RĪGAS TEHNISKĀ UNIVERSITĀTE
Datorzinātnes un informācijas tehnoloģijas fakultāte
Lietišķo datorsistēmu institūts

Aleksandrs SUHORUKOVS
Doktora studiju programmas „Datorsistēmas“ doktorants

DATORSISTĒMU TESTĒŠANAS AUTOMATIZĀCIJAS METODES, RĪKI UN EFEKTIVITĀTE

Promocijas darba kopsavilkums

Zinātniskais vadītājs
Dr. sc. ing., profesore
L. ZAICEVA

Rīga 2011

UDK 004.415.53 (043.2)
Su 264 d

Suhorukovs A. Datorsistēmu testēšanas automatizācijas metodes, rīki un efektivitāte. Promocijas darba kopsavilkums. -R.:RTU, 2011. -37 lpp.

Iespiests saskaņā ar DITF LDI institūta 2011.gada 1. jūlija lēmumu, protokols Nr. 72.

Šis darbs izstrādāts ar Eiropas Sociālā fonda atbalstu projektā „Atbalsts RTU doktora studiju īstenošanai“.

ISBN 978-9934-10-205-9

PROMOCI JAS DARBS
IZVIRZĪTS INŽENIERZINĀTŅU (datorsistēmu)
DOKTORA GRĀDA IEGŪŠANAI RĪGAS TEHNISKAJĀ UNIVERSITĀTĒ

Promocijas darbs inženierzinātņu (datorsistēmu) doktora grāda iegūšanai tiek publiski aizstāvēts 2011. g. 5. decembrī Rīgas Tehniskās universitātes Datorzinātnes un informācijas tehnoloģijas fakultātē, Meža ielā 1/3, 202. auditorijā.

OFICIĀLIE RECENZENTI

Profesors Dr.sc.ing. Uldis Sukovskis
Rīgas Tehniskā universitāte

Dr.sc.comp. Mārtiņš Gills
AS Norvik Banka

Profesors Ph.D. Radi Petrov Romansky
Technical University – Sofia, Bulgārija

APSTIPRINĀJUMS

Apstiprinu, ka esmu izstrādājis doto promocijas darbu, kas iesniegts izskatīšanai Rīgas Tehniskajā universitātē inženierzinātņu doktora grāda iegūšanai. Promocijas darbs nav iesniegts nevienā citā universitātē zinātniskā grāda iegūšanai.

Aleksandrs Suhorukovs …….(Paraksts)

Datums: ……

Promocijas darbs ir uzrakstīts latviešu valodā, satur ievadu, 6 nodaļas, nobeigumu, literatūras sarakstu, 30 zīmējumus un ilustrācijas, 8 tabulas, kopā 130 lappuses. Literatūras sarakstā ir 130 nosaukumi.

ANOTĀCIJA

Promocijas darbs ir veltīts automatizētās testēšanas jautājumiem: testēšanas procesa modelēšanai, automatizācijas rīku lietošanai, piemēroto testu kopuma izveidei testēšanai atvēlētajā laikā un to efektivitātes novērtēšanai.

Darbā ir izskatīti aktuālie automatizētās testēšanas attīstības virzieni, tiek klasificēti populāri testu automatizācijas rīki. Aprakstīts izstrādātais vienots automatizētās testēšanas modelis, kas augstā līmenī apraksta automatizēto testēšanu, atspoguļojot potenciāli automatizējamas programmatūras testēšanas aktivitātes. Aplūkots izstrādātais matemātiskais automatizētās testēšanas efektivitātes novērtēšanas modelis, kas ļauj panākt racionālu ierobežota testēšanas laika izmantošanu. Aprakstītas izstrādātās automatizētas testu komplektu ģenerēšanas un adaptīvas veiktspējas testēšanas metodes, uz kuru pamata tiek izstrādāti rīki, kas ir aprobēti reālos projektos.

SATURS

1. VISPĀRĪGS DARBA RAKSTUROJUMS

1.1. Pētījuma motivācija

Promocijas darbs veltīts datorsistēmu testēšanas automatizācijas metodēm un rīkiem, kā arī to efektivitātes novērtēšanai.

Mūsdienās arvien aktuālāks kļūst jautājums par programmatūras testu automatizāciju. Attīstoties tehnoloģijām, sistēmas kļūst lielākas un sarežģītākas, bet izstrādes cikli — ātrāki. Līdz ar to pieaug arī testu skaits un nepieciešamība pēc ātras testu izpildes. Šādos apstākļos, un ņemot vērā, ka programmatūras testēšana ir laikietilpīgs process, testēšanas efektivitāti var palielināt automatizētie testi, kuru izpilde prasa mazāk cilvēka laika, salīdzinot ar manuālo testu izpildi.

Automatizācijas lietderību nosaka nepieciešamība pēc testu daudzkārtējas atkārtošanas, piemēram, pārbaudot vairākus ieejas datus, pārbaudot sistēmu pēc veiktām izmaiņām utt. Testēšanas efektivitāti var ievērojami palielināt, ja šāda veida atkārtojošās darbības uzticēt veikt rīkam.

Automatizētais tests nekad nenogurst, nekļūst mazāk uzmanīgs un vienmēr precīzi veic paredzētu uzdevumu, turpretim manuālajai testēšanai piemīt vairāki citi trūkumi. Tā kā gan manuālajai, gan automatizētajai testēšanai ir gan priekšrocības, gan trūkumi, jautājums par to, kura no metodēm ir jālieto konkrētajā gadījumā, nav triviāls. Divas galējības — testu automatizācijas iespēju ignorēšana un centieni automatizēt visus testus — nav racionālas gan no darbietilpības, gan no testēšanas kvalitātes viedokļa [19]. Piemērota līdzsvara noteikšana ir problēma, kurai pagaidām nav viennozīmīga risinājuma.

Testēšanas automatizācijas tehnoloģijas mūsdienās ir labi attīstītas vairākos virzienos un piemērojamas dažādu, pat specifisku, sistēmu testēšanai [30]. Ir pieejami daudzi rīki funkcionālās un veiktspējas testēšanas automatizācijas nodrošināšanai. Vienībtestēšanas risinājumi ir pieejami gandrīz vai katrai pastāvošai programmēšanas valodai [12]. Rīku dažādība nodrošina plašāku automatizācijai piemērotu testu klāstu, bet tajā pašā laikā apgrūtina piemērotākā rīka izvēles uzdevumu.

Daudzas organizācijas gan Latvijā, gan pasaulē ir mēģinājušas, mēģina vai jau lieto testēšanas automatizācijas iespējas. Taču joprojām pastāv daudzi atšķirīgi viedokļi par to, kā jāveic testu automatizācija, un kādai ir jābūt automatizācijas vietai testēšanā un vispār

programmatūras izstrādē [21, 25]. Viedokļu dažādības dēļ ir grūti novērtēt kādas metodes un kādā apjomā jālieto konkrēta projekta vajadzībām. Tas var novest pie neefektīvas testēšanas plānošanas, kad vai nu no automatizācijas atsakās vispār, vai arī automatizē tādus testus, kuru manuālā izpilde būtu izdevīgāka.

Kad testēšanas automatizācija jau tiek pielietota, vai arī kad to tikai plāno ieviest, ir būtiski novērtēt vai prognozēt tās izmantošanas efektivitāti. Pagaidām nepastāv automatizētās testēšanas efektivitātes novērtēšanas metode, kas ļautu salīdzināt izvēlētas automatizācijas stratēģijas efektivitāti ar līdzvērtīgās manuālās testēšanas stratēģiju, vai ar citām alternatīvām automatizācijas stratēģijām.

Augstāk minētās problēmas nosaka šīs tēmas aktualitāti un nepieciešamību pēc padziļinātiem pētījumiem datorsistēmu testēšanas automatizācijas metožu un efektivitātes jomā.

1.2. Darba mērķis un uzdevumi

Promocijas darba mērķis ir balstoties uz testēšanas metožu analīzi un testu automatizācijas rīku pētīšanu, izstrādāt testēšanas automatizācijas metodes, kas ļautu panākt testēšanai veltītā laika ekonomiju un/vai labāku testēšanas kvalitāti, un automatizēto testu efektivitātes novērtēšanas modeli, kā arī aprobēt izstrādātās metodes un rīkus reālos projektos.

Lai sasniegtu izvirzīto mērķi ir jārisina šādi uzdevumi:

1) izpētīt dažādu testēšanas aktivitāšu automatizācijas esošas metodes un klasificēt mūsdienās pieejamus testu automatizācijas rīkus;
2) izstrādāt automatizētās testēšanas modeli, kas norādītu potenciāli automatizējamos procesus programmatūras testēšanā;
3) izstrādāt automatizēto testu efektivitātes novērtēšanas modeli;
4) izstrādāt testēšanas automatizācijas risinājumus ar augstākiem efektivitātes rādītājiem salīdzinājumā ar esošiem risinājumiem;
5) aprobēt izstrādātos risinājumus reālos projektos.

Formulētie uzdevumi atspoguļo darba struktūru. Katram uzdevumam tiek veltīta atsevišķa nodaļa, bet izstrādātajiem risinājumiem — divas nodaļas. Kopējie secinājumi tiek ievietoti nobeigumā.

1.3. Pētījuma metodika, zinātniskā novitāte, darba praktiskā nozīme

Pētījuma nozare ir datorsistēmu testēšana, izmantojot programmatūras rīkus, kuri automatizē testēšanas aktivitāšu izpildi.

Pētījuma priekšmets ir automatizētās datorsistēmu testēšanas metodes, efektivitāte un rīki, kas to nodrošina.

Pētījuma metodika ir kopu teorija, grafu teorija, algoritmu projektēšana un analīze.

Darba rezultāta jaunieguvumi ir šādi:

- izstrādāts automatizēto testu efektivitātes novērtēšanas modelis;
- izstrādātas automātiskās testu komplektu ģenerēšanas metodes, balstoties uz vienkāršo un salikto stāvokļu modeļiem;
- izstrādāts veiktspējas testēšanas adaptīvā slodzes plānotāja modelis un piedāvāts tā realizācijas algoritms.

Darba rezultātu galvenā praktiskā vērtība ir saistīta ar izstrādātajām testu komplektu ģenerēšanas metodēm un veiktspējas testēšanas adaptīvā slodzes plānotāja modeli, kas pie noteiktiem nosacījumiem izrādās efektīvāki par citām metodēm. Šīs metodes tika realizētas autora izstrādātajos rīkos, kuri tika sekmīgi pielietoti 14 reālajos testēšanas projektos dažādos uzņēmumos.

1.4. Promocijas darba struktūra

Darbs sastāv no ievada, sešām nodaļām un nobeiguma.

Pirmajā daļā tiek sniegts ieskats automatizētās testēšanas vēsturē, tiek apskatīti mūsdienās aktuālie testu ģenerēšanas metožu pētījumu virzieni, kā arī tiek piedāvāta testu automatizācijas rīku klasifikācija, pēc kuras tiek klasificēti 32 populārie rīki.

