failus un testējamajā sistēmā iebūvētos testa punktus.
- **Testa punkti** - komandas, pie kurām tiek izpildītas sistēmas testēšanas darbības. Precīzāk - testa punkts ir programmēšanas valodas komanda programmas tekstā, pirms vai pēc kuras tiek izsauktas testēšanas darbību komandas.

Paštestēšanas pieeja nosaka, ka testu punkti testējamā sistēmā jāizvieto tā, lai tie nosegtu sistēmas kritisko funkcionalitāti. Vienlaicīgi paštestēšanas pieeja pieļauj testa punktu izvietošanu tā, lai nodrošinātu:

- testēšanu pēc baltās kastes metodes, iestrādājot testa punktus, kas pārklāj visu sistēmas funkcionalitāti;
- svarīgākās funkcionalitātes testēšanu, piemēram, datu saglabāšanu, iestrādājot dažus svarīgus testa punktus.

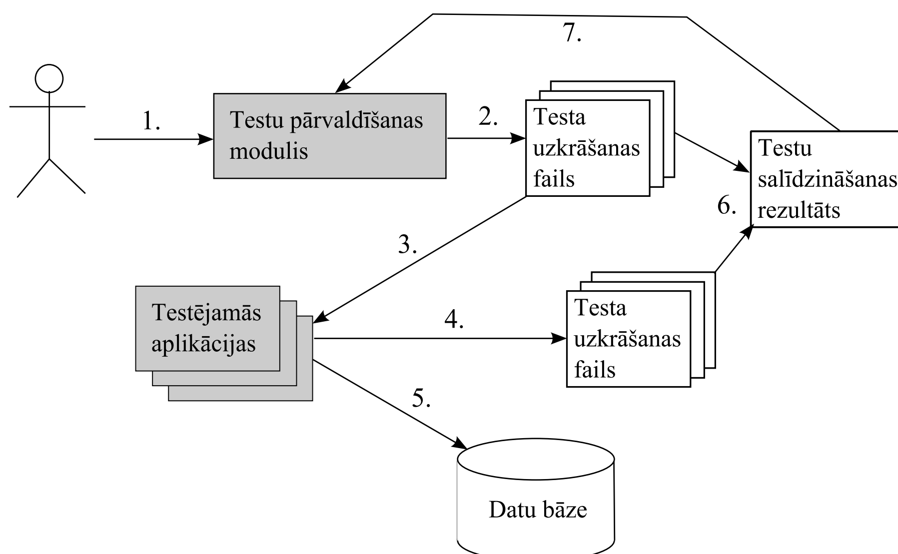
Programmētāja ziņā ir izvēlēties nepieciešamo testa punktu izvietošanas stratēģiju – kritiskās funkcionalitātes testēšana vai testēšana pēc baltās kastes metodes, vai testa punktu

izvietošana, ņemot vērā jutību pret pirmkoda izmaiņām. Jo mazāk sistēmā iestrādātu testa punktu, jo sistēma mazāk jutīga pret pirmkoda izmaiņām.

Sistēmu, kurā ir implementēta paštestēšana šeit piedāvātā veidā, iespējams darbināt vairākos režīmos:

- **Testu uzkrāšanas režīms**. Šajā režīmā tiek veikta jaunu testu definēšana un esošu testu rediģēšana un dzēšana.
- **Paštestēšanas režīms**. Šajā režīmā tiek veikta automatizēta programmatūras paštestēšana, automatizēti izpildot uzkrātos testa piemērus.
- **Lietošanas režīms**. Režīmā netiek veiktas nekādas testējošas darbības. Šajā režīmā sistēmas lietotājs izmanto sistēmas pamata funkcionalitāti.
- **Demonstrācijas režīms**. Ja datu bāzē ir uzkrāta informācija par korekti veiktajiem testiem, tad šo testu kopu būtu lietderīgi izmantot demonstrācijās, jaunu darbinieku apmācībai, sistēmas demonstrēšanai citiem interesentiem u.c. gadījumos.

Paštestēšanas režīma darbības shēma ir apskatāma attēlā (skat. 4. attēls).



4. attēls. Paštestēšanas režīms

Tā paredz šādas darbības:

1. lietotājs, lai izsauktu sistēmas paštestēšanu, atver testu vadības bloka logu;
2. logā lietotājs ielasa testu failus un norāda, kurus no ielasītajiem failiem izpildīt;
3. testu vadības bloks nolasa informāciju no testa faila, kas tiek izpildīts, un veic atbilstošās darbības, kas norādītas testa failā;
4. paštestēšanas režīmā, tāpat kā testu uzkrāšanas režīmā tiek veidots testa fails (tiek izmantota tā pati pieeja, kas - testa uzkrāšanas režīmā);

5. ja testa piemērā tiek veikta datu saglabāšana, tad dati tiek saglabāti datu bāzē (lietošanas vides gadījumā saistītie dati tiek ielasīti no reālās datu bāzes, savukārt, saglabāšana tiek veikta testu reģistrēšanas un izpildes datu bāzē, nevis reālajā datu bāzē);
6. pēc testu izpildes tiek salīdzināts testa uzkrāšanas režīmā izveidotais fails ar paštestēšanas režīmā izveidoto failu: ja failu saturs sakrīt, tad testa izpilde ir notikusi veiksmīgi, savukārt, ja nesakrīt, tad tas ir bijis neveiksmīgs;
7. informācija par testa rezultātu tiek attēlota lietotāja testu vadības blokā, kur neveiksmīga testa izpildes gadījumā lietotājam ir iespējams detalizēti iepazīties ar testu rezultātiem.

Risinājuma autors piedāvā ne tikai tehnoloģisku risinājumu, bet arī metodoloģisku aprakstu tā lietošanai. Autors izšķir trīs vides: izstrādes, testēšanas (ar šo tiek saprasta "akcepttesta" vide, kurā testēšanu veic problēmas apgabalu pārzinoši lietotāji) un lietošanas. Paštestēšanas implementēšana sākas jau programmatūras izstrādes sākumā izstrādes vidē. Lai nodrošinātu paštestēšanu, jau projektējot sistēmu, ir jāplāno paštestēšanas iebūvēšana. Izstrādes vidē izstrādātāji:

- sagatavo testa piemērus, kas pārklāj sistēmas kritisko funkcionalitāti;
- nodrošina testu piemēru importu no testa un lietošanas vides;
- izpilda testēšanu, to veicot automatizēti ar paštestēšanas funkcionalitātes atbalstu.

Programmētājs vai izstrādes programmētāju grupa veic sistēmas uzlabošanu tik ilgi, kamēr, lietojot paštestēšanu, tiek atklātas kļūdas. Iespējams, ka kļūdu iemesls ir programmatūra, bet varētu būt, ka tās iemesls ir nekorekts paštestēšanas mehānisms. Tādēļ tikai pēc veiksmīgas sistēmas testēšanas programmētājs nodod savu sistēmu neatkarīgai testēšanai testa vidē. Tādējādi jau izstrādes vidē tiek pārbaudīts būtisks kļūdu kopums. Turklāt paštestēšanā būtu iekļaujami scenāriji, kuri aptver plašāku piemēru kopu kā vienībtesti. Šīs metodes autors norāda, ka prakse rāda, ka, izmantojot paštestēšanas pieeju, neatkarīga testēšana reti atklāj programmatūras defektus.

Testa vidē testēšanu veic problēmas apgabalu pārzinoši lietotāji. Šeit tiek nodrošināti reālāki sistēmas izpildes apstākļi, un parasti šajā vidē ir iespēja testēt programmatūras ārējās saskarnes, kuras bieži vien izstrādes vidē nav pieejamas. Sistēmas konfigurācijas pārvaldnieks nodrošina testu piemēru uzstādīšanu no lietošanas vides un izstrādātāju piegādātās programmatūras, sagatavoto testu un to rezultātu uzstādīšanu testa vidē. Testētāji testa vidē:

- akceptē vai noraida no izstrādes vides saņemtos testus (izstrādes vidē sagatavotie testi testa vides paštestēšanā tiek iekļauti tikai tad, ja sistēmas testētājs testa vidē tos ir apstiprinājis);
- sagatavo testa piemērus, kas papildina izstrādātāju sagatavotos testa piemērus;
- veic piegādātās sistēmas testēšanu, tādā veidā pārliecinoties, vai programmatūras funkcionalitāte nav sabojāta ar jaunās funkcionalitātes vai izmaiņu ieviešanu, kā arī tiek pārbaudīts, vai testa vidē ir piegādāti visi ar jaunām izstrādēm saistītie nodevumi (datu struktūras, izejas tekstu faili, testu piemēri).

Lietošanas vidē ir pieejami reālie dati, kā arī ir pieejamas visas ārējās saskarnes lietošanas režīmā. Tādējādi lietošanas vidē ir iespējams konstatēt situācijas, kuras dažādu iemeslu dēļ nav saprotamas testa un izstrādes vidē. Sistēmas lietošanas vidē lietotāji:

- sagatavo jaunus testa piemērus, kas papildina izstrādes un testa vidē sagatavotos testa piemērus ar reāliem lietošanas vides piemēriem;
- veic automatizētu testu izpildi (sistēmas paštestēšanu), turklāt reālie dati lietošanas vidē netiek mainīti.

Lietošanas dati sistēmas paštestēšanas režīmā tiek izmantoti tikai lasīšanas (read-only) režīmā. Datu saglabāšana notiek testu izpildes datu bāzē. Testu rezultātā iespējams pārliecināties, vai piegādātā programmatūra nav nekorekti mainījusi esošo sistēmas funkcionalitāti un vai lietošanas vidē uzstādīti visi nepieciešamie programmatūras nodevumi.

Paštestēšanas pieejas autors ir novērtējis paštestēšanas efektivitāti [DIEB12]. Efektivitātes novērtēšanai tika izmantota reāla informācijas sistēma, tomēr paštestēšana tajā netika implementēta, jo tas neļautu novērtēt paštestēšanas efektivitāti - visas identificētās un novērstās kļūdas netiktu atklātas. Tādēļ efektivitātes novērtēšana tika veikta analītiski. Tika apskatīti 1171 reģistrētie sistēmas problēmu pieteikumi un tika novērtēts, vai tie tiktu identificēti ar paštestēšanas mehānismu. Autors novērtēja, ka 4/5 no kļūdām tiktu pamanītas automātiski.

Vērtējot paštestēšanu var iezīmēt daudzas pozitīvas iezīmes:

- paštestēšana rada papildus izstrādes darbu, tomēr ievērojami samazina vēlāko testēšanas darbu;
- paštestēšana nodrošina regresu testu izpildi ļoti plašai sistēmas funkcionalitātei, tādējādi atvieglojot sistēmas labojumu un modifikāciju testēšanu;

- paštestēšana pēc definīcijas ir automatizēta.

Tomēr paštestēšanai, salīdzinot ar izpildes laika verificēšanu, ir zināmi mīnusi - tā nav izpildāma nepārtraukti. Tā paredz visas vai daļējas testu kopas izpildi vai nu pēc lietotāja pieprasījuma, vai arī noteiktos sistēmas darbības brīžos, piemēram, pārstartējot tās darbību. Tomēr tiek ievērota viena no būtiskākajām izpildes laika verificēšanas prasībām: programmatūra tiek verificēta lietošanas vidē un ar lietošanas vides datiem (paštestēšanas gadījumā testa piemēri regulāri būtu jāpapildina ar lietošanas vides testa piemēriem).

2.2. Ārēji izpildes laika verificēšanas risinājumi

Šī verificēšanas risinājumu grupa paredz, ka pārbaudāmā programmatūrā netiek pielāgota verificēšanas mērķiem. Šādai pieejai ir vairāki plusi:

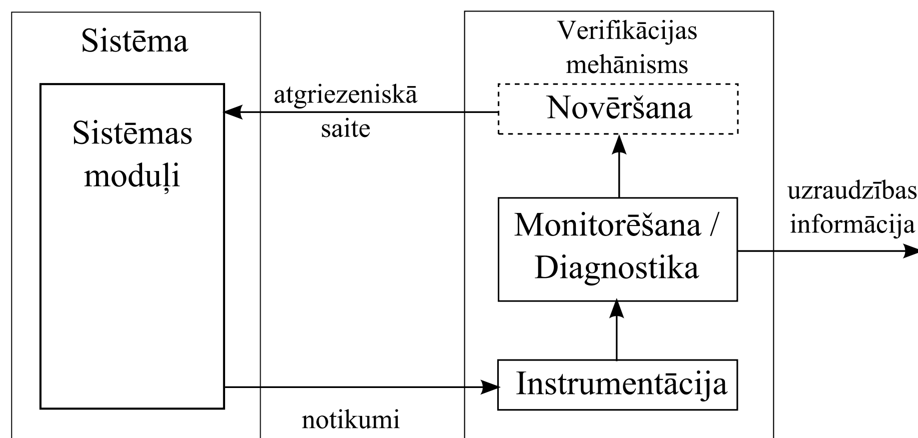
- iespējams verificēt programmatūru, to nemainot;
- kā vienu procesu ir iespējams verificēt vairākas sistēmas;
- ir iespējams verificēt mantotu programmatūru (no angļu valodas - *legacy software*).

Tomēr, acīmredzot, šai pieejai ir arī vairāki mīnusi:

- lai arī verificēšana ir ārēja, tā var negaidīti ietekmēt verificējamās programmatūras darbību, piemēram, izpildes trases faila lasīšana var radīt problēmas trases rakstīšanai;
- tā kā programmatūras kods var būt nepieejams, var trūkt pietiekami detalizētas informācijas procesa verificēšanas kritēriju definēšanai.

Ilgu laiku izpildes laika verificēšana tika attiecināta uz kritiskām sistēmām, kuras nodrošināja, piemēram, lidmašīnu vadību, medicīnas iekārtu darbību u.tml. Tomēr lielākoties šajos gadījumos tika implementēti iebūvēti verificēšanas risinājumi. Ārēji verificēšanas risinājumi pastiprināti tika pētīti, parādoties servisu orientētai arhitektūrai (SOA). SOA paredz, ka programmatūra tiek projektēta un izstrādāta kā atsevišķas komponentes, kas nodrošina zināmus servisus viena otrai. Lai servisu pielietojums būtu pēc iespējas plašs, tie tiek veidoti kā autonomas programmatūras vienības. Katru servisu raksturo tā kontraks jeb servisa apraksts (saskarne). Citas komponentes lieto nepieciešamos servisus atbilstoši to kontraktam. Tādējādi tiek uzskatīts, ka komponenšu, kuras lietos servisus, izveidei ir nepieciešams zināt tikai servisa kontraktu. Tas nodrošina savstarpēji atsaistītu servisu izstrādi, kur dažādus servisus nodrošina atšķirīgi uzturētāji. Tā kā šādi servisi var mainīties savstarpēji nesaskaņoti, ir svarīgi nodrošināt nepārtrauktu servisu darbības verificēšanu, lai savlaicīgi pamanītu saistītu servisu darbības anomālijas. Šādu izpildes laika verificēšanu vispārīgā gadījumā ir iespējams veikt tikai ar ārēju komponenti, jo, iespējams, daudzi no saistītajiem

servisiem verificētājam ir pieejami tikai kā to publiski pieejamās saskarnes. Tādējādi, attīstoties SOA, paralēli tika pētīti arī iespējamie SOA izpildes laika verificēšanas risinājumi.



5. attēls. Ārēja izpildes laika verificēšana

Ārējā izpildes laika verificēšanas mehānismā (skat. 5. attēls) visas verificēšanas komponentes ir implementētas ārējā verificēšanas mehānismā. Tajā ir iekļauta gan instrumentācija, gan monitorēšana, gan diagnosticēšana. Instrumentācija pēc ārejām pazīmēm konstatē paralēli strādājošas informācijas sistēmas notikumus, monitorēšana tos filtrē un diagnosticēšana identificē problēmas. Informāciju par visiem notikumiem sniedz verificēšanas mehānisms. Šādā mehānismā ir iespējams iebūvēt arī "novēršanas" komponentes, tomēr tas var būt visai sarežģīts uzdevums. Vispārīgā gadījumā tas var būt neiespējami: verificēšanas mehānismam jāinterpretē verificējamās sistēmas darbība un jāspēj to ietekmēt, lai novērstu konstatētās problēmas. Kaut kādā apjomā šādu uzdevumu veic operētājsistēmas, piemēram, apstādinot nekorekti strādājošus procesus vai piešķirot procesiem nepieciešamo papildus atmiņu vai samazinot atbrīvoto.

Turpmākajās nodaļās tiek apskatīti vairāki ārējas izpildes laika verificēšanas risinājumu piedāvājumi, no kuriem viens ir paredzēts SOA.

2.2.1. Programmatūras izpildes vides testēšana

Programmatūras izpildes vides pārbaude [RAUH09] tiek piedāvāta kā viens no viedtehnoloģiju veidiem. Šis risinājums ir attiecināms arī uz ārējiem izpildes laika verificēšanas risinājumiem.

Programmatūras izpilde vienmēr ir saistīta ar kādiem ārējiem apstākļiem, kas nosaka tās izpildi, piemēram, operētājsistēma, pieejamās iekārtas, citu resursu pieejamība u.c. Pētījumi norāda, ka 40% no visām programmatūras atteicēm tiek radīti nevis programmatūras kļūdu dēļ, bet gan nepareizas programmatūras pārvaldīšanas dēļ: kļūdas tās konfigurēšanā, vides nepiemērotībā u.c. [STER05]. Šajā nodaļā aprakstītā risinājuma autors piedāvā izveidot

universālu mehānismu sistēmu vides prasību aprakstīšanai un verificēšanai. Verificēšanu var veikt, sistēmu nedarbinot (ar ārēju komponenti), tomēr tikpat labi verificēšanas komponenti var izsaukt arī pati programmatūra.

Risinājuma autors piedāvā katrai sistēmai izveidot konfigurācijas jeb vides prasību profilu, kurā tiek aprakstīti visi verificējamie vides parametri, piemēram:

- sistēmas darbībai nepieciešamās bibliotēkas;
- nepieciešamie datu bāzu serveri;
- nepieciešamie citu serveru resursi;
- nepieciešamie konfigurācijas faili un tajos esošās vērtības (iespējama arī šo vērtību verificēšana).

Vienkārša apraksta piemērs ir parādīts nākamajā pirmkodā (skat. 1. pirmkods).

1. pirmkods. Konfigurācijas apraksta piemērs

```
<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>
```

Tomēr autors nepiedāvā vienu apraksta veidu, bet iespēju izvēlēties kādu no vairākiem industrijas standartiem: XML, RDF vai IUDD bāzētus, t.i., tiek apskatīti vairāki veidi, kā var veidot programmatūras konfigurācijas profilu.

Vides pārbaudi veic vides testēšanas koordinatos. Lai pārbaudītu visus konfigurācijas parametrus, testēšanas koordinators lieto prasību pārbaudes moduļus. Pārbaudes moduļu izvēle un ielāde nodrošina šī risinājuma universālo raksturu - pārbaudes moduļus var veidot dažādām vidēm un dažādiem parametriem. To klāstu var brīvi papildināt, mainoties videi. Lai, ieviešot jaunu pārbaudes moduli, nebūtu jāmaina testēšanas koordinators:

- ir noteikta pārbaudes moduļu saskarne;
- testēšanas koordinatorā tiek uzturēta pieejamo pārbaudes moduļu datu bāze.

Saskaņā ar autora ideju katram vides rādītājam atbilstošo pārbaudes moduli piemeklē prasību-pārbaužu kontrolieris. Tas uztur pārbaudes moduļu vārdnīcu, t.i., pārzina pieejamo vides pārbaudes moduļu klāstu un to, kuri moduļi spēj testēt katru no prasībām. Testēšanas koordinators izmanto prasību-pārbaužu kontroliera sniegto informāciju tālākam darbam: pēc kārtas atrod visu pārbaudes moduļu izpildāmos failus, tos ielādē atmiņā un izpilda pārbaudes.

Tā kā autors apskata dažādas koordinatora implementēšanas iespējas, tad viņš piedāvā apskatīt trīs dažādus pārbaudes moduļu darbināšanas veidus:

- **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 [FIRE14]. Līdz ar to pārbaudes moduļa veiktās pārbaudes attieksies uz pašas pārbaudes moduļa 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, 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, izmantojot koda funkciju izsaukumus.

Kā redzams, tad risinājuma autors norāda, ka pārbaudes moduļa ielāde lietojumprogrammas procesā var sniegt vislabākos verificēšanas rezultātus, jo tādējādi pārbaudes modulim ir pieejami tieši tie paši resursi, kuri lietojumprogrammai. Savukārt, šis

risinājums paredz konfigurācijas pārbaudes iebūvēšanu verificējamajā sistēmā. Tādējādi arī šeit tiek norādīts, ka, lai arī risinājums ir paredzēts "ārējai" lietošanai, t.i., kad testēšanas koordinators tiek darbināts neatkarīgi no verificējamās programmatūras, tomēr visprecīzākos verificēšanas rezultātus var iegūt, lietojot to kā "iebūvētu" verificēšanas risinājumu.

Visu vides parametru pārbaudi varētu iebūvēt arī pašā programmatūrā (tā arī lielākoties tiek darīts), tomēr risinājuma autors norāda vairākas sava risinājuma piekšrocības:

- 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 un ir noderīgi arī datorsistēmu uzturētājiem konfigurācijas maiņas gadījumos;
- izstrādātos vides pārbaudes rīkus iespējams atkārtoti izmantot citu programmatūras sistēmu būvē.

Papildus jānorāda, ka lielākā daļa no priekšrocībām ir spēkā neatkarīgi no tā, vai testēšanas koordinators tiek iebūvēts programmatūrā vai arī tiek lietots kā atsevišķa ārēja komponente. Priekšrocības nodrošina ārēji uzturēts konfigurācijas apraksts jeb sistēmas profils.

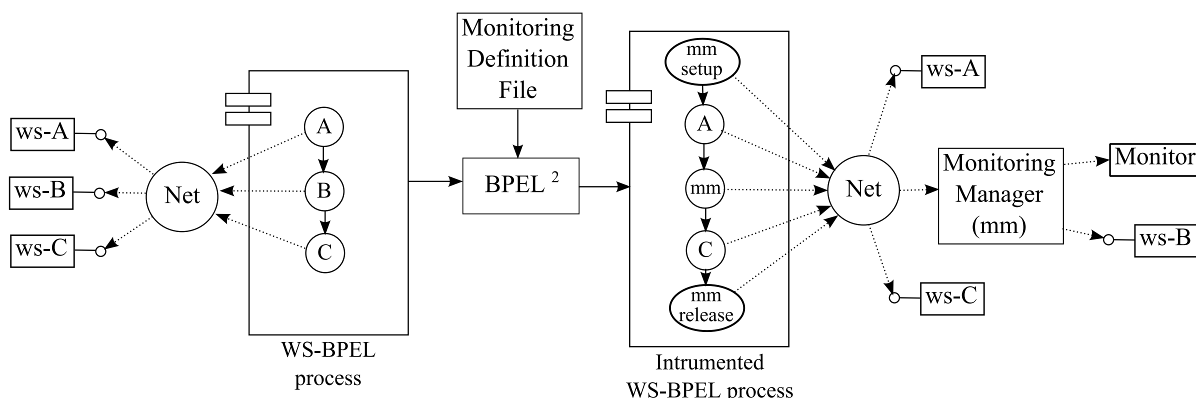
2.2.2. SOA risinājums WSCoL

Daudzi pētījumi ir veltīti SOA bāzētu izpildes laika verificēšanas risinājumiem. Risinājumu šādai videi piedāvā arī pētījuma [BARE05] autori. Viņi piedāvā dinamisku WS-BPEL [OASIS07] aprakstītu procesu verificēšanu.

Monitorēšanas nodrošināšanai katram verificējamam procesam tiek aprakstīti ārēji monitorēšanas likumi. Kā savā darbā norāda autori, ārēji definētu monitorēšanas likumu ieviešana ļauj realizēt vairākas monitorēšanas likumu kopas vienam un tam pašam WS-BPEL procesam. Monitorēšanas likumu definēšanai webservisi tiek aprakstīti ar UML [OMG11B] klasēm. Savukārt, pārbaudāmie apgalvojumi tiek definēti ar speciālas autoru izveidotas

valodas WSCoL (*Web Service Constraint Language*) palīdzību, par pamatu ņemot valodu JML (*Java Modeling Language* [LEAV06]).

Definētā WS-BPEL procesa verificēšanu veic starpniekserviss (skat. 6. attēls).



