RĪGAS TEHNISKĀ UNIVERSITĀTE

Datorzinātnes un informācijas tehnoloģijas fakultāte

Lietišķo datorsistēmu institūts

Lietišķo datorzinātņu katedra

Jurijs GRIGORJEVS

Doktora studiju programmas „Datorsistēmas” doktorants

IEGULTO SISTĒMU NEFUNKCIONĀLO ĪPAŠĪBU UZ MODEĻIEM BALSTĪTA TESTĒŠANAS METODE

Promocijas darbs

Zinātniskā vadītāja

Dr. sc. ing., profesore

O.ŅIKIFOROVA

Rīga 2012

Darbs ir izstrādāts ar Eiropas Sociālā fonda atbalstu Nacionālās programmas „Atbalsts doktorantūras programmu īstenošanai un pēcdoktorantūras pētījumiem" projekta „Atbalsts RTU doktorantūras attīstībai" ietvaros.

Anotācija

Iegultās sistēmas arvien vairāk parādās mūsu dzīvē, nodrošinot manuālo darbību automatizēšanu un ļaujot cilvēkam vadīt tos procesus, kurus viņi nevar pildīt bez tām. Sarežģītās medicīnas iekārtas, lidmašīnas, kosmiskie kuģi ir tikai daži piemēri šādām sistēmām. Lai nodrošinātu korektu funkcionēšanu, šādām kompleksām sistēmām piemīt virkne nefunkcionālu īpašību, piemēram, laika ierobežojumu kontrole, asinhronā darbība, sinhronizācija un citas. Regulāras sistēmu kļūmes parāda, ka sistēmu izstrādes procesā ir nepieciešami uzlabojumi. Eksistējošās nefunkcionālo īpašību testēšanas metodes balstās uz iepriekšējās paaudzes modelēšanas notācijām un nenodrošina automātisko testpiemēru ģenerēšanu no sistēmas modeļiem.

Tajā pašā laikā modernajā programmatūras izstrādes procesā parādās jaunās metodes un veselas metodoloģijas. Modeļvadāmā arhitektūra (angl. *Model Driven Architecture* - MDA) ir viena no populārākajām un strauji attīstošākajām programmatūras izstrādes tehnoloģijām, kas ļauj no sistēmu modeļiem ģenerēt pirmkodu. MDA izplatīšanu un popularizēšanu veicina pieejamie tehnoloģijas atbalsta standarti un rīki, kas nodrošina izstrādes procesu. Līdzīgas aktivitātes notiek arī testēšanas jomā. Parādās jaunās pieejas sistēmu testēšanai, balstoties uz to modeļiem. Savukārt, eksistējošās uz modeļiem balstītas testēšanas metodes fokusējas uz funkcionālo īpašību testēšanu.

Pētījumā rezultātā ir izstrādāta iegulto sistēmu nefunkcionālo īpašību testēšanas metode, kas balstās uz MDA principiem. Izstrādāta metode paredz testpiemēru ģenerēšanu no UML secību diagrammām un to prezentēšanu standartizētā UML testēšanas profila formātā. Piedāvātā metode ir implementēta ar nepieciešamo rīku kopu, kas nodrošina modeļu ielasīšanu XMI formātā, datu pārnešanu datubāzes tabulās, vienkāršotu sistēmas modeļa prezentāciju un tā transformāciju atbilstoši definētiem transformācijas likumiem uz testēšanas modeli.

Pētījumā izstrādāta metode un rīki ir aprobēti reālā laika maksājuma karšu sistēmas laicīguma īpašības testēšanā. Balstoties uz probācijas rezultātiem ir veikti secinājumi par metodes pielietošanas iespējām, tās priekšrocībām un turpmākiem attīstīšanas soļiem.

Darbs ir uzrakstīts latviešu valodā, satur ievadu, 4 nodaļas, nobeigumu, literatūras sarakstu ar 100 nosaukumiem, 5 pielikumus, kopā 147 lapaspuses.

Abstract

We regularly interact with a multitude of computer controlled systems of all shapes and sizes called embedded systems. Some of the most popular ones are the microwave oven, refrigerator, trucks and light motor vehicles, certain kinds of military systems and many others. Embedded systems have more non-functional properties, more complex software structure and development process compared to simple systems. In just the past few years, there have been a number of cases where errors in the software and hardware of embedded systems led to human casualties and massive losses. Errors in embedded systems are also discovered in smaller systems, such as motorcars, cell phones and elsewhere. Existing methods for testing embedded systems are incomplete and do not ensure automation of the testing process or correspondence to current trends in software development. These methods are based mostly on general testing standards and do not support testing of non-functional features of embedded systems.

To make development of embedded systems reliable, new methodologies, programming and specification languages are introduced. Standardized principles of model-driven architecture (MDA), the available development environments and tools stimulate automation of the entire software development cycle. One of the tools used by MDA is a Unified Modeling Language (UML), which provides a testing profile to support the testing process. However, even though the testing profile was standardized in 2005, there are still no generally accepted methods for automating the testing process and generating test cases based on the system model.

As a result of the research, a method was developed for testing the non-functional properties of embedded systems, as well as the set of tools developed to ensure transformation of UML models and generation of test cases. The suggested method is based on the fundamentals of model-driven software development and general principles of model transformation. To test the application of the method, it was approbated by verification of time constraints for a real-time payment card system, for which these properties are critical for performing standard activities.

The doctoral thesis has been written in Latvian, and includes Introduction, 4 Parts, Conclusion, 5 Appendices, Bibliography with 100 information sources, 37 figures and illustrations, 5 tables, in total 147 pages.

SATURS

IEVADS

Ikdienā mēs saskaramies ar vairākiem un dažādiem datoru vadītiem aparatūras līdzekļiem un tie veic arvien vairāk sadzīviskus un citus pakalpojumus, un to klāsts katru dienu pieaug. Ja padomā par tiem detalizētāk, tad šādas lielas un mazas sistēmas ir visapkārt mums un mēs pat reizēm neiedomājamies, ka saskaramies ar šādām sarežģītām sistēmām uz katra soļa. Visas šīs sistēmas sastāv no aparatūras un programmatūras un tiek sauktas par iegultām sistēmām. Dažas vienkāršākās un populārākās no tām ir: elektroniskā cepeškrāsns, mikroviļņu krāsns, ledusskapis, veļas mazgājamā mašīna, dažāda veida signalizācijas un trauksmes sistēmas, liftu sistēma, dzelzceļa pārbrauktuves sistēma, telpu apgaismojuma sistēmas, apkures un kondicionēšanas sistēmas, bankomāts, biļešu un vairākas citas sistēmas. Minētās sistēmas pārsvarā var attiecināt pie nelielām un vidēja izmēra iegultām sistēmām. Savukārt, pie lielo iegulto sistēmu piemēriem var minēt šādas: kravas un vieglās mašīnas, lidmašīnas, helikopteri, kuģi, dažādas militārās sistēmas, kosmosa kuģi, atomstacijas, dažādas automatizētās ražošanas un apkalpošanas industrijas roboti un citas sistēmas. Eksistē arī mazās iegultās sistēmas, kas pēc savas būtības sastāv no minimālās programmatūras un dažiem aparatūras līdzekļiem. Pie šādas kategorijas parasti tiek attiecinātas visāda veida rotaļlietas, vienkārša risinājuma apgaismojuma sistēmas, automātiski atveramo durvju sistēmas un citas. Galvenā atšķirība no iepriekšējo kategoriju sistēmām ir programmatūras un saslēgto aparatūras līdzekļu sarežģītība, kā arī izpildāmo funkciju nosacījumi un to kritiskums. Apskatot iegultās sistēmās, promocijas darba autors fokusējas uz vidējā un lielā izmēra sistēmām.

Vidējā un lielā izmēra iegultās sistēmas apvieno to specifika un funkcionēšanas īpašības. Tās parādās tāpēc, ka rodas nepieciešamība vienlaikus vadīt vairākās ārējās ierīces, vākt un apstrādāt informāciju no ārējās vides, patstāvīgi darboties cilvēkam neiejaucoties un iespēju robežās turpināt funkcionēt nenozīmīgu kļūmju gadījumos. Lai nodrošinātu korektu sistēmas funkcionēšanu, iegulto sistēmu specifikai arī ir jābūt pārbaudītai visos iespējamos gadījumos. Par dotā pētījuma objektu ir izvēlētās iegulto sistēmu nefunkcionālās īpašības.

Aktualitāte

Iegultām sistēmām piemīt vairāk nefunkcionālo īpašību, sarežģītākā programmatūras struktūra un tā izstrāde salīdzinājumā ar parastajām sistēmām. Pēdējo gadu laikā ir zināmi vairāki fakti par iegulto sistēmu programmatūras un aparatūras kļūmēm, kas noved pie cilvēku upuriem un milzīgiem zaudējumiem. Kaut arī visu lidmašīnu avāriju tehniskais iemesls ir ap 20% [ACRO 2011], kopējais skaits tomēr ir liels un tas ir atkarīgs no lidmašīnas programmatūras un aparatūras. Tā, piemēram, lielākās pēdējā laika lidmašīnu avārijas tehnisko iemeslu dēļ ir [LEY 2010] [BEA 2011] [BBC 2009]. Līdzīgi lidmašīnu problēmām šādas eksistē un regulāri parādās arī kosmiskajos aparātos [SVO 2011], kur atšķirībā no lidmašīnām tehnisko kļūmju statistika ir tuva 100%, jo tie tiek izlaisti orbītā, balstoties uz iepriekš ieprogrammētiem parametriem, cilvēkam neiejaucoties. Iegulto sistēmu kļūmes tiek atklātas arī mazākās sistēmās, piemēram, automašīnās [VWREC], kafijas automātos, mobilajos telefonos un citur. Īpaši aktuāls iegulto sistēmu izstrādē kļūst sistēmu testēšanas uzdevums.

Lai padarītu iegulto sistēmu izstrādi par drošu, tiek ieviestas jaunās metodoloģijas, programmēšanas un specificēšanas valodas un tiek ražoti atbalsta rīki. Modernajā programmatūras izstrādē automatizācija kļūst arvien populārāka. Pirmkoda ģenerācija ir zināma un vairākus gadus tiek pielietota, bet joprojām pilna izstrādes cikla automatizācija tiek pētīta un ir virzībā uz manuālās darbības aizvietošanu. Modeļvadāmās arhitektūras (angl. *Model-Driven Architecture* turpmāk MDA) [MDA] standartizētie principi, pieejamas izstrādes vides un rīki stimulē pilna programmatūras izstrādes dzīves cikla automatizāciju, tai skaitā arī testēšanas uzdevuma izpildē, kas parasti aizņem ap 50% no visa izstrādes laika. Viens no MDA izmantotiem līdzekļiem ir vienotā modelēšanas valoda (angl. *Unified Modeling Language* turpmāk UML), kas testēšanas procesa atbalstam piedāvā testēšanas profilu [UTP], kas savukārt nodrošina testēšanas procesa artefaktu specificēšanu standartizētā veidā. Kaut arī kopš 2005. gada, kad testēšanas profils tika standartizēts, joprojām neeksistē vispārīgi pieņemto metožu testēšanas procesa automatizācijai un testpiemēru ģenerēšanai no sistēmas modeļa.

Esošo risinājumu pārskatīšana

Eksistējošās iegulto sistēmu testēšanas metodes nav pilnīgas un nenodrošina testēšanas procesa automatizēšanu un atbilstību modernajām programmatūras izstrādes tendencēm. Metodes pārsvarā balstās uz vispārīgiem testēšanas standartiem un paņēmieniem. Ir zināmās specifiskās metodes, piemēram, nodrošināt pēc iespējas lielāko mezglu un ceļu pārklāšanu, automātiski ģenerējot testpiemērus, iegūstot maksimālo pārklāšanu [STH 2001] [WU 2007]. Metodei ir vairākas priekšrocības, ieskaitot metodes autoru uzstādījumu par automātiskās testpiemēru ģenerēšanas nepieciešamību. Tomēr šāda pārklāšana nevar nodrošināt iegulto sistēmu nefunkcionālo īpašību verificēšanu veiksmīgos un neveiksmīgos gadījumos. Eksistējošo testēšanas metožu formalizācijas mēģinājumi [AGR 2001] [MAT 1995] [KRS 2002] [ZHI 1999] neatbalsta automātisko testpiemēru ģenerēšanu no sistēmas modeļa, kas ir viens no uzdevumiem, kas risināms promocijas darba ietvaros.

Balstoties uz eksistējošo iegulto sistēmu testēšanas metožu analīzi var secināt, ka pašlaik neeksistē iegulto sistēmu testēšanas metodes, kas atbalstītu modernās programmatūras izstrādes tendences un nodrošinātu šādu sistēmu nefunkcionālo prasību verificēšanu.

Promocijas darba mērķis ir piedāvāt uz MDA principiem un standartiem balstītu testēšanas metodi, kas dod iespēju automātiski ģenerēt testpiemērus iegulto sistēmu nefunkcionālo īpašību testēšanai. Piedāvātai metodei jānodrošina iegulto sistēmu nefunkcionālo īpašību verificēšana, balstoties uz vispārpieņemtiem sistēmu modelēšanas standartiem.

Darba mērķa sasniegšanai autors izvirza **šādus uzdevumus:**

a. analizēt eksistējošās testēšanas metodes un klasificēt tās;
b. izpētīt iegulto sistēmu nefunkcionālās īpašības un to testēšanas metodes un paņēmienus;
c. veikt MDA principu analīzi un izvērtēt iespēju pielietot tos testpiemēru ģenerēšanā;
d. definēt testēšanas metodi, kas nodrošinātu testpiemēru ģenerēšanu no UML modeļiem;
e. pielietot piedāvāto metodi sistēmas nefunkcionālo īpašību testēšanai un secināt par metodes pielietošanas iespējām.

Pētījuma metode

Balstoties uz uzdevumu rezultātiem, tiek iegūts priekšstats par testēšanas specifiku, par nepieciešamo testēšanas veidu automatizāciju un modernām automatizācijas metodēm, kā arī tiek analizētas un definētas iegulto sistēmu nefunkcionālās prasības un to eksistējošās modelēšanas un testēšanas metodes. Darbā tiek izdalītās šādas vidējā un lielā izmēra iegulto sistēmu nefunkcionālās īpašības: laicīgums, asinhronā darbība, sinhronizācija, uzdevumu plānošana un drošums. Balstoties uz MDA principiem un aprakstītām īpašībām tiek izvirzīta hipotēze par testēšanas metodi automātisko testpiemēru ģenerēšanai. Hipotēze sniedz vīziju par testēšanas metodes eksistenci, tās pielietojumu. un tajā pašā laikā hipotēze nodrošina eksistējošo MDA standartu pielietošanu modeļu prezentēšanā, apstrādē un transformēšanā testēšanas modelī. UML un XML metadatu apmaiņas (angl. *XML Metadata Interchange* – XMI) standartu, kā arī vispārīgo MDA principu pielietošana veicina hipotētiskās metodes efektivitāti un universālo integrēšanu dažādos izstrādes procesos. Piedāvātā metode netiek saistīta ar kādu konkrētu izstrādes rīku un atbalsta vispārpieņemtus standartus. Metodes galvenās universālās īpašības ir sistēmas modeļa atbalsts XMI 2.1 versijas formāta un rezultātā iegūto testpiemēru saglabāšana UML definētā testēšanas profilā. Pirmā īpašība nodrošina neatkarību no sistēmas modeļu izstrādes rīkiem, jo eksistējošie modelēšanas un MDA atbalsta rīki, piemēram [EA] vai [EMF], nodrošina modeļu eksportēšanu XMI formātā. Savukārt, UML testēšanas profila izmantošana ļauj strukturēt transformācijas rezultātu un glabāt to standartizētā veidā. Šāds princips atbalsta uzģenerēto rezultātu turpmāko pielietošanu dažādos testēšanas vadības rīkos, kas nodrošina UML standartizēto testa datu importu no ārējiem avotiem.