Otrajā daļa tiek sniegts pārskats par testu automatizācijas rīku un testu programmatūras izstrādes procesiem, kā arī tiek aprakstīts izstrādātais vienots automatizētās testēšanas modelis.

Trešajā daļā tiek aprakstīts izstrādātais automatizētās testēšanas efektivitātes novērtēšanas modelis.

Ceturtajā daļā tiek aprakstīti izstrādātie automatizētās testu komplektu ģenerēšanas risinājumi.

Piektajā daļā tiek aprakstīts izstrādātais veiktspējas testēšanas rīks un izstrādātā adaptīvā veiktspējas testēšanas metode.

Sestajā daļā tiek īsi raksturota ceturtajā un piektajā daļā aprakstīto risinājumu ieviešanas un izmantošanas pieredze reālos projektos.

Darbs satur 130 lpp. teksta, 30 attēlus, 8 tabulas un 130 bibliogrāfiskos avotus.

2. PROMOCIJAS DARBA SATURS

2.1. Automatizācija programmatūras testēšanā

Balstoties uz vairākiem avotiem [1, 4, 9, 12, 18, 24] var izdalīt trīs būtiskus posmus, kuros tika iesākta mūsdienās aktuālo automatizētās testēšanas rīku attīstībā:

- 1970. gados — testu ģenerēšanas, simboliskās izpildes rīki;
- 1980. gados — melnās kastes testu automatizācijas, ierakstīšanas/atspēlēšanas rīki;
- 1990. gadu beigās — mūsdienu vienībtestēšanas ietvari.

Pirmajā promocijas darba nodaļā tiek klasificētas testu ģenerēšanas metodes, kā arī tiek izstrādāta testu automatizācijas rīku klasifikācija, pēc kuras tiek klasificēti vairāki mūsdienās pieejamie rīki.

Testpiemēru ģenerēšanas uzdevumu var raksturot šādi. Sistēmu var uzskatīt par funkciju, kas ievaddatus transformē izvaddatos. Par ievaddatiem varētu uzskatīt failus, ievade no tastatūras, darbības ar peli utt. Izvade varētu būt ģenerētie faili, attēlotas vērtības vai grafika utt. Testpiemēru ģenerēšanas uzdevums ir dotajai programmai atrast tādus ievaddatus vai to komplektus, kas atbilst noteiktiem kritērijiem.

Testpiemēru ģenerēšanas metodes var klasificēt dažādi. Darbā tiek aplūkotas četras klasifikācijas atkarībā no:

1) pirmkoda nozīmības;
2) programmas darbināšanas veida;
3) mērķa kritērijiem;
4) modeļu veidiem.

Pēc pirmkoda nozīmības testpiemēru ģenerēšanas metodes var iedalīt:

1) baltās kastes metodēs — testi tiek izvēlēti, balstoties uz informāciju par programmas implementāciju: ir zināma programmas iekšējā uzbūve un ir pieejams tās pirmkods;
2) melnās kastes metodēs — testus izvēlas, balstoties uz programmas ārējo uzvedību, ko nosaka prasību specifikācija vai arī pati izpildāmā programma;
3) pelēkās kastes metodēs — kombinē pirmās divas metodes, parasti tas nozīmē, ka pirmkodu izmanto testpiemēru projektēšanai, taču testu mērķus nosaka no programmas funkcionalitātes viedokļa.

Pēc darbināšanas veida testpiemēru ģenerēšanas metodes var iedalīt:

1) statiskajā — testi tiek ģenerēti, nedarbinot programmu, izmantojot pirmkoda statiskās analīzes un simboliskas izpildes metodes [10, 15];
2) dinamiskajā — programma tiek darbināta un testpiemēri tiek ģenerēti, balstoties uz darbināšanas laikā iegūtiem datiem [16];
3) hibrīdajā — apvieno abu iepriekšējo pieeju iespējas [11].

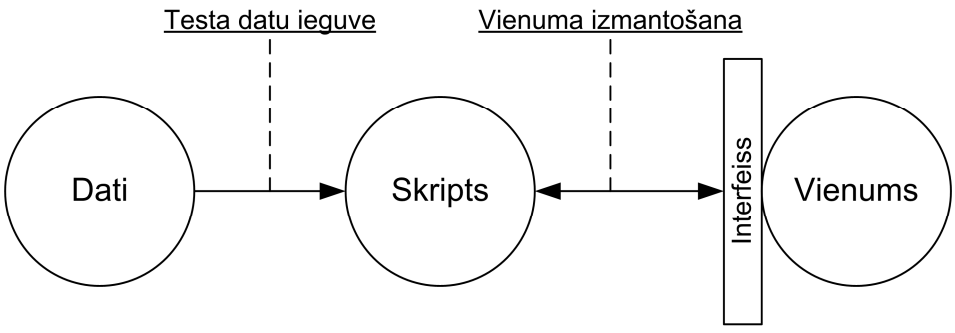
Pēc ceļu veidiem testpiemēru ģenerēšanas metodes var iedalīt [6]:

1) uz gadījuma ceļiem balstītajās — testi tiek izvēlēti, izmantojot gadījuma datus vai gadījuma izpildes ceļus;
2) uz mērķiem orientētajās — testpiemēri tiek ģenerēti, ievērojot ierobežojumu, ka programmai ir jāiziet ceļš ar noteiktām īpašībām;
3) uz ceļiem orientētajās — testpiemēri tiek ģenerēti tā, lai tos darbinot programma izietu konkrēti uzdotus ceļus.

Pēc modeļu veidiem testpiemēru ģenerēšanas metodes iedalās atbilstoši tam, kāda veida programmas modeļi tiek ņemti par pamatu [2]. Pastāv metodes, kas ļauj ģenerēt testpiemērus no kontraktu veida specifikācijām [17, 20], abstrakto datu tipu specifikācijām [8, 22], iezīmēto pāreju sistēmām [13, 14] un vairākiem citiem modeļiem.

Ģenerēšanas algoritmi lielā mērā ir atkarīgi no modelēšanas veida, ar ko ir izskaidrojams tas, ka testpiemēru automātiska ģenerēšana joprojām netiek plaši lietota programmatūras izstrādes industrijā — katrs atsevišķs modelis vai algoritms piedāvā tikai vienpusīgu, visai ierobežotu skatu uz testējamo programmu, bet plašākam aptvērumam ir jāizmanto vairāki modeļi un ģenerēšanas algoritmi, kas prasa pārāk daudz darba modelēšanai un daudz zināšanu algoritmu lietošanai.

Testu izpildes automatizācijas (jeb vienkārši testu automatizācijas) rīku klasifikācijai, ievērojot automatizētā testa izstrādes un darbināšanas kontekstu, tika izstrādāts dinamiskā automatizētā testa modelis, kas ir atspoguļots 2.1. attēlā.



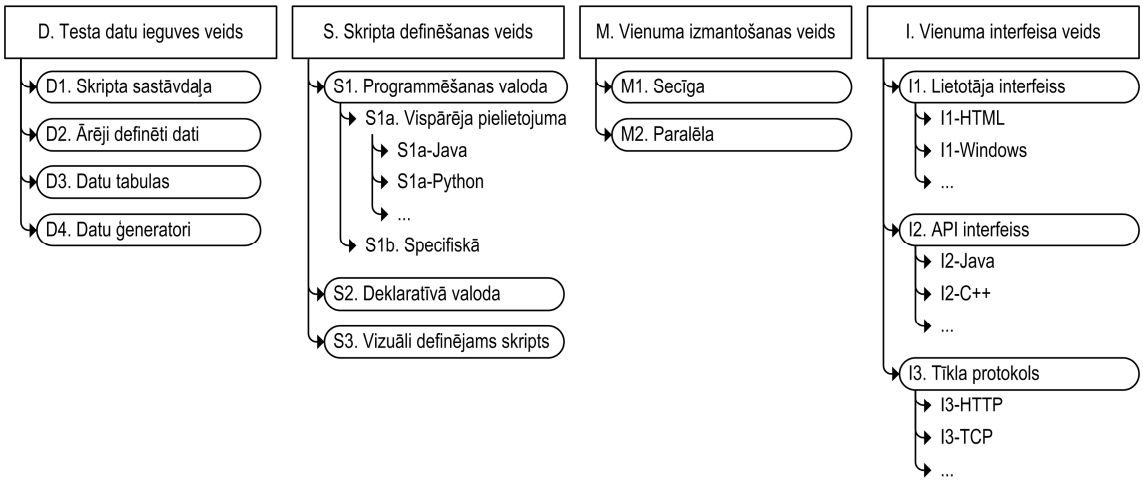
2.1. att. Automatizētā testa modelis

Uzbūvētajā modelī ir izdalāmi četri aspekti, kas tika izvēlēti par klasifikācijas kritērijiem (citi kritēriju piemēri, ko varētu atvasināt no šī modeļa, piemēram, tādi kā datu definēšanas veids vai vienuma veids, nav būtiski testu automatizācijas rīku piemērotības analīzei):

1. Testa datu ieguves veids. Šis kritērijs nosaka, kādā veidā skripts iegūst datus, vai dati ir atdalīti no skripta, vai ir skripta sastāvdaļa. Šis kritērijs tiks apzīmēts ar burtu D.
2. Skripta definēšanas veids. Nosaka, kādā veidā skripts tiek izstrādāts, kāda ir tā struktūra un kā tas tiek interpretēts. Šis kritērijs tiks apzīmēts ar burtu S.
3. Vienuma izmantošanas veids. Nosaka veidu, kādā izmanto vienumu, vai darbības tiek veiktas secīgi vai paralēli. Šis kritērijs tiks apzīmēts ar burtu M.
4. Vienuma interfeisa veids. Nosaka, ar kādu sistēmas līmeni sadarbojas skripts. Šis kritērijs tiks apzīmēts ar burtu I.

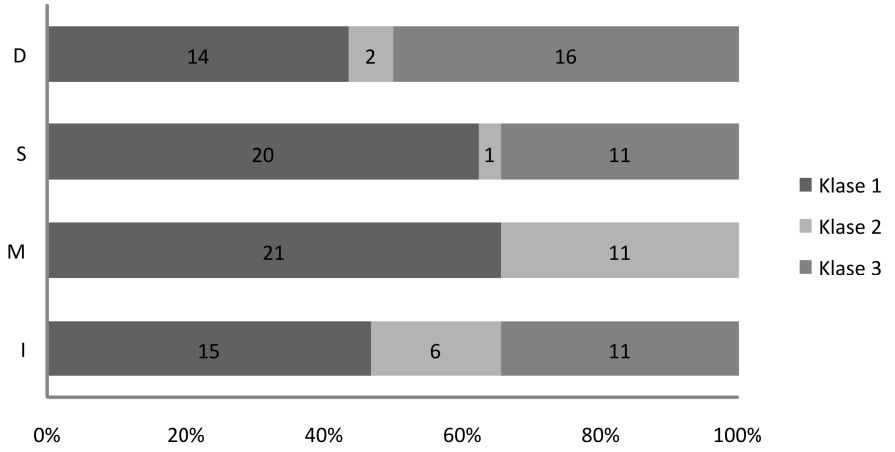
Katrs kritērijs ietver vairākus variantus (klases), kas tiks apzīmēti ar atbilstoša kritērija apzīmējuma burtu un klases numuru, piemēram, piemēram, D3, kur D ir kritērija apzīmējums, un 3 — apzīmē trešo klasi pēc kritērija D.