6. attēls. WS-BPEL procesa verificēšanas piemērs {schema:WSCoLachitecture}

Šajā shēmā autori parāda viņu risinājuma darbības piemēru:

- Attēla kreisajā pusē ir attēlots WS-BPEL process, kurā notiek trīs secīgi servisu izsaukumi.
- Monitorēšanas apraksta failu jeb monitorēšanas likumus var aprakstīt procesa izstrādes laikā vai arī vēlāk lietošanas laikā. Likumi tiek piesaistīti konkrētam definētā biznesa procesa elementam. Šajā gadījumā tas ir servisa B izsaukums.
- Kad tiek iespējota definētā likuma monitorēšana, *BPEL*^{2} pārslēdz WS-BPEL izpildi uz instrumentēto WS-BPEL procesa izpildi, kurā saskaņā ar monitorēšanas likumiem saistītie elementi tiek aizstāti ar monitorēšanas pārvaldnieka (*Monitoring Manager*) izsaukumiem. Šī piemēra gadījumā tas tiek attiecināts uz servisa B izsaukumu.
- Monitorēšanas pārvaldnieks izlemj, vai verificēt pieprasītā servisa izsaukumu. Tas tiek noteikts atbilstoši verificēšanas prioritātei: ja monitorēšanas likuma prioritēte ir zemāka par pārvaldniekā uzstādīto, tad serviss netiek verificēts. Pretējā gadījumā servisa izpilde tiek verificēta atbilstoši monitorēšanas likumam un rezultāts nosūtīts uz "Monitoru" (saskarne verificēšanas paziņojumu saņemšanai, apstrādei un analīzei).
- Lietojot Monitorēšanas pārvaldnieka saskarni, lietotājs var noteikt dažādus monitorēšanas līmeņus, iespējot un atspējot verificējamos likumus u.c.

Koda pārslēgšanu veic BPEL^2. Tā uzdevums ir analizēt monitorēšanas likumus un atbilstoši papildināt izpildāmo procesu ar izpildes elementiem, lai nodrošinātu dinamisku

monitorēšanu. Ja likumā ir norādīts pēcizpildes nosacījums (no angļu valodas - *post-condition*), tad BPEL² izsauc prasīto servisu ar Monitorēšanas pārvaldnieka starpniecību, pēc tam analizējot no pārvaldnieka saņemto rezultātu. Konkrētā nosacījuma pārbaudi nodrošina Monitorēšanas pārvaldnieks. Savukārt, BPEL² izlemj, vai, saņemot rezultātu, procesa izpilde drīkst tikt turpināta. Līdzīgi tiek apstrādāti arī priekšizpildes nosacījumi (no angļu valodas - *pre-condition*), ar vienīgo atšķirību, ka Monitorēšanas pārvaldnieks pārliecinās par nosacījuma izpildi pirms servisa izsaukuma un kļūdas gadījumā atgriež kļūdas paziņojumu, servisu neizsaucot.

Vienmēr pirms procesa uzsākšanas BPEL² nosūta inicializācijas pieprasījumu Monitorēšanas pārvaldniekam, norādot nepieciešamos monitorēšanas parametrus. Tāpat procesa izpildes beigās BPEL² nosūta Monitorēšanas pārvaldniekam paziņojumu par monitorēšanas pārtraukšanu.

Monitorēšanas likumu definīcijas fails sastāv no trīs daļām:

- pirmā daļa sniedz vispārēju aprakstu par WS-BPEL procesu, uz kuru šīs likumu definīcijas ir attiecināmas un kurš tiks monitorēts;
- otrā daļa apraksta kopējo procesa izpildi un ietekmē monitorēšanas aktivitāšu skaitu, kas tiks realizētas procesa izpildes laikā;
- savukārt, trešā sadaļa apraksta atsevišķos monitorēšanas likumus un ir uzskatāma par monitorēšanas pamatu.

Katrs likums tiek aprakstīs ar trīs komponentēm:

- pirmā no komponentēm apraksta precīzu WS-BPEL procesa pozīciju (elementu), kurā likums ir jāpielieto, un nosaka, vai tas ir pirmsizpildes vai pēcizpildes nosacījums;
- otrā komponente satur monitorēšanas parametrus (meta-informācija, kura definē monitorēšanas likuma kontekstu):
  - prioritāte - skaitlis no viens līdz pieci, kurš nosaka likuma izpildes prioritāti;
  - derīgums - laika periods, kurā likums ir derīgs;
  - sertificētie servisa nodrošinātāji (no angļu valodas - *Certified Providers*) - saraksts ar servisa nodrošinātājiem. Tā kā servisu var nodrošināt viens vai vairāki nodrošinātāji, tad šis saraksts ļauj definēt uzticamos nodrošinātājus, t.i., tādus, kuru servisu izsaukumus nevajag vai arī vajag verificēt.
- trešā komponente satur verificējamā nosacījuma izteiksmi.

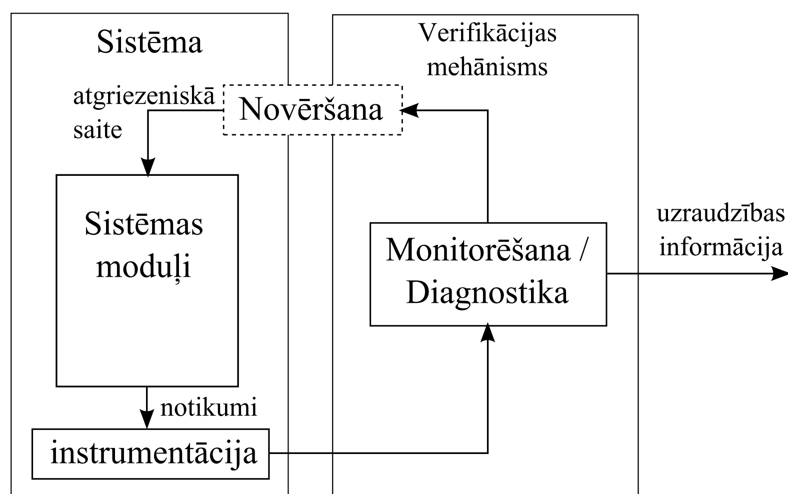
Šādi izmantojot ārēja starpniekservera (Monitorēšanas pārvaldnieks) palīdzību, autori ir piedāvājuši realizēt atsevišķu webservisu, kuri tiek lietoti procesa vajadzībām, izsaukumu verificēšanu. Monitorēšanas definīcijas fails ļauj aprakstīt gan visa WS-BPEL procesa verificēšanu, gan tikai kādu tā elementu apakškopu. Turklāt, lietojot dažādus monitorēšanas likumus, viena un tā paša elementa izsaukumu var verificēt no dažādiem aspektiem. Lai arī risinājums netiek iebūvēts verificējamajā procesā, tas strādā sinhroni ar verificējamo procesu, jo aizstāj procesa izpildes platformu: ja monitorēšana ir izslēgta, to izpilda standarta WS-BPEL serviss, savukārt, ja monitorēšana ir ieslēgta, izpilde tiek uzticēta instrumentētam WS-BPEL servisam.

2007. gadā WSCoL autori piedāvāja WSCoL valodas paplašinājumu - *Timed WSCoL*. Tas papildina valodu ar laika ierobežojumu verificēšanas nosacījumiem. Šajā gadījumā laika nosacījuma likumi nesatur norādi uz konkrēta WS-BPEL elementu, bet gan apraksta veselu elementu kopu, tādējādi norādot ietvaru, kurā izpildes laiks ir jāverificē. Valodas autori ievieš arī vairākus predikātus, kuri ir lietojami monitorēšanas likumos, piemēram, "kamēr" un "starp". Lai nodrošinātu pareizu likumu izpildi, autori piedāvā arī Monitorēšanas pārvaldnieka un BPEL[2] modifikācijas.

Šī risinājuma autori 2008. gadā ir izveidojuši ietvaru DynamoAOP [BIAN07], kurā ir realizēta sākotnējās autoru idejas un nepaplašinātā WSCoL valoda. Spriežot pēc turpmākajām publikācijām, valoda *Timed WSCoL* šobrīd nav implementēta un tās efektivitāte ar prototipu palīdzību nav novērtēta.

2.3. Kombinēti izpildes laika verificēšanas risinājumi

Kombinēti izpildes laika verificēšanas risinājumi paredz vienlaicīgi lietot gan "iebūvētas", gan "ārējas" komponentes (skat. 7. attēls).



7. attēls. Kombinēts izpildes laika verificēšanas risinājums{schema:verificationMixed}

Atšķirībā no iebūvētiem un ārējiem risinājumiem šajos risinājumos komponenšu izvietojums var būt visai atšķirīgs, tomēr pamatā ir zināmas kopīgas iezīmes:

- Instrumentācija parasti ir iebūvēta verificējamajā sistēmā, jo tas ir veids, kā notikumus identificēt visprecīzāk un visdetalizētāk.
- Monitorēšana un diagnosticēšana tiek realizēta ārējā programmatūrā (verificēšanas mehānismā).
- Novēršana var tikt realizēta atšķirīgi: gan iekļaujot sistēmā, gan - verificēšanas mehānismā. Abi no risinājumiem ir iespējami. Iebūvējot sistēmā, novēršana var tikt implementēta visefektīvāk, jo šādi ir pieejama visa informācija par sistēmu, tās funkcionalitāti un konfigurāciju. Šāds risinājums paredz, ka verificēšanas mehānismā ir implementēta saskarne, kura sniedz diagnostikas informāciju par verifikācijas konstatējumiem: precīzu problēmu aprakstu. Savukārt, ja novēršanas komponente tiek iebūvēta verifikācijas mehānismā, tad novēršanas iespējas ir visai ierobežotas, jo verificēšanas mehānismam ir ierobežota informācija par verificējamām sistēmām.

Daļa no jauktajiem risinājumiem paredz trasēšanas funkcionalitātes iebūvēšanu izstrādājamajā programmatūrā, nodrošinot universālus rīkus trases analīzei. Iebūvējot programmatūras trasēšanu, izstrādātāji paši var izlemt, kurā brīdī veikt trases papildināšanu. Atbilstoši autoru piedāvājumiem [HALL08, KORT01] var tikt apskatīti divi dažādi trases analīzes piedāvājumi: vieni paredz trases analīzi paralēli verificējamās programmatūras izpildei, citi - pēc programmatūras izpildes. Pēdējie no šiem tikai daļēji atbilst izpildes laika verificēšanas idejai, jo neidentificē izpildes problēmas to notikšanas brīdī, tomēr verificēšanu veic, balstoties uz lietošanas vides datiem.

Citi autori piedāvā automātiski ģenerēt instrumentācijas kodu, kurš tiek iekompilēts izstrādājamajā programmatūrā. Savukārt, monitorēšanas rīki arī šajos gadījumos parasti ir universāli.

Turpmākajās apakšnodaļās tiks apskatīti dažu autoru kombinēti izpildes laika verificēšanas risinājumu piedāvājumi.

2.3.1. Trases analizatori

Viena no autoru grupām piedāvā atbalstu C/C++ valodās rakstītas programmatūras verificēšanai [KORT01]. Autori piedāvā bibliotēku, kura nodrošina ērti konfigurējamas trases veidošanu. Bibliotēkā ir iekļautas:

- funkcijas trases inicializēšanai;
- plašs beznosacījuma un nosacījuma trases veidošanas funkciju klāsts;
- funkciju un metožu izsaukumu trasēšanas funkcijas.

Trasēšanas mehānisms ar konfigurācijas palīdzību var tikt pāradresēts vai nu uz failiem, vai arī uz SQL datu bāzēm, tādējādi ļaujot izstrādātājiem un lietotājiem izvēlēties ērtāko un vēlākai analīzei piemērotāko trasēšanas veidu. Autori ir izveidojuši risinājumu arī dalītu sistēmu lietošanas gadījumam. Tas ļauj izveidot centralizētu trases veidošanas serveri, uz kura informāciju nosūta visas darbstacijas vai serveri, kas izpilda verificējamo programmatūru.

Risinājuma autori piedāvā arī monitorēšanas rīku, kas ļauj analizēt ar trasēšanas palīdzību uzkrātos programmatūras izpildes datus. Autori norāda, ka bieži vien programmatūras izpildes datu anlīze un anomāliju identificēšana var būt visai sarežģīta, īpaši gadījumos, kad tiek analizēta dalītu sistēmu izpilde. Tāpēc autori piedāvā risinājumu, kurš ļauj identificēt notikumus pēc dažāda veida šabloniem. Kā viens no piemēriem varētu būt notikumu virkne: "ja ūdens līmenis akā pārsniedz noteiktu līmeni, ūdens sūknim ir jādarbojas 30 sekundes".

Lai identificētu notikumus pēc šādiem šabloniem, autori ir ieviesuši loģiku ITCL (*Interval Temporal Checking Logic*). Tajā ir ietverti divi bāzes primitīvi:

- **notikums**, kurš atbilst vienam trases ierakstam;
- **intervāls**, kurš raksturo attālumu starp diviem punktiem laikā.

Punkti laikā var būt kāda notikuma iestāšanās laiks vai relatīva laika nobīde pret notikuma iestāšanās brīdi. Aprakstot šablonus, lietotāji (programmatūras izstrādātāji vai testētāji) var lietot šo primitīvu kopas, tās apvienojot, šķeļot vai veicot iterācijas operācijas. Datu šabloni un attiecīgi arī iespējamā datu atlase tiek definēti, lietojot šīs ITCL izteiksmes.

Piemēram, ar ITCL izteiksmē var definēt nosacījumu: "vatmetram jāinformē par noslodzi, ja tvertnē ūdens līmenis ir virs līmeņa L un ir ieslēgts ūdenssūknis".

Autoru piedāvātās valodas risinājums ir datu analīzes rīkiem bagāts, un loģisko izteiksmju valoda ļauj aprakstīt pietiekami sarežģītus nosacījumu meklēšanas šablonus. Tomēr piedāvātais risinājums ļauj verificēt tikai valodās C un C++ rakstītu programmatūru, turklāt pēc programmatūras izpildes.

Cita autoru grupa piedāvā paralēli verificējamajai programmatūrai darbināt monitorēšanas programmatūru, kura izpildes laikā reaģē uz verificējamās programmatūras notikumiem [DUCA01]. Autori piedāvā verificējamajā programmatūrā iebūvēt pārtraukumpunktus (no angļu valodas - *breakpoint*), kuri tiek izsaukti pie zināmiem nosacījumiem (piemēram, "manīgā X vērtība ir 7"). Pārtraukumpunktā, izpildoties norādītajam nosacījumam, monitorēšanas procesam tiek nosūtīta notikumu raksturojoša informācija. Lai izstrādātājiem būtu vienkāršāka notikumu identificēšana, monitorēšanas rīkā ir iebūvēts definējamu filtru mehānisms, kurš ziņo izstrādātājam tikai par filtram atbilstošu notikumu iestāšanos.

2.3.2. Universāls analizators - ProM ietvars

ProM ietvara autori savu platformu jeb ietvaru sauc par procesu datu iegūšanas rīku [DONG05], tomēr tas zināmā mērā atbilst arī izpildes laika verificēšanas rīkiem: lai arī tas nenodrošina procesu verificēšanu to izpildes laikā, tomēr ar tā palīdzību ir iespējams verificēt pabeigtos procesus. Procesu datu iegūšana paredz informācijas iegūšanu no procesu trases vai transakciju failiem. Katram trases ierakstam vajadzētu atspoguļot vienu notikumu, norādot vismaz šādu informāciju:

- **aktivitāte** - katrs notikums attiecas uz vienu aktivitāti;
- **instance** - katrs notikums attiecas uz kādu procesa instanci jeb gadījumu;
- **iniciators** - katrai notikuma instancei ir tās izpildītājs jeb iniciators;
- **laika zīmogs** - visiem notikumiem ir jābūt laika zīmogam un sakārtojamiem to notikšanas secībā.

Autori piedāvā skatīties uz šāda veida datiem no trīs dažādām perspektīvām:

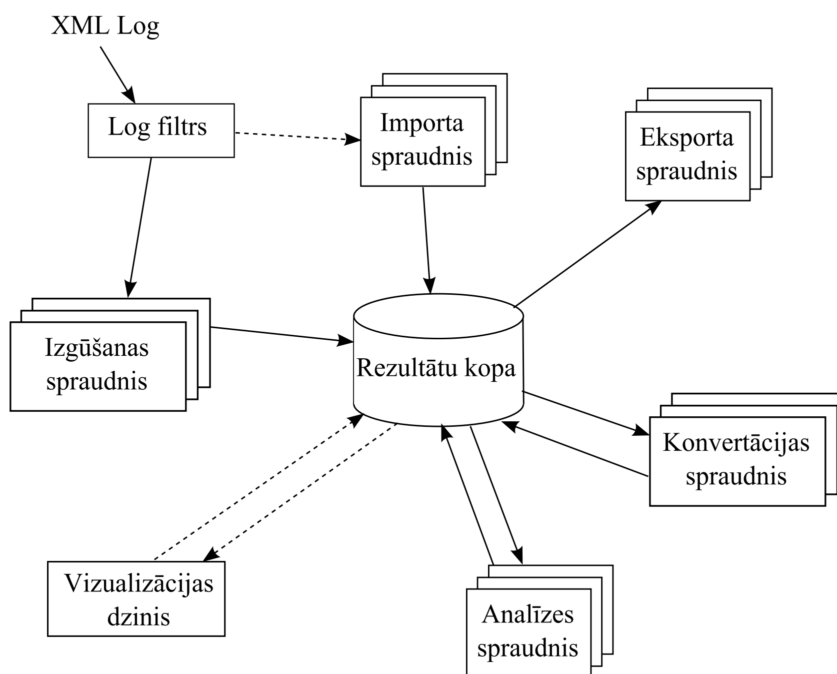
- **Procesu perspektīva** - datu analīze tiek vērsta uz procesu izpildes plūsmu, t.i., aktivitāšu secību. Šīs analīzes mērķis ir izveidot iespējamo procesu izpildes ceļu apkopojumu, attēlojot tos ar Petri tīklu [JENS09] vai EPC (*Event-driven Process Chain* [SCHE05]) palīdzību.
- **Organizācijas perspektīva** - analizē datus, par atskaites punktu izvēloties aktivitātes iniciatoru, t.i., kuri izpildītāji ir iesaistīti procesā un kā tie ir

saistīti. Šīs analīzes mērķis ir strukturizēt organizāciju, klasificējot iesaistītos izpildītājus pēc to lomām vai organizacionālās struktūras.

- **Instances perspektīva** - apskata instances izpildi. Šajā gadījumā datu analīze fokusējas uz procesu instanču izpildes ceļiem, kā arī iespējamajām procesu elementu vērtībām.

Ortogonāli norādītajām datu analīzes perspektīvām tiek analizēta procesu izpilde, apskatot loģiskās un ātrdarbības problēmas.

ProM ietvara arhitektūra ir attēlota nākamajā shēmā (skat. 8. attēls). Lai datu analīze būtu pēc iespējas universāla, ProM ietvara autori ir izveidojuši vispārēju datu glabāšanas formātu, kurā tiek glabāti visi trases ieraksti - *rezultātu datu bāze*. ProM spēj nolasīt plašu XML formāta trases failu klāstu. Lai varētu apstrādāt liela apjoma datu glabātuves, tiek lietots *Log filtrs*. Lietojot *Importa spraudņus*, sistēmā var importēt plašu dažādu modeļu klāstu, sākot ar Petrī tīkliem un beidzot ar loģiskām formulām. Savukārt, reālo datu izgūšanu nodrošina *Izgūšanas spraudņi*, kas saņem nolasīto log failu datus un ielasa tos rezultātu datu bāzē. Ja ir nepieciešamas datu konvertācijas, piemēram, transformācijas no EPC uz Petrī tīkliem, tiek lietoti *Konvertācijas spraudņi*. Datu analīzei, agregēšanai un cita veida apstrādes veikšanai tiek lietoti *Analīzes spraudņi*.



8. attēls. ProM ietvara arhitektūra{schema:ProMArchitecture}

ProM ietvara būtiskākais plus ir atvērtā spraudņu arhitektūra. Pateicoties tai, ProM nodrošina ļoti plašu datu formātu analīzes iespēju, kā arī dažādu modeļu datu importu un eksportu. Piemēram, no pielietojamajiem modeļiem, ProM ietvars šobrīd nodrošina ne tikai

Petrī tīklu un EPC lietojumu, bet arī Sociālos tīklus (no angļu valodas - *Social Networks*), BPMN un citus.

Lai arī ProM ietvars nav paredzēts procesu verificēšanai to izpildes laikā, tomēr tas nodrošina ļoti plašu procesu izpildes analīzi. Piemēram, tas nodrošina savlaicīgu ātrdarbības problēmu vai "šauro vietu" identificēšanu, populārāko procesu izpildes ceļu analīzi u.c.

2.3.3. Verificēšanas rīku ģenerēšana

Citi autori piedāvā izpildes laika verificēšanas rīku ģenerēšanu [BENS05]. Šī risinājuma piedāvātāji pamatā ir orientējušies uz robotu lietotņu (tātad reālā laika procesu) monitorēšanu un verificēšanu.

Lai izskaidrotu verificēšanas mehānismu, autori vispirms iepazīstina ar robotu pamatuzbūvi. Tipiski robotu kontroles aplikācijas tiek strukturētas divos hierarhiskos slāņos: augstākais slānis ir *plānošanas* slānis, bet zemākais - *izpildes* slānis. Plānošanas slānis seko līdzi ieejā padotajam plānam, kas detalizēti apraksta visus paveicamās misijas soļus. Plānošanas slānis, atbilstoši plānam, nosūta komandas izpildes slānim, kas mēģina tās implementēt un atgriež izpildes rezultātu, norādot statusa informāciju par veiksmīgu vai neveiksmīgu komandas izpildi. Balstoties uz izpildes slāņa sniegtajām atbildēm un realizējamo plānu, plānošanas slānis pieņem lēmumus par turpmāko soļu izpildi. Piedāvātā risinājuma autoru pamatmērķis ir piedāvāt risinājumu reālā laika sistēmu verificēšanai, kuras darbojas atbilstoši iepriekš sniegtajam aprakstam.

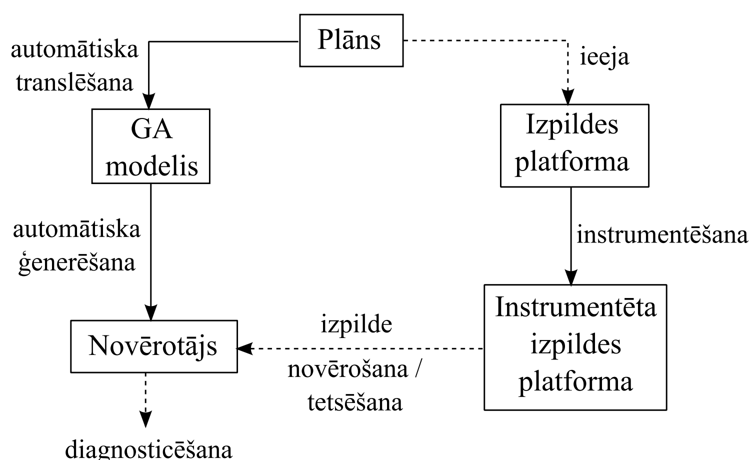
Autoru piedāvātais risinājums shematiski ir attēlots nākamajā shēmā (skat. 9. attēls - nepārtauktās līnijas attēlo modeļa un koda transformācijas, savukārt, raustītās - datu plūsmas). Tas sastāv no šādām fāzēm:

- no plāna automātiski tiek ģenerēts tam atbilstošs galīgs automāts [GAL14];
- izmantojot noģenerēto galīgo automātu, tiek ģenerēts "novērotājs" jeb monitors;
- testējamā sistēma, t.i., izpildes platforma tiek instrumentēta;
- testējamā sistēma tiek izpildīta un verificēta.

Pirmais solis konvertē plānu galīga automāta vai galīgu automātu tīkla formā (turpmāk - GA). Transformācijai ir jāsaglabā sākotnējā plāna semantika.

Pēc izpildes GA iegūšanas (t.i., precīza izpildes specifikācija), otrajā solī tiek ģenerēts "novērotājs". "Novērotājs" ir testēšanas rīks, kas verificē testējamo sistēmu, pārbaudot tās ģenerētās trases atbilstību ģenerētajam GA. Tiek pieņemts, ka "novērotājam" pieejamā trase saturēs novērojamo notikumu virknes ar precīzi norādītiem izpildes laika

zīmogiem. "Novērotājs" atzīst trasi par atbilstošu GA, ja GA izpilde protenciāli varētu ģenerēt šādu trasi.



9. attēls. Instrumentācijas ģenerēšana

Trešais solis ir izpildes platformas instrumentēšana. Tās mērķis ir informācijas nodošana iepriekš ģenerētajam "novērotājam". Autori izšķir divus informācijas nodošanas veidus:

- tiešsaistē - platformas izpildes laikā, nodrošinot reālā laikā pieslēgumu "novērotājam";
- savrupu - darbinot platformu vairākkārt, iegūst trases rezultātus un pēc tam tos nodod "novērotājam".

Jebkurā no šiem gadījumiem izpildes platformai ir jāsniedz pilna informācija par izpildes notikumiem. Vienīgā atšķirība varētu būt laika zīmogā: ja tiek lietots tiešsaistes režīms informācijas nodošanai, laika zīmogs var tikt nenodots, tomēr tiešsaistes režīmā ir jārēķinās ar potenciālām informācijas nodošanas aizturēm.

Autori norāda, ka instrumentācija var tikt implementēta manuāli vai arī automātiski, to ģenerējot. Tas pamatā ir atkarīgs no verificējamās sistēmas sarežģītības. Īpaša uzmanība ir jāpievērš, lai implementētā instrumentācija neietekmētu verificējamās sistēmas uzvedību, tostarp pievēršot uzmanību arī radītajai virsslodzei, kas var būtiski ietekmēt reālā laika sistēmu darbību. Tomēr šīs ir problēmas, kas piemīt jebkuras instrumentācijas risinājumam un autori šīs problēmas neapskata.