Hipotēzes pārbaude notiek balstoties uz izvēlētas nefunkcionālās īpašības modelēšanu un verificēšanu. Laicīgums ir viena no populārākajām īpašībām, kas tiek piemērota ne tikai iegultām, bet arī reālā laika un citām sistēmām ar laika ierobežojumiem. Eksistē daudz iespēju modelēt laicīgumu dinamiskajos modeļos [COR 2000] [WAN 1998] [PIL 2005] [DOU 2004], bet analizējot populārākos eksistējošus modeļus, autors secina, ka piemērotākā modelēšanas notācija ir UML valoda [GRI 2008a] [GRI 2008b], kas tajā pašā laika ir viens no standartizētiem līdzekļiem, ko piedāvā OMG kopā ar MDA.

Hipotēzes pārbaude tiek veikta uz reālas maksājuma karšu sistēmas lietojot reālus sistēmas modeļus ar laika ierobežojumiem. Maksājuma karšu sistēma Card Suite [1][TIE] atbilst reālā laika sistēmām ar „mīkstiem" laika ierobežojumiem (angl. *soft time constraints*), kas pieļauj laika ierobežojumu pārsniegšanu, bet tajā pašā laika pārsniegšanu skaits ir stingri ierobežots ar sistēmai izvirzītām prasībām. Laika ierobežojumu pārsniegšana sistēmas darbībā var novest pie finansiālām sekām, tāpēc ir svarīgi ievērot un verificēt tos. Balstoties uz Card Suite sistēmas specifiku, tā var tikt izmantota par hipotēzes pārbaudes platformu.

Hipotēzes verificēšanai tiek lietots finansu autorizācijas pilnais apstrādes cikls, kas iekļauj sevī vairākus laika ierobežojumus tās apstrādes gaitā. Card Suite izstrādē sistēmas modelēšanai tiek lietots Enterprise Architect [EA] modelēšanas un transformācijas rīks, kas nodrošina modeļu eksportēšanu XMI formātā. Sistēmas dinamiskās uzvedības modelēšanai tiek lietota UML secību diagramma, kas atbilst UML2.0 versijai un nodrošina laicīgu aspektu definēšanu modelī. Eksistējošā autorizāciju apstrādes secību diagramma ar laika ierobežojumiem tiek eksportēta XMI datnē (angl. *file*), nodrošinot ieejas datus hipotēzes pārbaudei. Tālāk ar autora izstrādātiem rīkiem izejas dati tiek ielādēti datu bāzē, sagatavoti transformācijai un apstrādāti ar transformācijas likumiem, ģenerējot testpiemērus. Hipotēzes validācija notiek salīdzinot automātiski ģenerēto rezultātu ar manuāli iegūtiem testpiemēriem, veicot modeļa analīzi pēc klasiskās testēšanas paņēmieniem. Validācijas secinājumi ļauj spriest par metodes trūkumiem, priekšrocībām un tās pielietošanas iespējām.

Promocijas darba novitāte

Automātiskā testpiemēru kopas ģenerēšanas metode iegulto sistēmu nefunkcionālām prasībām, kas balstās uz MDA principiem un eksistējošiem standartiem, ir dotā pētījuma novitāte. MDA principi galvenokārt tiek pielietoti pirmkoda ģenerācijai un reti testpiemēru izveidei, pārsvarā funkcionālo sistēmas īpašību testēšanai. Promocijas darba autors piedāvā metodi, kas spēj nodrošināt tieši nefunkcionālo prasību verificēšanu, balstoties uz modeļu transformācijas principiem MDA kontekstā. Tajā pašā laikā eksistējošās nefunkcionālo īpašību automātiskās

[1]Card Suite sistēma nodrošina karšu pieņemšanu vairākos pieņemšanas punktos, savienojumu un reālā laika transakciju apstrādi starp lokālo, VISA, MasterCard, American Express un citu starptautisko organizāciju tīkliem. Par pieprasījumu avotiem kalpo ATM (Automatic Teller Machine), POS (Point Of Sale) un citi termināļi, kā arī jau minētie starptautiskie tīkli. Card Suite bāzējas uz UNIX veida operētājsistēmām, par DBVS izmanto Oracle produktu un lieto ātrdarbīgu transakciju monitoru Oracle Tuxedo.

testēšanas metodes neatbalsta MDA principus un šodienas programmatūras izstrādes tendences, ko savukārt nodrošina piedāvātā metode. Darbā metode tiek realizēta un aprobēta laicīguma īpašībām ar automātisko testpiemēru ģenerēšanu. Tas ļauj secināt par metodes veiksmīgu pielietošanu reālo problēmu risināšanai un metodes principu pielietošanu pārēju nefunkcionālo īpašību verificēšanai, kas apzināti netika iekļauts dotā darba izstrādē. Metodes aprobācijā aprakstītais piemērs nodrošina testpiemēru ģenerēšanu laicīguma verificēšanu vienībtestēšanas līmenī.

Pētījuma ietvaros piedāvātā metode balstās uz izstrādāto rīku pielietošanu, kas atbalsta modeļu transformāciju un testpiemēru ģenerēšanu, kam autors ir definējis specifisku rīku lietošanas secību, t.s. rīku ķēdi (angl. *tool chain*). Rīku izstrādē apzināti tika pielietots komponenšu bāzēts projektējums, sadalot modeļu ielasīšanu XMI formātā datu bāzē, modeļu vienkāršotu reprezentēšanu un pašu transformācijas daļu. Modeļu vienkāršošanas posms ir nepieciešams lai nodrošinātu kvalitatīvu un saturisku transformācijas likumu definēšanu. Sadalīšana komponentēs ļauj nodrošināt katras komponentes savstarpējo neatkarību un fokusēšanos uz konkrētām operācijām. Šāds princips ļauj pielāgot un papildināt eksistējošus rīkus tā, lai piedāvāta metode varētu tikt pielietota arī pārējo nefunkcionālo īpašību verificēšanai un atbilstu UML un XMI standartu attīstībai nākotnē.

Metodes pielietošana ar automātisko testpiemēru ģenerēšanu ļauj ātrāk iegūt testējamus gadījumus un pasargāties no izlaistiem gadījumiem, jo ģenerēšana balstās uz sistēmas modeli un definētiem likumiem. Metode sagaida sistēmas modeli standarta UML diagrammu veidā un standarta XMI formātā, kas ļauj pielietot dažādus modelēšanas rīkus. Pateicoties šim, metode kļūst viegli integrējama eksistējošos izstrādes procesos ar vienīgo nosacījumu, ka tām jānodrošina uz modeļiem bāzēto sistēmu specificēšanu. Transformācijas likumu definēšana ir vienreizējs process, jo izveidotie likumi var tikt pielietoti jebkurai sistēmas elementu kopai. Tas nozīmē, ka laicīguma verificēšanai tiek izveidoti transformācijas likumi, kas spēj ģenerēt testpiemērus diagrammās aprakstītiem elementiem, kas atbilst transformācijas likumu nosacījumiem. Apskatot citu nefunkcionālo prasību, jādefinē jauna transformācijas likumu kopa. Šāda likumu izstrāde padara metodi efektīvāku īpaši produktu un ilglaicīgu projektu izstrādē, jo izstrādātie likumi tiek vairakkārt pielietoti vairākām sistēmas diagrammām un tās elementiem.

Promocijas darba praktiskā nozīme

Vairāk kā 10 gadu strādājot Tieto Latvia uzņēmumā par reālā laika maksājuma karšu sistēmas testētāju, Ieviešanas, testēšanas un integrācijas testēšanas nodaļu vadītāju, darba autors novēroja, kā nefunkcionālo īpašību nepietiekama testēšana var novest pie veselas sistēmas dīkstāves un citām negatīvām sekām. Dotā pētījuma rezultāti ir izstrādāti, balstoties uz iegūto pieredzi reālā laika sistēmu testēšanā un modernām tendencēm programmatūras izstrādē, kā arī piedāvātā metode tika veiksmīgi pielietota finansu ziņojumu apstrādes scenārija verificēšanā no laika ierobežojuma viedokļa. Praktiskā pielietošana ļauj spriest par metodes funkcionēšanu un nepieciešamo testpiemēru ģenerēšanu, balstoties uz standarta sistēmas modeļiem.

Daži no promocijas darba rezultātiem tika izstrādāti vai lietoti šādos zinātniskajos un mācību metodiskajos projektos, kuros autors piedalījās kā izpildītājs:

1. LZP lietišķo pētījumu projekta Nr. 09.1269 "Uz izkliedēta mākslīgā intelekta un tīmekļa tehnoloģijām balstītas metodes un modeļi intelektuālas lietišķās programmatūras un datorsistēmu arhitektūras izstrādei", virziena "Programmatūras izstrāde MDA ietvarā" (2009.-pašlaik).
2. Līdzdalība ESF līdzfinansētā projektā „Studiju moduļa izstrāde modeļvadāmai programmatūras attīstības tehnoloģijai datorsistēmas programmā" (Līgums Nr. 2007/0080/VPD1/ESF/PIAA/06/APK/3.2.3.2./0008/0007) (2006.-2008.).
3. IZM/RTU zinātniski pētnieciskais projekts R7389 "Programmatūras sistēmas klašu struktūras ģenerēšanas rīka prototipa izstrāde, pamatojoties uz divpusložu modeli" (2008).

Darba rezultātu publikācijas un to prezentācija konferencēs

Promocijas darba rezultāti ir publicēti 10 rakstos starptautiski recenzējamos krājumos un 6 publikācijās vietējos krājumos. Rezultāti ir prezentēti 12 starptautiskajās konferencēs, ka arī 4 vietējas testēšanas konferencēs un vienā RTU studentu zinātniskajā un tehniskajā konferencē.

Autora publikācijas starptautiskajos krājumos:

1. 2011. – Grigorjevs J. „Model-Driven Testing Approach for Embedded Systems Specifics Verification Based on UML Model Transformation", Proceedings of 3rd International Workshop on Model-Driven Architecture and Modeling-Driven Software Development, Beijing, China, pp. 26-35

2. 2011. – Grigorjevs J. „Model-Driven Testing Approach Based on UML Sequence Diagram", Scientific Journal of Riga Technical University, 5th series Computer Science Applied Computer Systems, Vol. 47, Riga, Latvia – 2011 – pp. 85-90

3. 2009. – Grigorjevs J., Nikiforova O. „Several Outlines on Model-Driven Approach for Testing of Embedded Systems", Scientific Journal of Riga Technical University, 5th series Computer Science Applied Computer Systems, 38. vol, pp. 108.-118.

4. 2008. – Nikiforova O., Pavlova N., Grigorjevs J. „Several Facilities of Class Diagram Generation from Two-Hemisphere Model in the Framework of MDA". 23rd International Symposium on Computer and Information Sciences, ISCIS 2008 : Proceedings. - Piscataway, NJ : IEEE, 2008. 6 p.

5. 2008. – Grigorjevs J., Nikiforova O. "Complience of Popular Modelling Notations to Non-functional Requirements of Embedded Systems", Proceedings of the International Scientific Conference Informatics in the Scientific Knowledge, Varna, Bulgaria, pp. 139-149

6. 2008. – Grigorjevs J. "Testing of Embedded System's Non-functional Requirements", Scientific Proceedings of the 8th International Baltic Conference Baltic DB&IS 2008, Tallinn, Estonia

7. 2008. – Grigorjevs J., Nikiforova O. "Modelling of Non-Functional Requirements of Embedded Systems", Scientific Proceedings of 42nd Spring International Conference MOSIS2008, Hradec nad Moravici, Czech Republic. – pp. 13-20

8. 2007. – Grigorjevs J., Nikiforova O. „Features of embedded systems that require specific testing approaches", Proceedings of the 48th Scientific Conference of Riga Technical University, 5th Series, Vol. 30, Computer Science, Applied Computer Systems, Vol. 26, Riga, Latvia. – pp. 47-56

9. 2006. – Grigorjevs J., Nikiforova O. „Unit Testing for Real Time Systems", Scientific Journal of Riga Technical University, Computer Science, Applied Computer Systems, 5th series, Vol. 26, Riga, Latvia: RTU Publishing. – pp. 67-77

10. 2005. – Grigorjevs J., Nikiforova O. „Testing Process Adjustment for Real Time Systems", Proceedings of the 46th Scientific Conference of Riga

Technical University, 5th Series, Vol. 22, Computer Science, Applied Computer Systems, Riga, Latvia: RTU Publishing. – pp. 229-241

Citas autora publikācijas:

1. 2011. – Grigorjevs J. „Card Suite Testing Toolbox – product launch case study", Proceedings of 12th Annual Software Testing Conference TAPOST2011, Riga, Latvia
2. 2010. – Grigorjevs J. „Several practical approaches in testing automation", Proceedings of 11th Annual Software Testing Conference TAPOST2010, Latvia
3. 2009. – Grigorjevs J. „Testing automation framework in combined technology environment", Proceedings of 10th Annual Software Testing Conference TAPOST2009, Riga, Latvia
4. 2008. – Grigorjevs J. „Testa gadījumu vadības sistēmas izvēle un ieviešana", 9. Latvijas ikgadējā konference „Testēšanas teorija un prakse", konferences materiālos, Rīga, Latvija
5. 2005. – Grigorjevs J., Ņikiforova O. „Testēšanas procesa pielāgošana reālā laika sistēmām", tēzes 46. RTU studentu zinātniskās un tehniskās konferences materiālos, Rīga, Latvija
6. 2003. – Grigorjevs J. "Sistēmas veiktspējas testēšana praksē. Nepieciešamās informācijas iegūšana, apstrāde un attēlošana", tēzes Latvijas 4. Testēšanas teorija un prakse konferencei, Rīga, Latvija

Par darba rezultātiem ir referēts starptautiskajās konferencēs:

1. 2011 – Grigorjevs J. „Model-Driven Testing Approach for Embedded Systems Specifics Verification Based on UML Model Transformation", 3rd International Workshop on Model-Driven Architecture and Modeling-Driven Software Development June 8-11, 2011, Beijing, China
2. 2011 – Grigorjevs J. „Card Suite Testing Toolbox – product launch case study", 12th Annual Software Testing Conference TAPOST2011, May 26, 2011, Riga, Latvia
3. 2010 – Grigorjevs J. „Model-Driven Testing Approach Based on UML Sequence Diagram", RTU 51st International Scientific Conference, October, 2010, Riga, Latvia
4. 2010 – Grigorjevs J. „Several practical approaches in testing automation", 11th Annual Software Testing Conference TAPOST2010, July6, 2010, Riga, Latvia