Visus kritērijus ar iespējamām vērtībām apkopo 2.2. attēls. Promocijas darbā visi kritēriji un to klases ir detalizēti aprakstīti un pamatoti.



2.2. att. Rīku klasifikācijas kritēriji

Uz piedāvātās klasifikācijas pamata tika analizēti un novērtēti 32 rīki, veidojot to rīku iedalījumu klasēs pēc katra kritērija. Promocijas darbā ir atrodama pilna šo rīku klasifikācijas tabula, bet kopējā statistika rīku sadalījumam klasēs ir atspoguļota 2.3. attēlā.



2.3. att. Rīku sadalījums klasēs

Vismazākais rīku skaits ir novērojams klasēs D4 (datu ģeneratori) — neviens rīks, D2 (dati ir atdalīti no skripta) — divi rīki un S2 (deklaratīvajā valodā veidojams skripts) — viens rīks. D2 un S2 klašu rīku neliels skaits ir izskaidrojams ar to, ka šīs klases varētu uzskatīt par pārejas formām. Datu atdalīšana no skripta ir vienkārši pilnveidojama un pārvēršama par datu tabulu funkcionalitāti (D3 klase), kas paver daudz plašākas testu automatizācijas iespējas. Tas pats attiecās arī uz klasi S2 — ja skripts ir definējams deklaratīvajā valodā, ir relatīvi viegli izveidot vizuālo interfeisu, kas atvieglos skripta izveidi, un rīks pārtop klasē S3.

No analizētajiem rīkiem pie klases D4 (datu ģenerators) nav pieskaitāms neviens rīks, kas liecina, ka šāda veida rīki ir relatīvi reti sastopami. Testa datu ģenerēšanas rīki pārsvarā pastāv kā atsevišķa rīku klase, kas nenodarbojas ar pašu testu automatizāciju.

Piedāvāto klasifikāciju var izmantot piemērotāko testu automatizācijas rīku klašu izvēlei atkarībā no uzdevuma īpašībām.

2.2. Automatizēto testu izstrādes procesi

Pastāv daudzi dažādi testu automatizācijas rīki, piemēroti dažādu testēšanas uzdevumu veikšanai, un katrs ar savām niansēm. Neskatoties uz šo dažādību, joprojām bieži var būt situācijas, kurās esošie rīki nav piemēroti, un ir jāizstrādā jauns, uzdevumam specifisks, risinājums. Promocijas darba otrajā nodaļā tiek analizētas testu automatizācijas rīku projektējuma īpašības.

No testu automatizācijas rīku projektēšanas viedokļa ir lietderīgi apskatīt testu automatizāciju trīs līmeņos.

- Lietotāja saskarnes līmenis (*UI*) (atbilst klasei S1). Tiek automatizētas darbības, kuras veic lietotājs, lietojot kādu vizuālu — grafisko jeb GUI (*Graphical User Interface*) vai konsoles jeb (*Console User Interface*) saskarni.
- Funkciju izsaukumu līmenis (atbilst klasei S2). Tiek automatizētas darbības, kuras veic citi moduļi (vai sistēmas) ar testējamo moduli (vai sistēmu), izpildot funkciju izsaukumus vai nu tieši, vai caur kādu augstāka līmeņa saskarni, piemēram, COM vai CORBA.
- Komunikācijas līmenis (atbilst klasei S3). Tiek automatizētas darbības, ko veic citi moduļi ar testējamo moduli, lietojot kādu komunikācijas protokolu, piemēram, tīklā (Ethernet, TCP, HTTP, utt.) vai tiešā savienojumā (COM, LPT, USB, utt.).

Pirmajā darba nodaļā (2.1. sadaļa) aplūkotais klasifikācijas kritērijs S visvairāk ietekmē testu automatizācijas rīka projektējumu (pārējie kritēriji ietekmē projektējuma detaļas). Katram no augstākminētajiem līmeņiem promocijas darba otrajā daļā ir aplūkoti būtiskākie šāda veida rīku projektēšanas aspekti.

Kad rīks ir izvēlēts, vai arī izstrādāts no jauna, var sākties automatizēto testu izstrāde, ko rīks spēj darbināt. Ja testu ir daudz, tad izstrādājamos automatizēto testu komplektus mēdz saukt par testu programmatūru (*testware*), tādējādi pasvītrojot to, ka automatizētie testi ir programmatūras paveids ar līdzīgām prasībām pēc modularitātes, elastības, uzturamības, utt. No otras puses, testu programmatūra tomēr ir specifisks programmatūras paveids, līdz ar to tai piemīt specifiskas īpašības.

Testu programmatūra saskaņā ar [7] sastāv no:

- testu kopas;
- skriptu kopas;
- datu kopas;
- utilītu kopas;
- testu komplektiem;
- testu programmatūras bibliotēkas;
- testu rezultātiem.

Šo elementu fiziska izvietošana ir atkarīga no pielietota testu automatizācijas rīka, kā arī no testu projektētāja izvēles. Testu efektivitāte lielā mērā būs atkarīga no tā, kādi dati ir atdalīti no skriptiem, bet kādi paliek konstantas skripta sastāvdaļas, kāda funkcionalitāte ir iznesta utilītās, bet kāda paliek pašos skriptos. Testu kopu kombinēšana testu komplektos ir

atkarīga no testēšanas mērķiem — kādi sistēmas aspekti būs jātestē un cik ātrai ir jābūt testu izpildei.

Atbilstoši [5] automatizēto testu programmatūras projektēšanas procesu var iedalīt divās fāzēs:

1) testēšanas prasību analīze, kuras rezultātā tiek izstrādāta testēšanas prasību matrica un novērtētas iespējamas testēšanas tehnikas (soļi: mērķu analīze, verifikācijas metožu izvēle, testēšanas prasību analīze, testēšanas prasību matricas izvēle, testēšanas tehniku kārtēšana);
2) testu programmatūras projektēšana, kuras rezultātā rodas testu procedūru definīcijas (specifikācijas) pietiekamā detalizācijas līmenī (soļi: testu programmatūras modeļa definēšana, testu arhitektūras definēšana, testu procedūru definēšana, automatizēto vai manuālo testu kartēšana, testa datu kartēšana).

Promocijas darbā tiek detalizēti aplūkots un analizēts šis process, kā arī apskatītas vairākas testu automatizācijas heiristikas.

Pastāv vairāki automatizētās testēšanas procesa modeļi. M. Fewster un D. Graham piedāvā procesa modeli, kas sastāv no piecām fāzēm [7]:

1) testēšanas nosacījumu identificēšana;
2) testu projektēšana;
3) testu izstrāde;
4) testu izpilde;
5) rezultātu salīdzināšana.

E. Dustin et al. piedāvā automatizētā testa dzīvescikla metodoloģiju (*ATLM, Automated Test Life-Cycle Methodology*) [5], kas sastāv no sešām fāzēm:

1) testa automatizācijas lēmums;
2) testēšanas rīka iegāde;
3) automatizētās testēšanas ieviešanas process;
4) testu plānošana, projektēšana un izstrāde;
5) testu izpilde un pārvaldība;
6) testēšanas un programmas caurskate un novērtēšana.

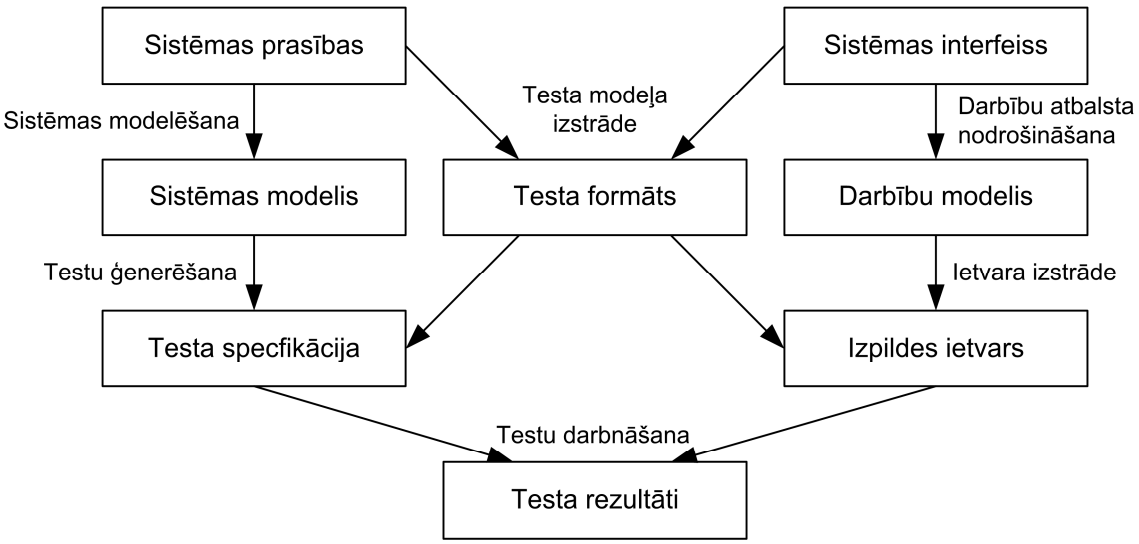
Salīdzinot šos divus modeļus, var secināt, ka Fewster-Graham procesa modelis ir šaurāks par ATLM un pārklāj tikai 4. un 5. ATLM fāzes.

Abos izskatītajos modeļos ir tāds trūkums, ka tie ignorē testu automātiskās ģenerēšanas iespēju, pieņemot ka visi automatizētie testi tiks izstrādāti manuāli. Lai novērstu šo trūkumu

tika izstrādāts vienots automatizētās testēšanas modelis, kas iekļauj arī testu ģenerēšanas iespēju.

Modeļa elementi ir aktivitātes un komponenti. Komponenti ir darba produkti, kas tiek radīti testēšanas procesa gaitā. Modeļa aktivitātes ir darbības, kas ir veicamās manuāli vai automatizēti, kuru rezultātā no viena veida komponentiem tiek iegūti citi.

Saistības starp modeļa aktivitātēm un komponentiem ir parādītas 2.4. attēlā (komponenti ir attēloti kā taisnstūri, aktivitātes — kā bultas).



2.4. att. Vienots automatizētās testēšanas modelis

Promocijas darbā tiek aprakstīts modeļa struktūras pamatojums un tā lietošanas iespējas.

Modeļa realizācija ir atkarīga no vairākiem aspektiem, tādiem kā testēšanas mērķi, testējamās sistēmas īpatnības un citiem. Modelis ir elastīgs tādā ziņā, ka ļauj atsevišķām aktivitātēm būt gan manuālām, gan automatizētām, ar izņēmumu, ka testu darbināšanai ir jābūt automātiskai, jo tas ir paredzēts automatizētās testēšanas modelēšanai. No otrās puses, darbību atbalsta nodrošināšana un jo vairāk ietvara izstrāde ir neautomatizējamas aktivitātes.