Ceturtais un pēdējais solis ir pati verificēšana. Verificējamā sistēma tiek darbināta, un izveidotā trase vai nu tiešsaistē, vai arī pēc izpildes tiek padota "novērotājam". Tas pieņem lēmumu par trases atbilstību. Ja trase neatbilst GA, tiek ziņots par kļūdainu darbību. Autori norāda, ka trases atbilstība GA tikai daļēji liecina par verificējamās sistēmas korektu izpildi.

Pārliecība par verificējamās sistēmas korektību pieaug, izpildot daudzus testus. Attiecīgi, trases ir jāanalizē atkārtoti pietiekoši plašam testpiemēru klāstam tā, lai kādi no izpildes ceļu pārklājumu kritērijiem tiktu apmierināti.

Autori norāda, ka metode ir aprobēta, pētot NASA robotu K9 Rover.

2.4. Esošo risinājumu sniegtā pieredze

Apskatot saistīto risinājumu klāstu, ir jāatzīmē vairākas iezīmes, kuras nodrošina plašu risinājuma pielietojumu:

- **Verificējamo iezīmju apraksts nošķirts no verificējamās programmatūras**. Sekojot šādam piemēram, vienmēr ir iespējams papildināt vai modificēt verificējamo iezīmju kopu, nemainot verificējamo programmatūru. Tas ļauj dinamiski papildināt verifikāciju lietošanas vidē, pat ja izstrādes un testēšanas vidē kāds no verificēšanas scenārijiem vai iezīmēm nav identificēts. Kā redzams, tad šī īpašība piemīt visiem risinājumiem - paštestēšana (skat. 2.1.2. nodaļā), vides konfigurācijas pārbaude (skat. 2.2.1. nodaļā), SOA risinājumi (skat. 2.2.2. nodaļā) utt.
- **Verificēšanas "kodola" nošķiršana no instrumentācijas moduļiem**. Veicot verificēšanu vairākās vidēs un veidojot universālus risinājumus, ir svarīgi, lai verificēšanas mehānisms būtu ērti paplašināms ar dažādiem instrumentācijas moduļiem. Tas ir panākams, instrumentāciju stigri nošķirot no verificēšanas pamatmoduļa jeb "kodola". Šādā gadījumā verificēšanas "kodols" pieprasa zināmu instrumentācijas moduļu saskarni un uztur tai pieejamo instrumentācijas moduļu vārdnīcu (moduļu un to verificējamo īpašību saraksts). Savukārt, instrumentācijas moduļi var tikt veidoti neatkarīgi dažādām vidēm, atbilstoši verificēšanas prasībām. Šāds mehānisms tiek piedāvāts vides konfigurācijas pārbaudē (2.2.1. nodaļa) un trases analīzes ietvarā ProM (2.3.1. nodaļa).
- **Ārēji mehānismi**. Kā jau norādīts vairākkārt, ārēji verificēšanas mehānismi piedāvā plašākas verificēšanas iespējas attiecībā pret verificējamo iezīmju tvērumu. Piemēram:
    - ir iespējams verificēt procesus, kurus implementē vairāk kā viena sistēma;
    - vairākas sistēmas ir iespējamas verificēt ar vienu un to pašu mehānismu.

Ārēja mehānisma gadījumā ir iespējams variēt ar mehānisma radīto verificēšanas virsslodzi: parasti, samazinot verificējamo iezīmju skaitu, samazināsies radītā virsslodze. Savukārt, iebūvēti mehānismi nodrošina precīzāku verificēšanu:

  - verificēšanas mehānismam pieejami tieši tie paši resursi, kas verificējamai programmatūrai;
  - verificēšanas punktus iespējams ielikt tieši tajās vietās, kur procesa verificēšana varētu būs visefektīvākā;
  - viegli nodrošināt verificēšanas sinhronitāti u.c.

Šīs īpašības būtu jāņem vērā, realizējot promocijas darbā izvirzītos verificēšanas mērķus. Skatoties uz šā darba autora pētījuma jomu (biznesa procesu verificēšana), ir iespējams norādīt vēra ņemamās īpašības:

- verificēšanas mehānismam jābūt ārējam;
- instrumentācija jānošķir no verificēšanas "kodola" - ir svarīgi, lai mehānisms ir pielāgojams dažādām procesu izpildes vidēm;
- verificējamo iezīmju apraksts atdalāms no verificējamā procesa - procesu lietošanas laikā var rasties nepieciešamība mainīt vai papildināt verificējamās iezīmes.

3. ASINHRONS MULTIAĢENTU VERIFICĒŠANAS MEHĀNISMS

Lai nodrošinātu biznesa procesu verificēšanu, darba autors piedāvā lietot asinhronu multiaģentu verificēšanas mehānismu:

- **asinhrons** - verificēšanas mehānisms darbojas neatkarīgi no verificējamā procesa, abus procesus nesinhronizējot;
- **multiaģentu** - verificēšanai nepieciešamos notikumus fiksē daudzi sadalīti aģenti.

Mehānisms paredz, ka verifikācijas procesu nodrošina centralizēts kontrolieris, kuram ir pieejams verificējamo procesu apraksts. Saskaņā ar verificējamo procesu aprakstu, kontrolieris pieprasa aģentiem fiksēt notikumus, kuri apliecina procesu soļu izpildi. Analizējot saņemtos procesa soļu izpildes apstiprinājumus, kontrolieris pieņem lēmumu par procesa izpildes pareizību. Precīzāks mehānisma apraksts ir iekļauts nākamajās nodaļās.

3.1. Verificējamie kritēriji

Lai realizētu procesu verificēšanu, svarīgi ir noteikt verificējamos kritērijus, kuri nosaka, vai process izpildās korekti vai nē. Darba ietvaros tika izvirzīti trīs šādi procesu izpildes korektuma kritēriji:

- process izpildās pa atļautu ceļu;
- visi izpildes ceļā esošie soļi tiek izpildīti;
- tiek ievēroti visi procesa izpildi ierobežojošie izpildes laiki.

Var likties, ka izvirzītie kritēriji ir ļoti vispārīgi un nenodrošina pietiekamu priekšstatu par izpildes korektību, tomēr jāņem vērā, ka šie kritēriji tiek attiecināti uz izpildes laika verificēšanu, t.i., tie nav kritēriji sistēmas statiskai vai dinamiskai testēšanai. Tāpat, izvirzot korektuma kritērijus, ir jāievēro verificēšanas mehānisma virsslodze un paralēlai notikumu apstrādei patērētais laiks. Viens no galvenajiem izpildes laika verificēšanas mērķiem ir savlaicīgi informēt par programmatūras izpildes problēmām. Tāpat jāņem vērā, ka izpildes laika verificēšanas uzdevums nav testēšanas aizstāšana, tādēļ detalizēta programmatūras pārbaude tiek atstāta kā statiskās un dinamiskās testēšanas uzdevums.

Attiecīgi šobrīd izvēlētie kritēriji ļauj noteikt, vai process tiek izpildīts un vai tas izpildās savlaicīgi. Īpaši svarīgi izvirzītie kritēriji ir gadījumos, kad biznesa procesu implementē vairākas informācijas sistēmas. Šajās situācijās sistēmas lietotājs (biznesa procesa izpildē iesaistītie cilvēki) bieži vien izpildes laika problēmas nepamana, jo īpaši vēlamā izpildes laika pārkāpumus.

3.2. Mehānisma apraksts

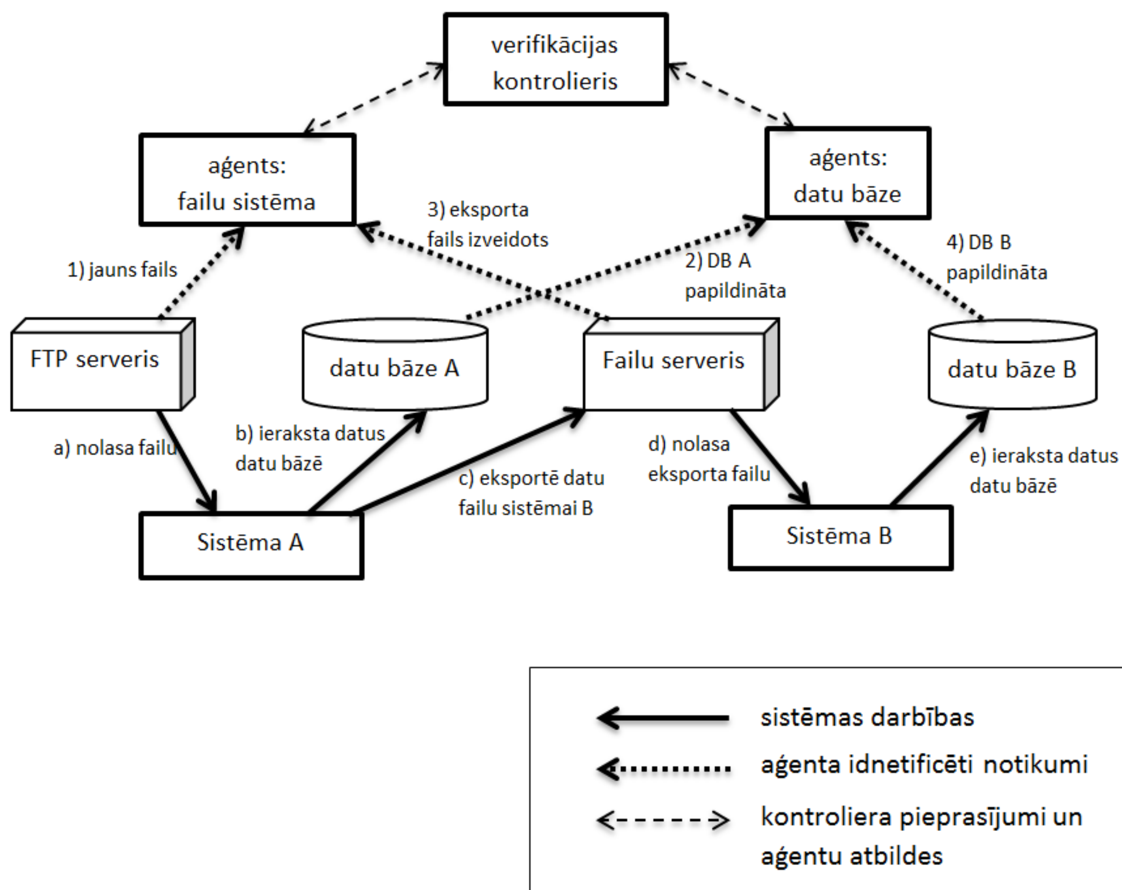
Lai nodrošinātu verificējamo kritēriju pārbaudi, vispirms ir jāspecificē visi procesa verificējamie aspekti jeb jāsagatavo procesu verificēšanas apraksts. Apraksta veidošanai ir radīta domēnspecifiska valoda (aprakstīta 3.4. nodaļā). Tajā tiek aprakstīti iespējamie procesu izpildes ceļi, notikumi, kuri liecina par katra atsevišķa procesa soļa izpildi, un laika ierobežojumi, kuri ir attiecināmi uz procesa soļu izpildi, piemēram, "vēstule jāapstrādā 5 dienās" vai "atbilde par faila apstrādi jānosūta 45s laikā pēc faila saņemšanas".

Verificēšanas procesā tiek pārbaudīts, vai verificējamais process izpildās atbilstoši specifikācijai jeb procesa verificēšanas aprakstam. Šīs pārbaudes nodrošināšanai ir paredzētas divu veidu programmatūras komponentes (precīzāk aprakstītas 3.3. nodaļā): kontrolieris un aģenti. Kontrolieris atbilstoši procesu verificēšanas aprakstam pieprasa aģentiem fiksēt procesa notikumu iestāšanos un pēc atbilžu saņemšanas no aģentiem verificē, vai notikumi ir iestājušies atbilstoši verificēšanas aprakstam un ievērojot laika ierobežojumus. Aģentu uzdevums ir pēc kontroliera pieprasījuma fiksēt procesu notikumus. Aģenti var būt vairāki, izvietoti dalītā vidē un veidoti dažādu notikumu veidu identificēšanai. Piemēram: failu sistēmas aģents spēj identificēt faila izveidošanu un dzēšanu, savukārt, datu bāzes notikumu aģents - jauna tabulas ieraksta izveidošanu un ieraksta maiņu.

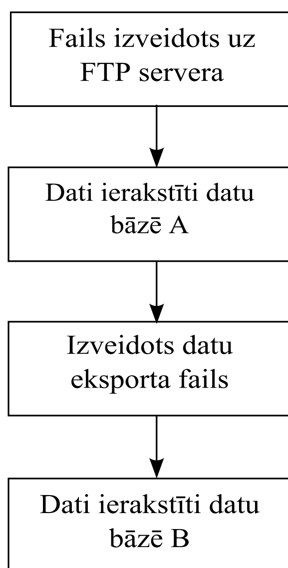
10. attēls parāda verifikācijas mehānisma darbības piemēru. Tajā tiek apskatītas divas sistēmas (A un B), kas abas kopā implementē vienu biznesa procesu. Kad FTP serverī tiek iekopēts datu fails, sistēma A to nolasa, datus ieraksta datu bāzē A un sagatavo datu eksporta failu sistēmai B. Savukārt, sistēma B nolasa sistēmas A sagatavoto eksporta datu failu un nolasīto informāciju ieraksta datu bāzē B. Šada procesa iespējamais verificēšanas apraksts satur četrus soļus (skat. 11. attēls).

Kā redzmas, tad piemērā sistēmas A un B lieto divas dažādas vides - failu sistēmu un datu bāzes. Attiecīgi šeit būtu nepieciešami divu veidu aģenti - failu sistēmas aģents un datu bāzes aģents. Saskaņā ar verificēšanas aprakstu kontrolierim par katru procesa instanci (šajā gadījumā viena instance ir viena saņemta datu faila apstrāde) jāpārbauda četri notikumi. Tādēļ verificējamajā procesā (skat. 10. attēls) varam izsekot šādām kontroliera un aģentu darbībām par katru faila apstrādes procesa instanci:

- Vispirms kontrolieris pieprasa failu sistēmas aģentam fiksēt jauna faila izveidošanu FTP serverī. Sekojot pieprasījumam, failu sistēmas aģents informē kontrolieri par katru FTP serverī izveidoto failu. Kontrolieris, saņemot apstiprinājumu par pirmā notikuma iestāšanos (FTP serverī izveidots fails), izveido verificēšanas procesa instanci katram no saņemtajiem failiem.



10. attēls. Verificēšanas mehānisma darbības piemērs



11. attēls. Verificēšanas apraksta piemērs

- Saskaņā ar verificēšanas aprakstu, kontrolieris nosūta pieprasījumu datu bāzes aģentam fiksēt datu saglabāšanu datu bāzē A. Kad aģents fiksē datu bāzes A papildināšanas notikumu, informācija par to tiek nosūtīta kontrolierim.

- Kontrolieris pieprasa failu sistēmas aģentam fiksēt notikumu, kad failu sistēmas serverī tiek izveidoti sistēmā A sagatavoti eksporta faili. Pēc eksporta faila pamanīšanas failu sistēmas aģents par šo notikumu informē kontrolieri.
- Kontrolieris pieprasa datu bāzes aģentam fiksēt datu saglabāšanu datu bāzē B. Kad šis notikums ir iestājies un kontrolieris ir saņēmis apstiprinājumu no datu bāzes aģenta, faila apstrādes procesa instance tiek atzīta par pabeigtu.

Gadījumā,ja kāds no notikumiem netiek savlaicīgi apstiprināts (kontrolieris nesaņem apstiprinājumu no attiecīgā aģenta), sistēmas uzturētāji tiek informēti par problēmas situāciju.

Kontrolieris paredz, ka katram verificēšanas procesam var būt vairākas procesa instances. Iepriekš minētajā piemērā tas nozīmē, ka paralēli FTP serverī varētu ienākt vairāki faili un katram no tiem tiktu izveidota sava verificēšanas procesa instance. Turklāt katra no tām var atrasties savā verificēšanas solī, tādēļ ir svarīgi, lai eksistētu pazīmes, kuras ļautu verificēšanas aprakstā sasaistīt verificējamos notikumus atbilstoši procesu instancēm, piemēram, pēc saņemtā faila nosaukuma.

Šajā piemērā var pamanīt, ka katram no serveriem (FTP un failu serveris) var tikt izveidots savs failu notikumu apstrādes aģents, tādējādi nepieciešamības gadījumā palielinot verificēšanas veiktspēju. Līdzīgi var rīkoties ar datu bāzes notikumu aģentu. Svarīgi, lai verificēšanas aprakstos būtu norādīti lietojamie aģenti un kontrolierim būtu pieejamas attiecīgo aģentu adreses.

Acīmredzami, ka piedāvātais verificēšanas mehānisms:

- ļauj vienlaicīgi verificēt daudzas procesu instances;
- neiejaucas verificējamā procesa izpildē;
- ir pietiekami elastīgs, ļaujot brīvi paplašināt kontrolierim pieejamo aģentu klāstu.

3.3. Programmatūras komponentes

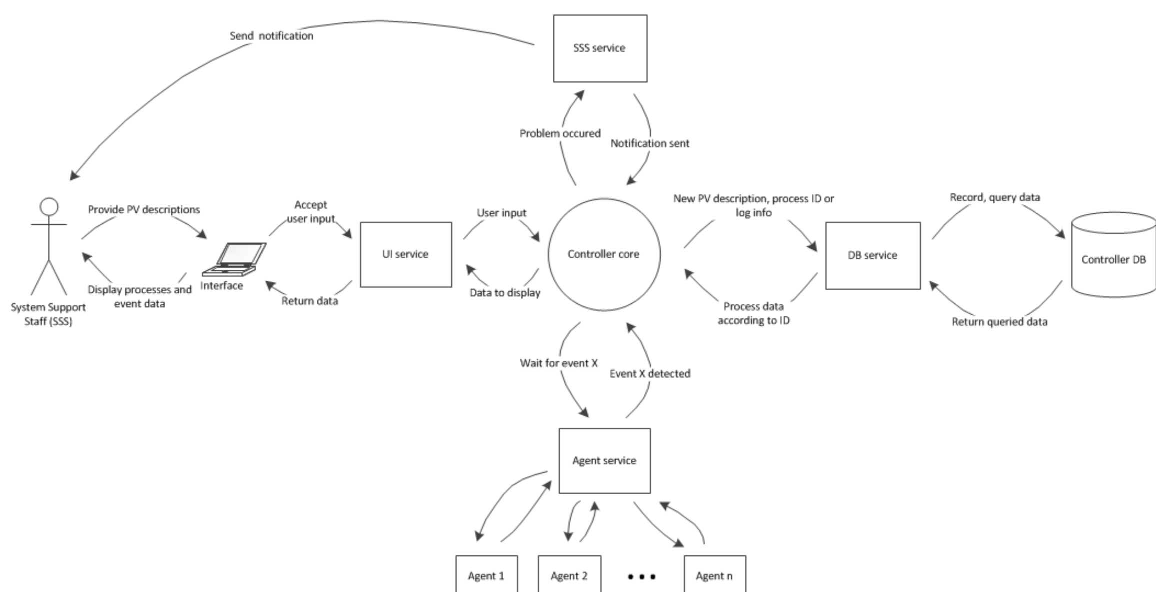
Verificēšanas mehānisms sastāv no divām komponentēm: kontroliera un aģentiem. Turpmākajās nodaļās ir izklāstīta šo komponenšu darbība un savstarpējā saskarne.

3.3.1. Kontrolieris

3.3.1.1. Darbība

Kontroliera iespējamo arhitektūru sava maģistra darba ietvaros aprakstīja K. Lapiņš [LAPI14]. Viņa darba ietvarā tika izveidots pirmais šī rīka prototips. Autors piedāvā kontrolieri konstruēt no pieciem moduļiem (autors tos sauc par servisiem - skat. 12. attēls):

- **UI modulis** ir lietotāja saskarnes serviss, kas nodrošina iespēju lietotājam iesniegt jaunu procesa verifikācijas aprakstu, lietot esošos aprakstus un saņemt informāciju par verificēšanas notikumiem.
- **SSS modulis** ir sistēmas serviss komunikācijai ar atbalsta personālu, ko kontroliera kodols izmanto, lai nosūtītu kļūdas paziņojumu personām, kuras ir atbildīgas par uzraugāmā procesa izpildi. Kļūdu paziņojumi tiek sūtīti gadījumā, ja novērota pretruna starp biznesa procesa tehnisko izpildījumu un atbilstošo procesa verificēšanas aprakstu.
- **Aģentu modulis** realizē kontroliera komunikāciju ar aģentiem: nodod pieprasījumus aģentiem sekot līdzi konkrētiem notikumiem (piemēram, „datne izveidota”) un saņem apstiprinājumus par atbilstošo notikumu iestāšanos.
- **DB modulis** ir datubāzes serviss, kas nodrošina transakcijas ar datu glabātuvi: pievieno jaunu procesu verificēšanas aprakstu datus, nolasa tos, kad nepieciešams uzsākt konkrētu procesu pārraudzīšanu, un saglabā verificēšanas trases informāciju.
- **kodols** ir modulis, kurš nodrošina verificēšanas procesu un sasaista pārējos moduļus.



12. attēls. Kontroliera prototipa loģiskā arhitektūra {controllerArchitecture}

Autors ir piedāvājis kontroliera datu bāzes aprakstu, kā arī pietiekami detalizētu kontroliera darbības aprakstu.

Tomēr, lai verificēšanas mehānismu veidotu maksimāli plaši lietojamu, tad būtu jānodrošina vairākas pamatprasības:

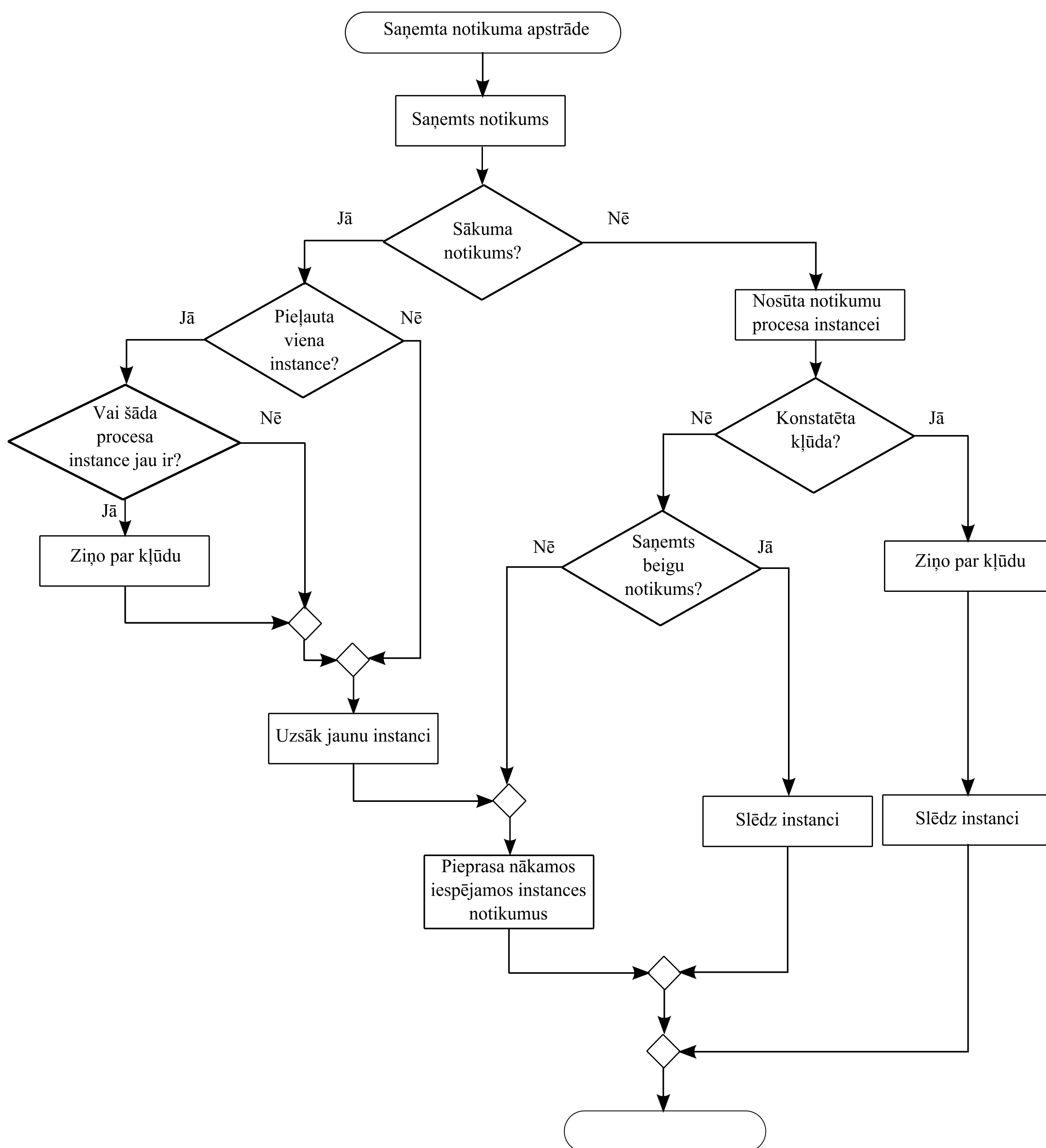
- procesu verificēšanas aprakstu kopu jāvar papildināt dinamiski, lai nodrošinātu vienkāršu jaunu procesu verificēšanas uzsākšanu;
- kontrolierim jāspēj verificēt daudzu procesu daudzas instances vienlaicīgi;
- ir jābūt iespējai papildināt kontroliera lietoto aģentu kopu, nemainot verificēšanas kontroliera programmatūru;
- kontrolierim jāuztur resursu vārdnīca, lai attiecīgos resursus varētu izmantot verificēšanas aprakstos (precīzāk aprakstīts 3.4. nodaļā);
- kontrolierim jāsaglabā visu verificēto procesu instances un to verificēšanas gaita, lai nepieciešamības gadījumā nodrošinātu vēlāku procesu analīzi.