5. 2009 – Grigorjevs J., Nikiforova O. „Several Outlines on Model-Driven Approach for Testing of Embedded Systems", RTU 50th International Scientific Conference, 12-16 October, 2009, Riga, Latvia
6. 2008 – Nikiforova O., Pavlova N., Grigorjevs J. „Several Facilities of Class Diagram Generation from Two-Hemisphere Model in the Framework of MDA", 23rd International Symposium on Computer and Information Sciences, ISCIS 2008, 27-29 October, 2008, Istanbul, Turkey
7. 2008 – Grigorjevs J., Nikiforova O. "Complience of Popular Modelling Notations to Non-functional Requirements of Embedded Systems", International Scientific Conference Informatics in the Scientific Knowledge, June, 2008, Varna, Bulgaria
8. 2008 – Grigorjevs J. "Testing of Embedded System's Non-functional Requirements", International Baltic Conference Baltic DB&IS 2008, June 2-5, 2008, Tallinn, Estonia
9. 2008 – Grigorjevs J., Nikiforova O. "Modelling of Non-Functional Requirements of Embedded Systems", 42nd Spring International Conference MOSIS2008, April 22-24, 2008, Hradec nad Moravici, Czech Republic
10. 2007 – Grigorjevs J., Nikiforova O. „Features of embedded systems that require specific testing approaches", RTU 48th International Scientific Conference, October, 2007, Riga, Latvia
11. 2006 – Grigorjevs J., Nikiforova O. „Testing Process Adjustment for Real Time Systems", RTU 47th International Scientific Conference, October, 2006, Riga, Latvia
12. 2005 – Grigorjevs J., Nikiforova O. „Unit Testing for Real Time Systems", RTU 46th International Scientific Conference, October, 2005, Riga, Latvia

Par darba rezultātiem ir referēts citās konferencēs un semināros:

1. 2008. – „Testa gadījumu vadības sistēmas izvēle un ieviešana", 9. Latvijas ikgadējā konference „Testēšanas teorija un prakse", Rīga, Latvija
2. 2008. – Seminārs „Modeļvadāmas programmatūras attīstības tehnoloģijas studiju moduļa zinātniskās vides izveide RTU studiju programmā Datorsistēmas", ar referātu, 14.05.2008, Rīga
3. 2005. – „Testēšanas procesa pielāgošana reālā laika sistēmām", 46. RTU studentu zinātniskās un tehniskās konferences materiālos, Rīga, Latvija

4. 2003. – “Sistēmas veiktspējas testēšana praksē. Nepieciešamās informācijas iegūšana, apstrāde un attēlošana.”, Latvijas IT uzņēmumu 4. Testēšanas teorija un prakse konferencei, Rīga, Latvija.

Promocijas darba struktūra

Darbs ir strukturēti sadalīts 4 nodaļās. Pirmās divas nodaļas ir veltītas pētījuma tēmas apskatam, analīzei un autora hipotēzes izvirzīšanai. Nākamajās nodaļās detalizēti tiek aprakstīta piedāvātā metode un tās aprobācija uz reālā laika sistēmas piemēra, kā arī metodes novērtēšana.

Pirmajā nodaļā ir sniegts ieskats iegulto sistēmu specifikā, tiek apskatītas dažāda veida iegultās sistēmas un to kopīgās īpašības. Turpat tiek apskatīti vispārīgie testēšanas principi, atsevišķi izdalot iegulto sistēmu testēšanas specifiku. Ir zināmi un vairākos darbos [SPI 2007] [MYE 2004] [KAN 1999] [COP 2004] ir aprakstīti kopīgie testēšanas principi, kas tiek pielietoti arī iegultām sistēmām. Nodaļas turpinājumā autors pievēršas iegulto sistēmu specifikai un to nefunkcionālām īpašībām. Detalizēti tiek definēta katra no tām, apskatot īpašības būtību, specificēšanas, modelēšanas un testēšanas paņēmienus.

Modernās tendences programmatūras izstrādē un modeļvadāmās izstrādes principi ir aprakstīti darba 2. nodaļā. Sistēmas modelis kļūst par pilnvērtīgu tās specifikāciju un UML vienotā modelēšanas valoda nodrošina un veicina pāreju uz strukturētiem sistēmas modeļiem. Uz šādiem modeļiem balstās MDA programmatūras izstrādes process, kas definē modeļu pielietošanas iespējas pateicoties modeļu transformācijas procesam. Nodaļā tiek apskatīts testēšanas process un programmatūras izveidošanas process ar MDA rīkiem un tiek izvirzīta hipotēze par jaunu testēšanas metodi, balstītu uz MDA transformācijas principiem, kas būtu pielietojama iegulto sistēmu nefunkcionālo īpašību testēšanai. Hipotēze definē metodes koncepciju un tās galvenās vadlīnijas.

3. nodaļa ir veltīta detalizētai metodes definēšanai un aprakstam. Piedāvātās metodes projektējums, realizācijas tehnoloģijas un izstrādes vides ir pamatotas un ir balstītas uz efektīvo metodes pielietojumu un integrāciju dažādos izstrādes procesos. Autors piedāvā uz komponentiem bāzēto metodes realizāciju un detaļās apraksta katru no tiem. Metodes piedāvātā realizācija tiek fokusēta uz laicīguma īpašību verificēšanu un pēc savas struktūras pārredz tās pielietošanu arī pārējo nefunkcionālo īpašību

testēšanai. Metodes demonstrācijas piemērs sniedz iespēju iepazīties ar tās darbības principiem un veikt sākotnējo izvērtēšanu.

4. nodaļa ir veltīta metodes aprobācijai uz reālā laika maksājuma karšu sistēmas. Vispārīgi ir apskatīts programmatūras izstrādes process SIA Tieto Latvia, kur notika piedāvātās metodes pielietošana reālās ekspluatācijas apstākļos. Tekstā detalizēti ir aprakstīts projekts, kura ietvaros notika metodes lietošana, nodrošinot reālā laika maksājuma karšu sistēmas laicīguma īpašību verificēšanu. Nodaļā tiek sniegts verificējamās sistēmas vispārīgs apraksts ar tās funkcionēšanas un realizācijas īpašībām. Metodes aprobācijai tiek izmantots veselas autorizācijas apstrādes piemērs no reālās dzīves ar vairākiem laika ierobežojumiem, kas tiek specificēti UML secību diagrammu veidā un tiek sagatavoti apstrādei XMI formātā. Apskatītais apstrādes piemērs tiek apstrādāts ar piedāvāto testēšanas metodi, paralēli manuāli veicot testpiemēru izstrādi. Piedāvātā aprobācija nodrošina testpiemēru ģenerēšanu laicīguma īpašības verificēšanai vienībtestēšanas līmenī. Balstoties uz aprobācijas rezultātiem, autors veic metodes izvērtēšanu, aprakstot tās ierobežojumus, priekšrocības un pielietošanas iespējas.

1 IEGULTO SISTĒMU TESTĒŠANAS PAMATI

Iegultās sistēmās ir tādas sistēmas, kuru sastāvdaļās ir aparatūra un programmatūra, un kas paredzētas specifiska lietojuma funkcionēšanai, cilvēkam neiejaucoties [JEN 2011].

Iegultās sistēmas ir specifiskas ar patstāvīgo funkcionēšanu un pieslēgto aparatūras līdzekļu vadību. Šāda specifika ienes papildprasības testēšanas procesam un pieprasa iegulto sistēmu īpašību speciālu verificēšanu. Iegulto sistēmu verificēšanā un validācijā pārsvarā pielieto klasiskās testēšanas metodes un tikai daļa no pieejām apskata šādu sistēmu specifikas papildus verificēšanu. Šajā nodaļā autors sniedz eksistējošo testēšanas metožu apskatu un klasificēšanu, definē iegulto sistēmu nefunkcionālās īpašības un apskata to modelēšanas un testēšanas paņēmienus.

1.1 Iegulto sistēmu īpašības

Programmatūras un aparatūras līdzekļiem attīstoties, parādās jaunas iespējas iegulto sistēmu attīstīšanai un jaunu sistēmu ražošanai. Sadzīves un citas tehnikas ražotāji nepārtraukti papildina tirgū piedāvātās ierīces ar jaunu funkcionalitāti un apakšsistēmām, šādā veidā padarot tās sarežģītākas un palielinot kļūmju skaitu tajās. Par piemēru šādai sistēmu attīstīšanai var kalpot mūsdienās ražotās Volkswagen automašīnas [VW]. Koncernam ir 3 izpētes laboratorijas, kur notiek jaunu sistēmu izpēte un inovāciju realizācija. Pēdējo gadu modeļos implementētās inovācijas skar nevis kādu vienu atsevišķu mašīnas daļu, bet veselu kompleksu no vairākām apakšsistēmām. Volkswagen mašīnu inovācijas var sašķirot un izdalīt sekojošas galvenās grupas: degvielas avotu dažādība un efektīva patērēšana (piemēram, biodegviela, hibrīdi dzinēji, CO^2 izmešanu un degvielas patērēšanas samazināšana), vadītāju asistēšana (parkošanas sistēmas, distances ievērošanas sistēmas un citas), komunikācijas un tīkli (navigācijas sistēmas, koplietošanas informācijas saņemšanas sistēmas un pieeja globālam tīmeklim), braukšanas stabilitātes un citas sistēmas [VW]. Ir acīmredzami, ka šādu apakšsistēmu esamība automašīnā un to paralēla funkcionēšana reālajā laikā prasa specifisku pieeju programmatūras izstrādei. Tomēr esošo laboratoriju un mašīnu vairāku verificēšanas un validēšanas posmu esamība nevar nodrošināt visu programmatūras un aparatūras kļūdu identificēšanu. [VWREC]

ir oficiāli pieejama informācija par Volkswagen automašīnu atsaukšanām uz servisa centriem ar mērķi atjaunot to programmatūru vai ierīces. Tā, piemēram, tikai Volkswagen Passat 2002-2008. gados ražotām automašīnām ir reģistrētās 8 globālās atsaukšanas. Tas ļauj secināt par to, ka joprojām sarežģīto iegulto sistēmu ražošanā nav pietiekamu metožu un rīku to verificēšanai.

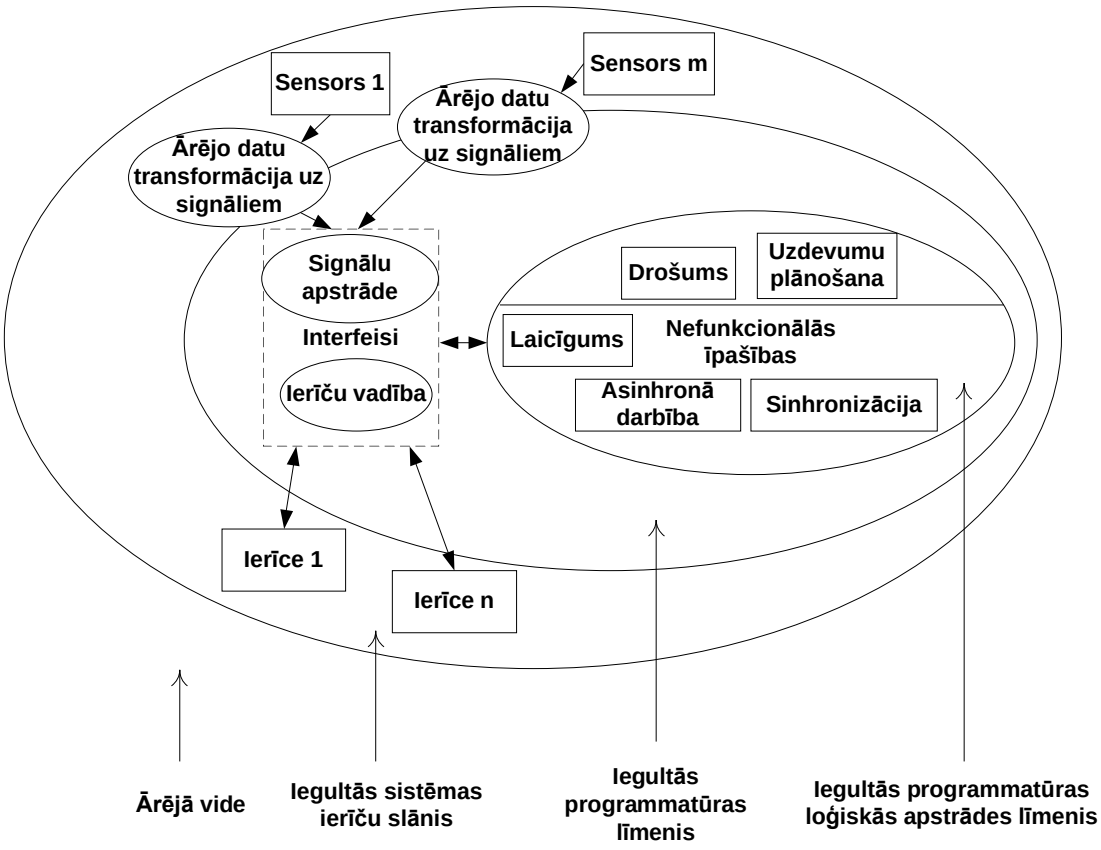
Iegulto sistēmu dažādība ir plaša, taču to vieno neatņemama saskarne ar ārējo vidi un pieslēgto ierīču vadība. Šī specifika arī definē papildus prasības iegulto sistēmu uzbūvei un funkcionēšanas principiem. Iegultām sistēmām ir jānodrošina šādas iespējas [ZIM 2009]:

- Fiziskā sasaiste (angl. *physical coupling*) – jāmāk transformēt fiziskos fenomenus analoga datos (to nodrošina visu veidu sensori).
- Saskarnes (angl. *interfaces*) – jāmāk no analoga datiem dabūt programmatūrai saprotamus datus.
- Laicīgums (angl. *timing*) – jānodrošina laika ierobežojumu specificēšana un to sasniegšana.
- Resursu kontrole (angl. *resource control*) – jāmāk pārvaldīt resursi.
    - Asinhronā darbība (angl. *asynchronisms*) – parasti nodrošina ar pārtraukumu apstrādes mehānismu, kas ļautu paralēli darboties vairākām saskarnēm un programmatūras daļām.
    - Datu sinhronizācija (angl. *synchronization*) – līdzi ar pārtraukumu pārķeršanu ir jānodrošina arī citi sinhronizācijas mehānismi, piemēram, semafori.
    - Uzdevumu izpildes plānošana (angl. *scheduling*) – multiapstrādes sistēmās vienmēr parādās konkurējošie procesi, tāpēc ir jāmāk atbilstoši izstrādātai stratēģijai vienlaicīgi apstrādāt vairākus uzdevumus.
- Ticamība, drošums (angl. *reliability*) – programmatūrai jābūt bojājumiecietīgai (angl. *fault tolerant*) un programmas kompilēšanas laikā ir jāmāk pārķert noteiktā tipa kļūdas (piemēram, tipizēšanas un sintakses kļūdas).

Visas šīs iespējas nav unikālas tieši iegultām sistēmām un dažas no tām piemīt arī citām sistēmām. Tā, piemērām, laicīgums ir reāla laika sistēmu neatņemama sastāvdaļa, savukārt, datu sinhronizācija un uzdevumu plānošana ir obligāta prasība

daudzu procesu sistēmām, bet fiziskās sasaistes līdzekļi ir sastopami ikdienā strādājot ar peli vai tastatūru. Tajā paša laikā, visām iegultām sistēmām visas šīs prasības ir obligātās.