Modeļa komponentu un aktivitāšu realizācija ir atkarīga arī no tā, kāda veida rīks (pēc darba 1. nodaļā aplūkotās klasifikācijas) tiek izmantots kā izpildes ietvars. Piemēram, D1 klases rīkiem, kuriem testa dati ir neatņemama skripta sastāvdaļa, par testa specifikāciju var kalpot pats skripts. Testu ģenerēšana varētu tikt nodrošināta, balstoties uz skripta šabloniem, ja rīks ir S1 vai S2 klasē, t.i., skripts ir definējams programmēšanas vai deklaratīvajā valodā. Ģenerators no šablona var izveidot vairākus skriptus, kas atbilst dažādiem testpiemēriem. Rīks (izpildes ietvars) šos skriptus paņem kā testu specifikācijas un interpretē, iegūstot testa rezultātus.

Promocijas darbā tiek aplūkotas tā realizācijas iespējas arī dažādām citām klasēm.

Modelis ir izmantojams ne tikai testu ģenerēšanas un izpildes uzdevumā, bet arī testu komplektu ģenerēšanā un izpildē. Par testu specifikāciju tādā gadījumā kalpo testu secība komplektā, testu ģenerators izveido šo secību, vadoties pēc esošu testu īpašībām, bet testu ietvara loma ir testu dzinim, kas šo secību izpilda. Detalizētāk šī pieeja tiek aplūkota darba 4. nodaļā, kur tiek aprakstīti izstrādātie testu komplektu ģenerēšanas risinājumi.

2.3. Automatizēto testu metrikas un efektivitātes novērtēšana

Testēšanas process ir paredzēts defektu atklāšanai. Tāpēc ir jābūt veidam, kā no potenciāli bezgalīgas iespējamo testu kopas izvēlēties tādu apakškopu, kas atklātu pēc iespējas vairāk defektu, cik ir iespējams testēšanai atvēlētajā laikā.

Promocijas darbā testēšanas efektivitātes definīcija ir balstīta uz Pfleeger [23] definīciju, ko var izteikt formulas veidā:

$$E = D/T, \tag{2.1}$$

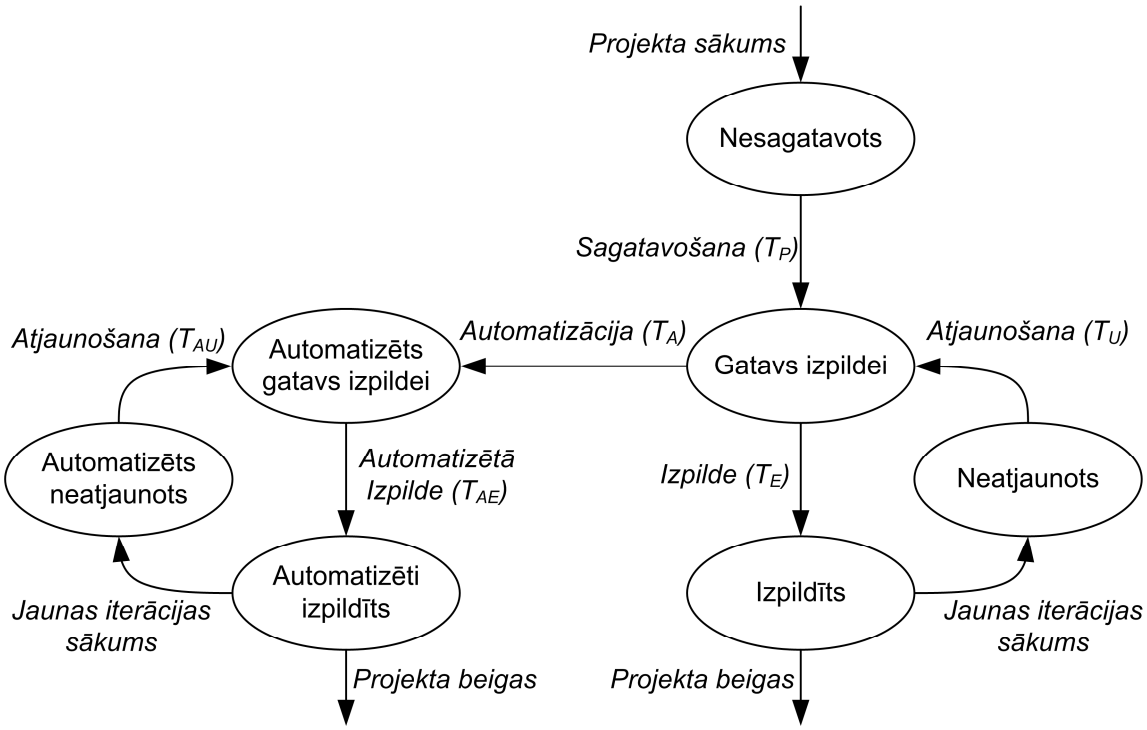
kur

E — testēšanas efektivitāte;

D — testēšanas laikā atklātu defektu skaits;

T — testēšanai patērētais laiks cilvēkstundās.

Tests, kas projekta gaitā tiek izpildīts vairakkārt vairākās iterācijās, katrā brīdī atrodas kādā no stāvokļiem, kuru diagramma ir parādīta 2.5. attēlā. Apzīmējumi T atbilst laikam, kas nepieciešams, lai veiktu pāreju (piemēram, lai automatizētu testu, izpildītu testu utt.)



2.5. att. Automatizējamā testa stāvokļi

Šo modeli var būtiski vienkāršot balstoties uz praktiskiem apsvērumiem.

1. Automatizētā testa izpilde neprasa cilvēka laiku, līdz ar to var uzskatīt, ka attiecīga darbietilpība ir vienāda ar nulli:

$$T_{AE} = 0. \tag{2.2}$$

2. Automatizētā testa sagatavošana un atjaunošana ir sarežģītāka par tā paša testa sagatavošanu un atjaunošanu manuālajā variantā. Šī sarežģītība nosaka attiecīgo darbietilpību palielināšanu automatizācijas gadījumā, ko ir lietderīgi izteikt kā automatizācijas sarežģītības koeficientu α (ievērojot, ka automatizētā testa sagatavošanas laiks ietver testa sagatavošanu kā tādu un tā automatizāciju):

$$\alpha = \frac{T_P + T_A}{T_P} = \frac{T_{AU}}{T_U}. \tag{2.3}$$

Testam τ veltītais laiks vienā i-tajā iterācijā ir atkarīgs no tā, kāds stāvoklis ir testam iterācijas sākumā un vai testu šajā iterācijā ir paredzēts automatizēt. Šo i-tajā iterācijā testam

τ veltīto laiku $T_{T\,i}(\tau)$ aprēķinu formulas ir apkopotas 2.1. tabulā (ievērojot ieviestos vienkāršojumus).

2.1. tabula

Testam patērētais laiks $T_{T\,i}$ vienā iterācijā

Stāvoklis iterācijas sākumā	**Stāvoklis iterācijas beigās**	
	Izpildīts	Automatizēti izpildīts
Nesagatavots	$T_P + T_E$	αT_P
Neatjaunots	$T_U + T_E$	$T_U + (\alpha - 1)T_P$
Automatizēts neatjaunots	—	αT_U

Zinot, kāds laiks ir jāvelta testam τ i-tajā iterācija, un testa svarīgumu $\vartheta_i(\tau)$ (kas ir testa mērķa defektu kopas risku summa), ir iespējams izskaitļot prognozējamo testa efektivitāti $E_i(\tau)$ šajā iterācijā. Balstoties uz formulu (2.1) un reducējot to uz viena testa gadījumu konkrētajā iterācijā, ir jāaizvieto defektu skaits ar testa svarīgumu (sagaidāmo defektpunktu skaitu, ko atklās tests) un par laiku jāpieņem testam veltīts laiks i-tajā iterācija. Rezultātā tiek iegūta formula:

$$E_i(\tau) = \frac{\vartheta_i(\tau)}{T_{T\,i}(\tau)}\ , \tag{2.4}$$

kur

$E_i(\tau)$ — testa τ efektivitāte i-tajā iterācija;

$\vartheta_i(\tau)$ — testa τ svarīgums i-tajā iterācija;

$T_{T\,i}(\tau)$ — testam τ patērētais laiks i-tajā iterācija.

Promocijas darba ietvaros tika iegūta formula (2.5), kas ļauj novērtēt testa automatizācijas relatīvo efektivitāti (efektivitāte, ko ir iespējams iegūt ar automatizēto testu pirmajās n iterācijās, dalīta tā paša testa efektivitāti ja to neautomatizē):

$$\mathcal{A}_n = \frac{1}{\alpha}\left(1 + \frac{nT_E}{T_P + (n-1)T_U}\right), \tag{2.5}$$

kur

α — automatizācijas sarežģītības koeficients;

T_P — manuālā testa sagatavošanas laiks;

T_U — manuālā testa atjaunošanas laiks;

T_E — manuālā testa izpildes laiks;

n — iterāciju skaits, kuram šī attiecība tiek rēķināta.

$\mathcal{A}_n$ vērtība, kas ir lielāka par 1, liecina par to, ka n pirmajās iterācijās automatizētais tests sasniedz augstāku efektivitāti, nekā tāds pats manuālais tests, un otrādi. Var konstatēt, ka ja izpildās nosacījums $\alpha T_U < T_U + T_E$, tad pie noteikta (varbūt arī liela) iterāciju skaita automatizācija sāks atmaksāties. Ja šis nosacījums neizpildās, šāda testa automatizācija neatmaksāsies nekad.

Promocijas darbā tiek piedāvāts praktisks algoritms, kas noteiktai iterācijai ļauj izvēlēties testu kopu ω, kuras efektivitāte ir tuva optimālajai.

1. Sākumā izvēlēto testu kopa ir tukša $\omega = \emptyset$, bet iterācijā atlikušais laiks $T_{\text{atl}} = T$ (kopējais iterācijas laiks).
2. Katram testam τ no iterācijā vēl nepaņemtiem testiem un kuriem $T_{T\,i}(\tau) < T_{\text{atl}}$, jāizrēķina efektivitāte $E_i(\tau)$, izmantojot datus par testa τ svarīgumu un tam nepieciešamo laiku (pēc 2.1. tabulas), apsverot arī automatizācijas iespēju.
3. Jāizvēlas tests τ_{ef}, kuram efektivitāte izrādījusies visaugstākā, jāpievieno τ_{ef} kopai ω, un iterācijā atlikušais laiks jāsamazina par testam τ_{ef} nepieciešamo laiku: $T_{\text{atl}} \coloneqq T_{\text{atl}} - T_{T\,i}(\tau_{\text{ef}})$.
4. Process jāturpina no 2. soļa kamēr paliek nepaņemti testi, kuriem $T_{T\,i}(\tau) < T_{\text{atl}}$.