Kontrolieris, uzsākot darbu, nolasa visus definētos procesu verificēšanas aprakstus. Par visiem verificēšanas procesiem, kuru instanču uzsākšana notiek ar sākuma notikuma iestāšanos, aģentiem tiek nosūtīti notikumu identificēšanas pieprasījumi. Paralēli tiek uzsākta laika ierobežojumu pārbaude: kādi no procesiem var saturēt ierobežojumus ar fiksētu laiku. Tāpat tiek uzstādīti taimeri, kuri identificēs tuvākos notikumu laika ierobežojumu pārkāpumus. Turpmākajā darbā kontrolieris balstās uz ziņojumiem, kas no aģentiem saņemti par notikušajiem notikumiem.

Saņemto notikumu ziņojumu apstrāde. Ziņojumus saņem *aģentu modulis* un nosūta kontroliera *kodolam*. Saņemot notikuma apstiprinājumu (skat. 13. attēls), kontrolieris identificē, vai tas ir procesa instances uzsākšanas ziņojums vai arī esošai procesa instancei atbilstošs ziņojums.

Saņemot ziņojumu, kas norāda uz jaunas procesa instances sākumu, tiek pārbaudīts, vai šim procesam drīkst būt viena vai vairākas instances. Gadījumā, ja drīkst būt tikai viena instance, bet jau viena vai vairākas ir strādājošas, tiek ziņots par kļūdu. Neatkarīgi no tā, vai tiek konstatēta kļūda vai arī viss ir korekti, tiek uzsākta jauna procesa verificēšanas instance.

Ja saņemtais ziņojums attiecas uz jau strādājošu procesa verificēšanas instanci, tas tiek "nosūtīts" attiecīgajai instancei apstrādei. Ja apstrādes laikā tiek konstatēta problēma (nokavēts kāds notikuma laiks vai nav korekta izpildes secība), tiek ziņots par kļūdu un procesa verificēšanas instance tiek slēgta. Tāpat procesa verificēšanas instance tiek slēgta, ja ir saņemts ziņojums par notikumu, kurš ir verificēšanas procesa beigu ziņojums.



13. attēls. Saņemta notikuma apstrāde

Konkrēta procesa verificēšanas instance, saņemot jauna notikuma ziņojumu, pārbauda, vai ir saņemts apstiprinājums par kādu no sagaidāmajiem notikumiem. Ja ir saņemts korekts notikums, tad instances ietvaros tiek pārbaudīti laika ierobežojumi. Savukārt, pēc veiksmīgas laika ierobežojumu pārbaudes, tiek pārbaudīti visi saistītie apakšnotikumi (precīzāk aprakstīts 3.4. nodaļā). Pēc saņemtā ziņojuma apstrādes, ja verificēšanas procesa instance nav nonākusi beigu aktivitātē, tiek pieprasīti iespējamie nākamie procesa instances notikumi. Noteikti tiek pieprasīti nākamie tieši sekojošie notikumi, kā arī iespējamie tālākie notikumi, ja ir pieejama visa nepieciešamā notikumu identificējošā informācija.

Laika ierobežojumu pārbaude. Verificējot laika ierobežojumus, ir jāizšķir divi atšķirīgi ierobežojumi:

- ierobežojumus var pārkāpt vienas procesa verificēšanas instances ietvaros;
- var nokavēt procesu instanču darbināšanu, kuras ir paredzētas darbināt vienā eksemplārā, t.i., var būt situācijas, kad sākuma notikumam ir uzstādīts fiksēta laika ierobežojums, bet sākuma notikums norādītajā laikā nav iestājies.

3.3.1.2. Saskarne

Obligāti nodrošināmā kontroliera ārējā saskarne *IControllerService* ir visai vienkārša. Kontrolierim ir jānodrošina izsaukums, kurš ļauj aģentam atgriezt informāciju par konstatētu notikumu - **AgentCollback**. Izsaucot šo metodi, aģents norāda:

- pieprasītā notikuma identifikatoru (kontrolieris to saņem no aģenta, pieprasot aģentam jaunu notikumu);
- notikumu raksturojošos rezultātus xml formātā (katrs rezultātu vienums satur divus xml elementus - vienuma nosaukumu un vērtību);
- informāciju par aģentu un notikumu (aģenta tips, notikuma tips un aģenta adrese).

Papildus šai saskarnei ir jānovērtē, kāds komunikāciju protokols tiek implementēts kontrolierī. Izvēlētais protokols noteiks, kā tiks implementēti visi verificēšanas aģenti.

3.3.2. Aģenti

Notikumu aģenti ir viena no verificēšanas mehānisma būtiskajām komponentēm. Skaidrojošā vārdnīcā vārdam "aģents" divi no skaidrojumiem ir: "Cilvēks, kas darbojas kāda interesēs" un "Spiegs, arī diversants". Šadā izpratnē varētu skatīties arī uz verificēšanas mehānisma notikumu aģentu:

- aģents darbojas kontroliera uzdevumā (kontrolieris pieprasa notikumu fiksēšanu);
- aģents novēro vidi, pēc iespējas mazāk ietekmējot tajā notiekošo (t.i., sevi slēpjot).

Papildus notikumu konstatēšanai aģenti ne tikai atgriež informāciju, vai notikums ir noticis, bet sniedz papildus notikumu raksturojošu informāciju. Piemēram, ja notikums ir "izveidots fails", tad atgrieztā notikuma informācija būtu:

- jaunā faila nosaukums;
- pilns faila nosaukums, iekļaujot ceļu līdz failam;
- direktorijas nosaukums, kur fails konstatēts.

Tālāk ir aprakstīti iespējamie aģentu veidi, to saskarne un darbība.

3.3.2.1. Aģentu veidi

Lai saprastu iespējamos aģentu veidus, vispirms būtu jāapskata iespējamie notikumu pieprasījuma veidi, kurus aģentiem vajadzētu apstrādāt. Tāpat jāapskata potenciālās aģentu darbības vides. Notikumu pieprasījumus var iedalīt divos veidos: sinhroni un asinhroni pieprasījumi.

- **Sinhrona** pieprasījuma gadījumā, aģents uzreiz ziņo kontrolierim par notikuma jeb precīzāk "jautājuma" rezultātu, piemēram, atbildi par notikumu "vai fails eksistē" failu sistēmas aģents sniegtu uzreiz: jā vai nē. Vienlaicīgi ar šo atbildi, aģents var sniegt papildus informāciju par notikumu. Piemēram, sinhronam notikumam "vai tabulā ir ieraksts", pieprasījumā jānorāda attiecīgā tabulas ieraksta identifikators, savukārt, kā atbilde varētu būt visi attiecīgās tabulas ieraksta lauki.
- **Asinhrona** pieprasījuma gadījumā, aģents ziņo kontrolierim par notikumu tikai brīdī, kad tas ir noticis, turklāt šādā gadījumā atbilde vienmēr būtu jāuzskata par "pozitīvu". Piemēram, notikuma "izveidots jauns fails" atbilde kontrolierim tiek nosūtīta tikai pēc jauna faila izveidošanas.

Aģentu analīzē nozīmīga loma ir arī videi, par kuras notikumiem tie ziņo. Vides kontekstā ir izšķirami divi gadījumi:

- tikai daži no vidē konstatējamajiem notikumiem ir nepieciešami verificēšanai, piemēram, datu bāzu gadījumā tikai dažu tabulu vai pat ierakstu pārmaiņas būtu nepieciešamas verificēšanai, t.i., tikai neliela daļa no visiem datu bāzē konstatējamajiem notikumiem;
- visi vidē konstatējamie notikumi ir noderīgi verificēšanai, piemēram, dokumentu vadības sistēmā visticamāk visas dokumentu statusu maiņas būtu verificējamas.

Darba autors atkarībā no vides un apstrādājamo notikumu pieprasījuma veidiem izdala trīs potenciālus aģentu veidus:

- **Vienkāršs aģents**. Tas darbojas vidē, kur visi notikumi nosūtāmi verificēšanai. Attiecīgi nav nepieciešams implementēt divvirziena komunikāciju, jo aģents visus notikumus bez kontroliera pieprasījuma drīkst nosūtīt kontrolierim. Šādi aģenti var tikt veidoti iegultās sistēmās vai, piemēram, vienai konkrētai informācijas sistēmai.
- **Komplekss aģents**. Tas darbojas intensīvā notikumu vidē, no kuriem tikai neliela daļa ir nepieciešama verificēšanai. Aģents ziņo par notikumiem, kurus

ir pieprasījis kontrolieris, attiecīgi ir jāimplementē divvirziena komunikācija, kā arī jāimplementē abu veidu notikumu pieprasījumi - sinhroni un asinhroni.

- **Paplašināts aģents**. Šis aģents ir vienkāršs aģents ar dažādiem papildinājumiem, piemēram, tajā varētu implementēt sinhronu notikumu pieprasīšanu vai pat asinhronu notikumu pieprasīšanu. Implementējot asinhronu notikumu pieprasīšanu, faktiski tiek realizēts pilnīgs komplekss aģents. Šādu aģentu lietojums var būt noderīgs, veidojot iegultus aģentus: par daļu no notikumiem aģents ziņo bez pieprasījuma, par daļu - pēc pieprasījuma saņemšanas. Tas ļautu papildināt sākotnējos sistēmai piesaistītos procesu verificēšanas aprakstus ar jauniem vai papildināt esošos pēc sistēmas uzstādīšanas lietošanas vidē. Veidojot jaunus verificēšanas aprakstus, visticamāk, būs nepieciešami asinhroni pieprasījumi.

3.3.2.2. Aģentu darbība

Lai nodrošinātu ērti paplašināmu aģentu klāstu, ir specificēta saskarne (3.3.2.3. nodaļa), kura jāimplementē katrā no aģentiem. Tāpat aģentu implementāciju ierobežo kontroliera atbalstītie komunikāciju protokoli. Savukārt, tālākā aģenta implementācija ir atkarīga no vides prasībām un izstrādātāju iespējām. Tomēr realizējot šāda veida monitorēšanas programmatūru, parasti tiek izvēlēts viens no diviem iespējamajiem implementācijas jeb aģentu darbības veidiem:

- **taimeru bāzēts** - notikuma iestāšanās tiek pārbaudīta ik pēc noteikta laika, piemēram, katras desmit sekundes;
- **notikumu bāzēts** - gadījumā, ja vide, kuras novērošanai aģents tiek lietots, atbalsta notikumu abonēšanu, aģenti tiek veidoti notikumu bāzēti, t.i., tie abonē zināmus vides notikumus un saņem informāciju no vides, iestājoties abonētajiem notikumiem.

Taimeru bāzēti aģenti parasti ir vieglāk implementējami, tomēr tie var atstāt pastāvīgu ietekmi uz novērojamo vidi: neatkarīgi no tā, vai gaidītais notikums ir iestājies vai nav, aģents ar noteiktu regularitāti ģenerē virsslodzi. Lai šo virsslodzi samazinātu, autori piedāvā dažādus optimizācijas mehānismus [NAVA13], piemēram, atļaut katram notikuma pieprasījumam norādīt savu laika kavējumu vai pat, analizējot notikumu iestāšanās statistiku, mēģināt pielāgot notikumu pārbaudes taimera kavējumu.

Implementējot asinhronu notikumu pieprasījumus, aģentos ir jāimplementē zināma papildus funkcionalitāte:

- asinhronu pieprasījumu kontrolieris var atsaukt, pat ja pieprasījumam atbilstošais notikums nav iestājies;
- asinhronam pieprasījumam var būt uzstādīts tā aktualitātes termiņš (no angļu valodas - *time to live*), kuru kontrolieris uzstāda pieprasījuma brīdī;
- asinhrons pieprasījums var būt vienreizējs vai vairākkārtējs: par vienreizēju pieprasījumu aģents "aizmirst" uzreiz pēc pirmās notikuma konstatēšanas un nosūtīšanas kontrolierim, savukārt, vairākkārtēja notikuma pieprasījums tiek apstrādāts (aģents turpina notikuma vērošanu) tik ilgi, kamēr kontrolieris to neatsauc vai nebeidzas tā aktualitātes termiņš.

3.3.2.3. Aģentu saskarne

Verifikācijas mehānisms paredz, ka kompleksam un paplašinātam aģentam ir jāimplementē *IAgentService* saskarne. Tajā ir iekļauti četri izsaukumi:

- **ProvidedEventTypes** - metode atgriež sarakstu ar aģenta nodrošinātajiem notikumu tipiem, aprakstot to izsaukumus un izpildes veidu (sinhrons vai asinhrons). Izsaukums nepieciešams, lai kontrolieris varētu pārbaudīt apstrādājamos procesu verificēšanas aprakstus un savlaicīgi pārliecināties, ka visi aprakstā norādītie pieprasījumi ir pieejami.
- **EnlistRequest** - metode reģistrē jauna notikuma pieprasījumu, t.i., pieprasa aģentu novērot vidi, vai nav iestājies noteikta veida notikums. Šādi pieprasījumi var būt vienreizēji vai vairākkārt izpildāmi: vienreizēju notikumu pieprasījumi pēc to iestāšanās tiek izņemti no novērojamo notikumu saraksta, savukārt, vairākkārtēji tiek fiksēti tik ilgi, kamēr kontrolieris tos atsauc.
- **DeleteRequest** - metode nodrošina kontrolierim iespēju atsaukt iepriekš pieprasītu notikumu uzraudzību.
- **Request** - izpilda sinhronus pieprasījumus, kā rezultātu agriežot xml dokumentu, kurā ir iekļauts gan notikuma rezultāts (True/False), gan to raksturojošas īpašības, piemēram, faila nosaukums.

Vienkāršiem aģentiem nekas no iepriekš minētā nav jāimplementē, jo tie nosūta kontrolierim notikumu informāciju bez kontroliera pieprasījuma. Savukārt, paplašināta aģenta gadījumā būtu jāimplementē vismaz divas metodes: *ProvidedEventTypes* un *Request*.

Aģentu implementācija ir atvērta un brīvi pieejama izstrādātājiem. Ja kādā no verificēšanas gadījumiem rodas nepieciešamība analizēt notikumus vidēs, kurām aģenti nav izveidoti, aģents var tikt implementēts, ievērojot iepriekš minētos saskarnes nosacījumus un

kontroliera atbalstītus komunikāciju protokolus. Savukārt kontrolierī, ieviešot jaunu aģentu, ir jāveic tikai konfigurācijas izmaiņas, norādot jaunā aģenta tipu un adresi.

3.4. Verificēšanas apraksta valoda

Šobrīd ir izstrādātas daudzas valodas procesu aprakstīšanai un modelēšanai. Atkarībā no apskatāmā procesa un nepieciešamās apraksta detalizētības var tikt piemeklētas dažādas apraksta valodas. Populārākās valodas ir UML aktivitāšu diagrammas (*UML Activity diagram* [OMG11B]), Biznesa procesu modelēšanas notācija (*Business Process Model and Notation* jeb BPMN [OMG11A]), Biznesa procesa izpildes valoda (*Business Process Execution Language* jeb BPEL [OASIS07]) un Notikumu vadītas procesu ķēdes (*Event-driven Process Chains* jeb EPC [SCHE05]):

- **UML aktivitāšu diagrammas**. Šīs diagrammas ir grafiska darba plūsmu aktivitāšu izpildes reprezentācija. Tās ir paredzētas kā datorizētu, tā arī tīri organizatorisku procesu attēlošanai. Aktivitāšu diagrammas parāda vispārēju aktivitāšu plūmas kontroli. Saskaņā ar diagrammu sintaksi [STOR04] tās ļauj attēlot aktivitātes, zarošanos, kā arī paralēlu plūsmu izpildi.
- **BPMN**. Šis ir biznesa procesu modelēšanas standarts, kas nodrošina grafisku notāciju biznesa procesa attēlošanai. Pēc piedāvātās tehnikas, šīs diagrammas ir ļoti līdzīgas UML aktivitāšu diagrammām. BPMN mērķis ir atbalstīt biznesa procesu pārvaldību kā tehniskiem speciālistiem, tā arī biznesa lietotājiem, piedāvājot intuitīvi saprotamu notāciju.
- **BPEL**. Šī valoda ir mērķtiecīgi veidota webservisu bāzētu biznesa procesu aprakstīšanai. Piedāvātie izteiksmes līdzekļi ļauj aprakstīt kā abstraktus biznesa procesus, tā arī veidot izpildāmas modeļu specifikācijas. BPEL apraksta ziņojumu apmaiņu starp dažādu sistēmu webservisiem.
- **EPC**. Līdzīgi kā iepriekš minētās valodas, arī EPC nodrošina biznesa procesu darba plūsmu attēlošanu. Sākotnēji EPC tika veidota specifiski priekš *SAP R/3*, bet vēlāk tā tika lietota arī nesaistīti ar konkrētajiem rīkiem. EPC diagrammas ir orientēti notikumu un funkciju grafi. Tajos ir iekļauti dažādi savienojoši elementi, kas atļauj gan alternatīvu, gan paralēlu izpildi [TSAI06]. Šo diagrammu pamatelementi iekļauj notikumus, funkcijas, procesa īpašniekus, organizatoriskās vienības, resursus, loģiskos savienojumus, kontroles un informācijas plūsmas u.c.

Minētās valodas sniedz plašas procesu apraksta iespējas, turklāt ļoti detalizētā līmenī. Teorētiski, ja kādā no šīm valodām būtu pieejams pietiekami detalizēts procesa apraksts,

implementēta porcesa izpildi varētu mēģināt verificēt atbilstoši šim aprakstam. Tā būtu modeļu bāzēta izpildes laika verificēšana [ZHAO09]. Šādā gadījumā nebūtu jāveido procesu verificēšanas valoda un atbilstoši arī verificēšanas apraksti. Tomēr praksē tas prasītu visiem verificējamajiem procesiem sagatavot pietiekami detalizētu procesa aprakstu. Ņemot vērā, ka autora veidotais mehānisms ir paredzēts arī mantotu sistēmu procesu verificēšanai, šādu aprakstu izveide varētu būt pietiekami sarežģīta. Tāpat būtu sarežģīti verificēt procesus, kurus implementē vairākas sistēmas: šajā gadījumā būtu jāveido modelis, kurš iekļautu vairākas sistēmas. Turklāt šāda pieeja tik un tā radītu problemātisku procesu bāzētu verificēšanu, t.i., verificēšanu, kura sistēmas apskata no implementēto procesu viedokļa. Attiecīgi var secināt, ka procesu verificēšanai būs jāveido specifiski procesu verificēšanas apraksti.

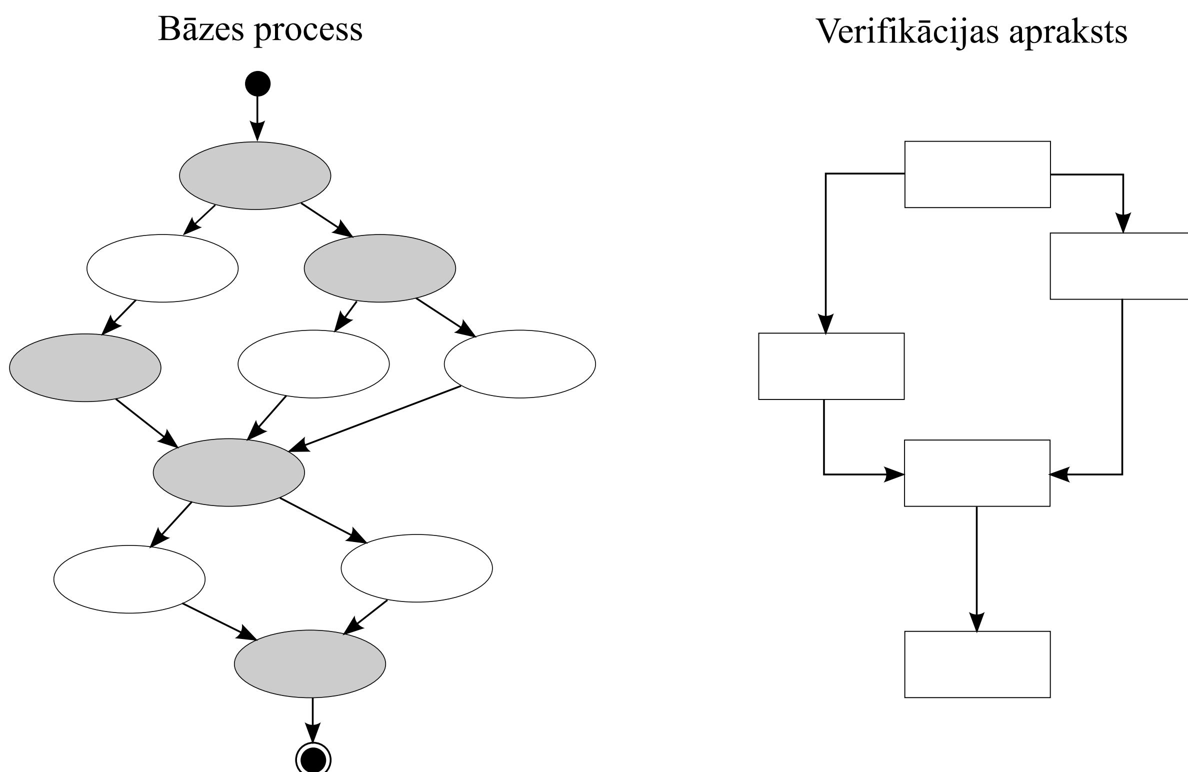
Analizējot esošo procesu apraksta valodu jēdzienus un piedāvātās iespējas, ir iespējams identificēt vairākas verificēšanai nepieciešamās iezīmes, kuras ar esošajām valodu iespējām aprakstīt ir grūti vai pat nav iespējams:

- kā noteikt, ka procesa solis ir pabeigts;
- cik ilgi procesa solis drīkst izpildīties;
- uz kura servera, darbstacijas vai iekārtas procesa soļi tiks izpildīti.

Tādējādi, lai aprakstītu visas verificēšanai nepieciešamās procesu iezīmes, esošajām valodām ir jāveido verificēšanas apraksta paplašinājumi.

Lai pēc iespējas ātri varētu nonākt pie pārbaudāma verificēšanas mehānisma, ņemot vērā nelielo procesu verificēšanai nepieciešamo jēdzienu daudzumu un domēnspecifisku valodu veidošanas cēloņus [MERN05], darba autors izveidoja domēnspecifisku valodu procesu verificēšanas aprakstiem. Valoda tika veidota uz XML bāzes [XML08], lai tajā sagatavotu aprakstu aptrāde būtu pēc iespējas vienkāršāka. Valodas sintakse ir definēta ar atbilstošu XML shēmu palīdzību. Lai arī ir izveidota jauna valoda, tomēr šāda pieeja nekādā veidā neierobežo citu valodu iespējamu lietojumu verificēšanas aprakstu sagatavošanai: attiecīgās procesu apraksta valodas var tikt papildinātas ar verificēšanai nepieciešamajiem paplašinājumiem un transformācijām paplašinājumu pārveidošanai par verifikācijas apraksta valodas aprakstiem.

Pieņemsim, ka procesa apraksts jeb *bāzes process* ir veidots, lietojot stāvokļu pārejas diagrammu (skat. 14. attēls). Lai izveidotu verifikācijas aprakstu, pirmais no uzdevumiem ir saprast, kurus no procesa izpildes soļiem identificē kādi "ārēji" novērojami notikumi (skat. 14. attēls – iekrāsoti pelēki). Nākamajā solī šīs aktivitātes, precīzāk, tās identificējošie notikumi, būtu sakārtojami to iespējamā iestāšanās secībā atbilstoši procesa izpildes aprakstam. Attiecīgi tiks iegūts procesa verifikācijas apraksts, t.i., apraksts, kurš norāda, kādā secībā var iestāties notikumi, kuri apliecina verificējamā procesa izpildi.