Ņemot vērā iepriekš aprakstītās prasības un sistēmu specifiku, 1.1. attēlā shematiski ir parādīta iegulto sistēmu vispārēja struktūra un to kopīgās īpašības.



1.1. att. Iegulto sistēmu struktūra un īpašības

Shēma ir sadalīta četros līmeņos:

1) Ārējā vide – pie šī līmeņa attiecas viss, kas nepieder un nav savienots ar sistēmu. Iegultā sistēma darbojas šajā vidē, nolasa vides parametrus (ārējas vides temperatūra, gaismas stari, fiziskā iedarbība, citi objekti utt.) un var ietekmēt to.
2) Iegultā sistēma – līmenis reprezentē veselu iegulto sistēmu ar visiem pieslēgtiem aparatūras līdzekļiem (sensori, ierīces, pati iegultā sistēma).
3) Iegultās sistēmas programmas nodrošinājuma līmenis – visa iegultā programmatūra, sakot ar saskarnēm ar ārējo vidi, līdz iegultās sistēmas iekšējiem procesiem un programmām.

4) Iegultās sistēmas loģiskās apstrādes līmenis – šajā līmenī notiek sistēmas vadības procesi, kas strādā ar jau apstrādātiem datiem.

Shēmā speciāli netiek izdalīti procesi, kurus nodrošina operētājsistēma un specifiskā programmas nodrošinājuma uzdevumi, jo iegultās sistēmas var tikt būvētas uz operētājsistēmām un bez tām.

2. līmeņa struktūra atšķirībā no 3. līmeņa satur visas aparatūras ierīces, ar kurām strādā iegultā sistēma. Par ierīču programmas nodrošinājumu jeb dzini atbild konkrētas ierīces izstrādātāji, kas arī nodrošina saskarnes korekto darbību ar ierīcēm atbilstoši noteiktam aprakstam. Savukārt, 3. līmeņa specifika atdala ierīču saskarnes no tās funkcionālās apstrādes. Šīs dekompozīcijas rezultātā autors grib panākt standartizētās funkcionalitātes izdalīšanu un fokusēšanos uz pašas iegultās sistēmas funkcionēšanu un programmas nodrošinājumu. Tas nozīme, ka pēdējais 4. līmenis atbild par sistēmas loģisko uzvedību.

Pēdējā līmenī ir izdalītas iegulto sistēmu nefunkcionālās prasības, kas ir kopīgas visām iegultām sistēmām. Lai nodrošinātu ierīču pārvaldību un nepieciešamo funkciju izpildi, iegultām sistēmām ir jāatbalsta fundamentālās nefunkcionālās prasības šādas specifikas sistēmām. Paralēlie procesi, attālināta vadība un pieslēgtās ierīces pieprasa šādu īpašību realizāciju: sinhronizāciju, asinhrono darbību, laicīgumu, uzdevumu plānošanu un drošumu.

Balstoties uz iepriekš minēto, var secināt, ka iegultām sistēmām ir sekojošie faktori, kas prasa detalizētāko un specifisko testēšanas procesu:

- iegultās sistēmas ir sarežģītas sistēmas [CSYS];
- iegultās sistēmas ir kritiskas sistēmas, no kurām ir atkarīgas cilvēku dzīvības;
- iegultās sistēmas ir patstāvīgas, noslēgtas un dažreiz fiziski nepieejamas;
- tās darbojās heterogēnā dinamiskā vidē;
- to darbību nodrošināšanai ir nepieciešama virkne ar nefunkcionālām prasībām.

Minētie faktori definē specifiskās prasības sistēmu programmatūrai un testēšanas procesam. Tālākai spriešanai par iegulto sistēmu testēšanas procesa īpašībām ir nepieciešams veikt vispārīgo testēšanas metožu un to pielietošanas iespēju analīzi.

1.2 Iegulto sistēmu testēšanas pamatprincipi un metodes

Kā bija minēts iepriekšējā nodaļā iegulto sistēmu specifika izvirza paaugstinātas prasības šādu sistēmu testēšanai, nekā to var prasīt vispārīgā lietojuma programmatūras sistēmas. Tomēr arī iegulto sistēmu testēšanas process bāzējas uz sistēmas testēšanas procesu kā tas ir definēts vispārīgām sistēmām. Līdz ar to šajā nodaļā ir aprakstīti testēšanas pamatprincipi, kas tiek lietoti arī iegulto sistēmu testēšanā.

1.2.1 Testēšanas definīcija un būtība

Testēšana ir sistēmas darbināšanas un analīzes process, ar mērķi gūt pārliecību, ka tā spēs ilglaicīgi funkcionēt noteiktajos apstākļos un izpildīt iepriekš noteiktās darbības. Pēc autora viedokļa izsmeļošākās testēšanas definīcijas ir šādas:

Testēšana ir programmas izpildīšanas un apskatīšanas process ar mērķi atrast kļūdas [MYE 2004] – vispārīgs testēšanas definējums.

Sistēmas/komponentes darbināšanas process ar noteiktiem nosacījumiem, pārbaudot sistēmas/komponentes kādu aspektu. Darbināšanas procesa rezultāti tiek pierakstīti un analizēti [COP 2004] – vairāk formāls definējums, ko piedāvā IEEE Standard 610.12-1990.

Kaut arī šīs definīcijas ir atšķirīgas, taču kopīgais aspekts abās ir tas, ka tās pārredz sistēmas darbināšanu pie definētiem standarta un izņēmumu apstākļiem. Pirmais definējums motivē sistēmas darbināšanu īpašos gadījumos un ar īpašiem datiem, lai atrastu pēc iespējas vairāk programmas kļūdu. Otrā definīcija fokusējas uz pierādījumu vākšanu par sistēmas korekto funkcionēšanu iepriekš definētājos apstākļos. Autora viedoklis ir, ka abas definīcijas ir svarīgas testēšanas procesā un tajās uzstādītais fokuss var tikt definēts atkarībā no testējamās programmatūras.

Neatkarīgi no pielietojamas programmatūras izstrādes metodoloģijas, programmas nodrošinājuma testēšana ieņem tai atvēlēto vietu. Izdala sekojošus testēšanas procesa mērķus [BUR 2003]:

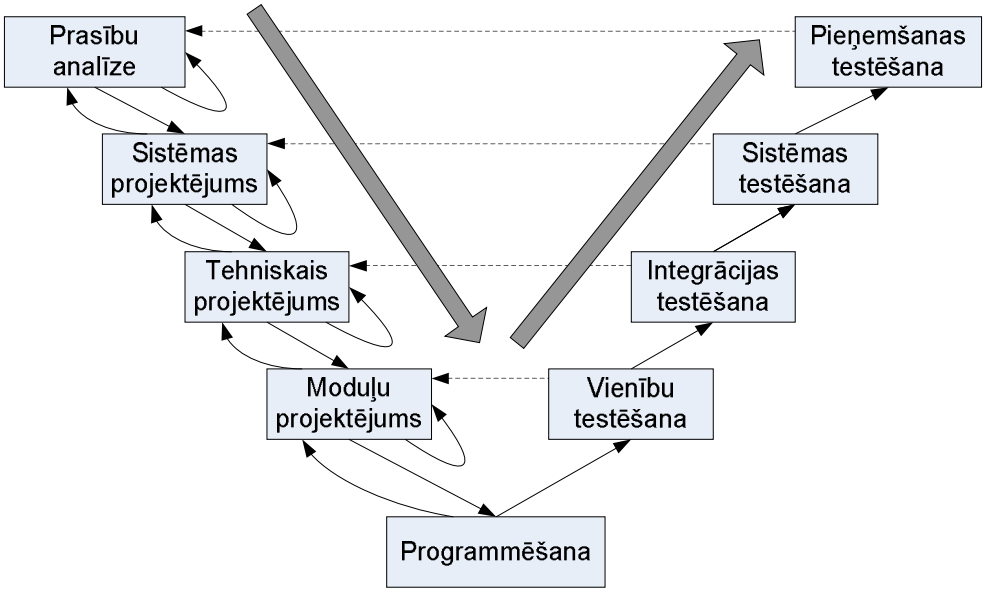
- pārbaudīt, vai izstrādes procesā aplūkojamā posma rezultāti atbilst tā sākumā definētiem noteikumiem un prasībām – verifikācija;

- pārbaudīt, vai izstrādātā programmatūra atbilst iepriekš formulētām prasībām – validācija;
- atrast kļūdas izstrādātā programmatūrā;
- novērtēt, vai izstrādāto produktu var lietot reālajos apstākļos.

Iegulto sistēmu testēšana konceptuāli atbilst vispārīgām un vispārpieņemtām testēšanas procesam un tā mērķiem. Tas sastāv no sekojošiem posmiem [SPI 2007]: plānošana (1. fāze), specificēšana (2. fāze), darbināšana (3. fāze), pierakstīšana (4. fāze) (piemēram, rezultātu dokumentēšana), pārbaude par plānoto darbu izpildi (5. fāze) un slēgšanas aktivitātes (6.fāze) (piemēram, gala rezultātu izvērtēšana).

Pirmā fāze ietver nepieciešamo resursu plānošanu un testēšanas stratēģijas definēšanu: testēšanas metožu noteikšana, sasniedzamo pārklājuma un testu pabeigšanas kritēriju specificēšana, testu strukturēšana un prioritāšu izvirzīšana, kā arī testēšanas atbalsta rīku izvēle un testēšanas vides aprakstīšana [SPI 2007]. Testu specificēšanā atbilstošie testpiemēri tiek definēti, balstoties uz metodēm, aprakstītām testēšanas plānā. Testu darbināšana nozīmē testpiemēru un testu scenāriju palaišana pret testējamo sistēmu. Balstoties uz darbināšanas rezultātiem, izveidotie pieraksti padara testēšanas procesu saprotamu priekš cilvēkiem, kas nav tieši saistīts ar testēšanas procesu (piemēram, pasūtītāji), un nodrošina pieradījumus, ka plānotās testēšanas aktivitātes tika izpildītas saskaņā ar testēšanas plānu. Noslēgumā fāzē tiek analizēta iegūtā informācija par visu testu izpildes rezultātiem, lai fiksētu iegūto pieredzi un, balstoties uz apkopotiem datiem, veiktu analīzi testēšanas procesam [SPI 2007].

Testpiemēru iegūšana un pielietošana notiek 2.-4. soļos. Šajā darbā uzmanība tiek pievērsta tieši šiem soļiem, kas apskata testējamo sistēmu specifiku, tai izvirzītās prasības un dažādās testpiemēru ģenerēšanas metodes. Izstrādājamie testpiemēri, it īpaši lielām sistēmām, tiek ražoti no vairākām perspektīvām jeb fokusēšanas apgabaliem. Programmatūras izstrādes ciklā ir jāveido testpiemēri sistēmas pieņemšanas pārbaudei, atsevišķiem sistēmas moduļiem, izdalītām apakšsistēmām, noteiktām ārējām saskarnēm un citiem apgabaliem. Šādu testēšanas klasifikāciju apraksta populārais programmatūras izstrādes V modelis, kas ir parādīts 1.2. attēlā.



1.2. att. Programmatūras izstrādes V modelis [MYE 2004] [KAN 1999]

Mūsdienās programmatūras izstrāde vairs neatbilst klasiskajam V modelim tā ciešās saites ar ūdenskrituma modeli dēļ [LIV 2005]. Tajā pašā laikā V modeļa prezentētie testēšanas veidi joprojām ir aktuāli un tiem jābūt izpildītiem, neskatoties uz izvēlēto programmatūras izstrādes modeli [SPI 2007]. Sasniedzamais mērķis katrā no testēšanas veidiem ir atšķirīgs, bet tie visi ir virzīti uz kopīgo mērķi – nodrošināt kvalitatīvu programmatūru. Iegulto sistēmu specifika lielākā vai mazākā mērā ietekmē katru no minētiem testēšanas veidiem. Minētājos testos ģenerētiem testpiemēriem ir jāpārklāj funkcionālo testējamo apgabalu, ka arī nefunkcionālās iegulto sistēmu īpašības.

1.2.2 Testēšanas metodes un paņēmieni

Testēšanas paveids, kur notiek testpiemēru izstrāde, tiek saukts par dinamisko testēšanu un paredz programmatūras fizisko darbināšanu. Testēšanas procesa būtība un galvenās idejas rodas testpiemēru specificēšanas jeb 2. posmā, jo būtiski ir daudzpusīgi pārklāt sistēmas funkcionālo apgabalu ar ģenerētiem testpiemēriem. Ir zināms liels dinamiskās testēšanas metožu skaits, kas atkarībā no pieejamās informācijas ļauj ģenerēt testpiemērus. Eksistē sekojošas dinamiskās testēšanas metodes [SPI 2007] [MEY 2004]: sadalīšana ekvivalences klasēs (angl. *equivalence*

class partitioning), robežvērtību analīze (angl. *boundary value analysis*), pētnieciskā testēšana (angl. *exploratory testing*), situācijai piekārtota testēšana (angl. *ad hoc testing*), uz formālām specifikācijām bāzēta testēšana (angl. *formal specification-based testing*), uz vadības plūsmām bāzēta testēšana (angl. *control flow-based testing*) , uz stāvokļu pārejām bāzēta testēšana (angl. *state transition testing*), uz datu plūsmām bāzēta testēšana (angl. *data flow-based testing*), kļūdu minēšana (angl. *error guessing*), gadījuma testēšana (angl. *random testing*), lēmumu pieņemšanas tabulas (angl. *decision table*), uz izsaukumu grafiem bāzēta testēšana (angl. *call graph for code-based testing*), uz plūsmas grafiem bāzēta testēšana (angl. *flow graph for code-based testing*), mutāciju testēšana (angl. *mutation testing*), cēloņu un aktivitāšu grafi (angl. *cause-effect graph*), uz lietošanas gadījumiem bāzēta testēšana (angl. *use case testing*), dūmu testi (angl. *smoke test*), priekšrakstu pārklāšana (angl. *statement coverage*), programmas zaru pārklāšana (angl. *branch coverage*), nosacījumu testi (angl. *condition testing*), programmas ceļu pārklāšana (angl. *path testing*) un citas. Iegultās sistēmas atšķiras savā starpā un katrai sistēmai pielāgo savu testēšanas pieeju atkarībā no sistēmas funkcionēšanas īpatnībām un pārvaldāmām ierīcēm. Tādejādi neeksistē vienotas testēšanas metodes, kas būtu efektīvi pielietojamas visām iegultām sistēmām. Tomēr konkrētas pielāgotas pieejas balstās uz kopīgiem standartiem, testēšanas principiem un metodēm [BRO 2003].