Šis testu izvēles algoritms nodrošina testu kopas izvēli, kuras efektivitāte ir tuva visaugstākajai vienā konkrētajā iterācijā. Taču ja iterāciju skaits ir paredzams liels, šis algoritms ir pārāk alkatīgs. Piemēram, testa automatizācija reti atmaksājas tajā pašā iterācijā, kad to veic — automatizācijas priekšrocības parādās vēlāk.

Tāpēc var būt lietderīgi izvēlēties testus tā, lai testu kopas efektivitāte būtu pēc iespējas lielāka nevis vienā — tekošajā iterācijā, bet lai būtu pēc iespējas lielāka tās kumulatīvā efektivitāte nākamajās tuvākajās k iterācijās. Tādā gadījumā piedāvātajā algoritmā ir jāmaina 2. solis un katram testam jārēķina nevis efektivitāte $E_i(\tau)$, bet kumulatīvā efektivitāte $E_{i:k}(\tau)$:

$$E_{i:k}(\tau) = \frac{\sum_{j=i}^{i+k-1} \vartheta_j(\tau)}{\sum_{j=i}^{i+k-1} T_{T\,j}(\tau)}\ , \tag{2.6}$$

kur

$E_{i:k}(\tau)$ — testa τ kumulatīvā efektivitāte k iterācijās, sākot ar i-to iterāciju;

$\vartheta_j(\tau)$ — testa τ svarīgums j-tajā iterācija;

$T_{T\ j}(\tau)$ — testam τ patērētais laiks j-tajā iterācija.

Tātad izstrādāts testa efektivitātes matemātiskais modelis, kas ļauj aprēķināt prognozējamo testu efektivitāti vienā noteiktajā, kā arī vairākās testēšanas procesa iterācijās uz priekšu, bet izstrādātais efektīvās testu kopas izvēles algoritms var palīdzēt panākt testēšanas laika racionālāku izmantošanu, koncentrējoties uz testiem, kas spēj atklāt defektus ar lielāku varbūtību.

2.4. Automatizēto testu komplektu ģenerēšanas risinājuma izstrāde

Viena testpiemēra automatizēta izpilde ir relatīvi vienkāršs process. Kad testu kopa satur daudz testpiemēru, katra testpiemēra atsevišķa izpilde ir neefektīva. Tāpēc ir svarīgi atrast metodes lielu testu kopu izpildes automatizācijai. Promocijas darba ceturtajā nodaļā tiek ir veltīta testu kopu izpildes alternatīvām un testu dziņu projektēšanas iespējām. Rezultātā tiek izstrādāta bibliotēka, kas nodrošina testu komplektu ģenerēšanu no testu kopas.

Testu kopu automatizēto izpildi var klasificēt pēc diviem parametriem:

1) pēc izpildes aktivizēšanas tipa — testu kopa var tikt palaista manuāli vai automātiski;
2) pēc rezultātu tipa — testu kopas izpildes rezultātā var iegūt neapstrādātus rezultātus, atbilstību sagaidāmajiem rezultātiem vai orākula novērtējumu.

Lai izpildītu testu kopu, ir jāizpilda visi tajā esošie testpiemēri, palaižot attiecīgus testu skriptus. Programmatūras vienību, kas veic šo darbību sauc par testu dzini (*test driver*). Testu dziņus var realizēt vairākos veidos, bet visas testu dziņu realizācijas var iedalīt divās būtiski atšķirīgajās kategorijās:

1) statiskajos — testpiemēru izpildes secība ir viennozīmīgi definēta, ko nosaka dziņa izstrādātājs;
2) dinamiskajos — testpiemēru secība netiek uzdota tiešā veidā, dziņa automātiskās plānošanas algoritms izveido testpiemēru secību automātiski no atsevišķu testpiemēru izpildes nosacījumiem.

Promocijas darbā tās kategorijas tiek padziļināti analizētas. Statiskie dziņi tiek iedalīti lineārajos un strukturētajos, dinamiskie dziņi var būt balstīti uz atkarībām starp testiem, uz atkarībām ar nosacījumiem, uz stāvokļiem un uz stāvokļu sistēmām.

Automātiskās testu komplektu ģenerēšanas jeb testu komplektu sastādīšanas no testpiemēriem, uzdevuma veikšanai tika izstrādāta bibliotēka TSGL (*Test Suite Generation Library*). Atbilstoši 2.2. sadaļā aprakstītā vienotā automatizētās testēšanas modeļa (2.4. att.) apzīmējumiem TSGL nodrošina papildus funkcijas testu ģeneratoram un izpildes ietvaram. Tās izvade var būt gan XML formātā specificēta testu izpildes secība, kurai tādā gadījumā ir testa specifikācijas loma, gan tajā ir nodrošinātas funkcijas, kas spēj izsaukt testu izpildi šajā secībā. Par ievadu kalpo nestrukturēta testu kopa un testu metadati, kuros ir ievietota informācija par katra testa sākuma un beigu stāvokli. Pašu testpiemēru izveidošana un izpildes nodrošināšana nav TSGL kompetencē, tam ir nepieciešami citi rīki, piemēram, kāds no promocijas darba pirmajā nodaļā minētajiem.

TSGL bibliotēkas pamatā ir dinamiskais testu dzinis, kas ir balstīts uz testējamās programmas stāvokļu informāciju. Dziņa galvenais komponents ir testu komplektētājs, kas no dotās testpiemēru kopas izveido testpiemēru secību tādā veidā, ka katrs nākamais tests sākās stāvoklī kurā beidzās iepriekšējais. Bibliotēkā ir iestrādāti divi komplektētāju veidi:

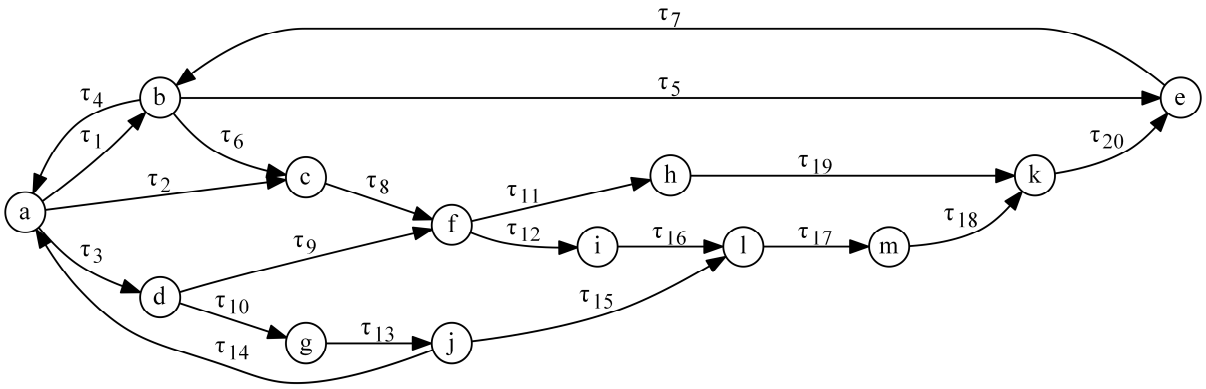
1) balstīts uz vienkāršo stāvokļu sistēmām;
2) balstīts uz salikto stāvokļu sistēmām.

Par ievadu testu komplektētājam kalpo testpiemēru kopa [31]. Taču no komplektētāja viedokļa būtisks ir nevis testpiemēru saturs, bet to metadati:

- testpiemēra sākuma stāvoklis, kurā ir jāsākas testpiemēra izpildei;
- testpiemēra beigu stāvoklis, kurā testpiemēram ir jābeidzas.

Komplektētāja darbība sastāv no sekojošajiem soļiem:

1) katram testpiemēram nolasa sākuma un beigu stāvokļu informāciju;
2) konstruē atmiņā stāvokļu pāreju grafu, kurā virsotnes atbilst testējamās programmas stāvokļiem, bet loki — testpiemēriem τ_i, kā 2.6. attēlā parādītajā piemērā;
3) konstruē ceļu grafā, kas sākās dotajā sistēmas sākumā stāvoklī, satur visus grafa lokus, un beidzās tajā pašā sistēmas sākuma stāvoklī — šis ceļš arī nosaka rezultāta testu komplektu, vai, precīzāk, loku secība ceļā nosaka testpiemēru secību komplektā.



2.6. att. Stāvokļu pāreju grafa piemērs

Kaut arī īsāki testu komplekti (tie, kas satur mazāku loku skaitu) ir labāki, nav nepieciešams izvēlēties visīsāko ceļu, jo ja testu komplekts izpildīsies automātiski, tas nepatērēs cilvēka laiku. Taču testu komplektam jebkurā gadījumā ir jābūt pieņemami īsam.

Balstīta uz vienkāršajiem stāvokļiem testu komplektu ģenerēšanas metode ir ierobežota, it īpaši gadījumā ar lietotāja saskarnes līmeņa automatizētajiem testiem. Saliktos stāvokļus var modelēt kā vārdu-vērtību pāru kopas [28]. Atsevišķus šādus vārdu-vērtību pārus sauksim par stāvokļa komponentiem. Stāvokļu komponenti, kas ir svarīgi vienam testam, var būt nesvarīgi citam. Ir iespējami trīs testa informētības tipi attiecībā uz stāvokļa komponenta vērtībām. Tie tipi tiks apzīmēti šādi:

- $R(x, y)$ — tests pieprasa noteiktu stāvokļa komponenta vērtību x kā priekšnosacījumu un uzstāda tam vērtību y kā pēcnosacījumu;
- $S(y)$ — testam nav svarīga stāvokļa komponenta vērtība (tas var strādāt ar jebkādu) kā priekšnosacījums, bet darbības rezultātā uzstāda tā vērtību par y kā pēcnosacījumu;
- U — testam nav svarīga stāvokļa komponenta vērtība.

Divus testus τ_1 un τ_2 ir iespējams savienot ceļa posmā $<\tau_1, \tau_2>$, ja nevienam stāvokļa komponentam nerodas konflikts starp τ_1 pēcnosacījumiem un τ_2 priekšnosacījumiem. Šādam ceļu savienojumam arī būs savi priekšnosacījumi, pēcnosacījumi un informētības tips. Pēc līdzīga principa ceļa posmā var savienot arī testu ar jau esošo ceļa posmu vai divus ceļa posmus. Ja atsevišķus testus uzskatīt par ceļa posmiem, kas sastāv no viena testa, tad šī savienošanas operācija ir definēta uz ceļa posmu kopas.

2.2. tabula parāda kā no savienojamo ceļa posmu C_1 un C_2 informētības tipiem tiek veidots rezultāta ceļa posma $C_3 = <C_1, C_2>$ informētības tips.