14. attēls. Bāzes un verificēšanas procesu apraksti

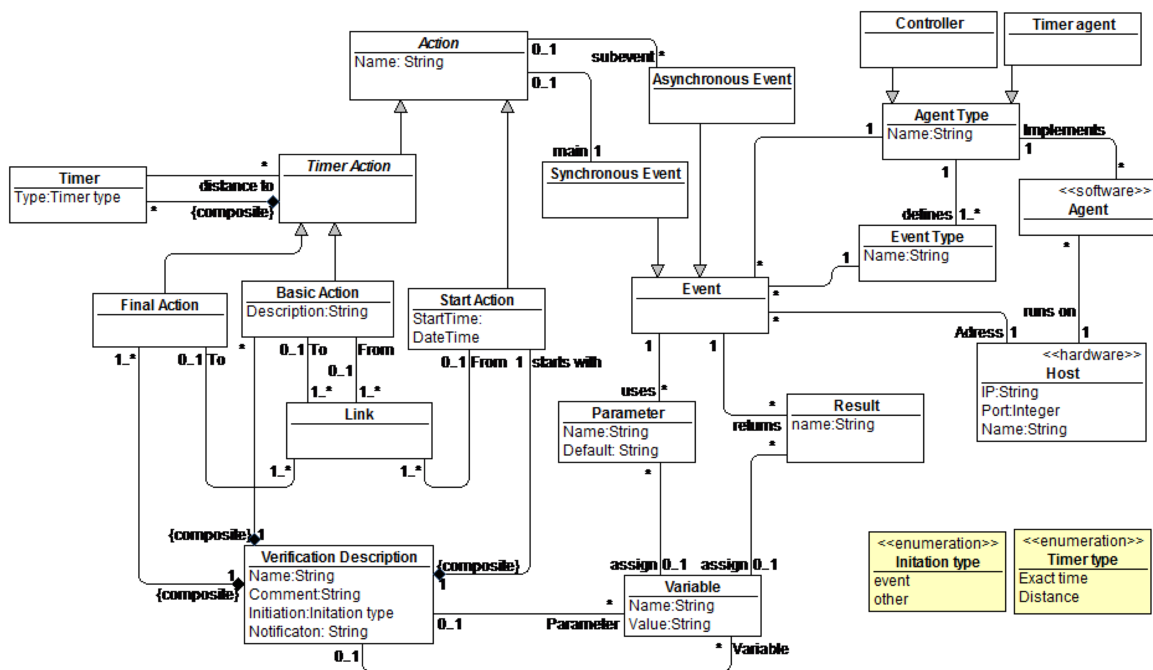
Apskatot verificējamā procesa apraksta stāvokļu diagrammu (14. attēls) un tai atbilstošo procesa verificēšanas diagrammu, verificēšanas procesa apraksta grafa virsotnes ir notikumi, kuri raksturo stāvokļa iestāšanos. Savukārt, orientētā grafa virsotņu saites attēlo iespējamo notikumu iestāšanās secību.

3.4.1. Verificēšanas apraksta valodas elementi

Procesu verificēšanas apraksta valodā ir trīs pamatelementi: novērojamās aktivitātes, tās identificējošie notikumi un orientētas aktivitātes savienojošas saites. Pilns elementu pārskats ir pieejams klašu diagrammā (15. attēls).

Tā kā saitēm starp notikumiem, atskaitot to virzienu, nav citu raksturojošu atribūtu, tad turpmākajā nodaļas izklāstā tiks apskatītas aktivitātes un notikuma jēdziens.

Katrai no aprakstā iekļautajām verificējamajām aktivitātēm atbilst vismaz viens notikums, kurš apliecina šīs aktivitātes izpildi. Var būt gadījumi, kad vienu aktivitāti raksturo vairāki notikumi, tomēr viens notikums tiek uzskatīts par galveno notikumu, savukārt, pārējie par saistītajiem jeb apakšnotikumiem.



15. attēls. Verificēšanas apraksta valodas elementi

Katrai aktivitātei ir nosaukums un tās identifikators. Svarīga aktivitātes apraksta sastāvdaļa ir laika ierobežojumi. Vienai aktivitātei tie var būt vairāki, turklāt divu dažādu veidu:

- **absolūts laiks**, piemēram, "līdz plkst. 14:00" vai "pēc 15:00";
- **laika nobīde** attiecībā pret citām iepriekš notikušām aktivitātēm, piemēram, "aktivitātei jānotiek piecu minūšu laikā pēc sākuma aktivitātes".

Attiecīgie laika ierobežojumi var tikt kombinēti, piemēram, vienai aktivitātei varētu būt šādi ierobežojumi:

- līdz plkst. 13:00;
- 10s pēc aktivitātes X;
- 15s pēc aktivitātes Y

Pirmais ierobežojums nosaka, ka jebkurā gadījumā šai aktivitātei ir jābūt pabeigtai līdz plkst. 13:00. Savukārt, pēdējie divi ir atkarīgi no verificēšanas aprakstā noteiktās aktivitāšu izpildes secības:

- ja apraksts paredz, ka aktivitātes X un Y izpildīsies secīgi, tad tiek kontrolēti abi no laikiem;
- ja apraksts paredz, ka var izpildīties vai nu aktivitāte X, vai arī - Y (t.i., divi iespējami procesa izpildes ceļi), tad tiek kontrolēts laiks attiecībā pret to aktivitāti, kura ir izpildījusies.

Aprakstot aktivitātes raksturojošos notikumus (gan galvenos notikumus, gan apakšnotikumus), katram notikumam tiek norādītas šādas īpašības:

- notikuma aģenta tips, piemēram, *FileSystem* (t.i., failu sistēmas notikumu aģents);
- notikuma tips, piemēram, *NewFileCreated* (t.i., izveidots jauns fails);
- notikuma identificēšanai nepieciešamie parametri, piemēram, jauna faila izveides gadījumā - direktorija un faila nosaukuma šablons;
- aģenta adrese, t.i., servera nosaukums vai IP adrese (un arī ports), kur aģents ir pieejams;
- rezultātu piešķiršana, t.i., kādas konstatētā notikuma vērtības ir saglabājamas, piemēram, izveidotā faila nosaukums.

Visi aktivitātes laika ierobežojumi attiecas tikai uz galveno aktivitātes notikumu, jo apakšnotikumi tiek verificēti tikai pēc galvenā notikuma iestāšanās. Acīm redzami, ka par galveno notikumu var kalpot asinhroni notikumi, savukārt, sinhroni - kā apakšnotikumi, t.i., apakšnotikumu pārbaudēm ir jānotiek nekavējoties.

Papildus aktivitāšu un to saišu aprakstiem katrs no verifikācijas aprakstiem satur vispārēju to raksturojošu informāciju:

- procesa nosaukums un identifikators;
- verifikācijas uzsākšanas pazīme: verifikācijas procesa instance tiek uzsākta pēc sākuma notikuma konstatēšanas vai arī to uzsāk cits verifikācijas process;
- apraksta parametri (konstantes, piemēram, pārbaudāmās direktorijas, serveru nosaukumi u.c.);
- aprakstā lietotie mainīgie;
- problēmu ziņojumu saņēmēji un ziņošanas veidi (e-pasts, īsziņa u.c.)

3.4.2. Verificēšanas apraksta valodas sintakse

Procesu verificēšanas apraksta valoda ir XML bāzēta. Tai ir definēta XSD shēma, saskaņā ar kuru ir iespējams veikt sagatavoto aprakstu pārbaudi. Valodas piemērs ir iekļauts darba pielikumā.

Saskaņā ar shēmu katrā aprakstā ir trīs sadaļas: galvene, aktivitātes un saites:

Verifikācijas apraksta sadaļu piemērs

```
<process_description>
  <header>
    <id>Sample-3</id>
    <name>Sample process verification description with subevent</name>
    <initiation>start-event</initiation>
    <instances>multiple</instances>
    <parameters> ... </parameters>
    <variables> ... </variables>
    <message_receivers> ... </message_receivers>
  </header>
  <actions>
    <action>...</action>
    <action>...</action>
    <action>...</action>
  </actions>
  <links>
    <link from="action-1" to="action-2"/>
    <link from="action-2" to="action-3"/>
  </links>
</process_description>
```

3.4.2.1. Galvene

Kā jau tika minēts valodas elementu aprakstā, katrs procesa verifikācijas apraksts satur tā identifikatoru un nosaukumu. Tāpat tiek norādīta verifikācijas procesa inicializācijas pazīme (<initiation>), kura var pieņemt vienu no divām vērtībām:

- **start-event** - verifikācijas procesa instanci iniciē sākuma aktivitāti aprakstošais notikums;
- **other** - verifikācijas procesa instanci iniciē cits verifikācijas process.

Papildus tiek norādīta pazīme (elements <instances>), kura norāda, cik procesa instances vienlaicīgi var būt aktīvas. Šim elementam ir divas iespējamās vērtības:

- **multiple** - pieļaujamas vairākas procesa instances;
- **single** - pieļaujama tikai viena procesa instance.

Galvenē ir iekļautas parametru (<parameters>) un mainīgo (<variables>) sadaļas. Šajās sadaļās atbilstoši ir vai nu parametru (elementi <parameter>), vai arī mainīgo (elementi <variable>) saraksts. Gan parametru, gan mainīgo apraksta līdzīgi elementi: nosaukums (<name>) un sākotnējā vērtība (<default>).

Parametra apraksta elementu piemērs

```
<parameter>
  <name>X</name>
  <default>456</default>
</parameter>
```

Parametru gadījumā noklusētajai vērtībai ir jābūt obligāti norādītai, savukārt, mainīgo gadījumā sākuma vērtība drīkst nebūt, t.i., ir pieļaujams šāds mainīgā apraksts:

Mainīgā apraksta elementu piemērs

```
<variable>
  <name>Y</name>
</variable>
```

Tāpat galvenē ir iekļauta sekcija (<message_receiver>), kas apraksta ziņošanu par izpildes problēmām. Šajā sadaļā tiek norādīti visi ziņojumu saņēmēji, ziņojuma veids (e-pasts vai īsziņa), ziņojuma tēma un ziņojuma saturs. Aprakstot ziņojuma tēmu un saturu, ir iespējams lietot verifikācijas aprakstā norādītos parametrus un mainīgos, tādējādi ļaujot sniegt pēc iespējas precīzas ziņas par konstatētajām problēmām.

3.4.2.2. Aktivitātes

Aktivitātes elementa (<action>) apraksts sastāv no vairākiem elementiem:

- aktivitātes elementam ir divi atribūti:
  - aktivitātes identifikators (atribūts *id*), kuram jābūt unikālam un tas tiek izmantots, atsaucoties uz šo aktivitāti citās apraksta sadaļās);
  - aktivitātes tips (atribūts type): *start*, *normal* vai *finish*;
- laika ierobežojums (elements <time>);
- nosaukums (elements <name>);
- galvenais notikums (elements <main_event>);
- apakšnotikumi (elements <sub_events>).

Aktivitātes elementa piemērs

```
<action id="file-deleted" type="finish">
  <time>
    <distances>
      <distance>
        <ref_event_id>new-file-created</ref_event_id>
        <period>00:00:15</period>
      </distance>
    </distances>
  </time>
  <name>File deleted</name>
  <main_event> ... </main_event>
  <sub_events>
    <sub_event> ... <sub_event>
    <sub_event> ... <sub_event>
  </sub_events>
</action>
```

Laika ierobežojums (elements <time>) var saturēt vairākus apakšelementus:

- <exact_time> - formātā hh:mm:ss norādīts konkrēts laiks;
- <distances> - elements, kurā tiek iekļautas atsevišķas laika "distances" (elements <distance>) līdz citām aktivitātēm, katrā norādot saistīto aktivitāti (elements <ref_event_id>) un "distanci" (elements <period>) formātā hh:mm:ss;
- <no_time> - norāda, ja aktivitātei nav nekādu laika ierobežojumu.

Elementi <exact_time> un <distances> var tikt lietoti kopā, savukārt, <no_time> - var tikt lietots tikai atsevišķi.

3.4.2.3. Notikums

Galvenais notikums (elements <main_event>) un apakšnotikumi (elements <sub_event>) tiek aprakstīti vienādi.

Notikuma elementa piemērs

```
<..._event>
  <agent>FileSystem</agent>
  <event_type>FileCreated</event_type>
  <address>file-server</address>
  <parameters>
    <parameter>
      <name>Directory</name>
      <assign>Inbox</assign>
    </parameter>
    ...
  </parameters>
  <results>
    <result>
      <name>FileName</name>
      <assign>fileName</assign>
    </result>
  </results>
</..._event>
```

Notikumam tiek norādīts aģenta tips (elements <agent>), notikuma tips (elements <event_type>) un adrese (elements <address>). Pārbaudot procesa verificēšanas aprakstu, kotrolierim ir jāpārbauda, vai šāda tipa aģents norādītajā adresē ir pieejams un nodrošina norādīto notikumu. Lai visos aprakstos nebūtu jānorāda precīzas aģentu adreses un nepieciešamības gadījumā tās nebūtu jāmaina daudzviet, ir pieļaujams norādīt adreses nosaukumu (iepriekš minētajā piemērā - *file-server*). Šajā gadījumā precīzo adrešu sarakstu un adrešu nosaukumu vārdnīca ir jāuztur verificēšanas kontrolierim.

Notikumā ir iekļauta notikuma pieprasīšanai nepieciešamo parametru (elements <parametrs>/<parmater>) un rezultātu (elements <results>/<result>) sadaļas. Katrs parametra elements <parameter> satur divus apakšelementus:

- **name** - parametra nosaukums;
- **assign** vai **value** - ja tiek norādīts elements <assign>, tad šajā elementā tiek norādīts atbilstošā procesa verifikācijas apraksta mainīgā vai parametra nosaukums, savukārt, norādot elementu <value> tiek norādīta konstanta vērtība (iepriekš minētajā piemērā tiek norādīts elements <assign>, un tas nozīmē, ka notikuma parametram ar nosaukumu "Directory" tiks piešķirta mainīgā ar nosaukumu "Inbox" vērtība).

Katrs rezultātu elements <results>, līdzīgi kā parametri, satur divus elementus <name> un <assign> (elements <value> nav pieļaujams), un saturiski tas norāda, kādas notikuma rezultātu vērtības kādiem verifikācijas apraksta mainīgajiem tiks piešķirtas. Iepriekšējā piemērā ir norādīts, ka notikuma rezultātā atgrieztā vērtība ar nosaukumu "FileName" tiks piešķirta mainīgajam ar nosaukumu "fileName".

3.4.2.4. Saites

Saišu sadaļa (<links>) satur aktivitāšu pārejas. Katrs tās elements ir viena saite (<link>), un katram elementam ir divi atribūti: aktivitāte, no kuras saite iziet, un aktivitāte, kurā saite ieiet. Pārbaudot saišu definīciju, būtu jāpārliecinās:

- no sākuma aktivitātes iziet vismaz viena saite, bet neviena tajā neieiet;
- beigu aktivitātē ieiet vismaz viena saite;
- visām pārējām aktivitātēm ir vismaz viena ienākoša un izejoša saite;
- līdz katrai no aktivitātēm ir jābūt vismaz vienam ceļam no sākuma aktivitātes.

Kā redzams no valodas sintakses, tad procesa verificēšanas apraksts neparedz noteikta veida aģentu lietojumu, t.i., ja verificēšanas kontrolierim ir pieejams kāds notikumu aģents, kurš implementē prasīto saskarni, tad tas var tikt brīvi lietots procesu verificēšanas aprakstos. Protams, lietojot aģentus un aprakstot fiksējamos notikumus, ir jāzina, kādi parametri notikuma fiksēšanai ir jānorāda un kādi rezultāti, saņemot fiksēto notikumu, būs pieejami. Tādējādi piedāvātais verifikācijas mehānisms ir atvērts un ļauj pēc nepieciešamības papildināt pieejamo aģentu klāstu ar jauniem.

3.4.3. Verificēšanas apraksta valodas piemērs

Verificēšanas apraksta piemērs ir apskatāms darba pielikumā. Tajā ir aprakstīts failu apstrādes process, kurš tiks apskatīts tālākajās nodaļās kā aprobācijas piemērs (skat. 4.

nodaļu). Verificējamais process ar tā stāvokļu pārēju diagrammu ir attēlots 16. attēlā, bet procesa verifikāciju skaidrojošā shēma – 17. attēlā.

Atbilstoši galvenei procesa verifikācija tiek uzsākta ar sākuma notikumu (<initiation>start-event</initiation>). Verificējamajam procesam var būt vairākas vienlaicīgas instances (<instances>multiple</instances>). Verifikācijas galvenē ir iekļauti pieci parametri un četri mainīgie. Pirmie divi no parametriem norāda direktorijas, kurā ir meklējami ienākošie un izejošie faili. Nākamie divi parametri norāda uz datu bāzu serveri, kas tiek lietots verificējamās sistēmas datu glabāšanai. Savukārt, pēdējais no parametriem satur bankas identifikatoru (šīs bankas, EKS dalībnieces, faili tiks apstrādāti). Savukārt, četri mainīgie tiks lietoti procesa verificēšanas notikumu sasaistei, t.i., konkrētās procesa instances izsekošanai:

- F_ISN - faila, kurš tiek apstrādāts, identifikators;
- F_EncrName - šifrēta ienākošā faila nosaukums;
- F_EncrFullName - šifrēta ienākošā faila nosaukums, iekļaujot pilnu ceļu;
- F_OUT_EncrName - šifrēta izejošā faila nosaukums.

Procesa verificēšanas aprakstā ir noteiktas trīs verificējamas aktivitātes. Katrai no tām ir noteikts galvenais notikums. Pēdējām divām no aktivitātēm ir noteikts arī apakšnotikums. Pirmā no aktivitātēm ir "file-received" (saņemts jauns fails). Ar šo aktivitāti ir saistīts faila izveidošanas notikums:

- *aģents* - FileSystem;
- *notikums* - FileCreated;
- *adrese* - EKS-file-server (saīsinājums, kurš ir definēts kontrolierim pieejamo aģentu aprakstā).

Kā parametri šim notikumam tiek nodotas trīs vētrības:

- *Dircetory* - direktorija, kurā failu veidošana ir sagaidāma, un šo vērtību uzstāda ar procesa parametru;
- *Pattern* - meklējamo failu šablons, kas šajā gadījumā ir konstante "*.*";
- *Subdirs* - norāda, vai faili meklējami arī apakšdirektorijās, un šajā gadījumā vērtība ir konstante "False".

Kad šis notikums iestājas un kotrolieris saņem tā apstiprinājumu, kontrolieris saņem vairākas rezultātu vērtības, no kurām procesa verificēšanai tiek saglabātas divas:

- *FileName* - faila nosaukums tiek saglabāts procesa mainīgajā F_EncrName;
- *FileFullName* - pilns faila nosaukums ar ceļu uz to tiek saglabāts procesa mainīgajā F_EncrFullName.

Otrajai no aktivitātēm ir arī apakšnotikums. Galvenais notikums ir datu bāzes notikums, kurš iestājas, kad saņemtais fails tiek piereģistrēts datu bāzē un ir uzsākta tā apstrāde. Pēc šī notikuma tiek pārbaudīts viens apakšnotikums - saņemtajam failam ir jābūt izdzēstam no saņemto failu direktorijas. Šo notikumu nodrošina:

- *aģents* - FileSystem;
- *notikums* - FileExists;
- *adrese* - EKS-file-server (saīsinājums tāpat kā pirmās aktivitātes gadījumā).

Kā parametri šim notikumam tiek nodotas divas vērtības: meklējamais fails un pazīme, kura norāda, vai fails eksistē vai neekistē, t.i., atkarībā no norādītās vērtības tiek pārbaudīta vai nu faila eksistence, vai arī tā neeksistence.

Līdzīgi ir aprakstīta trešā aktivitāte "response-delivered" jeb beigu aktivitāte (<action id="response-delivered" type="finish">). Tās bāzes notikums ir datu bāzes notikums, kurā ar SQL procedūras palīdzību tiek pārbaudīts, vai atbildes fails ir sagatavots (atzīme datu bāzē). Ja atbildes fails ir sagatavots, tad ar apakšnotikuma palīdzību tiek pārbaudīts, vai atbildes fails ir nosūtīts.

Otrajai un trešajai aktivitātei ir norādīti laika ierobežojumi:

- "file-registered" ir jābūt sasniegtai minūtes laikā pēc "file-received";
- "response-delivered" ir jābūt sasniegtai divu minūšu laikā pēc "file-registered".

Pēdējā apraksta sadaļa satur aktivitāšu izpildes secību jeb saites starp aktivitātēm. Piemēra gadījumā process ir vienkāršs bez zarošanās: "file-received" -> "file-registered" -> "response-delivered".

3.5. Verificēšanas mehānisma darbības ierobežojumi

Piedāvātajam izpildes laika verificēšanas mehānismam ir zināmi ātrdarbības ierobežojumi. Piemēram, nevar tikt verificēti pilnīgi visi uzraugāmā procesa soļi, jo, pat ja visus šos soļus varētu verificēt, tad to verificēšana prasītu tikpat daudz laika vai vairāk nekā paša procesa izpilde. Attiecīgi verificēšana izpildītos lēnāk par verificējamo procesu. Tāpēc ir svarīgi apzināties, kādas ir darbā piedāvātā izpildes laika verificēšanas mehānisma darbības robežas. Tas tiešā veidā ietekmē, ko var verificēt un cik precīzi ir iespējams veikt verificēšanu.

Apskatot iespējamos ātrdarbības ierobežojumus, tie ir attiecināmi kā uz kontrolieri, tā uz aģentiem (sadalot kontrolieri vairākās komponentēs, būtu jāapskata arī to ietekme). Attiecībā uz aģentiem būtu jāapskata divi dažādi gadījumi:

- aģenta radītā virsslodze, tātad ietekme uz novērojamo procesu;

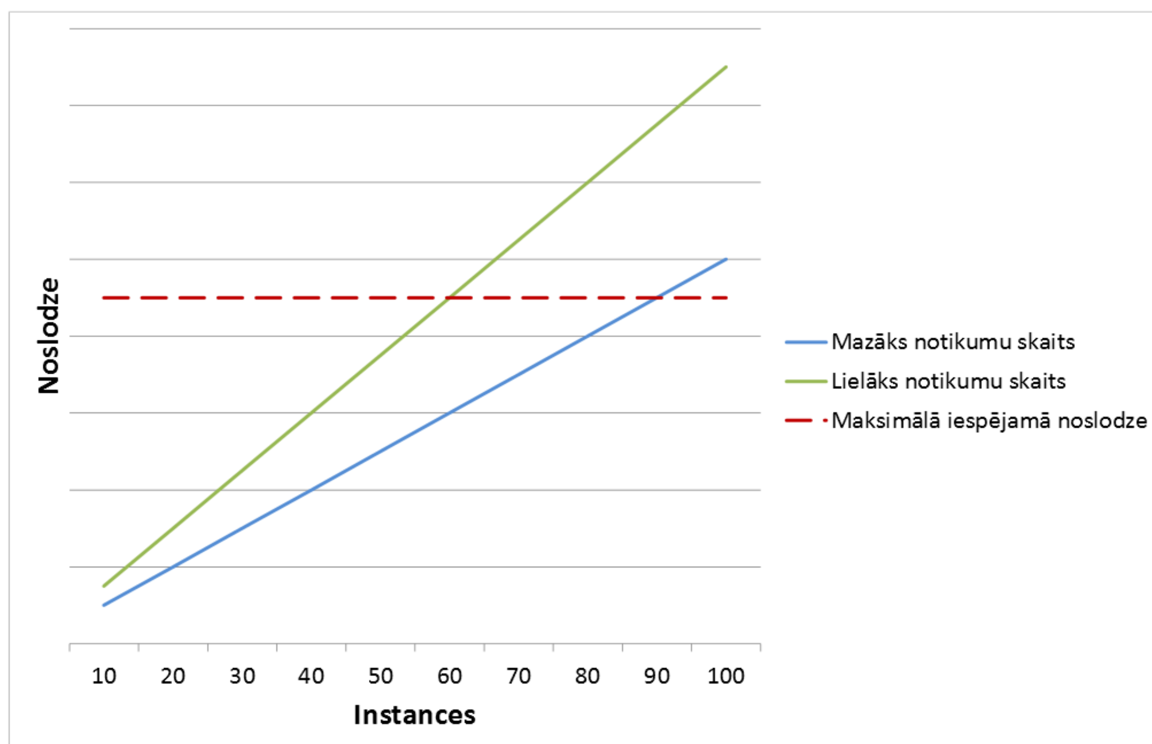
- notikumu intensitāte, kādu aģents spēj apstrādāt.

Nākamajās divās nodaļās tiks apskatīti atsevišķi kontroliera darbības un aģentu darbības ierobežojumi.

3.5.1. Kontroliera ierobežojumi

Procesu verifikācijas kontroliera ātrdarbība un verificējamo notikumu skaits, protams, ir atkarīgs no tā implementācijas, kā arī no kontrolierim izdalītajiem tehniskajiem resursiem. Tomēr, apskatot kādu abstraktu kontroliera implementāciju, ir iespējams noteikt faktorus, kuri var ietekmēt tā darbību, un noteikt to savstarpējo ietekmi. Analizējot kontroliera darbību, var identificēt, ka izpildīto procesora operāciju skaits ir saistīts ar diviem lielumiem: verificējamo procesu instanču skaitu (P) un vienā laika vienībā saņemto notikumu atstiprinājumu skaitu (N(n)) (n ir procesa instance). Attiecīgi kopējais vienā laika vienībā apstrādājamo notikumu skaits būtu visu (N(n)) summa. Pieņem, ka vienā laika vienībā kontrolieris spēj apstrādāt notikumus (MaxN). Acīmredzot, (MaxN) ir jābūt lielākam par kopējo vienlaicīgo notikumu summu.

Grafikā (1. grafiks) redzami divi iespējami noslodzes gadījumi.