Otrā lielā testēšanas metožu klase ir statiskā testēšana. Tā balstās uz sistēmas un tās saistītu īpašību aprakstu, specifikāciju izpēti un analīzi, fiziski nedarbinot pašu sistēmu. Populārākās statiskās testēšanas metodes ir statiskā analīze un dažādi revīzijas paveidi. Statiskā testēšana ir visai efektīva un ļauj ātri atrast kļūdas pie mazām pašizmaksām [SPI 2007]. Ar statisko testēšanu, atšķirībā no dinamiskās testēšanas, ir iespējams pārbaudīt, ko domā izstrādātāju grupa un vai tiešām visiem komandas dalībniekiem ir vienots un pareizs priekšstats par problēmas vidi un izstrādājamo risinājumu. Statiskā testēšana pārbauda programmatūras uzbūvi, funkcionēšanas apgabalus, dažādus ar programmatūru saistītus aspektus (dokumentācija, apmācība, nodevums, utt.), kā arī izstrādātāju komandas zināšanas. Statiskā testēšana ir neaizvietojams posms sarežģīto sistēmu ražošanā. Iegulto sistēmu statiskā testēšana atbilst vispārīgām procesam un balstās uz tām pašām metodēm, jo statiskās testēšanas procesā verificējamie artefakti piemīt arī iegultām sistēmām.

Viens no populārākiem statiskās testēšanas veidiem ir revīzija. Eksistē liels revīzijas paveidu skaits. Daži no tiem ir [SPI 2007]: caurskate (angl. *walkthrough*),

formālā inspicēšana jeb formālā revīzija (angl. *inspection*), tehniskā revīzija (angl. *technical review*), neformālā revīzija (angl. *informal review*). Minētās metodes pārsvarā balstās uz klientu prasību, sistēmas specifikāciju, risinājuma apraksta analīzi un sistēmas modeļiem. Lietojot aprakstošo dokumentāciju, izstrādāto pirmkodu eksperts jeb ekspertu grupa veic detalizēto analīzi ar vai bez mutiskās apspriešanas un sagatavo secinājumus par apskatīto problēmu. Iegulto sistēmu kontekstā revīzija ieņem līdzvērtīgu un svarīgu vietu izstrādes procesā, jo tās mērķi paliek nemainīgi un aktuāli. Turklāt iegulto sistēmu specifiskās īpašības definē papildus prasības programmatūrai, padarot revīzijas procesu vēl aktuālāku.

Otrais populārākais statiskās testēšanas veids ir statiskā analīze. Galvenā atšķirība no revīzijas ir rīku pielietošanā. Statiskā analīze balstās uz rīku pielietošanu kāda noteikta objekta pārbaudei. Teksta dokumentu pārbaudei bieži pielietots statiskās analīzes rīks ir pareizrakstības pārbaudītājs, kas nodrošina gramatisko kļūdu meklēšanu tekstā. Vēl viens statiskās analīzes rīku veids ir modeļu verificēšanas rīki, kas nodrošina modeļu korektumu. Šo rīku pieeja ir līdzīga visiem statiskās analīzes rīkiem: par verificējamo objektu tiek paņemts kāds formāli aprakstīts objekts (varētu būt MS Word dokuments, parasts teksta vai arī XML/HTML datne utt.), tad rīks, pārvaldot to konkrētu formātu veic objekta pārbaudi atbilstoši iepriekš definētā formāta specifikācijai. Priekšnosacījums šādu rīku pielietošanai ir verificējamo objektu formālais pieraksts. Sistēmu modeļi tiek prezentēti noteiktā formāta veidā, tāpēc tie var kļūt par statiskās analīzes objektiem. Iegulto sistēmu īpašības arī var tikt modelētas ar dažādām notācijām, prezentētas noteiktajos formātos un statiski analizētas ar iepriekš sagatavotiem rīkiem.

1.2.3 Testpiemēra uzbūve

Testpiemēri (angl. *test case*) ir dinamiskās testēšanas pamata koncepti, balstoties uz kuriem notiek sistēmas darbināšana pie dažādiem apstākļiem. Viens no dotā pētījuma uzdevumiem ir piedāvāt metodi automātiskai testpiemēru ģenerēšanai, līdz ar to tie ir būtiskākie šī darba elementi.

[IEEE 829] piedāvā šādu testpiemēra definīciju:

Testpiemēra apraksts ir dokuments, kas testējamai programmatūrai nosaka ievaddatus, prognozējamo rezultātu un darbināšanas nosacījumu kopu [IEEE 829].

Minētā definīcija ir vispārīgā pielietojuma testpiemēra apraksts, kas atspoguļo tā būtību un nozīmi. Turpat [IEEE 829] definē testpiemēra dokumenta struktūru, iekļaujamus elementus un to secību:

a) testpiemēra identifikators – unikāls testpiemēra identifikators lai nodrošinātu to izsekojamību un vadību;
b) testējamais objekts – identificē un īsi apraksta vienību un īpašību, kas tiek eksaminēta testēšanas procesa gaitā. Definējot testējamo objektu, tiek sniegtas atsauces uz prasību un projektējuma specifikācijām, lietotāja un citām instrukcijām;
c) ievaddatu apraksts – apraksta ievades parametru, kas tiek lietots testpiemēra palaišanas brīdī. Daži ievaddati var būt atsevišķi atribūti ar noteiktām vērtībām, savukārt, citi, piemēram, statiskās tabulas vai transakciju datnes, var tikt specificēti tikai ar atbilstošu nosaukumu. Pie ievaddatu apraksta tiek definētas arī to savstarpējas atkarības, ieskaitot laicīguma aspektu;
d) izvaddatu apraksts – detalizētā formā tiek specificēts programmas darbības rezultāts un iegūtie dati, ieskaitot mainīgo vērtības un formātus, ka arī laika atribūtus;
e) darbināmas vides īpašības – nosaka prasības programmatūrai, aparatūrai un atkarībās ar citam specifiskām sistēmām, kas var ietekmēt testpiemēra apstrādes procesu;
f) speciālas instrukcijas - tiek definētas citās palaišanas un atsekošanas darbības, kas var tikt pieprasītas, lai korekti izpildītu aprakstāmo testpiemēru. Šādas instrukcijas tiek izpildītas, lai iegūtu testējamās programmatūras uzvedību pie nestandarta apstākļiem, piemēram, lai simulētu galējo izpildes termiņu (angl. *deadline*), noraut komunikāciju kanālus, vai realizētu kādu citu specifisku gadījumu, kas reti tiek sastapts pie standarta apstrādes;
g) starpgadījumu atkarības – tiek aprakstītas atkarības starp citiem testpiemēriem, norādot priekšnosacījumus tekošā testpiemēra izpildei.

Testpiemēra dokumentā minētiem elementiem jābūt izvietotiem tajā pašā secībā, kādā tie ir definēti augstāk. Papildus šiem elementiem, dokuments var saturēt arī citu papildus aprakstu, kas tiek novietoti pēc šiem obligātiem atribūtiem.

IEEE 829 standarts [IEEE 829] fokusējas uz vispārīgo testpiemēra specificēšanu, kas pēc būtības ir norādījumi testēšanas dokumentācijas izstrādei. Savukārt, testpiemēri automatizētajā dinamiskajā testēšanā var atšķirties no minētiem manuālās testēšanas aprakstiem. Kaut arī testpiemēra būtība paliek nemainīga (notestēt izstrādātas sistēmas funkcionēšanu pie atbilstošiem ievaddatiem un definētiem nosacījumiem), testpiemēru struktūra var atšķirties no iepriekš aprakstītas konstrukcijas. Par standartam pietuvināto struktūru var uzskatīt UML testēšanas profilu (angl. *UML Testing profile*), kas piedāvā konceptuālo testēšanas elementu modeli lietošanai automatizētajā testēšanā un kas ir detalizētāk apskatīts šī darba 3.1.2 nodaļā. Aprakstot testēšanas procesā lietojamus elementus, profils nesniedz detalizētu informāciju par šo elementu uzbūvi un atstāj to konkrētās testēšanas automatizācijas izstrādātājiem. Tajā pašā laikā piedāvātais modelis satur lielāku daļu no IEEE 829 standartā aprakstītiem elementiem, papildinot un sadalot tos detalizētākajos klasifikatoros.

1.2.4 Testēšanas procesa automatizācija

Papildus rīku pielietošana un testēšanas procesa automatizēšana dod iespēju racionalizēt programmu un ar tām saistītu aspektu verificēšanu un validēšanu. Testēšanas procesa automatizācija ir dotā pētījuma tēma, jo viens no uzstādītiem darba mērķiem ir testpiemēru ģenerēšana.

Testēšanas procesa automatizācija ir vispārīgs jēdziens un ietvers sevī dažāda veida darbību paātrināšanu. Pie šādām darbībām tiek attiecinātas: testēšanas plānu un scenāriju veidošana un vadība [GRI 2008d], programmas kļūdu reģistrēšana, testa datu un datu plūsmu ģenerēšana [GRI 2009b] [GRI 2010], pirmkoda analīze, programmu darbināšana [GRI 2011c] un citas. Procesu paātrināšana un manuālo darbību vai soļu aizvietošana tiek iegūta ar dažāda veida testēšanas rīkiem. Ir zināmas vairākas pieejas testēšanas rīku klasifikācijai un aprakstīšanai, bet pēc autora viedokļa [BAT 2008] sniedz strukturētu un plašu rīku sadalīšanu septiņās kategorijās:

- testēšanas vadības rīki (angl. *test management tools*) – iekļauj testpiemēru, prasību, konfigurācijas vadības rīkus, ka arī kļūdu atsekošanas un citus rīkus. Šīs kategorijas rīki ļauj atvieglot testēšanas un projektu vadības procesus.
- Kļūdu injicēšanas rīki (angl. *fault seeding and fault injection tools*) – šādas programmas ir spējīgas mainīt pirmkodu tā, lai oriģinālais pirmteksts strādātu ar kļūdām, kur mainītās programmas tiek sauktas par mutantiem. Šāda uzvedība tiek iegūta ar mērķi pārliecināties, ka uzrakstītie testpiemēri ir spējīgi atrast nepareizo sistēmas uzvedību, un pārbaudīt citu sistēmas daļu funkcionēšanu pie šādiem apstākļiem.
- Simulācijas un emulācijas rīki (angl. *simulation and emulation tools*) – pirmie tiek lietoti lai simulētu atbildes no programmatūras vai aparatūras komponentēm, kas sadarbojas ar testējamo programmatūru. Emulatori ir simulācijas rīku apakškopa, kas pilnīgi vai daļēji imitē kādas aparatūras uzvedību, tādejādi nodrošinot interfeisu (angl. *interface*) testēšanu ar šīm ierīcēm.
- Statiskās un dinamiskās analīzes rīki (angl. *static and dynamic analysis tools*) – statiskās analīzes rīki eksaminē pirmkodu, nedarbinot to, kur rezultātā var tikt atrasts riskants un nedrošs kods. Savukārt, dinamiskās analīzes rīki analīze programmas darbināšanas laikā un meklē kļūdainus atmiņas rādītājus, atmiņas noplūdes un citas programmas kļūdas.
- Ātrdarbības rīki (angl. *performance tools*) – ļauj pārbaudīt programmu funkcionēšanu pie dažādām slodzēm.
- Tīmekļa rīki (angl. *web tools*) – plaša rīku kopa, kas veidota tīmekļa lietotņu testēšanas atbalstam. Šie rīki tiek lietoti lai pārbaudītu hipersaites, konstruēt lapas koku, kā arī novērot lietotāju aktivitātes, ierakstīt to uzvedību ar laicīguma aspektu un atkārtot to. Rīki, kas tiek lietoti lietojamības testēšanā, arī tiek attiecināti pie šīs kategorijas.
- Traucējummeklēšanas rīki (angl. *troubleshooting*) – tiek lietoti detalizētai programmas darbības analīzei, kad ir nepieciešams apskatīties programmā inicializētus mainīgus, iegūt informāciju par

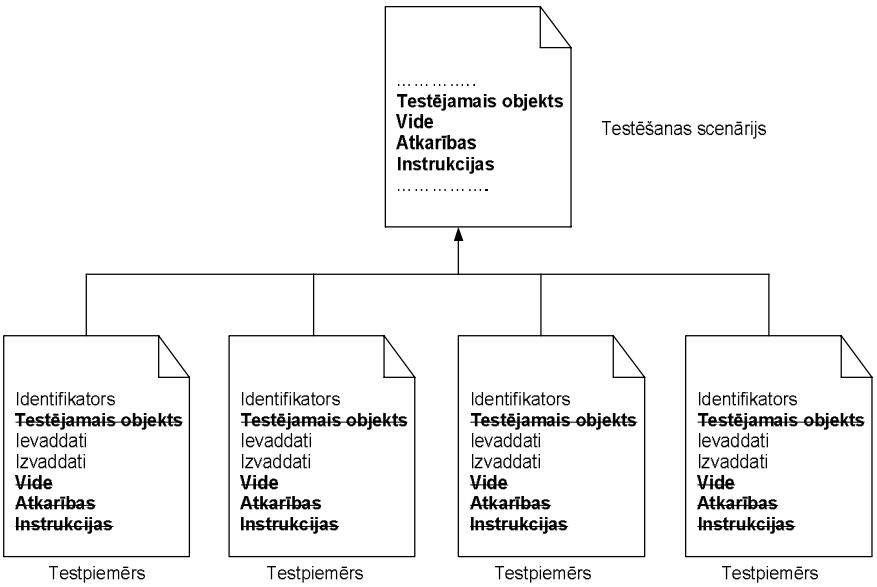
programmu atteikumiem vai arī sekot solis pa solim līdzi programmas izpildei.

Visas minētas rīku kategorijas tiek lietotas iegulto sistēmu testēšanā. Testpiemēru ģenerēšanas rīkus var attiecināt pie statiskās analīzes rīkiem, jo tie neparedz programmu darbināšanu. Testpiemēri tiek ģenerēti balstoties uz definētiem likumiem, kas nosaka testpiemēra īpašības, un sistēmas funkcionēšanas aprakstu. Modeļvadāmajā programmatūras izstrādē testpiemēru ģenerēšanas rīki analīzē sistēmas modeļu elementus, to atribūtus un saites starp tiem un veido atbilstošus gadījumus to verificēšanai. Šādā vidē rīku koncepts paliek nemainīgs un galvenā atšķirība no vispārējiem testpiemēru ģenerēšanas rīkiem ir analizējamā objekta prezentācijas dažādība.

Eksistē vispārīgie testēšanas rīki, kas paredz testpiemēru ģenerēšanu un to darbināšanu [STH 2001]. Šādus rīkus var attiecināt pie divu kategoriju rīkiem – statiskās analīzes rīkiem, jo tie eksaminē sistēmas specifikācijas bez programmas palaišanas, un simulācijas rīkiem, nodrošinot testējamās sistēmas vai to komponenšu izsaukšanu pie specifiskiem nosacījumiem un ar noteiktiem ievaddatiem.

Testpiemēru ģenerēšanas procesā to pierakstīšanas formāts var atšķirties starp rīkiem. Kaut arī eksistē UML testēšanas profils, kas piedāvā konceptus un struktūru testpiemēru un to aspektu vadībai, tomēr eksistējošo rīku ģenerēto gadījumu pieraksts atšķīrās starp tiem.

Analizējot dažāda veida testēšanas automatizācijas rīkus [GRI 2008d] [GRI 2009b] [GRI 2010] [GRI 2011c], kas tieši ir saistīti ar testpiemēriem, var secināt, ka kaut arī atsevišķā testpiemēra koncepts ir līdzīgs IEEE 829 aprakstītai struktūrai, tam piemīt noteiktā specifika. Tā būtība ir definēt testpiemēriem pēc iespējas mazāk vispārīgas informācijas un kopīgus datus aprakstīt citā līmenī, piemērām, testēšanas scenārija vai citā izdalītā dokumentā. Pateicoties šādai testpiemēra apraksta minimizēšanai un dublējošās informācijas vienreizējo aprakstīšanu, ir iespējams samazināt testpiemēra aprakstāmus atribūtus un tas, savukārt, var ietekmēt uz testpiemēru izstrādāšanas un uzturēšanas laiku. Šī specifika detalizētāk ir parādīta 1.3. attēlā, kur vairāku testpiemēru atribūti tiek iznesti testēšanas scenārijā līmenī.