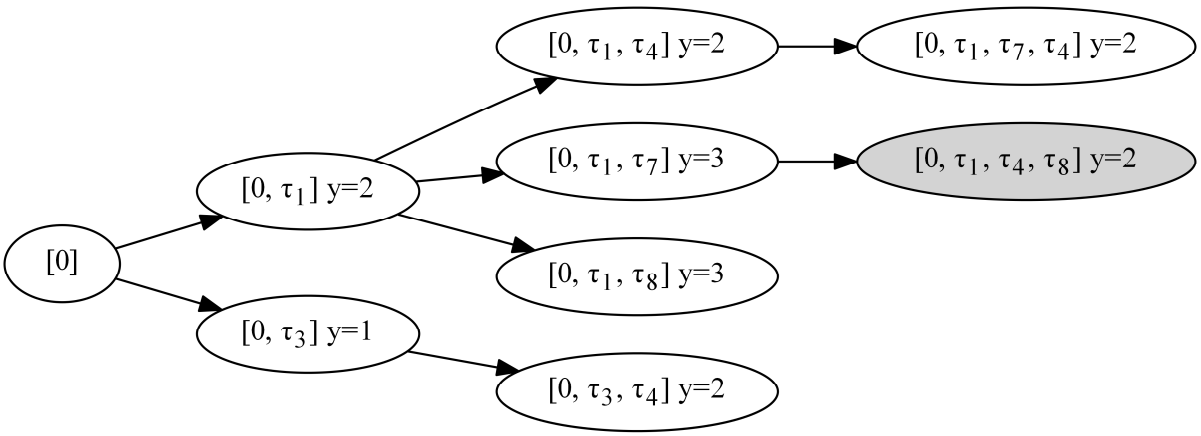
2.2. tabula

Ceļu posmu savienošanas likumi

Kreisā ceļa posma informētības tips	**Labā ceļa posma informētības tips**		
	***R*(x_2, y_2)**	***S*(y_2)**	***U***
R(x_1, y_1)	*R*(x_1, y_2), ja $y_1=x_2$ savādāk – nav iespējams	*R*(x_1, y_2)	*R*(x_1, y_1)
S(y_1)	*R*(x_2, y_2), ja $y_1=x_2$ savādāk – nav iespējams	*S*(y_2)	*S*(y_1)
U	*R*(x_2, y_2)	*S*(y_2)	*U*

TSGL realizē metodi, kas ģenerē testu komplektu, balstoties uz ceļu posmu koku.

1. Par koka sakni kalpo fiktīvas „nulles tests“. Nulles tests modelē sistēmas sākuma stāvokli, pasargājot sākuma vērtības visiem stāvokļa komponentiem. Nulles testam ir informētības tips *R*(*x*, *x*) visiem stāvokļa komponentiem, jo ir nepieciešams, lai testu komplekts beigtos tajā pašā stāvoklī, kādā tas sākās. Katrs koka mezgls reprezentēs kādu ceļa posmu.
2. Katrs nākamais koka līmenis tiek veidots no iepriekšējā. Katram līmeņa mezglam ir nepieciešams atrast tādus testus, kas var būt pielikti klāt (no labās puses) šim mezglam atbilstošajam ceļa posmam. Jauni ceļa posmi (jau par vienu elementu lielāki) kļūst par nākamā līmeņa mezgliem — dotā mezgla bērniem. Šo procesu ilustrē 2.7. attēls, kurā stāvoklis ietver vienu komponentu *y* ilustrācijas vienkāršības labā.



2.7. att. Ceļu koka fragmenta piemērs

Šis process ir jāatkārto kamēr netiks atrasts ceļa posms, kas satur visus testus un beidzās ar nulles testu. Algoritmam ir eksponenciālā sarežģītība un praksē, kad testu skaits ir liels, šī metode tās tīrā veidā nav piemērota. Darbā tiek aprakstītas TSGL izmantotās heiristikas, kas šo metodi padara par praktiskāku.

Izstrādāto TSGL bibliotēku ir iespējams izmantot automātiskajai testu komplektu ģenerēšanai no testu kopas gadījumos, kad atkarības starp testiem ir reprezentējamas kā vienkāršo vai salikto stāvokļu sistēmas.

2.5. Veiktspējas testēšanas rīka Picus izstrāde

Programmatūras veiktspējas testēšana ir svarīga vairāklietotāju sistēmu kvalitātes novērtēšanas sastāvdaļa. Lai notestētu sistēmas veiktspēju ir nepieciešams savākt veiktspējas metrikas vairāku lietotāju vienlaicīga darba apstākļos.

Divas pamatfunkcijas, kas jānodrošina veiktspējas testēšanas rīku klasē, ir:

- reālā lietotāja darbību emulēšana, kā arī spēja emulēt vairāku lietotāju darbu vienlaicīgi;
- sistēmas atbilžu laika un tā korektuma mērīšana.

Darba gaitā tika izstrādāts veiktspējas testēšanas rīks Picus, kura būtiskā īpašība ir paplašināšanas iespēju nodrošināšana [27]. Tajā ir realizētas lietojumprogrammu saskarnes (API) priekš:

1) protokoliem (nodrošina iespēju komunikācijas līmeņa testa skriptiem darboties ar dažāda veida sistēmām);
2) laika plāniem (nosaka, kā mainās noslodze testa laikā);
3) statistikas apstrādātājiem (nosaka stratēģiju, kā veiktspējas statistikas dati tiek apstrādāti un nodoti augstāka slāņa apstrādātājiem un kā tiek veidoti gala rezultāti);
4) komunikatoriem (nosaka, kādā veidā rīka centrālais komponents konsole sadarbojas ar slodzes aģentiem, kuri ļauj mērogot testa kapacitāti).

Pēc izstrādātās klasifikācijas (2.2. att.) Picus pieder klasēm D3 (atbalsta datu tabulas), S1a-Java (izmanto Java programmēšanas valodu skriptu izveidei), M2 (atbalsta paralēlu skriptu izpildi), I3-HTTP (strādā protokola līmenī, pagaidām atbalstot tikai HTTP). Atbilstoši izstrādātā vienotā automatizētās testēšanas modeļa (2.4. att.) apzīmējumiem Picus ir tipiska izpildes ietvara realizācija, kas var būt pielāgojama dažāda veida testu specifikāciju apstrādei.

Promocijas darbā ir detalizēti aprakstīta un pamatota Picus uzbūve, realizācijas un izmantošanas īpatnības.

Klasiskā pieeja darba slodzes veidošanai veiktspējas testos ir slodzes scenāriju projektēšana pirms testa darbināšanas. Gan skriptu, gan slodzes scenāriju rūpīga projektēšana ir svarīga testa mērķu sasniegšanai [26]. Šajā pieejā slodzes scenārija specifikācija kalpo par ieeju slodzes aģenta komponentam, kuru mēs sauksim par plānotāju (*scheduler*), kas ir atbildīgs par slodzes scenārijam atbilstošas darba slodzes nodrošināšanu. Veiktspējas rādītāji tiek mērīti un veido testa izvadi.

Vairākos gadījumu fiksēta slodzes scenārija projektēšana un izmantošana ir pietiekama un pat ir labākā metode. Piemēram, ja mērķis ir novērtēt dažādas veiktspējas metrikas pie dažādiem slodzes līmeņiem, būtu jāuzprojektē slodzes scenārijs, kurā virtuālo lietotāju skaits pakāpeniski palielinās no nulles līdz nepieciešamai testējamās sistēmas kapacitātei, jāizpilda šis tests un jāapstrādā rezultāti, lai iegūtu savāktus veiktspējas rādītājus kā funkcijas no virtuālo lietotāju skaita [3]:

$$p = F(v), \tag{2.7}$$

kur

p — veiktspējas rādītāja vērtība;

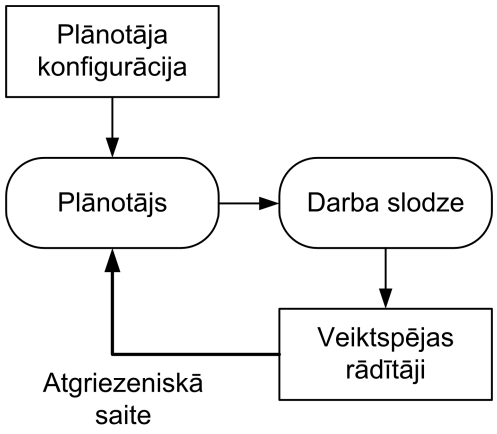
v — virtuālo lietotāju skaits;

F — pētāmā funkcija.

Šādu funkciju esamība un to reprezentēšana grafiskā formā var būt ļoti noderīga testējamās sistēmas šauro vietu noteikšanai, kā arī novērtēšanai, vai reālā testējamās sistēmas kapacitāte atbilst prasītajai.

Šādas pieejas trūkums ir relatīvi ilgs laiks, kas ir nepieciešams testa darbināšanai. Ja mērķis ir šaurāks, nekā piemērā minētais, tas var būt neefektīvi.

Promocijas darbā tiek piedāvāta cita metode, kurā plānotājs automātiski adaptē slodzes scenāriju, lai ātrāk noteiktu v_0 punktu dotajam p_0, tādu, ka $F(v_0) \geq p_0$ un $F(v_0 - 1) < p_0$ (gadījumā ja F ir nedilstoša). Šāda metode ir pielietojama, kad ir jāatrod tāds slodzes līmenis, pie kura interesējošs veiktspējas rādītājs sasniedz noteiktu vērtību p_0. Metode ir ilustrēta 2.8. attēlā.



2.8. att. Adaptīva slodzes scenārija modelis

Šajā modelī plānotājs nav tikai iepriekš specificētā slodzes scenārija izpildītājs, bet ir aktīvs lēmumu pieņēmējs par to, ko darīt tālāk [29]. Lēmumi, ko pieņem plānotājs ir atkarīgi no diviem faktoriem.

1. Plānotāja parametru konfigurācija. Šādi parametri nosaka mērķus, kuri ir jāsasniedz testa laikā un kurus plānotāja algoritms ir spējīgs apstrādāt.
2. Veiktspējas rādītāji, kas tika mērīti pirms lēmuma pieņemšanas brīža. Plānotājs var analizēt iepriekš iegūtos mērījumus un var balstīt uz tiem slodzes izmaiņas lēmumus.

Galvenā šī modeļa atšķirība no fiksēta slodzes scenārija modeļa ir atgriezeniska saite, ko iegūst plānotājs veiktspējas rādītāju formā.

Zemāk tiks aprakstīts vienkāršs adaptīvā plānotāja algoritms, kas var būt noderīgs dažādu veiktspējas testēšanas mērķu sasniegšanai. Bez vispārīguma zaudēšanas var pieņemt, ka funkcija $F(v)$ ir monotoni nedilstoša. Algoritms sastāv no divām fāzēm.

Pirmās fāzes mērķis ir intervāla noteikšana, kuram pieder punkts v_0. Lai tas varētu notikt ātri, lietotāju skaits pieaug eksponenciāli laikā, iteratīvi divkāršojot virtuālo lietotāju skaitu.

1. Sākt ar 1 virtuālo lietotāju ($v = 1$) un nomērīt $F(v)$.
2. Kamēr $F(v) < p_0$ divkāršot virtuālo lietotāju skaitu ($v := 2v$) un atkārtoti nomērīt $F(v)$.
3. Kad tiek sasniegta v vērtība, pie kuras $F(v) \geq p_0$, var uzskatīt, ka v_0 pieder intervālam $[v/2, v]$.