1. grafiks. Kontroliera noslodze

Viena no līknēm attēlo noslodzes pieaugumu mazāka verificējamo notikumu skaita gadījumā, otra - lielāka notikumu skaita gadījumā. Ja maksimālā pieļaujamā kontroliera

noslodze ir ar sarkano līniju norādītajā līmenī, tad pirmajā gadījumā ir iespējams verificēt nosacīti 60 procesa instances, bet otrā - 90.

Attiecīgi, kontroliera darbības nodrošināšanai ir jāmeklē zināms kompromiss starp verificējamo procesu instanču skaitu un verificējamo soļu skaitu: palielinot verificējamo procesu instanču skaitu, ir jāsamazina verificējamo notikumu skaitu, un otrādi - ja tiek samazināts verificējamo procesu instanču skaits, var tikt palielināta verificēšanas precizitāte, t.i, verificējamo soļu jeb notikumu skaits. Tostarp jāņem vērā arī verificējamā soļu skaita ietekme uz aģentu darbību: palielinot vienlaicīgi verificējamo soļu skaitu, pieaugs arī aģentu noslodze.

3.5.2. Aģentu ierobežojumi

Kā jau iepriekš tika minēts, aģentu darbības ierobežojumus var sadalīt divās grupās:

- aģenta radītā virsslodze;
- notikumu intensitāte, kādu aģents spēj apstrādāt.

Pirmā no problēmām ļoti plaši tiek apskatīta dažādos ar izpildes laika verificēšanu saistītos pētījumos (piemēram, [WU13] un [NAVA13]). Jo īpaši šis jautājums tiek apskatīts iegultu sistēmu verificēšanā, kur verificēšanas un monitorēšanas virslodze var radīt nozīmīgus iekārtas darbības traucējumus. Virsslodzes minimizēšanai tiek piedāvāti dažāda veida risinājumi, piemēram:

- veikt selektīvu procesa instanču verificēšanu (verificē tikai dažas instances un tikai daļēju instances izpildes laiku), tādējādi samazinot ietekmi uz katru atsevišķu instanci;
- taimeru bāzētu monitorēšanu aizstāt ar notikumu bāzētu vai mēģināt pielāgoties novērojamam procesam un dinamiski pielāgot taimerus.

Otrā problēma, notikumu intensitāte, ir saistīta ar piedāvāto risinājumu. Notikumu intensitāti var noteikt divi faktori:

- saņemto viena veida notikumu skaits;
- pieprasīto notikumu skaits.

Abu faktoru ietekmi var aplūkot ar piemēra, failsistēmas notikuma "jauns fails", palīdzību. Kontrolieris var pieprasīt šo notikumu vienā direktorijā, kur faili tiek veidoti ar lietu intensitāti. Piemēram, iekopējot vienlaicīgi direktorijā 1000 failus, aģents saņems informāciju par 1000 failiem un nosūtīs tos kontrolierim. Lai arī reāli 1000 faili direktorijā vienlaicīgi neizveidosies (tie tiks izveidot zināmā secībā), aģentam būs nepārtraukta noslodze. Līdzīga problēma var rasties arī otrā gadījumā - kontrolieris var pieprasīt šo notikumu par

simts atšķirīgām direktorijām. Šajā gadījumā noslodzi radīs gan daudzu pieprasījumu uzturēšana, gan daudzi vienlaicīgi notikuši notikumi.

Jebkurā no iepriekš minētajiem piemēriem nozīmīgākais aģenta noslodzi ietekmējošais faktors ir viena notikuma apstrādes ātrums, t.i., cik ātri aģents apstrādā informāciju par notikušu notikumu un nosūta to kontrolierim. Lai aģents spētu apstrādāt notikumus, ilglaicīgi nedrīkst pieļaut situāciju, ka notikumu iestāšanās ātrums pārsniedz notikumu apstrādes ātrumu. Pretējā gadījumā:

- *notikumu bāzētam* aģentam kādā brīdī var pārpildīties rinda, kurā ir iekļauti apstrādi gaidošie notikumi;
- *timera bāzēts* aģents kādu no notikumiem pazaudēs, jo notikumu iestāšanās pārbaude notiks retāk par notikumu iestāšanos: starp notikumu pārbaudēm ir notikumu apstrāde, savukārt, tā prasa vairāk laika nekā jauna notikuma iestāšanās.

Attiecīgi, plānojot intensīvi strādājošu procesu verificēšanu, ir rūpīgi jāizvērtē verificējamo notikumu skaits: vai aģents to spēs apstrādāt. Tieši tāds bija nākamajās nodaļās aprakstītās aprobācijas uzdevums - novērtēt potenciālo aģentu darbību intensīvas notikumu apstrādes gadījumā.

Tādējādi līdzīgi kā kontroliera ātrdarbības gadījumā aģentu ātrdarbība ir tiešā veidā atkarīga no to implementācijas un izpildes vides (arī aparatūras). Tomēr acīmredzami, ka arī aģentu gadījumā pastāv analoģiska sakarība starp notikumu pieprasījumu un atsevišķo notikumu skaitu. Šos lielumus ir iespējams regulēt, mainot verificējamo procesu aprakstus un izvērtējot nepieciešamo verifikācijas detalizētības pakāpi.

4. APROBĀCIJA

Procesu verificēšanas mehānismam tika izveidots prototips, kura darbība tika pārbaudīta vairāku procesu verificēšanai, tostarp arī reālu sistēmu darbināšanā. Aprobācijai bija divējāds uzdevums: pārliecināties par tās principiālu atbilstību procesu verificēšanas prasībām un kaut kādā mērā novērtēt procesu verificēšanas tehniskos ierobežojumus. Tā kā darba autors ilgu laiku ir strādājis pie starpbanku maksājumu sistēmu izstrādes, tad šo sistēmu procesi tika apskatīti arī prototipa vajadzībām. Turklāt maksājumu sistēmas ir labi piemērotas šāda prototipa darbības pārbaudei, jo:

- procesi notiek pietiekmi ātri, lai eksperimenti būtu ērti atkārtojami;
- sistēmas ir labi izsekojamas, lai salīdzinātu verificēšanas rezultātus un reālo procesu izpildi;
- atsevišķu procesu izpilde parasti ir tuva reālā laika apstrādei, nodrošinot iespēju pārliecināties par verificēšanas mehānisma efektivitāti.

Aprobācijas mērķis bija pārliecināties, ka ir iespējams efektīvi implementēt autora piedāvāto verificēšanas mehānismu, kurš nodrošinātu:

- vienlaicīgu vairāku procesu verificēšanu;
- vienlaicīgu daudzu procesu instanču verificēšanu;
- vienlaicīgu sadarbību ar vairākiem notikumu aģentiem;
- papildināt aģentu sarakstu, nemainot kontrolieri.

Neapšaubāmi, piedāvātais mehānisms spēj verificēt biznesa procesus, kuru izpilde notiek lēni, t.i., katra atsevišķa procesa soļa izpildes laiks nav samērojams ar viena notikuma verificēšanu. Piemēram, parasti dokumentu apstrādes sistēmā izpildes soļi nenotiek biežāk kā reizi desmit minūtēs. Savukārt, šādu izpildīto soļu identificēšanai (t.i., pamanīt datu izmaiņas datu bāzē) ir nepieciešamas milisekundes. Pat pieņemot, ka datu bāzes notikumi tiek verificēti, piemēram, reizi desmit sekundēs, starp diviem procesa soļiem verificējamais notikums tiks pārbaudīts 60 reizes. Attiecīgi prototipēšanas laikā netika apskatīti procesi ar lēnu izpildi. Pētīti tika procesi, kuru izpilde ir pietiekami strauja, lai radītu problēmas verificēšanas mehānisma darbībai:

- procesu skaits ir pietiekami liels un to instances tiek uzsāktas bieži (pieaug kontroliera noslodze);
- viena procesa stāvokļu maiņas notiek bieži salīdzinājumā ar aģenta notikumu identificēšanas biežumu (augsta aģenta noslodze).

4.1. Implementācijas apraksts

Verificēšanas mehānisma prototips tika izveidots, lietojot kompānijas *Microsoft* tehnoloģijas,.*Net* ietvaru. Gan aģenti, gan kontrolieris tika veidoti kā *Windows* servisi. Aģentu, kontrolieru un citu servisu komunikācija tika realizēta, lietojot *WCF* tīmekļa pakalpojumus. Programmēšanai tika lietota programmēšanas valoda *C#*. Apstrādājamo datu glabāšanai tika lietota *MongoDB* [MONG14] datu bāze.

Kontroliera datu bāzes izvēli noteica lietotnes prototipa raksturs. Tā kā *MongoDB* ir dokumentu bāzēta ne-SQL datu bāzu vadības sistēma, tad tā ļauj vienkārši mainīt saglabājamos datus un to struktūru. Neskatoties uz iespēju saglabāt ļoti atšķirīgus datus, datu bāze nodrošina ātru un efektīvu datu meklēšanu. Tādējādi ne-SQL datu bāze šādam prototipa projektam ir ļoti piemērota, jo visā prototipa veidošanas laikā saglabājamie datu vienumi tika bieži mainīti.

4.1.1. Kontroliera implementācija

Kontrolieris tika implementēts saskaņā ar 3.3. nodaļā piedāvāto kontroliera arhitektūru. Lai minimizētu implementācijai nepieciešamos resursus, UI un SSS modulis netika implementēti, savukārt, DB modulis tika aizstāts ar vienkāršu spraudni, kas nodrošina datu saglabāšanu izvēlētajā datu bāzē. Savukārt, aģentu modulis un kontroliera kodols tika implementēti, nodrošinot:

- aģentu modulis nodrošina saskarni ar aģentiem atbilstoši aprakstītajai aģentu un kontroliera saskarnei;
- kontroliera kodos, nodrošina:
  - aģentu direktorijas uzturēšanu, nepiesaistoties konkrētu aģentu implementācijai, bet izmantojot tikai universālu saskarnes mehānismu;
  - procesu verificēšanas apraksti nav iebūvēti kodolā, bet kodols tos nolasa no aprakstu definīcijas direktorijas, tādējādi nodrošinot brīvu jaunu verificēšanas aprakstu pievienošanu un esošo aprakstu mainīšanu.

Lai aģentu pievienošana kontroliera direktorijai būtu pēc iespējas vienkārša, aģentu aprakstīšanai tika izveidots atsevišķs XML fails, kurā par katru aģentu ir norādīts aģenta tips un aģenta atrašanās adrese. Aģentu nodrošinātos notikumu tipus kotroliera kodols noskaidro, pieprasot aģentiem nodrošināto notikumu tipu sarakstu. Tādējādi jauna aģenta reģistrēšana ir vēl viena ieraksta pievienošana aģentu apraksta failā. Pēc šī ieraksta pievienošanas un

kontroliera pārstartēšanas, kontrolieris spēj šo aģentu lietot atbilstoši procesu verificēšanas aprakstiem.

Abi no moduļiem ir implementēti kā *Windows* servisi. Tie darbojas kā divi savstarpēji saistīti servisi, tomēr tos katru atsevišķi ir iespējams apstādināt, tādējādi nodrošinot pietiekami ērtu aprobācijas procesa pārvaldīšanu: lai pieslēgtu jaunus vai mainītu esošos procesu verificēšanas aprakstus, nepieciešams pārstartēt kontroliera kodola servisu, savukārt, gadījumā, ja nepieciešams pārtraukt aģentu ziņu saņemšanu, pietiek pārstartēt aģentu servisu.

4.1.2. Aģentu implementācija

Aģentu implementāciju noteica izvēlētie procesu piemēri. Ņemot vērā verificējamo procesu specifiku, tika izvēlēts izveidot divus notikumu aģentus: failu sistēmas aģentu un datu bāzu aģentu. Tā kā verificējamā sistēma izmanto kompānijas *Microsoft* piedāvāto infrastruktūru, tad tika izveidoti šādi aģenti:

- *Windows server* failu sistēmas notikumu aģents - *AgentFileSystem* - ar kuru var identificēt šādus notikumus:
  - jauna faila izveidošana (asinhrons);
  - faila dzēšana (asinhrons);
  - fails ir/nav atrodams direktorijā (sinhrons);
- *SQL Server* datu bāzes notikumu aģents - *AgentMSSQL* - atgriež informāciju, ja datu bāzes procedūra atgriež rezultātu.

Tā kā *Windows Server* failu sistēma nodrošina efektīvu notikumu apstrādes mehānismu, **Failu sistēmas aģents** tika bāzēts uz notikumiem. Pretēji *Failu sistēmas aģentam*, **Datu bāzes aģents** tika veidots, bāzējoties uz taimeriem. Turklāt notikumu pārbaudes taimeri netika fiksēti kodā, bet gan katram atsevišķam notikumam var tikt noteikts savs taimeris. Tas ļauj regulēt un pielāgot notikumu pieprasījumu intensitāti atbilstoši verificējamo procesu intensitātei.

4.2. Reālu sistēmu verificēšana

4.2.1. Piemēra process

Verificēšanas mehānisma pārbaudei prototips tika darbināts uz reālas sistēmas piemēra testēšanas vidē. Kā sistēma tika izvēlēta Elektroniskā klīringa sistēma (turpmāk tekstā - EKS), kura nodrošina neliela apjoma starpbanku maksājumu neto norēķinus. "EKS ir vienīgā Latvijā funkcionējošā klīringa (neto norēķinu) sistēma liela skaita klientu kredīta pārvedumu veikšanai eiro. EKS darbadienā ir septiņi klīringa cikli norēķiniem eiro (EKS eiro norēķiniem darbojas saskaņā ar TARGET2 darbadienu kalendāru)" [EKS13]. Šī sistēma ir

interesanta verificēšanas mehānisma pārbaudei, jo tajā ir realizēti gan īslaicīgi, gan ilglaicīgi procesi, turklāt tiem ir izvirzīti dažādi laika ierobežojumi. Piemēram, katram no norēķinu cikliem ir noteikti zināmi laika ierobežojumi:

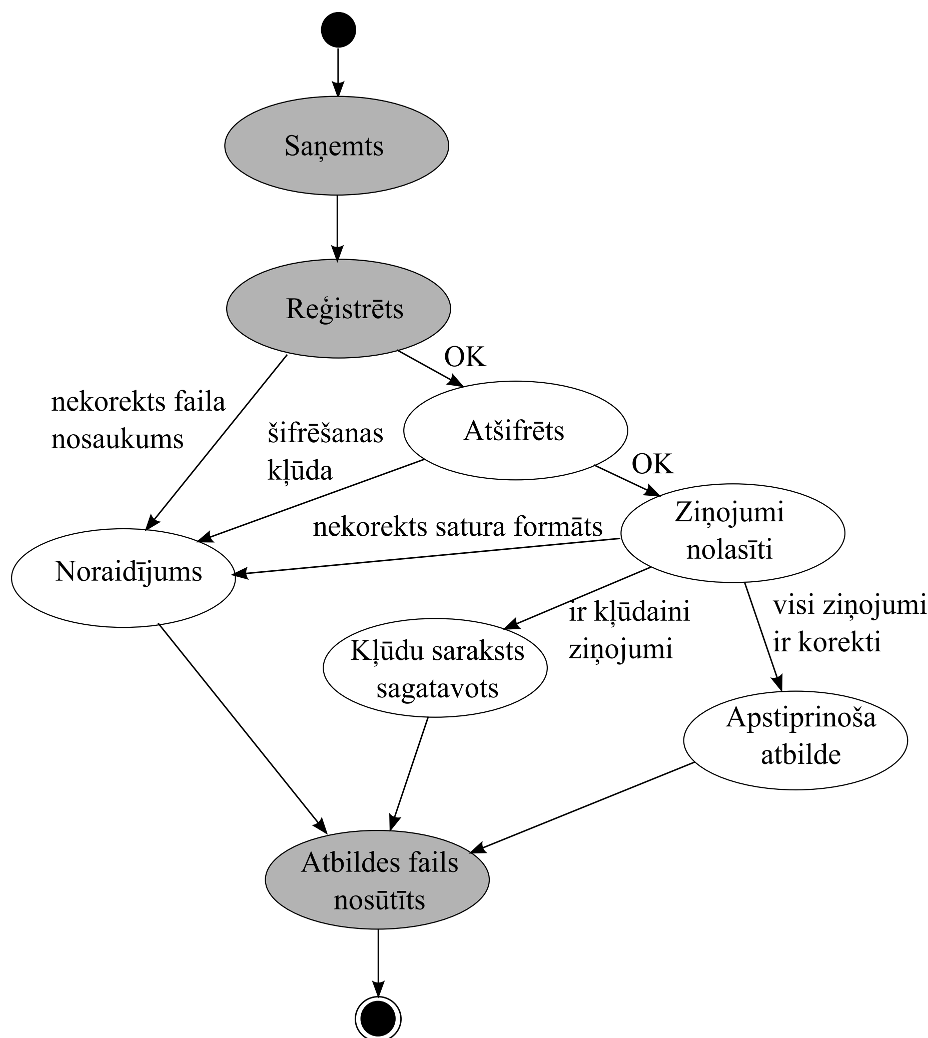
- noteikts laiks, līdz kuram tiek pieņemti maksājumi katrā no norēķinu cikliem;
- noteikts pieļaujamais norēķina laiks;
- noteikts laiks, cik ilgā laikā sistēmas klientiem (bankām) tiek nosūtīti norēķina rezultāti un adresētie maksājumi;
- sistēmas darbības laiks ir saskaņots ar citu Eiropas norēķinu sistēmu darbības laikiem (t.i., kāda EKS norēķinu cikla kavēšanās var izraisīt tālāko pārrobežu maksājumu iekļaušanu tālākajos norēķinos);
- failu apstrādes laiks.

EKS maksājumus no bankām saņem failu veidā, katrā failā iekļaujot ne vairāk kā 15000 maksājumus. Lai arī noteikumos nav noteikts maksimālais viena faila apstrādes laiks, Latvijas Banka, EKS sistēmas uzturētājs, vēlas nodrošināt pēc iespējas ātru atbildes sniegšanu bankām par maksājumu iekļaušanu norēķinā vai noraidīšanu. Tādējādi banka uzņemas zināmas servisa līmeņa saistības. Tāpat failu apstrādes ātrums ne tikai nodrošina savlaicīgu atgriezeniskās saites sniegšanu bankām, maksājumu iesniedzējam, bet arī nodrošina savlaicīgu norēķina veikšanu. Piemēram, failu pieņemšana otrajam norēķinu ciklam tiek pārtraukta plkst. 9:20, savukārt, rezultāti sistēmas dalībniekiem ir jānosūta līdz plkst. 10:00. Šajā laikā ir jāveic norēķins saistītajā bruto norēķinu sistēmā un jāsagatavo visi rezultāti. Pat ja jaunu failu pieņemšana pēc 9:20 tiek pārtraukta, lēnas failu apstrādes rezultātā var izveidoties rinda ar apstrādājamiem savlaicīgi iesūtītiem maksājumu failiem. Piemēram, lēna 50 failu apstrāde, kuri iesūtīti laika posmā no 9:18 - 9:20, var izraisīt nepieļaujamu norēķinu kavēšanos. Šādā gadījumā, ja savlaicīgi netiek pamanīta failu apstrādes lēndarbība, var aizkavēties viss sistēmas darba grafiks, iespējams, pat maksājumu nosūtīšana uz citām Eiropas norēķinu sistēmām. Tāpēc failu apstrādes ātruma un apstrādājamo failu rindas monitorēšana ir vieni no būtiskākajiem uzraugāmajiem lielumiem.

Ņemot vērā iepriekš teikto, failu apstrāde tika ņemta kā pamata piemērs verificēšanas mehānisma pārbaudei, jo:

- failu apstrāde notiek pietiekami ātri un tā ir salīdzināma ar aģentu darbību;
- ģenerējot failus, var brīvi veidot nepieciešamo procesu instanču skaitu, lai pārliecinātos par verificēšanas kontroliera darbības ierobežojumiem;
- katra atsevišķa procesa instance izpildās pietiekami īsā laikā, lai eksperimentus varētu salīdzinoši viegli atkārtot;

- verificēšanai pietiek ar nelielu aģentu skaitu un tie strādā saskaņotā vidē, potenciāli neradot tehnoloģiskas problēmas, saskaņojot vides.



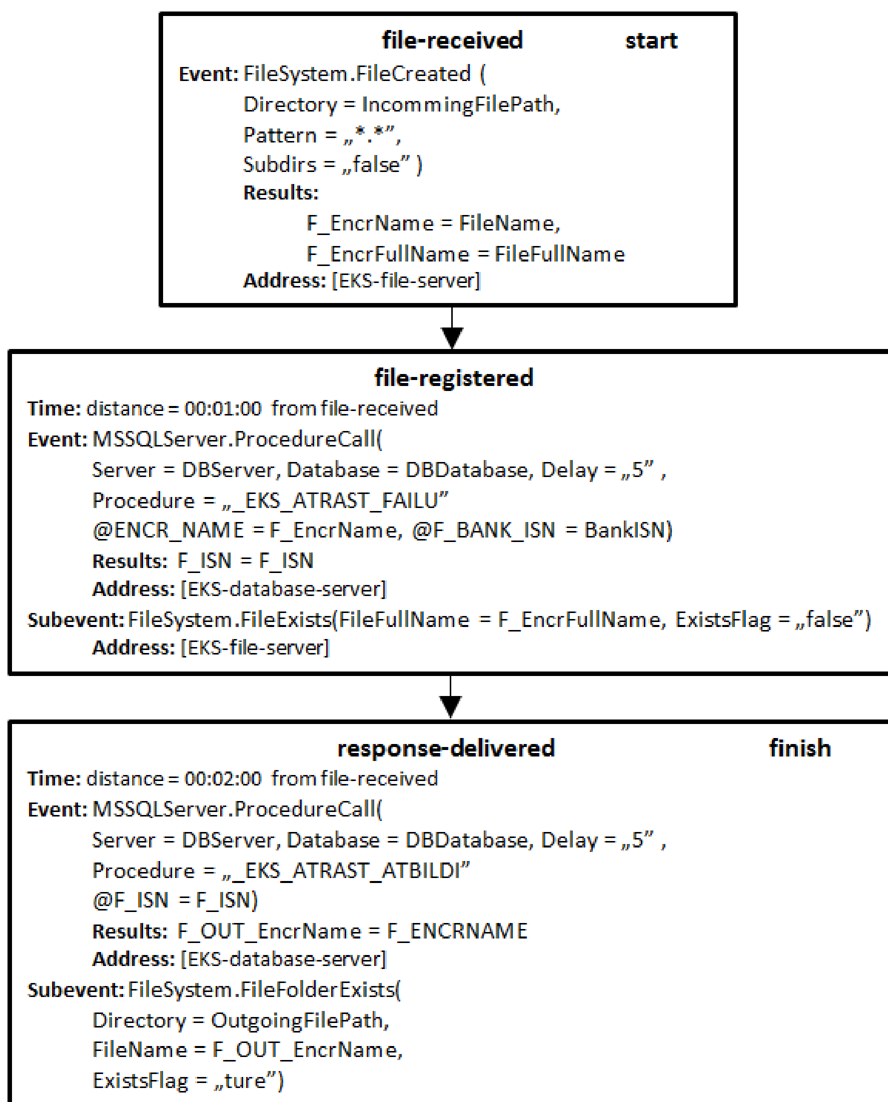
16. attēls. Faila apstrādes stāvokļu pārejas diagramma

Vienkāršota viena faila apstrādes stāvokļu pārejas diagramma ir attēlota nākamajā shēmā (skat. 16. attēls). Vispirms EKS dalībnieks failu iesūta failu apmaiņas serverī, un fails nonāk stāvoklī "saņemts". Sistēma, pamanot faila saņemšanu, piereģistrē failu datu bāzē, pārvieto uz darba direktoriju un uzsāk tā apstrādi. Pirmais apstrādes solis ir faila nosaukuma pārbaude. Ja faila nosaukums ir korekts, fails tiek atšifrēts, un tiek pārbaudīts faila šifrētāja paraksts. Ja faila nosaukums ir nekorekts vai arī ir kādas atšifrēšanas vai paraksta problēmas, tiek sagatavots atbildes faila saturs: faila apstrādes noraidījums. Savukārt, ja atšifrēšana ir beigusies veiksmīgi, EKS pārbauda faila formātu un nolasa ziņojumus. Ja faila formāts nav korekts, tam tiek sagatavota atbilde, noraidījums. Ja faila ziņojumus izdodas nolasīt, tiek pārbaudīta to korektība, un, ja kāds no ziņojumiem ir kļūdains, tiek sagatavota atbilde ar kļūdaino maksājumu noraidījumu. Pretējā gadījumā, ja visi maksājumi ir korekti, EKS dalībniekam tiek sagatavots maksājumu pieņemšanas apstiprinājums. Pēdējā solī sagatavotā

atbilde (faila apstrādes noraidījums, kļūdaino maksājumu noraidījums vai apstrādes apstiprinājums) tiek nosūtīta klientam: izveidots fails uz diska, nošifrēts un iekopēts dalībnieka izejošo failu "pastkastē" (direktorija uz failu apmaiņas servera).