1.3. att. Testpiemēru kopīgu atribūtu noslēgšana testēšanas scenārija līmenī

1.3. attēls parāda, ka vairāku testpiemēru kopīgie atribūti tie definēti testēšanas scenārijā līmenī. Automatizētās testpiemēros, kad rīki darbina testējamo sistēmu pie noteiktiem apstākļiem, testējamā objekta, vides, atkarību starp citiem testpiemēriem apraksts un papildus instrukcijas var tikt izdalītas no testpiemēriem un definētas kopīgajā atsevišķā vietā.

Aprakstītā pieeja tiek pielietota gadījumos, kad apskatāmai testpiemēru kopai ir vienādi izdalāmie atribūti un to vienīga atšķirība ir dažādos ievaddatos. Šāds testpiemēru izstrādāšanas princips tiek saukts par uz datiem balstītu testēšanu (angl. *data-driven testing*) [BAT 2008] un bieži tiek pielietots automatizētajā testēšanā [GRI 2008d] [GRI 2009] [GRI 2010] [GRI 2011c].

Iegulto sistēmu testēšanas automatizācija ar testpiemēru ģenerēšanu paredz nefunkcionālo īpašību verificēšanu pie dažādiem apstākļiem neapskatot sistēmas funkcionālo aspektu. Tas nozīme, ka ģenerētie testpiemēri netiek balstīti uz ievaddatu dažādību, bet fokusējas uz programmatūras apstrādes specifiku. Ņemot vērā šo aspektu var secināt, ka iepriekš aprakstītais testpiemēru vienkāršotais pierakstīšanas formāts neatbilst iegulto sistēmu nefunkcionālo īpašību automātiskai testpiemēru ģenerēšanai un par pamatu pēdējām var tikt izvēlēts eksistējošais formāls testpiemēra pieraksts, ko piedāvā UML testēšanas profils.

1.2.5 Eksistējošie pētījumi par iegulto sistēmu testēšanu

Vairākās praksē pielietojamās iegulto sistēmu testēšanas metodes balstās uz klasiskām pieejām, fokusējoties uz sistēmas funkcionālā apgabala verificēšanu. Par šādām pieejām kalpo sadalīšana ekvivalences klasēs, rebēžvērtību analīze un citas plaši pielietojamās atzītās metodes.

Ir zināmas arī atsevišķas vai no vispārīgām pieejām atvasinātās iegulto sistēmu testēšanas metodes, kas tiek efektīvi pielietotas praktiskajā sistēmu izstrādē. [WU 2007] piedāvā dinamiskās testēšanas paņēmienu, kas fokusējas uz pirmkoda pārklājumu. Pielietojot automatizācijas rīkus, tiek nodrošināta sistēmas darbināšana ar vairākiem ievaddatiem un tiek mērīts pirmkoda pārklājuma līmenis. [WU 2007] savā darbā pārnesa koda pārklājuma noteikšanas metodi no vispārīgām uz Linux operētājsistēmām bāzētām sistēmām uz iegultām. Šāda pieeja ļauj uzlabot un papildināt funkcionālo testēšanu ar vairāku gadījumu verificēšanu, tomēr iegulto sistēmu specifiskās īpašības paliek nepārklātas.

Cita pētnieku grupa no DaimlerChrysler korporācijas piedāvā savu evolucionāro iegulto sistēmu testēšanas pieeju [STH 2001], kas arī balstās uz sistēmas darbināšanu ar dažādiem ieejas datiem ar mērķi verificēt pēc iespējas vairāk kvalitatīvus testpiemērus. Labāko testpiemēru identificēšana notiek, balstoties uz heiristiskās meklēšanas paņēmieniem, analizējot sistēmas darbības rezultātu. Šis pētījums un praktiskais pielietojums līdzīgi kā [WU 2007] ir fokusēts uz funkcionālo testēšanu ar vairākiem testpiemēriem, lai nodrošinātu plašāku ieejas datu pārklājumu testēšanas rezultātā. Šādas pieejas ir efektīvas praktiskajā pielietojumā, jo tās paredz automātisko testpiemēru ģenerēšanu, sistēmu darbināšanu un automātisko rezultātu analizēšanu. Tomēr ir jāatzīmē, ka abas metodes neapskata iegulto sistēmu specifiskās īpašības un neparedz speciālas testpiemēru kopas ģenerēšanu ar mērķi verificēt tās.

Ir zināmās arī formālās metodes iegulto sistēmu īpašību verificēšanai. Šādu metožu popularitāte un attīstīšana notika 80.-90. gados. Metodes fokusējās uz konkrētu īpašību verificēšanu, piemēram, laicīguma, paralēlās apstrādes, uzdevumu plānošanu, sinhronizēšanu un citām. Izdalītā uzdevuma risināšanas dēļ, katra metode parasti paredzēja kādas konkrētās valodas lietošanu, kas padara kopīgu izstrādes procesu sarežģītāku un neparedz vienotu principu pielietošanu sistēmu implementēšanas un testēšanas laikā. Par vienu no šādām metodēm var uzskatīt

ASTRAL formālo specificēšanas valodu [COE 1997], kas tika izstrādāta lai nodrošinātu reālā laika sistēmu formālo izstrādi un to īpašību verificēšanu. Metode paredz reāla laika sistēmas darbības modelēšanu ar vairākām stāvokļu mašīnām un vienu kopīgu globālu aprakstu. Globālais apraksts satur konstantu, mainīgo un datu tipu deklarācijas, kas var tikt koplietoti starp dažādiem procesu tipiem. Katra stāvokļu mašīnas specifikācija apraksta noteiktu procesa tipu, kas var tikt statiski vairots vairākās instancēs. Savukārt, katra procesa instance asinhroni un paralēli var izsaukt cita procesa instances. Implementējot stāvokļu mašīnas ar laika specificēšanu, tiek iegūts formāls sistēmas darbības apraksts, kas var tikt automātiski verificēts, ar to nodrošinot specifikācijas automātisko pārbaudi. Valodas praktiskai pielietošanai eksistē ASTRAL programmatūras izstrādes vide (angl. *software development environment* jeb *SDE*), kas nodrošina sistēmu specificēšanu, analīzi, izstrādātas specifikācijas verificēšanu un specifikācijas atkārtotu lietošanu. Kaut arī piedāvātā valoda ļauj definēt vairāku līmeņu uz komponentēm dalītu sistēmas struktūru, tās ierobežojumu un trūkumu dēļ šī valoda nevar konkurēt ar UML modelēšanas valodu iegulto sistēmu specificēšanai. Metode ļauj specificēt sistēmas darbību pie vienkāršiem laicīguma ierobežojumiem, savukārt, komplicētus laika ierobežojumus, kas ir aprakstīti 1.3.3 sadaļā, valoda ASTRAL nevar nodrošināt.

Viena no vecākām un populārākām paralēlās darbības un sinhronizācijas modelēšanas notācijām, Petri tīkls (angl. *Petri net*), formāli apkopotā veidā tika piedāvāta tālajos 70-80-tājos gados vairākās publikācijās un izdevumus [REI 1985] [MUR 1989] [WAN 1998]. Savukārt, pirmie pētījumi datējas ar 60-tajiem gadiem [PET 1966]. Klasisks Petri tīkls ļāva formāli aprakstīt daudzu procesu sistēmu uzvedību, kā arī piedāvāja grafisko notāciju modeļa prezentēšanai. Pateicoties šīm divām priekšrocībām, metodes pielietošana ļāva nodrošināt formālo izstrādātā sistēmas modeļa verifikāciju un uzskatāmo sistēmas darbības prezentāciju. Notācijas pieejamība un popularitāte veicināja tās attīstību un turpmākajos gados tika izstrādāti daudzie atbalsta rīki, kas nodrošina modeļu izveidi un to automātisko verificēšanu. Tajā pašā laikā tiek piedāvāti vairāki notācijas paplašinājumi, piemēram, krāsainie Petri tīkli (angl. *coloured Petri net*), kas izdala un specificē programmatūras dažādu datu tipu mainīgu apstrādi, ar laika atribūtiem paplašinātie Petri tīkli (angl. *timed Petri net*) un citus. Pētnieku grupa (Luis Alejandro Cortés, Petru Eles un Zebo Peng) no Linčepingas Universitātes Zviedrijā izgudroja un pielietoja iegultām sistēmām pielāgotu Petri tīkla modeli. Šo Petri tīkla paplašinājumu viņi nosauca par PRES+

modeli (angl. *Petri net based Representation for Embedded Systems*). Iegultām sistēmām pielāgotais modelis ir modificēts un ar laika parametriem paplašināts klasiskais Petri tīkla modelis [COR 2000]. PRES+ notācija ļauj definēt un verificēt pamata laika ierobežojumus reālā laika un iegultām sistēmām.

Kaut arī Petri tīklu pielietošana bija un ir vispāratzītā prakse, tai piemīt virkne ar problēmām un ierobežojumiem. Viena no būtiskām Petri tīklu problēmām ir modeļa automātiskā verificēšana un lēmumu pieņemšana par galēja stāvokļa sasniegšanas, ņemot vērā bezgalīga cikla iestāšanos. Kaut arī ir pieejami vairākas šīs problēmas risinājumi [KOS 1982] [LAM 1992], tie nenodrošina precīzu programmas ceļu verificēšanu [ATI 2009]. Tajā pašā laikā Petri tīklu modeļos specificēta informācija ierobežojas ar paralēlo procesu modelēšanas specifiku un modeļi nav pietiekami iegulto sistēmu nepieciešamo nefunkcionālo īpašību modelēšanas.

Pēdējos gados parādās arvien vairākas testēšanas metodes, kas atbalsta MDA principus. [ZAN 2009] piedāvā uz modeļiem balstītu pieeju automatizētam testēšanas procesam. Aprakstīta metode detalizēti apskata dažādas pieejas testpiemēru darbināšanai un rezultātu atsekošanai un fokusējas uz reālās sistēmas darbināšanas realizāciju. Pētījums piedāvā oriģinālo un daļēji automatizēto sistēmas verificēšanas pieeju, sākot ar testpiemēru ģenerēšanu un beidzot ar iegūto rezultātu verificēšanu. Minēta metode nodrošina ieejas datu ģenerēšanu un sistēmas funkcionalitātes pārbaudi, balstoties uz sistēmas sagatavotiem modeļiem. Šis metodes autors apraksta iegulto sistēmu testēšanas metodi ar MiLEST ietvara palīdzību, kas ir balstīts uz MATLAB, Simulink un Stateflow komerciāliem rīkiem. Metodes koncepts paredz signālu pielietošanu sistēmu darbināšanai (nodrošinot ieejas datus) un rezultāta atsekošanai (t.i. iespēju nolasīt sistēmas stāvokļus un uzvedību). Darbā piedāvātais ietvars nodrošina sistēmas darbināšanu ar iepriekš sagatavotiem testa datiem un reālā laikā tās uzvedības atsekošanu. Kaut arī aprakstīta metode ir domāta iegulto sistēmu testēšanai, tā pārsvarā koncentrējas tieši uz praktisko funkcionālo prasību testēšanas procesu un tā realizāciju. Iegulto sistēmu īpašības, tās modelēšanas un reprezentēšanas iespējas, kā arī nefunkcionālo prasību testpiemēru kopas ģenerēšana netiek apskatīta minētājā pētījumā.

Šajā apakšnodaļā uzskaitītie pētījumi iegulto sistēmu testēšanas jomā pārsvarā fokusējās uz sistēmu funkcionālo īpašību verificēšanu, nodrošinot ātrāku un plašāku testpiemēru ģenerēšanu un izpildi. Savukārt, metodes, kas paredz nefunkcionālo īpašību verificēšanu, balstās uz vecām modelēšanas notācijām, tādejādi fokusējoties

uz konkrētas īpašības pārbaudi, nelietojot modernās universālās modelēšanas iespējas un rīkus. Šādu apgalvojumu apstiprina arī citi pētnieki, kas analizēja dažādas metodes un pieejas gan funkcionālo, gan nefunkcionālo īpašību verificēšanai un to sarežģītību iegultajās sistēmās. Par tādiem var minēt iegulto sistēmu projektēšanas procesa pētniekus [ACK 2008], iegulto sistēmu kompilatoru realizācijas metodes autorus [KUG 2008], kā arī MARTE ietvara autorus [PER 2010] u.c. Šī promocijas darba autora pētījums fokusējās tieši uz sistēmas nefunkcionālo īpašību testēšanu, kas varētu tikt atbalstīta ar modelēšanas un izstrādes rīku lietošanu promocijas darba autora piedāvātas metodes kontekstā.

1.3 Iegulto sistēmu īpašības, to modelēšanas un testēšanas paņēmieni

Iegulto sistēmu specifikas verificēšanai vispārīgās metodes ir nepietiekamas un šim nolūkam pielieto specifiskus paņēmienus, kas ir fokusēti konkrētu īpašību verificēšanai. Iegulto sistēmu nefunkcionālās īpašības ietver asinhrono darbību, sinhronizāciju, laicīgumu, uzdevumu plānošanu un drošumu. Šīs īpašības ir speciāli izceltas 1.1.attēlā, definējot iegulto sistēmu struktūru un funkcionēšanas īpašības. tieši šo īpašību izpēte testēšanas uzdevumā ir darba autora pētījuma objekts. Lai nodrošinātu korektu sistēmas funkcionēšanu un izvairītos no kļūdām šo īpašību realizācijā, tās ir nepieciešams modelēt un verificēt . Tajā pašā laikā sistēmas modeļu esamība ir pamata prasība MDA programmatūras izstrādes konceptos.

Asinhronās darbības, sinhronizācijas un laicīguma īpašības ir neatņemamas dažāda veida iegultās programmatūras īpašības. Šīs īpašības tiek implementētas programmatūrās abos gadījumos, kad tiek un netiek lietota operētājsistēma. Savukārt, uzdevumu plānošana pārsvarā tiek attiecināta uz operētājsistēmas funkcionalitāti, kā viena no īpašībām, kas ir jānodrošina. Tieši tāpēc mūsdienās arvien biežāk iegulto sistēmu programmatūra neimplementē šo īpašību un atstāj to operētājsistēmas pārziņā. Drošuma īpašību nodrošināšana ir komplicēts un joprojām pētīts jautājums. Kaut arī pastāv konkrētas metodes drošuma izskaitļošanai un modelēšanai, no implementācijas viedokļa šī īpašība joprojām ir atsevišķs pētniecības objekts. Lai pietiekami detalizēti izklāstītu iegulto sistēmu nefunkcionālo īpašību verificēšanu un tajā pašā laikā fokusējoties uz aktuālākām nefunkcionālām īpašībām, darba autors promocijas darba

ietvaros apskata asinhrono darbību, sinhronizāciju un laicīgumu, kas detalizēti ir aprakstītas 1.3.nodaļā.