Otrās fāzes mērķis ir sašaurināt atrasto intervālu $[v_{\text{low}}, v_{\text{high}}]$:

1. Ja $v_{\text{high}} - v_{\text{low}} > 1$ punktam $v := v_{\text{low}} + (v_{\text{high}} - v_{\text{low}})/2$ nomērīt $F(v)$.
2. Ja $F(v) \geq p_0$, tad piešķiram $v_{\text{high}} := v$, savādāk $v_{\text{low}} := v$.

3. Atkārtot no soļa (1) līdz $v_{\text{high}} - v_{\text{low}} = 1$. Šajā brīdī v_{high} arī ir meklējamā v_0 vērtība.

Citiem vārdiem sakot, otrajā fāzē tiek pielietots binārās meklēšanas algoritms, lai atrastu v_0 intervālā $[v_{\text{low}}, v_{\text{high}}]$. Kopējo funkcijas $F(v)$ novērtējumu skaitu n aprakstītajam algoritmam var novērtēt ar formulu:

$$n = 2\lceil \log_2 v_0 \rceil, \tag{2.8}$$

kur

v_0 — meklējamais virtuālo lietotāju skaits.

Tā kā funkcijas $F(v)$ novērtējumu skaits ir logaritmiski atkarīgs no v_0, šāda algoritma efektivitāte ir augsta, salīdzinot ar fiksēta slodzes scenārija pieeju, kur novērtējumu skaits ir lineāri atkarīgs no v_0.

Lai algoritms varētu būt efektīvs, ir jāņem vērā daži citi aspekti:

- Pēc slodzes palielināšanas plānotājam ir jāgaida, kamēr visi jauni virtuālie lietotāji iestrādājas parastajā darba ritmā un tikai tad jāmēra nepieciešamie vidējie veiktspējas rādītāji. Gaidīšanas laikam jābūt vismaz divas reizes ilgākam, nekā aizņem viena skripta izpilde.
- Pēc slodzes samazināšanas plānotājam ir jāgaida, kamēr visi apturētie virtuālie lietotāji korekti pabeidz savas darbības un sistēma stabilizējas pēc noslodzes, ko ģenerēja šie lietotāji, un tikai tad ir jāmēra nepieciešamie vidējie veiktspējas rādītāji.

Ar skripta izpildes laiku ir saprotama laika, kas ir nepieciešams visu skripta transakciju izpildei, un visu aizturu starp transakcijām summa. Šādas piesardzības ir nepieciešamas, lai labāk nomērītu slodzes efektu un novērstu troksni mērāmajos datos, ko var radīt slodzes izmaiņu efekti.

Ir nepieciešams norādīt, ka aprakstītais algoritms varētu nenostrādāt gadījumā, ja ir citi faktori, kas var ietekmēt veiktspējas rādītājus vairāk, nekā slodzes līmenis, ko ģenerē slodzes aģents. Tādā gadījumā funkcijā $F(v)$ var būt stohastiski fluktuējoša un pieņēmums, ka tā ir monotoni nedilstoša, var būt aplams.

Promocijas darbā ir eksperimentāli parādīts, ka atsevišķiem uzdevumiem adaptīvā plānotāja pielietošana ļauj panākt būtiski labāku laika efektivitāti, samazinot testa izpildes laiku no lineārā līdz logaritmiskajam.

2.6. Izstrādāto risinājumu ieviešana

Promocijas darba sestajā nodaļā tiek īsi apkopota informācija par iepriekšējās divās nodaļās aprakstīto risinājumu lietošanu reālos projektos, sniedzot testēšanas ārpakalpojumus. Pasūtītāji šajos projektos bija vairākas Latvijas bankas un valsts iestādes, testētas tika lielas sistēmas ar tīmekļa saskarni, kuras raksturo liels lietotāju skaits.

Darba gaitā izstrādātie testu komplektu ģenerēšanas risinājumi un bibliotēka TSGL tika izmantota divos SIA „Jauno Tehnoloģiju Centrs" testēšanas projektos. Bibliotēka TSGL tika izmantota tikai divas reizes tāpēc, ka tajā nodrošinātas iespējas kļūst nepieciešamas tikai ļoti netriviālajos gadījumos. Vienkāršajos testu automatizācijas uzdevumos parasti pietiek ar statiskajiem testu datiņiem un TSGL lietošana nevajadzīgi sarežģītu testu programmatūru.

Taču vienā testēšanas projektā automatizācijas uzdevums izrādījās pietiekami netriviāls, lai TSGL ieviešana dotu pozitīvu efektu. Uzdevuma būtība ir šāda: bija jāpārbauda, vai lietotājiem ir tiesības veikt tās un tikai tās darbības sistēmā, kas atbilst viņa lomai (tiesību līmenim) sistēmā. Uzdevuma sarežģītību raksturo šādi lielumi:

- pārbaudāmo darbību skaits: ap 120;
- lomu skaits: ap 45.

Sistēmas integritātes kritiskuma dēļ, bija jāpārbauda visas lomu un darbību kombinācijas. Vienas darbības izpildes pārbaudei vienai lietotāja lomai manuālās testēšanas gadījumā aizņemtu 10 minūtes, tāpēc visu kombināciju pārbaude prasītu 900 cilvēkstundas jeb 5,6 cilvēkmēnešus.

Automatizējot testus, bija iespējams samazināt vienas kombinācijas testēšanai nepieciešamo laiku līdz aptuveni vienai minūtei. Tādā gadījumā pilna testa izpilde aizņemtu 90 stundas, kas joprojām nebija pieņemami, jo tika sagaidīts, ka tiks atklātas neatbilstības, tās tiks labotas un pilnais tests būs jāizpilda atkārtoti. Četras diennaktis ilgs tests nebija pieņemams.

Vienkāršojot testu skriptus un uzticot TSGL testu komplektu ģenerēšanu, pilnajam testam nepieciešamais laiks samazinājās līdz 30 stundām. Paralelizējot testa izpildi uz trīs datoriem, ir izdevies testa kopējo laiku samazināt līdz 10 stundām, kas jau bija pieņemami. Atklāto neatbilstību dēļ tests bija jāatkārto 4 reizes (notika četri testēšanas-labošanas cikli), tāpēc uz TSGL balstītās pieejas izmantošana attaisnojās.

Šajā testā tika izmantots uz saliktajiem stāvokļiem balstītais komplektētājs, un stāvokļa komponenti bija šādi:

1) lietotāja loma — tiesību līmenis, kas ir tekošajam sistēmas lietotājam;
2) aktīvais logs — kurš logs sistēma dotajā brīdī aktīvs.

Būtiskākā TSGL loma bija tāda, ka testu skripti varēja neiekļaut darbības, kas atver nepieciešamo logu, un beigās atgriežas sākuma stāvoklī — pietika norādīt, kurā logā katram testam ir jāsākas, un kurā jābeidzas — pareizo izpildes secību TSGL izvēlējās automātiski, līdz ar to tika panākta laika ekonomija uz liekas logu atvēršanas un aizvēršanas. Bez TSGL tas arī būtu iespējams, taču tas sarežģītu skriptu izstrādi ar to, ka secība būtu jāveido manuāli, un manuāli jāpārveido pēc skriptu atjaunošanas.

Otrajā projektā, kur tika izmantota bibliotēka TSGL, tā tika lietota saiknē ar Picus, nodrošinot virtuālā lietotāja darbību secības ģenerēšanu. TSGL izmantošanas lietderība bija pamatota ar to, ka veicamo darbību skaits projektā bija liels (ap 50) un darbību veikšanas iespējamība lielā mērā bija atkarīga no tekoša sistēmas stāvokļa. Taču šajā gadījumā TSGL izmantošanas atdeve nebija tik liela kā pirmajā projektā, jo darbību skaits tomēr bija ievērojami mazāks.

Izstrādātais veiktspējas testēšanas rīks Picus tika izmantots 13 SIA „Jauno Tehnoloģiju Centrs“ testēšanas projektos. Visi projekti bija saistīti ar tīmekļa sistēmu veiktspējas testēšanu, līdz ar to tika izmantots esošais Picus modulis HTTP protokola atbalstam. Lai ilustrētu projektu dažādību, četri no tiem tiek aprakstīti nedaudz detalizētāk:

1. Testējamā sistēma bija izstrādāta, izmantojot valodu Java un ietvaru Struts. Tomcat bija izmantojams kā lietojumu konteineris un tīmekļa serveris. Tika izstrādāti divi Picus skripti. Vienam skriptam bija nepieciešams sūtīt sistēmai unikālus specifiska formāta datus, tāpēc lai nodrošinātu to iespēju, Picus tika paplašināts ar datu tabulu atbalstu. Realizācijā datu tabulām ir jābūt CSV formas failiem, no kuriem skripts ņem vienu kārtējo rindu, kad tas ir nepieciešams. Pietiekami lielas datu tabulas garantē, ka skripti katru reizi izmanto savu datu komplektu.
2. Testējamā sistēma bija izstrādāta, izmantojot Oracle rīkus, un Oracle Web Application Server tika izmantots kā tīmekļa serveris. Tika izstrādāti pieci skripti, kas izpildīja piecus dažādus datu atlasīšanas scenārijus. Pieprasījumos tika izmantoti lieli datu apjomi ar neparastu kodējumu, tāpēc uz šī projekta precedenta Picus HTTP protokola modulī tika ieviests šādu pieprasījumu atbalsts.
3. Testējamā sistēma bija izstrādāta, izmantojot Ruby-on-Rails un Microsoft Infopath tehnoloģiju kombināciju. MS IIS tika izmantots kā tīmekļa serveris. Lai nodrošinātu darba ar Infopath emulāciju, bija nepieciešams rīkam Picus pievienot iespēju apmainīties ar XML datiem caur HTTP protokolu. Līdz tam brīdim esošais sīkdatņu

(*cookies*) apstrādes mehānisms nebija savietojams ar testējamo sistēmu, tāpēc tika uzlabotas sīkdatņu apstrādes iespējas HTTP protokola modulī.

4. Sistēma bija izstrādāta, izmantojot valodas Java un JSP. Tomcat bija izmantots kā lietojumu konteineris un tīmekļa serveris. Šai sistēmai jau nebija nepieciešams paplašināt Picus iespējas — pietika ar jau esošo funkcionalitāti.

Kā var redzēt no šiem piemēriem, reālu projektu veikšanas rezultātā vairākos aspektos tika uzlabots HTTP protokola modulis. Pašlaik tas ir pietiekami stabils, lai bez izmaiņām varētu atbalstīt gandrīz jebkuru tīmekļa sistēmu testēšanu.

Visi minētie, kā arī citi projekti bija sekmīgi izpildīti un aizņēma no vienas līdz trim nedēļām, skaitot no testēšanas prasību fiksēšanas brīža līdz rezultātu kopsavilkuma atskaites sagatavošanai. Tehniskais testu sagatavošanas darbs, ieskaitot skriptu izstrādi, Picus pielāgošanu un testa parametru konfigurēšanu, aizņēma no divām līdz piecām darba dienām. Gūta pieredze rāda, ka rīks Picus ir pietiekami labi atbilst savam nolūkam. Laika daudzums, kas bija jāpatērē šiem projektiem ir tuvs laikam, kas bija jāvelta līdzīgiem projektiem, izmantojot citus rīkus, vai pat labāks. Tas ir izskaidrojams ar Picus pielāgojamību pat pietiekami netriviālām situācijām.