Stāvokļu pārejas diagrammā tika identificēti trīs stāvokļi (skat. 16. attēls, atbilstošie stāvokļi iekrāsoti pelēkā krāsā), kuru iestāšanos varētu verificēt. Praktiski verificēts varētu tikt arī lielāks stāvokļu skaits, jo visas šīs stāvokļu pārejas tiek fiksētas sistēmas datu bāzē. Tomēr no procesa verificēšanas viedokļa, šo stāvokļu pārbaude nav nepieciešama. No konkrētā uzdevuma viedokļa svarīgs ir kopējais faila apstrādes laiks.

Initiation: start event
Instances: multiple
Parameters: IncommingFilePath, OutgoingFilePath, DBServer, DBDatabase, BankISN
Variables: F_ISN, F_EncrName, F_EncrFullName, F_OUT_EncrName



17. attēls. Faila apstrādes procesa verificēšanas apraksts {graph:fileProcessVerification}

Turklāt tā kā ir iespējamas tikai trīs atbildes un no procesa viedokļa tās visas ir legālas, tad atbildes sagatavošana apliecina, ka atbilde ir sagatavota korekti. Tāpat ir jāņem vērā datu bāzes aģenta (faila apstrādes statusi tiek fiksēti datu bāzē, tādēļ tos ir iespējams verificēt tikai ar datu bāzes aģenta palīdzību) darbība - taimera bāzēta notikumu konstatēšana. Mazu failu apstrādes gadījumā faila stāvokļi var mainīties ievērojami ātrāk kā uzstādītais taimeris. Tādēļ kāda faila statusa maiņa var tikt nepamanīta.

Atbilstoši faila apstrādes stāvokļu diagrammai tika izveidots faila apstrādes procesa verificēšanas apraksts (17. attēls). Lai arī varētu rasties iespaids, ka mehānisma pārbaude tika veikta ļoti vienkāršotam verificēšanas aprakstam, tomēr jau šis piemērs deva pietiekamu ieskatu mehānisma vērtējumā. Izmantojot šo modeli, bija iespējams pārbaudīt gan verificēšanas mehānisma un aģentu radīto virsslodzi, gan novērot robežgadījumus, gan kontroliera un aģentu pārslodzi.

4.2.2. Iegūtie rezultāti

Kā jau iepriekš tika minēts, verificēšanas mehānisma pārbaudei, tika lietots EKS failu apstrādes process, kura apraksts minēts iepriekšējā nodaļā (skat. 4.2.1. nodaļu). Verificēšanas procesa apraksts paredz viena EKS dalībnieka failu aptrādes verificēšanu. Lai nodrošinātu vairāku dalībnieku failu apstrādes verificēšanu, tika izveidoti trīs verificēšanas apraksti trijiem dalībniekiem - visi apraksti bija analogi ar vienīgo atšķirību parametru *BankISN*, *IncommingFilePath* un *OutgoingFilePath* vērtībās. Lai arī apraksti bija analogi, no verificēšanas mehānisma viedokļa tie bija trīs dažādi procesu apraksti, un šo procesu instances tiks verificētas atblistoši atšķirīgiem aprakstiem.

Testa vidē tika noģenerēti 80 maksājumu faili no trīs dažādiem dalībniekiem (skat. 1. tabula).

1. tabula. Testēšanā lietotais failu skaits {tab:aprobFiles}

Dalībnieks	Sagatavotie faili	Maksājumi
Banka 1	50	3070
Banka 2	10	200
Banka 3	20	400
Kopā	**80**	**3670**

Failu izveidošana testa vidē nenotika atbilstoši reālajai sistēmas lietošanas situācijai: dalībnieki failus iekopē failu apmaiņas serverī pakāpeniski. Savukārt, testa faili tika iekopēti vienkopus uzreiz. Šāda rīcība izraisīja 80 vienlaicīgu procesa verificēšanas instanču

izveidošanu - failu sistēmas aģents notikumus fiksē bāzējoties uz failsistēmas notikumiem, tādēļ katra faila iekopēšana nekavējoties izraisīja faila izveidošanas notikumu.

Savukārt, tālākā failu procesu verificēšana notika pakāpeniski, atbilstoši failu apstrādei. Ir jāņem vērā, ka EKS veic vairāku failu apstrādi vienlaicīgi. Attiecīgi arī kontrolieris turpmākos failu apstrādes notikumus saņēma pakāpeniski par vairākiem procesiem pamīšus. Tāpat sinhronie notikumi tika pieprasīti pamīšus.

Lai arī EKS failu apstrādi veic pamīšus, visu 80 failu apstrāde nenotiek vienlaicīgi. Tādēļ, atkarībā no verificēšanas aprakstā uzstādītā laika intervāla, daži no pēdējiem apstrādē pieņemtajiem failiem tika apstrādāti ar nokavēšanos. Attiecīgi verificēšanas mehānisms šos failu apstrādes procesus identificēja kā kļūdainus un informēja par procesu kļūdu. 11 no failiem tika atzīmēti kā kavēti, jo pirmais solis neiestājās savlaicīgi.

Interesanti, ka EKS izstrādātāji apgavoja, ka dalībnieku faili tiek pieņemti vienmērīgi, t.i., ja dalībnieks A atsūta 100 failus un pēc tam dalībnieks B atsūta 5 failus, tad dalībnieku A un B faili tiks apstrādāti pamīšus, lai dalībniekam B atbildes nebūtu jāgaida pēc visu dalībnieka A failu apstrādes. Verificēšanas rezultātā par kavētu tika atzīti 11 dalībnieka *Banka 1* (1. tabula) faili, t.i., tā dalībnieka faili, kurš iesūtīja lielāko failu skaitu. Tādējādi, lietojot izpildes laika verificēšanas mehānismu izdevās pārbaudīt, vai implementētā pakāpeniskā failu apstrāde darbojas, kā tas tika apgalvots.

2. tabula. Aizkavēto failu pārskats

Nr.	**Dalībnieks**	**Fails**	**Apstrādes uzsākšana**	**Aizkavētais notikums**	**Problēmas konstatēšanas laiks**
1	Banka 1	PE2100560.p7m	09:41:47	file-registered	09:46:47
2	Banka 1	PE2100571.p7m	09:41:47	file-registered	09:46:47
3	Banka 1	PE2100572.p7m	09:41:47	file-registered	09:46:47
4	Banka 1	PE2100573.p7m	09:41:47	file-registered	09:46:47
5	Banka 1	PE2100574.p7m	09:41:47	file-registered	09:46:47
6	Banka 1	PE2100575.p7m	09:41:47	file-registered	09:46:47
7	Banka 1	PE2100576.p7m	09:41:47	file-registered	09:46:47
8	Banka 1	PE2100577.p7m	09:41:47	file-registered	09:46:47
9	Banka 1	PE2100578.p7m	09:41:47	file-registered	09:46:47
10	Banka 1	PE2100579.p7m	09:41:47	file-registered	09:46:47
11	Banka 1	PE2100580.p7m	09:41:47	file-registered	09:46:47

Svarīgi, ka failu apstrādes kavēšanās tika konstatēta precīzi noteiktajā laikā (skat. 2. tabula) - visi faili tika iekopēti vienlaicīgi (9:41:47) un to reģistrēšanas aizkavēšanās tika fiksēta tieši noteiktajā laikā - 5 minūtes pēc failu saņemšanas.

Ar prototipa palīdzību tika vērtēta arī aģentu radītā virsslodze. Kontroliera process un failu sistēmas aģents tika darbināti uz atsevišķi izdalīta servera. Savukārt, datu bāzes aģents tika darbināts uz EKS sistēmas datubāzes servera - pēc iespējas tuvu vietai, kur notiek novērojamie notikumi. Tā kā failu sistēmas notikumu aģents ir notikumu bāzēts, procesora noslodzes monitorā pamanāmu virslodzi tas radīja tikai EKS apstrādājamo failu iekopēšanas brīdī (uzreiz tika uzsākta 80 failu apstrāde). Savukārt, vēlākajā darba posmā, lai arī failu notikumi bija pietiekami blīvi, virslodze nebija novērojama. Pretēji failu apstrādes aģentam, datu bāzes aģents ir taimera bāzēts, tāpēc uzmanīgi iedziļinoties datu bāzes servera procesora noslodzē, varēja novērot regulāru aģenta radītu virslodzi, tomēr arī šī aģenta radītā virsslodze bija nenozīmīga, un vidējais failu apstrādes ātrums sakrita ar apstrādes ātrumu bez paralēlas procesu verificēšanas.

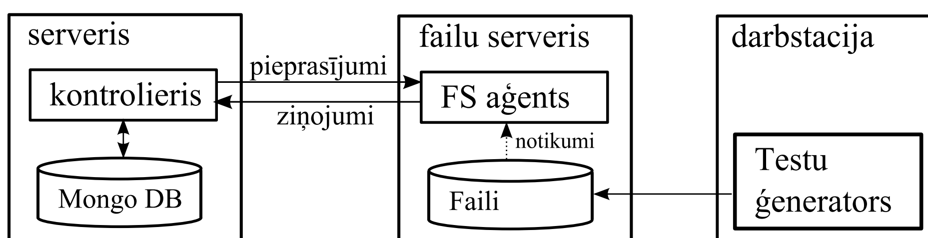
4.3. Aprobēšana simulētā vidē

Viens no aprobācijas mērķiem bija novērtēt piedāvātā risinājuma ātrdarbības iespējas. Lai to izdarītu, tika izveidota īpaša testēšanas vide, kā arī programmatūra testa procesu ģenerēšanai un izpildei. Tādējādi tika izveidoti apstākļi, kuros var novērtēt:

- procesu notikumu izpildes un to „pamanīšanas” laiku;
- nepamanīto notikumu daudzumu;
- nepamanīto procesu instanču daudzumu.

Testa programmatūras darbības pierakstos tika fiksētas visas izveidotās procesu instances un to apstrādes stāvokļu maiņas. Savukārt procesu izpildes verificēšanas rīka prototips reģistrē visas procesu aprakstiem atbilstošās procesu instances un to pārmaiņas. Salīdzinot abus pierakstus, ir iespējams salīdzināt reālo izpildi un verificēto. Mainot dažādus ģeneratora parametrus, ir iespējams mainīt vienlaicīgo procesu instanču skaitu un instanču izpildes intensitāti. Tādējādi ir iespējams novērtēt verifikācijas menānisma darbību dažādas noslodzes apstākļos.

Lai samazinātu dažādu blakusfaktoru ietekmi uz novērtējumu un nesarežģītu iegūto rezultātu skaidrojumu, testa procesu izpildei tika lietots viens serveris un tikai viens notikumu aģents – failu notikumu apstrādes aģents. Tāpat visiem verificējamajiem procesiem aprakstā netika uzstādīti izpildes laika ierobežojumi, pretējā gadījumā katrā stituācijā, kur procesa instances verificēšana beigusies ar kļūdas ziņojumu, būtu jāanalizē, vai tās cēlonis ir kāda procesa izpildes soļa pazaudēšana vai arī izpildes kavēšanās.

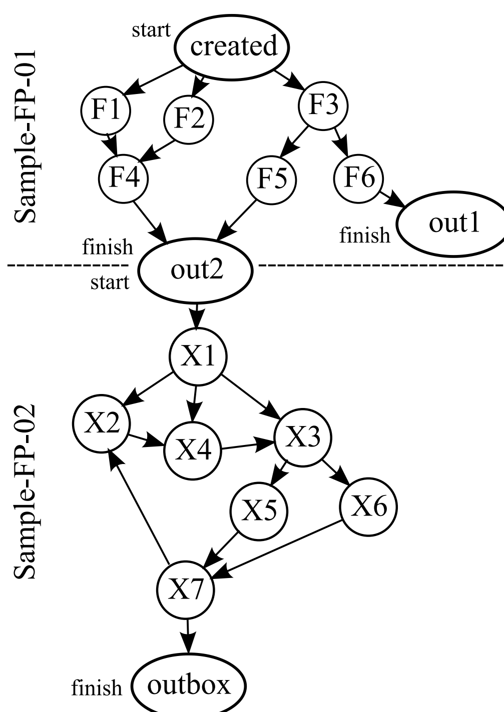


18. attēls. Aprobācijas vides arhitektūra

4.3.1. Apskatītie piemēra procesi

Šajā aprobācijas solī tika lietoti vairāki procesi: četri vienkārši (bez zarošanās) un divi sarežģīti. Vienkāršie procesi satāv no diviem vai trīs soļiem, no kuriem pirmais ir faila izveidošana, bet pēdējais – faila dzēšana.

Savukārt, sarežģītie procesi (*Sample-FP-01* un *Sample-FP-02*) sastāv no vairākiem soļiem un pieļauj arī procesu zarošanos. Turklāt, atbilstoši testēšanas plānam otrais no sarežgītajiem procesiem, *Sample-FP-02*, tiek uzsākts gadījumā, ja *Sample-FP-01* tiek pabeigts noteiktā beigu stāvoklī.



19. attēls. Procesu Sample-FP-01 un Sample-FP-02 stāvokļu diagrammas

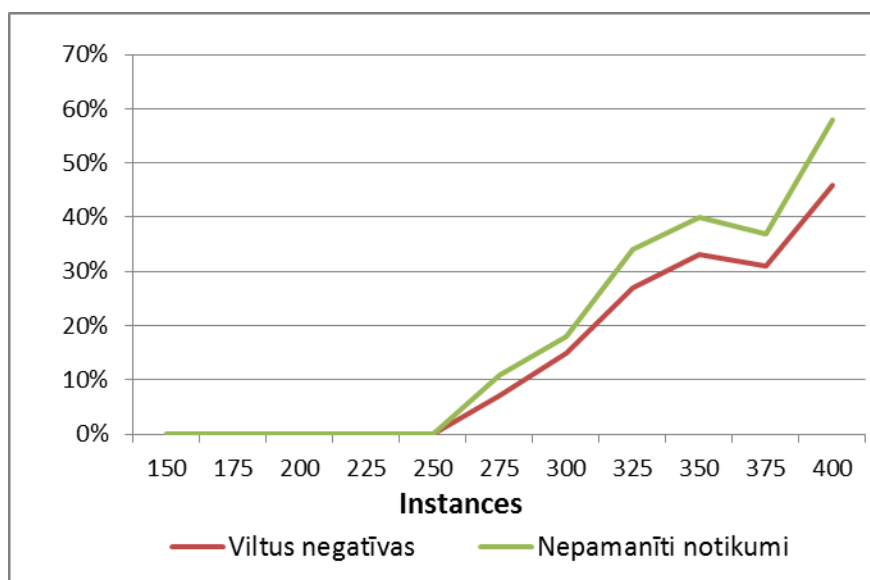
Piemēra procesu ietvaros izveidotie faili (katrs fails identificē vienu procesa instanci) tika pārvietoti starp direktorijām ar stāvokļiem atbilstošiem nosaukumiem. Attiecīgi procesa *Sample-FP-01* viena no beigu stāvokļiem (*out2* – skat 19. attēlu) realizēšana, nozīmē, ka direktorijā ..*\out2* tika novietots procesa instancei atbilstošais fails. Faila izveidošana direktorijā ..*\out2* izraisa jaunas procesa *Sample-FP-02* instances verificēšanas uzsākšanu, jo

šī direktorija tika atzīmētā kā atbilstoša otra procesa sākuma stāvoklis *out2*. Tādējādi tika iegūti divi savstarpēji saistīti procesi.

Izveidotais ģenerators nodrošināja vienmērīgu jaunu procesu instanču izveidošanu. Vienīgais izņēmums bija process *Sample-FP-02*, jo tas bija saistīts ar procesa *Sample-FP-01* instanču izpildi. Atbilstoši testa piemēru ģeneratora uzstādījumiem, 80% no *Sample-FP-01* instancēm nonāca procesa beigu stāvoklī *out2*. Attiecīgi procesa *Sample-FP-02* instances bija 80% no procesa *Sample-FP-01* instanču skaita.

4.3.2. Iegūtie rezultāti

Atbilstoši sākotnējam novērtējumam, sasniedzot zināmu procesu izpildes intensitāti, izveidotais verificēšānas mehānisma prototips nepaspēja konstatēt notikuma iestāšanos. Konkrētajā gadījumā (skat. 2. grafiku) tika uzstādīts, ka katra procesa instance ģenerē vienu notikumu katrās 4.5 sekundēs. Pie šādas procesu izpildes intensitātes, verificēšanas mehānisms strādāja korekti, kamēr paralēli izpildāmo procesu instanču skaits nepārsniedza 250 instanes.



2. grafiks. Verificējamo instanču skaita un verificēšanas kļūdu atbilstība

Sasniedzot norādīto robežu, procesu instances ģenerēja lielāku notiku skaitu, kā kontrolieris spēja apstrādāt. Attiecīgi, kontrolieris, verificējot instanču izpildi, nosūtīja pieprasījumus notikumu reģistrēšanai pēc notikumu iestāšanās. Tādējādi kontrolieris nekonstatēja procesu instanču korektu izpildi, un attiecīgās instances tika atzītas par nekorekti izpildītām. Iepriekš minētais grafiks parāda, ka, pieaugot kopējai sistēmas noslodzei, pieaug nepamanīto notikumu skaits un atbilstoši – arī viltus negatīvu (no angļu valodas – *false negative*) verifikācijas ziņojumu skaits.

Protams, mainot sistēmas konfigurāciju, uzlabojot prototipa implementāciju un palielinot pieejamos sistēmu resursus, minēto paralēli verificējamo procesu instanču skaitu varētu palielināt. Tomēr ņemot vērā risinājumu sadalīto raksturu, vienmēr būs zināma procesu izpildes robežintensitāte, kuru pārsniedzot, verifikācijas mehānims kādus no procesu notikumiem pazaudēs. Savukārt, pazaudējot kādu no notikumiem, procesa instances izpilde tiek atzīta par kļūdainu.

4.4. Aprobācijas secinājumi

Apkopojot aprobācijas rezultātus, varam secināt, ka, lai arī sākotnēji piedāvātais izpildes laika verifikācijas mehānisms var izskatīties vienkāršs, tomēr tas ir darboties spējīgs ar pietiekami plašu pielietojumu:

- procesu verificēšana ir vienkārši izveidojama (jaunu aprakstu pievienošana) un nepieciešamības gadījumā modificējama;
- verificēšanas mehānisms ir paplašināms, tādējādi nodrošinot iespēju verificēt atšķirīgas lietošanas vides;
- lai arī piedāvātajam risinājumam ir ātrdarbības ierobežojumi, tomēr aprobācijā apskatīto procesu izpildes ātrums ievērojami pārsniedza tipiskus biznesa procesus un faktiski ir pielīdzināms reālā laika sistēmu izpildei;
- risinājums nodrošina plaša spektra procesu verificēšanu.

Testējot piedāvāto verificēšanas mehānismu, tika konstatēti vairāki potenciāli procesu verificēšanas apraksta valodas papildinājumi. Pirmkārt, valodā būtu noderīgi ieviest ceļu zarošanās nosacījumus. Tie palīdzētu precīzāk aprakstīt verificēšanas procesu. Piemēram, ja šāds izteiksmes līdzeklis būtu iespējams, aprobācijā lietotajā failu apstrādes piemērā varētu iekļaut nosacījumu saskaņā ar kuru tiktu verificēts, vai tiek sagatavots pareizs atbildes fails: saņemtā faila apstrādes apstiprinājums, ziņojumu kļūdu paziņojums vai aptrādes noraidījuma fails.

Otrkārt, šobrīd esošie verificēšanas apraksta valodas izteiksmes līdzekļi neļauj attēlot paralēlas izpildes plūsmas. Šāda plūsmu attēlošana ir nozīmīga, piemēram, dokumentu pārvaldes sistēmās, kur tiek implementēta dokumentu apstiprināšana: vairāki lietotāji dažādās lomās apstiprina vienu un to pašu dokumentu vienlaicīgi. Attiecīgi nav iespējams noteikt, kurš no apstiprinājumiem būs pirmais, kurš - pēdējais. Tāpat paralēlu notikumu aprakstīšana ļautu verificēt procesa notikumus, kuri laikā ir ļoti tuvi: verificēšanas aprakstā tos iekļaujot paralēli, būtu iespējams izvairīties no situācijas, kad kaut kādas kontroliera vai aģenta darbības aiztures dēļ netiek pamanīts kāds no notikumiem.

Aprobācija sniedza arī metodoloģisku pieredzi par verificēšanas mehānisma pielietošanu. Lai arī aprobētais verificēšanas mehānisms pilnībā nodrošina verificēšanu, neiejaucoties uzraugāmā procesa darbībā, lai veiksmīgi aprakstītu procesa verificēšanu, ir jābūt labām zināšanām par procesa implementāciju. Pretējā gadījumā ir grūti sagatavot pietiekami detalizētu verificēšanas aprakstu. Apskatot aprobācijā lietoto piemēra procesu (skat. 4.2.1. nodaļu), pārzinot tikai sistēmas konfigurēšanu, būtu iespējams definēt sistēmas ārējās saskarnes notikumus: failu izveidošanu un dzēšanu. Lai precīzāk verificētu failu apstrādes plūsmu, t.i, izmantotu failu apstrādes statusus, ir jāzina, kā datu bāzē tiek saglabāta informācija par failu apstrādi.

SECINĀJUMI

Lai arī izpildes laika verificēšana ir ilgu laiku lietota informācijas sistēmu kvalitātes nodrošināšanas metode, tomēr tās tipiskais pielietojums ir bijis reālā laika sistēmās, robotikā un dažādās elektroniskās ierīcēs. Tāpat izpildes laika verifikācija ir bijusi pielietota informācijas sistēmām to testēšanas un stabilizēšanas posmā, tomēr reti kad tā tiek lietota visā sistēmu darbības laikā. Tas pamatā ir saistīts ar verificēšanas risinājumu radīto virsslodzi, salīdzinoši sarežģītajiem izpildes laika verificēšanas risinājumiem un nepieciešamību savlaicīgi iebūvēt programmatūrā verificēšanai nepieciešamo papildus kodu. Savukārt šī darba autors ir piedāvā izpildes laika verificēšanas metodi, kura ir viekārši pielietojama, turklāt bez esošo sistēmu modificēšanas. Tātad **tiek apstiprinātas sākotnēji izvirzītās tēzes**:

- izpildes laika verificēšana kā kvalitātes nodrošināšanas metode ir pielietojama arī biznesa procesiem;
- to var lietot procesu izpildes laikā ikdienā;
- piedāvātais izpildes laika verificēšanas mehānisms nerada nozīmīgu virsslodzi;
- lietojot piedāvāto aģentu risinājumu, risinājums ir paplašināms dažādām biznes proceu izpildei nepieciešamām platformām.

Tādējādi šī **darba zinātniskā novitāte ir**:

- praktiski pielietojama biznesa procesu izpildes laika verificēšanas risinājuma izveide;
- ārējs verificēšanas risinājums, kurš tiešā veidā neiejaucas verificējamo sistēmu darībā un ir pielietojams jau eksistējošām informācijas sistēmām.

Apkopojot citu autoru teikto un iepriekš darbā rakstīto, būtu svarīgi uzsvērt:

- Pirmkārt, sekojot citu autoru izteikumiem, jāpiekrīt, ka izpildes laika verificēšana ir nozīmīga programmatūras kvalitātes nodrošināšanas komponente. Pieaugot sistēmu un to izpildes vides sarežģītībai, arvien nozīmīgāk ir nepārtraukti sekot līdzi programmatūras izpildes korektumam jeb verificēt to izpildes laikā. Pretējā gadījumā varbūt sākotnēji nenozīmīgas sistēmu izpildes anomālijas ar nenozīmīgu iespaidu uz kopējo procesu to konstatēšanas brīdī (t.i., notiekot izpildes kļūdai) var nozīmīgi ietekmēt visu procesa izpildi.

- Otrkārt, ir svarīgi norādīt, ka izpildes laika verificēšana biznesa procesu jomai ir ne mazāk nozīmīga kā programmatūrai. Biznesa procesu gadījumā izpildes laika verificēšana parasti tiek aizmirsta. Labākajā gadījumā izpildes laika verificēšanai līdzīgi mehānismi tiek iebūvēti, piemēram, lietojot dokumentu pārvaldības sistēmu plūsmu pārvaldīšanas komponentes. Tomēr arī tie ir tikai daļēji verificēšanas risinājumi, un tiek lietoti tikai gadījumos, kad izstrādātāji to ir ieplānojuši. Lielākoties procesu lietotāji pat nenojauš, ka arī biznesa procesu izpildi varētu verificēt automātiski, un šāda veida kontroles tiek veiktas manuāli, piemēram, kvalitates vadības sistēmu ietvaros.

Darba autors piedāvā risinājumu šāda veida problēmām, t.i., praktiski lietojamu izpildes laika verificēšanas risinājumu, kas primāri ir paredzēts biznesa procesiem. Tas samērā vienkāršā veidā ļauj verificēt biznesa procesu izpildes atbilstību nozīmīgākajiem procesu korektuma kritērijiem:

- procesa izpildes laika ierobežojumi;
- procesa izpildes atļautie ceļi.

Kā nozīmīgs ieguvums ir jāatzīmē procesa apraksts, lietojot procesu lietotājiem saprotamus jēdzienus. Lai arī procesu aprakstā esošās notikumu definīcijas ir tehniski pietiekami sarežģītas, tomēr pati procesu plūsma tiek aprakstīta lietotājam saprotamos jēdzienos – verificējamie procesu soļi un to secība lietotājiem ir ikdienišķi un ikdienā lietoti.