1.3.1 Asinhronā darbība

Asinhronisms ir attiecība starp divām vai vairākām aktivitātēm, kas ir neatkarīgas laikā. Asinhronā darbība ir sistēmu tehniskā īpašība, kas tiek lietota, lai nodrošinātu nepieciešamo funkciju izpildi. Iegulto sistēmu prasības iekļauj vairāku pieslēgto ierīču vadību un var iekļaut paralēlo uzdevumu izpildi, kas attiecas uz vidēja un liela izmēra iegultām sistēmām. Šādas funkcionālās prasībās prasa programmatūras izstrādātājiem implementēt sistēmas, balstoties uz asinhronās parādības principiem.

Lai veiksmīgi realizētu asinhronās darbības mehānismu izstrādājamā programmatūrā, ir nepieciešams to projektēt programmatūras izstrādes sākuma fāzē. Ir pieejami daži pētnieciskie un praktiskie darbi par šīs parādības specificēšanu un realizēšanu [YAK 1996] [ZIM 2009] [HAL 2006]. Balstoties uz minēto darbu analīzi, tiek iegūts saraksts ar tehniskām īpašībām, kas realizē asinhrono darbību. Šis saraksts ļauj izvērtēt visus aspektus, kas ir nepieciešami asinhronās darbības projektēšanai. Šis saraksts iekļauj sevī šādas tehniskās īpašības:

- pārtraukumi un to vadība – viennozīmīga pārtraukumu identificēšana (pārtraukumu vektori un pārtraukumi steki), grupēšana, šifrēšana, prioritātes piešķiršana, pārtraukumu avoti, to ieslēgšana un izslēgšana;
- izņēmumu apstrāde – izņēmumu definēšana un pārtveršana;
- konteksta pārslēgšana procesora līmenī – procesora stāvokļa saglabāšana un atjaunošana, uzdevumu pārslēgšana;
- vairāku procesu paralēla darbība – vairāku procesu sistēmas, vairāku ieroču pārvaldība, pavedieni, procesu savstarpējā sadarbība;
- kritiskā sekcija – programmas ar kritiskām sadaļām, atomārās darbības;
- laicīguma aspekts – laika ierobežojumi.

Visas šīs īpašības, izņemot konteksta pārslēgšanu procesora līmenī, ir attiecināmās uz iegulto sistēmu programmatūras realizāciju. Savukārt, konteksta pārslēgšana ir uzskatīta par vienu no operētājsistēmas izpildāmām funkcijām, kas ir ārpus šī darba pētījuma robežās.

Izņēmumu un pārtraukumu apstrāde ir ierasts mehānisms pieslēgto ierīču vadībai. Izņēmumi parasti tiek definēti un apstrādāti lietojuma programmā, bet pārtraukumi aparatūras vai operētājsistēmas līmeņos. Pārtraukumus ir iespējams apstrādāt četros līmeņos: konkrētās ierīcēs, kopējā sistēmas pārtraukumu vadības, operētājsistēmas un programmas līmeņos. Neatkarīgi no izvēlēta līmeņa, lai nodrošinātu pārtraukumu vadību ir jāizpildās trim prasībām [ZIM 2009]:

- uzstādīt prioritātes atbilstoši katram pārtraukuma identifikatoram; grupēšana, šifrēšana, prioritātes piešķiršana, pārtraukumu avota ieslēgšana un izslēgšana;
- nodrošināt konteksta pārslēgšanu procesora līmenī: procesora stāvokļa saglabāšana un atjaunošana, uzdevumu pārslēgšana;
- identificēt pārtraukumus; pārtraukumu vektori un pārtraukumi steki.

Aparatūras pārtraukumiem un asinhroniem ziņojumu apmaiņas uzdevumiem tiek lietoti signāli (signāls ir asinhronais ziņojums, kādi strukturēti dati, kuri tiek pārsūtīti no viena procesa uz otru [JEN 2011]). Unix veida operētājsistēmās eksistē ap 30 signālu veidu un viens no tiem ir SIGINT, kas nodrošina jau minēto pārtraukumu vadību. Vēl viens no populārākiem signāliem ir SIGKILL, ar kuru palīdzību Unix veida operētājsistēmās var tikt pārtraukts jebkurš process.

Iegultās sistēmās ar asinhrono vidi tiek lietotas atomārās darbības, kas palielina sistēmas drošību un stabilitāti. Darbību sauc par atomāru, ja izpildās sekojoši nosacījumi:

- esošajam procesam nav informācijas par jebkādu citu aktīvu procesu un jebkāds cits aktīvs process ”nezina” par palaižamo darbību;
- nav komunikācijas ar citiem procesiem;
- nevar atpazīt ārējo stāvokļu izmaiņas un netiek atklātas arī savas stāvokļu izmaiņas, kamēr darbība netiek pabeigta;
- darbība tiek pieņemta par nedalāmu un momentānu.

Atomāra aktivitāte vai nu izpildās pilnībā, vai neizpildās vispār, kā arī izpildes rezultāts ir nepareizs, ja kaut vienas aktivitātes daļas rezultāts ir aplams.

Pētījuma ietvaros tiek analizētas iegulto sistēmu īpašību modelēšanas iespējas. Ņemot vērā augstāk minētas asinhronās darbības īpašības, grafiskajām notācijām ir jāspēj modelēt šādus aspektus:

- attēlot notikumus un pārtraukumus;

- modelēt kritisko sekciju;
- modelēt paralēlo darbību;
- attēlot notikumu un pārtraukumu parametrus (prioritāte, notikuma numurs/nosaukums, notikuma avots, pārtraukuma apstrādes sekcija);
- attēlot laiku;
- modelēt notikumu apstrādātājus.

Asinhronās darbības modelēšanai tiek pielietoti Petri tīkli un to atvasinājumi, kā arī standartizētās UML diagrammas. Pašlaik zināmas un biežāk pielietotās asinhronās parādības modelēšanas notācijas ir sekojošas:

- Petri tīkls (angl. *Petri net*) – ir bipartīts orientēts grafs, kura virsotņu kopa ir sadalīta divās apakškopās un neeksistē loks, kas savienotu divas virsotnes no vienas apakškopas [GIR 2002] [JEN 1997];
- PRES+ modelis – klasisks Petri tīkla modelis papildināts ar laika atribūtiem [COR 2000];
- signālu pāreju grafi (angl. *signal transition graphs*) – no Petri tīkla atvasinātā grafiskā notācija, kas primāri tika piedāvāta procesoru darbības modelēšanai [YAK] [YAK 1996];
- Petri diagrammas (angl. *Petri charts*) – stāvokļu diagrammas un Petri tīklu kombinējums, kas vienā diagrammā ļauj definēt dažādu abstrakcijas līmeņu modeļus [HOL 1995];
- laicīguma Petri tīkli (angl. *Timed Petri net*) – Petri tīkls, kura pārejas starp stāvokļiem ir papildinātas ar laika intervāliem, specificējot laika ierobežojumus pāreju darbināšanai [WAN 1998] [GHE 1989];
- UML secību un stāvokļu diagrammas – klasiskās UML diagrammas, kas ļauj modelēt paralēlu procesu darbību un sistēmas objektu stāvokļus [PIL 2005].

UML 2.x versijas veicināja stāvokļu un secību diagrammu papildināšanu ar laicīguma īpašībām un ir spējīgas reprezentēt sistēmu dinamisko uzvedību ar laika ierobežojumiem. Tajā pašā laikā UML diagrammas ir spējīgas modelēt iepriekš minētās prasības asinhronās darbības prezentēšanai, ieskaitot paralēlus procesus, kritiskās sekcijas, signālus un to atribūtus.

Asinhronās darbības verificēšanas metodes var iedalīt 2 grupās: formālās metodes, kas ir balstītas uz stingri definētiem matemātiskiem modeļiem, un daļēji

formālās metodes, balstītas uz vispārīgām un grafiskām sistēmas specifikācijām. Formālās verificēšanas metodes (piemēram, [THA 2004]) paredz matemātiski aprakstītus un validētus asinhronās darbības pārbaudes algoritmus. Savukārt, pārējas metodes atbilst vispārīgam testēšanas procesam un paredz atbilstošu testpiemēru ģenerēšanu un izpildi, balstoties uz iepriekš sagatavotām sistēmas specifikācijām, kas apraksta asinhronās darbības īpašības. Savukārt, kaut arī uz modeļiem balstīta programmatūras izstrāde paredz jebkura formāli aprakstīta modeļa lietošanu, tā balstās uz vispārpieņemtiem standartiem – tādiem kā UML.

1.3.2 Sinhronizācija

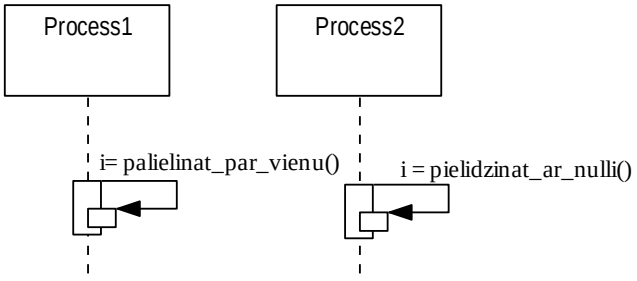
Asinhronās sistēmās un vairāku aparatūras līdzekļu sistēmās sinhronizācija ir neatņemama nefunkcionālā prasība, kas ir jānodrošina, lai iegultā sistēma varētu funkcionēt. Eksistē vairāki sinhronizācijas lietojumi un definīcijas. Datorzinātnē izdala divus sinhronizācijas konceptus: datu sinhronizācija un procesu sinhronizācija. Datu un procesu sinhronizācijām, pēc autora domām, visatbilstošākās definīcijas ir sekojošās:

„*Datu sinhronizācija* – darbība, kas nodrošina datu kopas vairāku eksemplāru atbilstību/aktualitāti" [ENC].

„*Procesu sinhronizācija* – darbība, kas nodrošina noteiktu notikumu sakritību laikā divu vai vairāku asinhronu procedūru izpildes gaitā" [TER].

Iegultās sistēmas ir multiapstrādes sistēmas, tāpēc procesu sinhronizācijas problēma ir būtiska programmatūras izstrādē, savukārt datu sinhronizācijas specifika nav pilnvērtīgi izteikta vispārīgo iegulto sistēmu izstrādē.

Apskatot sinhronizāciju iegultās sistēmās, pievērsīsim uzmanību vienkāršam piemēram, kad vienu mainīgo maina divi paralēlie procesi. 1.4.attēlā ir paradīt nesinhronizētu procesu modelis.



1.4. att. Nesinhronizētu procesu piemērs

Pieņemot, ka *i* ir kopīgs *integer* tipa objekts, tad dotajā piemērā nav iespējams noteikt izpildes rezultātu, jo nevar spriest par aprakstīto operāciju atomāritāti un operācijas izpildi vienā solī. Uz operāciju izpildes rezultātu ietekmē sekojošās detaļas:

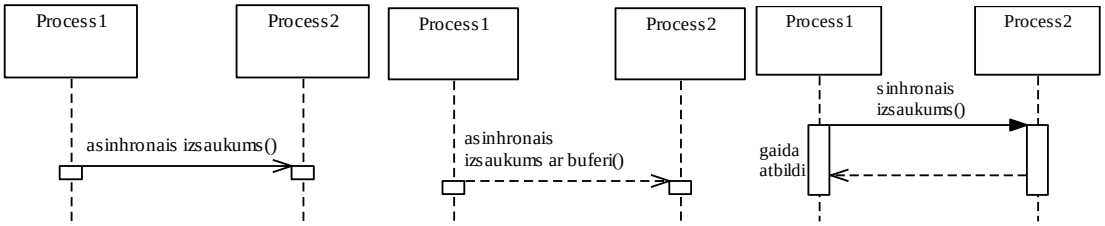
- ja *integer* ir 64 bitu garš un tiek lietots 8 vai 16 bitu kontrolieris, tad operācijas nebūs atomārās (bet iespējams, ka tas ir 8 bitu garš *integer*);
- jebkura manipulācija ar operatīvo atmiņu nebūs atomāra operācija (bet iespējams, ka darbs notiek ar reģistru).

Kļūdainas situācijas ar šo operāciju sinhronizāciju ir reti sastopamas, bet sarežģīto sistēmu projektēšanā, sistēmas, kas atbild par cilvēku dzīvību, visiem šādiem gadījumiem jābūt sinhronizētiem un verificētiem.

Izšķir divas vispārējas sinhronizēšanas metodes: uz koplietojamiem atmiņas apgabaliem (angl. *shared memory*) un uz ziņojumiem bāzētās sinhronizācijas. Uz koplietojamiem atmiņas apgabaliem balstītā sinhronizācijā izšķir šādus veidus:

- sinhronizācija ar karodziņa palīdzību jeb arī savstarpēja izslēgšana (angl. *mutual exclusion* vai saīsināti *mutex*) – pats vienkāršākais sinhronizācijas piemērs, kad procesu kopa sinhronizācijas nolūkiem lieto mainīgo, ko sauc par karodziņu [ZIM 2009]. Bieži vien tas ir Boolean tipa mainīgais;
- sinhronizācija ar semaforiem – procesu kopa nosaka kopīgo mainīgo S, kas tiks lietots sinhronizācijas nolūkiem; tiek definēta atomāra operācija P: [if S > 0 then S:= S – 1] (tiek aizņemta resursa viena vienība); tiek definēta atomāra operācija V: [S:= S + 1] (tiek atbrīvota resursa viena vienība; šādu S mainīgo sauc par semaforu (angl. *semaphore*);
- kritiskās sadaļas sinhronizācija (angl. *synchronization of critical sections*) – kritiskās sadaļas ir pirmkodu rindas dažādos procesos, kuri nevar tikt palaisti, kad cita kritiskā sekcija vēl strādā;
- speciāls sinhronizācijas veids ir sinhronizācija ar aizsargātiem objektiem. Šajā gadījumā visas operācijas ir kritiskās sadaļas un darbojas līdzīgi tai.

Uz ziņojumiem bāzētā sinhronizācija var tikt nodrošinātā ar sinhroniem un asinhroniem ziņojumiem. Ir zināmas vairākas sinhronizācijas struktūras un dažas no tām ir apskatītas 1.5.attēlā.



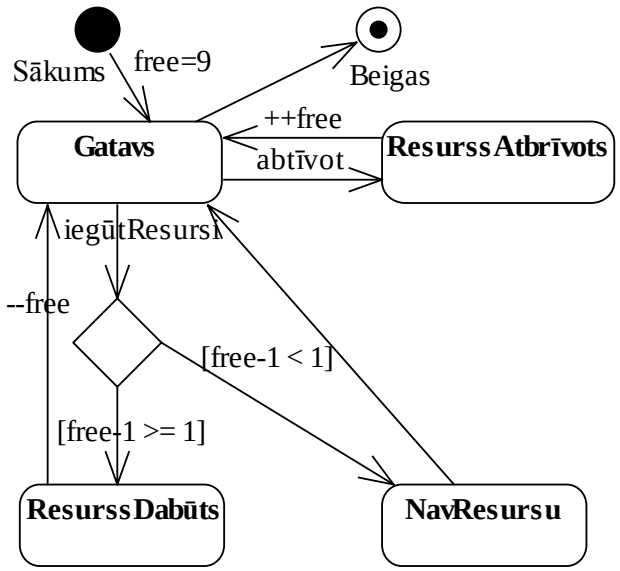
1.5.att. Uz ziņojumiem bāzētas sinhronizācijas piemēri

Ņemot vērā augstāk aprakstītas īpašības, darba autors izdala sekojošus kritērijus modelēšanas notācijām, kas būtu jānodrošina modelējot sinhronizācijas īpašības. Tie ir šādi:

- jāspēj attēlot sinhronie un asinhronie notikumi;
- jāspēj modelēt kritiskā sekcija;
- jāspēj modelēt paralēlā darbība;
- jāspēj modelēt laika ierobežojumi;
- jāspēj attēlot koplietojamus atmiņas apgabalus (semafori, karodziņi) un to stāvokļus;
- jāspēj attēlot aizsargātos objektus.