3. DARBA REZULTĀTI

Darba mērķis bija izstrādāt programmatūras testēšanas automatizācijas metodes, kas ļautu padarīt automatizēto testēšanu efektīvāku. Darba rezultātā tika izstrādātas metodes automatizētai testu komplektu ģenerēšanai, kā arī adaptīvajai veiktspējas testēšanai, kas tika aprobēti reālos projektos.

Būtiskākie darba sasniegumi.

1. Izstrādāta automatizētās testēšanas rīku klasifikācija, pēc kuras tika klasificēti 32 testu automatizācijas rīki, kas ļauj izvēlēties piemērotāko rīku konkrētas programmatūras testēšanai.
2. Tika izstrādāts vienots automatizētās testēšanas modelis, kas ļauj identificēt potenciāli automatizējamos procesus programmatūras testēšanā. Modelis ir piemērojams dažādos testu automatizācijas kontekstos un dažādi implementējams atkarībā no rīka klases.
3. Tika izstrādāts manuālo un automatizēto testu efektivitātes novērtēšanas matemātiskais modelis. Uz modeļa pamata ir izstrādāts efektīvās testu kopas izvēles algoritms, kas ir izmantojams, lai balstoties uz defektu risku un testēšanas darbietilpības novērtējumiem izvēlēties testus, kas ar lielāku varbūtību atklās vairāk defektu ierobežotā laikā.
4. Tika izstrādātas divas automatizētās testu komplektu ģenerēšanas metodes, balstītās uz testējamu sistēmu stāvokļiem: vienkāršajiem un saliktajiem. Metodes tika implementētas bibliotēkā TSGL, kas tika aprobēta divos reālajos projektos.
5. Tika izstrādāts veiktspējas rīks Picus, īpašs ar tā elastību un paplašināšanas iespējām. Picus rīks tika aprobēts 13 reālajos projektos.
6. Tika izstrādāta adaptīvās veiktspējas testēšanas metode, kas tika realizēta kā Picus rīka spraudnis.

Izstrādātajiem risinājumiem ir tāda praktiskā nozīme, ka tos ir iespējams izmantot programmatūras izstrādes projektos automatizētās testēšanas efektivitātes uzlabošanai. Izstrādātie rīki un pieredze to izmantošanā projektos liecina, ka izstrādātie risinājumi ir reāli ieviešami un praktiski noderīgi.

4. DARBA APROBĀCIJA

4.1. Uzstāšanās konferencēs

Par darba rezultātiem ir ziņots šādās starptautiskajās konferencēs:

1. 50th International Scientific Conference of Riga Technical University, 12-16 October 2009, Riga, Latvia.
2. 13th East-European Conference on Advances in Databases and Information Systems (ADBIS 2009), Workshop on Intelligent Educational Systems and Technology-enhanced Learning (INTEL-EDU), 7 September 2009, Riga, Latvia.
3. International Conference on Advanced Learning Technologies (ICALT-2009), 15-17 July 2009, Riga, Latvia.
4. 49th International Scientific Conference of Riga Technical University, 13-15 October 2008, Riga, Latvia.
5. International Conference on Information Technologies (InfoTech-2008), 19-20 September 2008, Varna, Bulgaria.
6. International Conference on Engineering Education & Research (ICEER 2007), 2-7 December 2007, Melbourne, Australia.
7. 48th International Scientific Conference of Riga Technical University, 11-13 October 2007, Riga, Latvia.

4.2. Publikācijas

Darba rezultāti ir publicēti šādos starptautiskajos izdevumos:

1. Sukhorukov A. Automated Test Chaining Based on Compound States // Proceedings of the International Conference on Information Technologies (InfoTech-2010). – Sofia, Bulgaria: Publishing House of Technical University, 2010. – pp. 127–132.
2. Sukhorukov A. Self-Directed Performance Testing // Scientific Journal of Riga Technical University. Series 5 – Computer Science. Applied Computer Systems. – 2010. – Vol. 43. – pp. 84–89.

3. Сухоруков А. Целенаправленное обучение на примере модели и классификатора инструментов автоматизации тестирования ПО // Образовательные технологии и общество. – Казань, Татарстан, РФ: Казанский государственный технологический университет, 2010. – Том 13, № 1. – стр. 370–377.
4. Sukhorukov A. Test Case Generation for Validation of E-Learning Course // Advances in Databases and Information Systems, 13th East-European Conference, ADBIS 2009 Associated Workshops and Doctoral Consortium. Local Proceedings. – Riga, Latvia: Riga Technical University, 2009. – pp. 230–237.
5. Sukhorukov A. Architecture for Automated Validation of E-Learning Courses // Proceedings of the Ninth IEEE International Conference on Advanced Learning Technologies (ICALT-2009). – Washington, DC, USA: IEEE Computer Society, 2009. – pp. 152–153.
6. Sukhorukov A. Problems of Test-Driven Aspect-Oriented Development // Scientific Proceedings of Riga Technical University. Series 5 – Computer Science. Applied Computer Systems. – 2009. – Vol. 38. – pp. 180–186.
7. Sukhorukov A. Performance testing tool Picus // Proceedings of the 22nd International Conference on Systems for Automation of Engineering and Research (SAER-2008). – Bulgaria: King, 2008. – pp. 165–172.
8. Sukhorukov A., Zaitseva L. Automated test state management framework // Scientific Proceedings of Riga Technical University. Series 5 – Computer Science. Applied Computer Systems. – 2008. – Vol. 34. – pp. 215–224.
9. Sukhorukov A., Zaitseva L. Applicability of Automated Testing to E-Learning Software Validation // Proceedings of the International Conference on Engineering Education & Research (ICEER 2007). – Melbourne, Australia: Victoria University, 2007. – pp. 1–6.

LITERATŪRA[1]

1. Adrion W., Branstad M., Cherniavsky J. Validation, Verification, and Testing of Computer Software // ACM Computing Surveys. – 1982. – Vol. 14, No. 2. – pp. 159–192.
2. Aichernig B. et al. State of the Art Survey - Part a: Model-based Test Case Generation. Technical Report 1-19a on project MOGENTES. – Graz University of Technology. – 2008.
3. Amza C. et al. Specification and implementation of dynamic web site benchmarks // 5th IEEE Workshop on Workload Characterization (WWC-5). – IEEE Press, 2002. – pp. 3–13.
4. Boehm B. W. Seven Basic Principles of Software Engineering // Journal of Systems and Software. – 1983. – Vol. 3, No. 1. – pp. 3–24.
5. Dustin E., Rashka J., Paul J. Automated Software Testing: Introduction, Management and Performance. – Boston, MA, USA, 1999. – 608 p.
6. Ferguson R., Korel B. The chaining approach for software test data generation // ACM Transactions on Software Engineering and Methodology. – 1996. – Vol. 5, No. 1. – pp. 63–86.
7. Fewster M., Graham D. Software Test Automation: Effective Use of Test Execution Tools. – New York, NY, USA: ACM Press/Addison-Wesley, 1999. – 596 p.
8. Futatsugi K. et al. Principles of OBJ2 // Proceedings of the 12th ACM Symposium on Principles of Programming Languages. – 1995. – pp. 21–28.
9. Gelperin D., Hetzel B. The Growth of Software Testing // Communications of the ACM. – 1988. – Vol. 31, No. 6. – pp. 687–695.
10. Godefroid P. Compositional dynamic test generation // Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. – 2007. – pp. 47–54.
11. Gupta N., Mathur A., Soffa M. Automated test data generation using an iterative relaxation method // Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering. – 1998. – pp. 231–244.
12. Hamill P. Unit Testing Frameworks. – Sebastopol, CA, USA: O'Reilly, 2004. – 304 p.

[1] Kopasvilkumā ir saīsināts literatūras saraksts. Promocijas darbā bibliogrāfiskajā sarakstā ir 130 avoti.

13. ISO 8807:1989. Information processing systems – Open Systems Interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour. – 1989.
14. ITU-T Recommendation Z.100 (11/99). Specification and Description Language (SDL). – 2000. – 246 p.
15. King J. Symbolic execution and program testing // Communications of the ACM. – 1976. – Vol. 19, No. 7. – pp. 385–394.
16. Korel B. Automated software test data generation // IEEE Transactions on Software Engineering. – 1990. – Vol. 16, No. 8. – pp. 870–879.
17. Leavens G. T., Baker A. L., Ruby C. Preliminary design of JML: A behavioral interface specification language for Java // ACM SIGSOFT Software Engineering Notes. – 2006. – Vol. 31, No. 3. – pp. 1–38.
18. Lewis W. E. Software Testing and Continuous Quality Improvement. 3rd ed. – Auerbach Publications, 2008. – 688 p.
19. Marick B. When should a test be automated // Proceedings of The 11th International Software/Internet Quality Week. – 1998. – pp. 1–20.
20. Meyer B. Eiffel: The Language. – Prentice Hall, 1991. – 300 p.
21. Mosley D. J., Posey B. A. Just Enough Software Test Automation. – Prentice Hall, 2002. – 288 p.
22. Mosses P. D. CASL: A guided tour of its design // Proceedings of WADT'98. – Springer-Verlag, 1999. – pp. 216–240.
23. Pfleeger S. L. Software Engineering: Theory and Practice. 2nd ed. – Prentice Hall, 2001. – 659 p.
24. Pressman R. S. Software Engineering: A Practicioner's Approach. 4th ed. – McGraw-Hill, 1997. – 852 p.
25. Runeson P. A survey of unit testing practices // IEEE Software. – 2006. – Vol. 23, No. 4. – pp. 22–29.
26. Subraya B. M. Integrated approach to web performance testing: A practitioner's guide. – PA, USA: IRM Press, 2006. – 368 p.
27. Suhorukovs A. Veiktspējas testēšanas rīka Picus izstrāde un pielietošanas iespējas // Latvijas IT uzņēmumu 9. konference „Testēšanas teorija un prakse": konferences materiāli. – Rīga, Latvija, 2008. – lpp. 31. –39.

28. Sukhorukov A. Automated Test Chaining Based on Compound States // Proceedings of the International Conference on Information Technologies (InfoTech-2010). – Sofia, Bulgaria: Publishing House of Technical University, 2010. – pp. 127–132.
29. Sukhorukov A. Self-Directed Performancd Testing // Scientific Journal of RTU. Series 5. – 2010. – Vol. 43. – pp. 84–89.
30. Sukhorukov A., Zaitseva L. Applicability of Automated Testing to E-Learning Software Validation // Proceedings of the International Conference on Engineering Education & Research (ICEER 2007). – Melbourne, Australia: Victoria University, 2007. – pp. 1–6.
31. Sukhorukov A., Zaitseva L. Automated test state management framework // Scientific Proceedings of Riga Technical University. Series 5. – 2008. – Vol. 34. – pp. 215–224.