Tomēr neskatoties uz it kā vienkāršo autora piedāvāto verificēšanas mehānisma risinājumu, tas, salīdzinot ar citiem risinājumiem, ir pielietojams plašam uzdevumu klāstam, piemēram:

- Risinājums ir pielietojams gadījumos, kad procesi izpildās salīdzinoši ātrāk nekā ikdienišķi biznesa procesi un dažādas izpildes anomālijas būtu jāpamana pēc iespējas drīz. Šāda procesa piemērs tika apskatīts darba aprobācijas laikā (aprakstīts 4. nodaļā).
- Aprobācijas nolūkos izmantotajā klīringa sistēmā maksājumu norēķini ir sadalīti septiņos norēķina ciklos, kuriem ir jāizpildās noteiktos laikos. Tā kā dažādu ar sistēmu nesaistītu iemeslu dēļ šie izpildes laiki var tikt mainīti, tad sistēmā nav iebūvēta stingra termiņu kontrole. Vēl vairāk - laika gaitā var mainīties ne tikai ciklu izpildes laiki, bet arī ciklu skaits (piemēram, 2007. gadā EKS bija tikai divi norēķinu cikli). Lietojot autora piedāvāto verificēšanas risinājumu, būtu iespējams kontrolēt ciklu izpildes laiku ievērošanu un ziņot par pārkāpumiem.

- Eksistē procesi, kuru izpildē ir manuālas operācijas, kurās nav iesaistītas informācijas sistēmas. Piemēram, izsludinot valsts iepirkumus, tie ir jāpublicē Iepirkumu uzraudzības biroja (turpmāk tekstā - IUB) mājas lapā. Pieņemot, ka iepirkuma dokumentu apstiprināšana notiek, lietojot kādu datorizētu dokumentu pārvaldības risinājumu, visticamāk publicēšanu IUB mājas lapā veic cilvēks. Aprakstot šādu procesu autora piedāvātajā verificēšanas apraksta valodā, kā pēdējais procesa notikums (t.i., pēdējais solis) būtu jānorāda iepirkuma informācijas parādīšanās IUB mājas lapā. Tas nozīmētu, ka aģentam būtu jāseko IUB mājas lapā publicētajiem iepirkumiem. Tad verificēšanas kontrolieris varētu automātiski kontrolēt arī šo manuālo darbību, par korekti pabeigtām atzīstot tikai tās iepirkuma procesa instances, kuru dati ir publicēti IUB mājas lapā. Pārējos gadījumos tiktu paziņots par kļūdu, t.i., atbildīgais darbinieks ir aizmirsis nopublicēt šos datus.
- Verificēšanas risinājums ir lietojams, izstrādājot jaunu programmatūru, kuras izstrādātāji vēlas nodrošināt savai sistēmai izpildes laika verificēšanu. Šādā gadījumā izstrādātājiem sistēmā ir jāiebūvē vienkāršs aģents, kurš sniegtu informāciju universālam verificēšanas kontrolierim.

Šādi piemēri varētu tikt minēti vēl, tomēr arī šie parāda, ka piedāvātajam verifikācijas risinājumam var būt visai plašs pielietojums.

Piedāvātais risinājums nodrošina verificēšanu neierobežotam platformu skaitam. Vienīgais nosacījums ir atbilstoša aģenta izvietošanas iespēja attiecīgajā platformā. Saglabājot universālu verifikācijas kontrolieri, izstrādātāji var veidot pēc patikas plašu aģentu klāstu notikumu identificēšanai. Realizējot aģentus, tajos ir jāimplementē autora norādītā saskarne un aģents jāreģistrē kontroliera direktorijā. Savukārt, tālākā lietošana visiem aģentiem ir analoģiska, turklāt dažādu aģentu lietojums neietekmē verificēšanas aprakstus (atskaitot notikumu parametru un atgriezto rezultātu definēšanu). Tādējādi darbā piedāvātais risinājums ir atvērts un viegli paplašināms. To apliecina risinājuma aprobācijas mērķiem izveidotais kontroliera prototips, kas implementē nozīmīgākās risinājuma īpašības.

Kā vēl viena pozitīva risinājuma iezīme ir vienkārša jaunu procesu verificēšanas aprakstu sagatavošanas vai esošo modificēšanas iespēja. Lai arī izvēlētais apraksta formāts (XML) parastam sistēmu lietotājam nav ērti pārskatāms, pieredzējušiem speciālistiem tā lietošana problēmas nesagādās, jo aprakstu sagatavošanai un modificēšanai ir izveidotas atbilstošas XML shēmas. Turklāt lietotā verifikācijas apraksta valoda satur nelielu jēdzienu skaitu, kas tās lietošanu padara īpaši vienkāršu. Tāpat arī kontrolierī, implementējot efektīvas

kontroles un izsmeļošus problēmu skaidrojumus, var ātri identificēt nepareizi sagatavotus aprakstus.

Labai verificēšanas implementēšanai ir nepieciešamas labas zināšanas par verificējamo procesu. Protams, vislabāk procesa verificēšanas aprakstus varētu sagatavot sistēmu izstrādātāji (tehniskā verifikācija) sadarbībā ar biznesa pārstāvjiem (biznesa procesu verifikācija). Tas pat būtu uzskatāms par labo praksi: piegādāt sistēmu kopā ar izpildes laika verificēšanas aprakstiem, kas lietošanas vidē būtu jānokonfigurē, norādot dažādus vides rādītājus, piemēram, servera nosaukumus. Tomēr, mantotu sistēmu verificēšanas aprakstus, visticamāk, sagatavos speciālisti, kuri nav šo sistēmu izstrādātāji. Attiecīgi precīzu verificēšanas aprakstu sagatavošanai būs jāveic nozīmīgs sistēmu izpētes darbs.

Pateicoties veiktajai aprobācijai, var secināt, ka darbā piedāvātā verifikācijas risinājuma implementācija nav pārlieku sarežģīta. Tā balstās uz universālu kontrolieri un daudziem koda apjoma ziņā nelieliem aģentiem. Turklāt aprobācija parāda, ka tādu uzdevumu klasēm, kādas tiek risinātas šajā pētījumā, aģentu radītā virsslodze ir nenozīmīga un verificējamās programmatūras darbs netiek traucēts. Darbā ir analizēti potenciālie kontroliera un aģentu darbības ierobežojumi, tomēr tie ir acīm redzami un, visticamāk, veidojot procesu verifikācijas aprakstus, izstrādātāji šos ierobežojumus ņems vērā:

- ja procesu izpildes ātrums ir salīdzināms ar kontroliera vai aģentu darbības ātrumu, tad verificēt ir iespējams ievērojami mazāku soļu skaitu;
- verificējamo procesa instanču skaitam ir jābūt salāgotam ar kontroliera veiktspēju: pārsniedzot zināmu daudzumu (šis lielums ir atkarīgs no kontroliera implementācijas, darbības vides, komunikāciju ātruma u.c.), kontrolieris tērēs vairāk laika instanču izpildes verificēšanai un, pieaugot instanču skaitam, nespēs apstrādāt paziņojumus par to verificēšanas notikumu konstatēšanu.

Lai arī esošais risinājums jau šobrīd ir reāli pielietojams, tomēr tajā ir vismaz trīs jautājumi, kas būtu apskatāmi turpmākajos pētījumos:

1) Kā jau tika secināts aprobācijas laikā, zināmi papildinājumi būtu noderīgi procesa verificēšanas apraksta valodā: lai arī šobrīd esošie līdzekļi ir pietiekami procesu aprakstīšanā, tomēr efektīvākai verificēšanai būtu noderīga zarošanās nosacījumu definēšana un paralēlu soļu izpilde.
2) Šobrīd autors ir apskatījis un implementējis tikai divus notikumu aģentus. Pilnvērtīgai procesu verificēšanai būtu nepieciešams daudz plašāks aģentu klāsts. Attiecīgi būtu svarīgi apzināties nepieciešamos aģentus,

implementējamos notikumus, aģentu darbības veidu, tomēr šī izstrāde būtu saskaņojama ar plānotajiem verificēšanas pielietojumiem.

3) Verificēšanas mehānisma asinhronā darbība var izraisīt kļūdu paziņojumu sniegšanu par korekti izpildītām procesu instancēm (no angļu valodas - *false negative*). Šādus gadījumus varētu minimizēt, optimizējot kontroliera darbības algoritmu.

Tādējādi lai arī šobrīd piedāvātais verifikācijas mehānisms ir strādājošs un pielietojams, tomēr tas vēl ir papildināms un paver iespējas turpmākiem pētījumiem.

LITERATŪRAS SARAKSTS

[FIRE14] Add a program to the exceptions list. Pieejams tiešsaistē (2014): http://technet.microsoft.com/enus/library/cc775783.aspx.

[OASIS07] Web Services Business Process Execution Language. OASIS, version 2.0 edition. (2007)

[XML08] Extensible markup language (xml) 1.0 (fifth edition). Pieejams tiešsaistē (2008): http://www.w3.org/TR/2008/REC-xml-20081126/.

[OMG11A] Business Process Model and Notation (BPMN). Object Management Group, version 2.0 edition. (2011)

[OMG11B] OMG Unified Modeling Language (OMG UML), Superstructure. Object Management Group, version 2.4.1 edition. (2011)

[COX12] Coxworth, B. BTPS replaces bike tire pressure gauge with sensors and a phone, pieejams tiešsaistē (21.12.2012) http:// http://www.gizmag.com/btps-bike-tire-pressure-system/25542/

[EKS13] Latvijas Bankas elektroniskās klīringa sistēmas (EKS sistēmas) funkcionālais apraksts. Latvijas Banka (2013)

[DYNA14] Dynamic testing. Pieejams tiešsaistē (2014): http://en.wikipedia.org/wiki/Dynamic_testing.

[GAL14] Galīgs automāts. Pieejams tiešsaistē (2014): http://lv.wikipedia.org/wiki/Galīgs_automāts.

[HALT14] Halting problem. Pieejams tiešsaistē (2014): http://en.wikipedia.org/wiki/Halting_problem.

[SPPS14] Microsoft sharepoint. Pieejams tiešsaistē (2014): http://office.microsoft.com/en-us/sharepoint/.

[MONG14] Mongodb. Pieejams tiešsaistē (2014): http://www.mongodb.org.

[MURP14] Murphy’s computers laws. Pieejams tiešsaistē (2014): http://www.murphyslaws.com/murphy/murphy-computer.html.

[NOVE14] Novell vibe. Pieejams tiešsaistē (2014): http://www.novell.com/products/vibe/.

[RUNV14] Runtime verification. Pieejams tiešsaistē (2014): http://en.wikipedia.org/wiki/Runtime_verification.

[STAT14] Static testing. Pieejams tiešsaistē (2014): http://en.wikipedia.org/wiki/Static_testing

[BARB03] Barbier, F. and Belloir, N. (2003). Component behavior prediction and monitoring through built-in test. In Engineering of Computer-Based Systems,

2003. Proceedings. 10th IEEE International Conference and Workshop on the, pages 17–22. IEEE.

[BARE07] Baresi, L., Bianculli, D., Ghezzi, C., Guinea, S., and Spoletini, P. (2007). A timed extension of wscol. In Web Services, 2007. ICWS 2007. IEEE International Conference on, pages 663–670. IEEE.

[BARE05] Baresi, L. and Guinea, S. (2005). Towards dynamic monitoring of ws-bpel processes. In Service-Oriented Computing-ICSOC 2005, pages 269–282. Springer.

[BENS05] Bensalem, S., Bozga, M., Krichen, M., and Tripakis, S. (2005). Testing conformance of real-time applications by automatic generation of observers. Electronic Notes in Theoretical Computer Science, 113:23–43.

[BERG98] Bergstra, J. A. and Klint, P. (1998). The discrete time toolbus – a software coordination architecture. Science of Computer Programming, 31(2):205–229.

[BIAN07] Bianculli, D. and Ghezzi, C. (2007). Dynamo-aop user manual.

[BICE07] Bičevska, Z. and Bičevskis, J. (2007). Smart technologies in software life cycle. In Product-Focused Software Process Improvement, pages 262–272. Springer.

[BICE08] Bičevska, Z. and Bičevskis, J. (2008). Application of smart technologies in software development: automated version updating. Datorzinātne un informācijas tehnoloģijas, page 24.

[BICE10A] Bičevska, Z. (2010). Viedās tehnoloģijas un to efektivitāte. Promocijas darbs, Lavtijas Universitāte.

[BICE10B] Bičevskis, J., Ceriņa-Bērziņa, J., Karnītis, Ğ., Lāce, L., Medvedis, I., and Ņesterovs, S. (2010). Practitioners view on domain specific business process modeling. In Databases and information systems VI, selected papers from the ninth International Baltic Conference, DB&IS, pages 169–182.

[BICE15A] Z. Bičevska, J. Bičevskis, I. Odītis. Smart technologies for improved software maintenance. In: Computer Science and Information Systems (FedCSIS), 2015 Federated Conference on. IEEE, 2015. p. 1533-1538.

[BICE15B] J. Bičevskis, Z. Bičevska., K. Rauhvargers, E. Diebelis, I. Odītis, J. Borzovs A Practitioner’s Approach to Achieve Autonomic Computing Goals. Baltic Journal of Modern Computing, Vol 3. (2015) No. 4: 273-293. Ieguldījums 20%. Publikācija tiks ir indeksēta datu bāzē Web Of Science.

[BICE16] J. Bičevskis, Z. Bičevska, I. Odītis. Self-management of Information Systems, Baltic DB&IS 2016.

[BIND00] Binder, R. V. (2000). Testing object-oriented systems: models, patterns, and tools. Addison-Wesley Professional.

[CALI11] Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R., and Tamburrelli, G. (2011). Dynamic qos management and optimization in service-based systems. Software Engineering, IEEE Transactions on, 37(3):387–409.

[CHAR05] Charette, R. N. (2005). Why software fails [software failure]. Spectrum, IEEE, 42(9):42–49.

[CIAN96] Ciancarini, P. and Hankin, C. (1996). Coordination Languages and Models: First International Conference, COORDINATION'96, Cesena, Italy, April 15-17, 1996. Proceedings., volume 1. Springer.

[DIEB12] Diebelis, E. (2012). Programmatūras paštestēšana. PhD thesis, Latvijas Universitāte.

[DIEB09] Diebelis, E., Takeris, V., and Bičevskis, J. (2009). Self-testing – new approach to software quality assurance. In Proceedings of the 13th East-European Conference on Advances in Databases and Information Systems (ADBIS 2009), pages 62–77.

[DUCA01] Ducassé, M. and Jahier, E. (2001). Efficient automated trace analysis: Examples with morphine. Electronic Notes in Theoretical Computer Science, 55(2):118–133.

[FAIR78] Fairley, R. E. (1978). Tutorial: Static analysis and dynamic testing of computer software. Computer, 11(4):14–23.

[GHEZ07] Ghezzi, C. and Guinea, S. (2007). Run-time monitoring in serviceoriented architectures. In Test and analysis of web services, pages 237–264. Springer.

[HALL08] Halle, S. and Villemaire, R. (2008). Runtime monitoring of message-based workflows with data. In Enterprise Distributed Object Computing Conference, 2008. EDOC'08. 12th International IEEE, pages 63–72. IEEE.

[HORN01] Horn, P. (2001). Autonomic computing: IBM's perspective on the state of information technology.

[JENS09] Jensen, K. and Kristensen, L. M. (2009). Coloured Petri nets: modelling and validation of concurrent systems. Springer.

[KANE06] Kaner, C. (2006). Exploratory testing. In Presinted at Quality Assurance Institute Worldwide Annual Software Testing Conference, Orlando, FL. Florida Institute of Technology.

[KEPH11] Kephart, J. O. (2011). Autonomic computing: the first decade. In ICAC, pages 1–2.

[KEPH03] Kephart, J. O. and Chess, D. M. (2003). The vision of autonomic computing. Computer, 36(1):41–50.

[KORT01] Kortenkamp, D., Milam, T., Simmons, R., and Fernandez, J. L. (2001). Collecting and analyzing data from distributed control programs. Electronic Notes in Theoretical Computer Science, 55(2):236–254.

[LAPI14] Lapiņš, K. (2014). Automatizētas informācijas sistēmu darbības uzraudzības implementācija. Master's thesis, Latvijas Universitāte, Datorikas fakultāte.

[LEAV06] Leavens, G. T., Baker, A. L., and Ruby, C. (2006). Preliminary design of jml: A behavioral interface specification language for java. ACM SIGSOFT Software Engineering Notes, 31(3):1–38.

[LEUC09] Leucker, M. and Schallhart, C. (2009). A brief account of runtime verification. The Journal of Logic and Algebraic Programming, 78(5):293 – 303. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

[MERN05] Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. ACM computing surveys (CSUR), 37(4):316–344.

[NAVA13] Navabpour, S., Bonakdarpour, B., and Fischmeister, S. (2013). Pathaware time-triggered runtime verification. In Runtime Verification, pages 199–213. Springer.

[ODIT10] Odītis, I. and Bičevskis, J. (2010). The concept of automated process control. Computer Science and Information Technologies, Scientific Papers, University of Latvia, 756:193–203.

[ODIT14] Odītis, I. and Bičevskis, J. (2014). Runtime verification of business processes. In Haav, H.-M., Kalja, A., and Robal, T., editors, Databases and Information Systems, Proceedings of the 11th International Baltic Conference, Baltic DB&IS 2014, pages 363–370. Tallin University of Technology Press.

[ODIT15] I. Odītis, J. Bičevskis. Asynchronous Runtime Verification of Business Processes. In: Computational Intelligence, Communication Systems and Networks (CICSyN), 2015 7th International Conference on. IEEE, 2015. p. 103-108.

[ODTI16] I. Odītis, J. Bičevskis. Asynchronous Runtime Verification of Business Processes: Proof of Concept. International Journal of Simulation Systems, Science & Technology (2016).

[PHIL13] Philippe Lalanda, Julie A. McCann, A. D. (2013). Autonomic Computing. Principles, Design and Implementation. Springer.

[PROJ02] Project, R. (2002). The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology.

[RAUH10] Rauhvargers, K. (2010). Programmatūras izpildes vides testēˇsana. PhD thesis, Latvijas Universitāte.

[RAUH09] Rauhvargers, K. and Bičevskis, J. (2009). Environment testing enabled software–a step towards execution context awareness. In Databases and Information Systems, Selected Papers from the 8th International Baltic Conference, volume 187, pages 169–179.

[SCHE05] Scheer, A.-W., Thomas, O., and Adam, O. (2005). Process modeling using event-driven process chains. Process-Aware Information Systems, pages 119–146.

[STER05] Sterritt, R. and Hinchey, M. G. (2005). Why computer-based systems should be autonomic.

[STOR04] Störrle, H. and Hausmann, J. (2004). semantics of uml 2.0 activities. In Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing.

[TSAI06] Tsai, A., Wang, J., Tepfenhart, W., and Rosca, D. (2006). Epc workflow model to wifa model conversion. In Systems, Man and Cybernetics, 2006. SMC'06. IEEE International Conference on, volume 4, pages 2758–2763. IEEE.

[DONG05] van Dongen, B. F., de Medeiros, A. K. A., Verbeek, H., Weijters, A., and Van Der Aalst, W. M. (2005). The prom framework: A new era in process mining tool support. In Applications and Theory of Petri Nets 2005, pages 444–454. Springer.

[WU13] Wu, C. W. W. (2013). Methods for Reducing Monitoring Overhead in Runtime Verification. Promocijas darbs, University of Waterloo.

[ZHAO09] Zhao, Y. and Rammig, F. (2009). Model-based runtime verification framework. Electronic Notes in Theoretical Computer Science, 253(1):179–193.

PIELIKUMS

Procesa verificēšanas apraksta piemērs. Pievienotais verificēšānas apraksts atbilst aprobācijā apskatītajam procesam (skat. 4. nodaļu)

```
<?xml version="1.0" encoding="utf-8" ?>
<process_description>
  <header>
    <id>BANK_FILE_PROCESS</id>
    <name>verificaton of bank file processing</name>
    <comment></comment>
    <initiation>start-event</initiation>
    <instances>multiple</instances>
    <parameters>
      <parameter>
        <name>IncommingFilePath</name>
        <default>\\server\share\path\in</default>
      </parameter>
      <parameter>
        <name>OutgoingFilePath</name>
        <default>\\server\share\path\out</default>
      </parameter>
      <parameter>
        <name>DBServer</name>
        <default>db_server_name</default>
      </parameter>
      <parameter>
        <name>DBDatabase</name>
        <default>db_name</default>
      </parameter>
      <parameter>
        <name>BankISN</name>
        <default>12</default>
      </parameter>
    </parameters>
    <variables>
      <variable>
        <name>F_ISN</name>
      </variable>
      <variable>
        <name>F_EncrName</name>
      </variable>
      <variable>
        <name>F_EncrFullName</name>
      </variable>
      <variable>
        <name>F_OUT_EncrName</name>
      </variable>
    </variables>
    <message_receivers>
      <receiver id="default-email">
        <default>true</default>
        <address>email@domain.org</address>
        <type>email</type>
      </receiver>
    </message_receivers>
  </header>
  <actions>
    <action id="file-received" type="start">
      <time>
```

```
      <no_time/>
    </time>
    <name>File received</name>
    <main_event>
      <agent>FileSystem</agent>
      <event_type>FileCreated</event_type>
      <address>EKS-file-server</address>
      <parameters>
        <parameter>
          <name>Directory</name>
          <assign>IncommingFilePath</assign>
        </parameter>
        <parameter>
          <name>Pattern</name>
          <value>*.*</value>
        </parameter>
        <parameter>
          <name>Subdirs</name>
          <value>false</value>
        </parameter>
      </parameters>
      <results>
        <result>
          <name>FileName</name>
          <assign>F_EncrName</assign>
        </result>
        <result>
          <name>FileFullName</name>
          <assign>F_EncrFullName</assign>
        </result>
      </results>
    </main_event>
  </action>
  <action id="file-registered" type="normal">
    <time>
      <distances>
        <distance>
          <ref_event_id>file-received</ref_event_id>
          <period>00:01:00</period>
        </distance>
      </distances>
    </time>
    <name>File registered</name>
    <main_event>
      <agent>MSSQLServer</agent>
      <event_type>ProcedureCall</event_type>
      <address>EKS-database-server</address>
      <parameters>
        <parameter>
          <name>Server</name>
          <assign>DBServer</assign>
        </parameter>
        <parameter>
          <name>Database</name>
          <assign>DBDatabase</assign>
        </parameter>
        <parameter>
          <name>Delay</name>
          <value>5</value>
        </parameter>
        <parameter>
          <name>Procedure</name>
          <assign>ODS_EKS_ATRAST_FAILU</assign>
```

```
      </parameter>
      <parameter>
        <name>@F_ENCNAME</name>
        <assign>F_EncrName</assign>
      </parameter>
      <parameter>
        <name>@F_BANK_ISN</name>
        <assign>BankISN</assign>
      </parameter>
    </parameters>
    <results>
      <result>
        <name>F_ISN</name>
        <assign>F_ISN</assign>
      </result>
    </results>
  </main_event>
  <sub_events>
    <sub_event>
      <agent>FileSystem</agent>
      <event_type>FileExists</event_type>
      <address>EKS-file-server</address>
      <parameters>
        <parameter>
          <name>FileFullName</name>
          <assign>F_EncrFullName</assign>
        </parameter>
        <parameter>
          <name>ExistFlag</name>
          <value>false</value>
        </parameter>
      </parameters>
    </sub_event>
  </sub_events>
</action>
<action id="response-delivered" type="finish">
  <time>
    <distances>
      <distance>
        <ref_event_id>file-registered</ref_event_id>
        <period>00:02:00</period>
      </distance>
    </distances>
  </time>
  <name>Response delivered</name>
  <main_event>
    <agent>MSSQLServer</agent>
    <event_type>ProcedureCall</event_type>
    <address>EKS-database-server</address>
    <parameters>
      <parameter>
        <name>Server</name>
        <assign>DBServer</assign>
      </parameter>
      <parameter>
        <name>Database</name>
        <assign>DBDatabase</assign>
      </parameter>
      <parameter>
        <name>Delay</name>
        <value>5</value>
      </parameter>
      <parameter>
```

```
            <name>Procedure</name>
            <assign>ODS_EKS_ATRAST_ATBILDI</assign>
          </parameter>
          <parameter>
            <name>@F_ISN</name>
            <assign>F_ISN</assign>
          </parameter>
        </parameters>
        <results>
          <result>
            <name>F_ENCNAME</name>
            <assign>F_OUT_EncrName</assign>
          </result>
        </results>
      </main_event>
      <sub_events>
        <sub_event>
          <agent>FileSystem</agent>
          <event_type>FileFolderExists</event_type>
          <address>EKS-file-server</address>
          <parameters>
            <parameter>
              <name>Dircetory</name>
              <assign>OutgoingFilePath</assign>
            </parameter>
            <parameter>
              <name>FileName</name>
              <assign>F_OUT_EncrName</assign>
            </parameter>
            <parameter>
              <name>ExistFlag</name>
              <value>true</value>
            </parameter>
          </parameters>
        </sub_event>
      </sub_events>
    </action>
  </actions>
  <links>
    <link from="file-received" to="file-registered"/>
    <link from="file-registered" to="response-delivered"/>
  </links>
</process_description>
```