Ir zināmas vairākas metodes sinhronizācijas īpašības modelēšanai. No populārākām un veiksmīgākām var izdalīt Petri tīklus (angl. *Petri net*) un UML secību un stāvokļu diagrammas [PIL 2005]. Savukārt, MDA ietvaros UML ieņem tām atvēlēto vietu, tāpēc vienotai modelēšanās valodai ir priekšrocība attiecībā uz pielietošanas iespējām. Ņemot vērā šo aspektu, primāri tiek izskatīta iespēja pielietot UML stāvokļu un secību diagrammas sinhronizācijas īpašības modelēšanai.

UML stāvokļu diagramma var tikt lietota par sinhronizācijas modelēšanas notāciju gadījumos, kad ir jāmodelē uz koplietojamiem atmiņas apgabaliem balstītu sinhronizāciju. Šim apgalvojumam ir sekojošiem pamatojumi: tā nodrošina iepriekš minēto aspektu specificēšanu un stāvokļu diagrammu detalizācijas līmeni pietiekami, lai aprakstītu objektu uzvedību nepieciešamos stāvokļos, turklāt detalizācijas līmeni var izvēlēties pēc nepieciešamības. 1.6.attēlā ir parādīta stāvokļu diagramma, kas prezentē uz koplietojamās atmiņas balstītu sinhronizēšanas piemēru.



1.6.att. Uz koplietojamās atmiņas sinhronizācijas attēlošana ar stāvokļu diagrammu

Atšķirībā no koplietošanas atmiņas apgabalu balstītās sinhronizācijas, uz ziņojumiem balstīto metožu specificēšanai un modelēšanai piemērotāka ir UML secību diagramma, kas ir paredzēta vairāku objektu savstarpējās mijiedarbības definēšanai. Objekti tiek attēloti ar tā saucamām dzīves līnijām, kuru virzība uz leju reprezentē objekta dzīves ciklu laikā, un to mijiedarbība tiek attēlota ar bultu palīdzību, nosakot iedarbības avotu un izsaucamo objektu. Par iedarbību var būt sinhronie vai asinhronie izsaukumi vai signāli. UML secību diagrammas piemērs tika apskatīts 1.5. attēlā, reprezentējot dažāda veida uz ziņojumiem balstītas sinhronizācijas piemērus.

Eksistē arī formālās metodes sinhronizācijas verificēšanai. [STI 2008] piedāvā uz CSP [HOA 2004] balstītu sinhronizācijas verificēšanas metodi. Aprakstīta metode paredz FDR un ProBE [FSS] speciālo rīku lietošanu, kas nodrošina CSP modeļu automātisko verificēšanu. FDR rīks ļauj pierakstīt sistēmas CSP modeļus, pārbaudīt to loģisko struktūru un tajos meklēt strupsaķeres un bezgalīgus ciklus. Minētais darbs apraksta vairākus sinhronizēšanas paņēmienus un piedāvā to realizāciju CSP modeļos un Java programmās. Savukārt, dinamiskā programmas testēšanas netiek apskatīta un ir ārpus šī pētījuma. Cita formāla pieeja ir balstīta uz LTL formālo modeli [KSH 1998] un līdzīgi iepriekšējai metodei paredz sistēmas modeļa izstrādi noteiktajā formālajā notācijā, modeļa verificēšanu pret strupsaķerēm, ka arī C programmu automātisko darbināšanu ar mērķi atrast sinhronizācijas kļūdas.

1.3.3 Laicīgums

Laicīgums ir reālā laika sistēmu galvenā nefunkcionālā īpašība, jo laika ierobežojumu sasniegšana tieši ietekmē operāciju izpildes rezultātu un veselas sistēmas darbspēju. Iegultās sistēmas pārsvarā ir arī reālā laika sistēmas, tāpēc laicīguma īpašība ir tik pat svarīga arī tām.

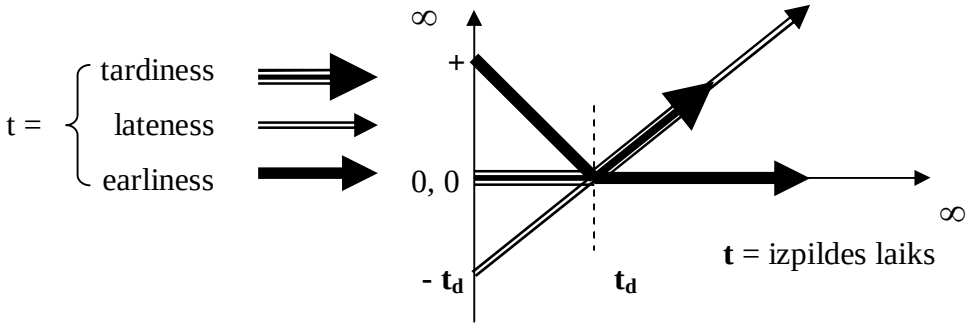
Laika ierobežojumi nosaka kādas darbības lietderību atkarībā no laika momenta. Pazīstamākais laika ierobežojumu piemērs ir galējais izpildes termiņš (angl. *deadline*) – laiks, līdz kuram aktivitātes izpildei ir lielāka lietderība, un pēc kura tai pašai aktivitātes izpildei ir mazāka lietderība [JEN 2011]. Bez galējā izpildes termiņa ir citi laika ierobežojumi, piemēram, aktivitātes lietderību var noteikt kāda speciāla funkcija atkarībā no izpildes laika.

„Cietais" galējais izpildes termiņš (angl. *hard deadline*) ir gadījums, kad lietderība ir bināra vērtību kopa {1, 0}: pabeidzot aktivitātes izpildi šajā laikā sistēma saņem maksimālo lietderību, kas ir iespējama šai aktivitātei, bet pārsniedzot to – lietderība ir 0 vai arī negatīvs skaitlis (tiek patērēti resursi, kas nedeva pozitīvus rezultātus).

Vispārīgam jeb „mīkstajam" galējā izpildes termiņa gadījumam (angl. *soft deadline*) lietderība tiek mērīta ar trīs parametriem [JEN 2011]:

- kavēšanās laiks (angl. *lateness*): izpildes laiks atskaitot galējo izpildes termiņu; šīs mērījums nosaka vispārīgo kavēšanas apjomu un var būt pozitīvs (laika ierobežojums ir pārsniegts) vai negatīvs (laika ierobežojums vēl nav sasniegts);
- nokavētais laiks (angl. *tardiness*): mērījums nosaka nokavēto laiku un ir vienāds ar nulli kamēr laika ierobežojums nav sasniegts un ir lielāks par nulli, ierobežojumu pārsniegšanas gadījumos.
- apsteigšanas laiks (angl. *earliness*): mērījums parāda cik daudz paliek līdz galējām izpildes termiņam un tā iespējamās vērtības ir no nulles līdz galējām izpildes termiņam.

Šie mēri tiek parādīti 1.7. attēlā.



1.7. att. „Mīkstā" galējā izpildes termiņa mēri

Lielākās *tardiness* vai *lateness* vērtības nozīmē mazāko lietderību un atbilstoši lielākās *earliness* – lielāko lietderību. Šie mēri tiek lietoti lai noteiktu strādājošā vai nostrādāta uzdevuma lietderību, ka arī uzdevumu plānošanas algoritmos. Šie mēri ļauj novērtēt konkrēto plānošanas algoritmu, jo tie reprezentē kavējumu lielumu un tieksme uz lielāku *earliness* mēru ļauj optimizēt šos algoritmus.

Laika ierobežojumi arvien biežāk parādās sistēmu implementācijas, nodrošinot stabilu to funkcionēšanu. Tieši tāpēc laika specificēšana jau sen netiek uzskatīta par specifisko īpašību un tiek nodrošināta populārākajās modelēšanas notācijas, piemēram, laicīguma Petri tīkli un UML 2.0 versija, kurā laika ierobežojumi kļuva par UML standarta modeļu sastāvdaļām un vairāk neprasa atsevišķu profilu to implementācijai. Runājot par UML diagrammām, tad tie ļauj specificēt laicīguma īpašībai nepieciešamus funkcionālus aspektus:

- uzdevuma, procedūras vai arī kāda cita funkcionālā apgabala maksimālais un minimālais apstrādes laiks;
- tekošais sistēmas laiks (laika dimensija);
- dažādu notikumu laiks;
- universāla laika funkcija, kas derētu jebkāda laika ierobežojuma specificēšanai (galējais izpildes termiņš ir īpašs gadījums);
- laika attēlošana nosacījumu izteiksmēs (sistēmas darbība atkarībā no laika).

Līdz UML 2.0 versijas izlaišanai laika ierobežojumu un citu aspektu specificēšanai tika lietots speciāls profils uzdevumu plānošanai, ātrdarbībai un laikam (angl. *profile for schedulability, performance and time specification*) [SPT], kura idejas par laika modelēšanu tika integrētas vispārīgajā UML 2.0 versijā.

1.4 Nodaļas secinājumi

Nodaļā tiek apskatīta iegulto sistēmu specifika un to nefunkcionālās īpašības. Vispārējo testēšanas metožu lietojums iegulto sistēmu verificēšanas uzdevumā, it īpaši to nefunkcionālo īpašību korektuma pārbaudei, noved pie situācijas, kad programmatūras sistēmā paliek neatklātas kļūdas, kas savukārt noved pie nopietniem zaudējumiem. Tas liecina par esoša iegulto sistēmu testēšanas procesa nepilnībām un tā uzlabojuma nepieciešamību. Tādas nefunkcionālās īpašības kā laicīgums, paralēlā darbība un sinhronizācija ir kritiskās iegultām programmatūrām, jo šo īpašību nodrošināšana ir neatņemama prasība veselas sistēmas patstāvīgai darbībai.

Iegulto sistēmu testēšanā tiek lietotas vispārīgas un šīm sistēmām specifiskās metodes un rīki, tomēr tiem visiem ir savi trūkumi. Neskatoties uz to, ka eksistē dažas nefunkcionālo īpašību testēšanas metodes, tās ir novecojušās un nespēj nodrošināt nepieciešamo automatizācijas līmeni, ko pieprasa mūsdienās pielietojamās tehnoloģijas. Rezumējot nodaļā aprakstītu iegulto sistēmu specifiku un to īpašību eksistējošās testēšanas metodes, var secināt sekojošo:

- Iegultās sistēmas ir komplicētas, līdz ar to, lai verificētu to specifiskās nefunkcionālās īpašības, nav pietiekami izmantot standarta metodes šādu sistēmu testēšanai.
- Eksistē daudz testēšanas pieeju un paņēmienu, kuru mērķis ir nodrošināt dažāda līmeņa sistēmu verificēšanu un validēšanu, tomēr tās pārsvarā pēc savas būtības fokusējas uz sistēmas funkcionēšanu.
- Iegulto sistēmu īpašības ir iespējams klasificēt, atsevišķi analizēt un verificēt, un šāda klasifikācija izceļ iegulto sistēmu tieši nefunkcionālo īpašību svarīgumu šādu sistēmu definētās darbības atbalstam.
- Iegulto sistēmu īpašības ir iespējams modelēt, izmantojot speciālus notācijas līdzekļus, piemēram, UML un to paplašinājumus, tomēr pilnīgu verificēšanu tie nenodrošina nepietiekamās automatizēšanas dēļ.

2 MODEĻVADĀMĀS ARHITEKTŪRAS BĀZES ELEMENTU ATTĒLOŠANA TESTĒŠANAS ARTEFAKTOS

Iegulto sistēmu specifika uzstāda paaugstinātas prasības to nefunkcionālo īpašību testēšanai. UML dod iespēju testēšanas uzdevuma izpildīšanai veidot sistēmas modeļus, kas detalizēti atspoguļo katras nefunkcionālās īpašības būtību. Viens no moderniem risinājumiem dažāda tipa sistēmu izstrādē ir izmantot modeļus un to transformāciju izstrādes procesa automatizācijai. Šajā nodaļā autors izvirza hipotēzi par to, ka ir iespējams modeļvadāmās programmatūras izstrādes principus izmantot iegulto sistēmu verificēšanas automatizācijai.

2.1 Modeļvadāmās arhitektūras pamata principi

Modeļvadāmās arhitektūras princips programmatūras izstrādē ienesa kardinālās izmaiņas programmatūras izstrādes procesā. Pirmkoda rakstīšanas fāze kļuva neobligāta un izstrādātājiem tika piedāvāts projektēt un būvēt sistēmas modeļu līmenī un šādā veidā atturēties no tehniskām programmu realizācijām. Tas ļāva atrisināt sekojošās problēmas, kas joprojām bija aktuālas, gan inkrementālajā, gan arī iteratīvajā programmatūras izstrādē [KLE 2003] [MEL 2004]:

- Reālajā izstrādes procesā laika periods no specifikācijas pabeigšanas līdz koda gatavībai ir pietiekami liels un šī perioda laikā nobīde starp specifikāciju un kodu palielinās un rezultātā kods vairs neatbilst specifikācijai. Lai izvairītos no šādām neatbilstībām mākslīgi un manuāli ir nepieciešams nepārtraukti uzturēt aktuālo specifikāciju. Šāda aktivitāte reālos apstākļos kļūst grūti realizējamā nepieciešamo resursu un apšaubāma mērķa dēļ.
- Parastajā programmatūras izstrādē, kur netiek lietoti modeļvadāmās arhitektūras principi, koda rakstīšana ir manuāls process, kas dabīgā veidā prasa noteikto laiku starp modeļa jeb specifikācijas izveidi līdz gatavai programmai. Šādā dinamiskajā vidē kā programmatūras izstrāde laiks spēlē vienu no noteicošām lomām, tāpēc arī pielietojot

automātisko modeļu transformāciju kodā, ir iespējams paātrināt šo posmu.

- Veiklajās metodoloģijās viens no galvenajiem faktoriem ir cilvēku kompetences, spēja sadarboties komandā un komandas pastāvīgums, lai komandu dalībnieki pilnīgi pārzinātu izstrādājamo programmu un problēmas sfēru. Gadījumos, kad šādi izstrādāts projekts tiek nodots, jeb mainās izstrādātāju komanda, jaunie cilvēki tiek iekļauti izstrādes vai uzturēšanas procesā un tie kļūst par vāju posmu nepilnīgās specificēšanas un dokumentācijas dēļ. MDA ietvaros izveidotie sistēmu modeļi visu laiku ir aktuāli un principā tās ir programmas. Šādā veidā modeļi kalpo par vienotu programmas aprakstu un tajā paša laikā ir arī pati programma.

Ir zināmas vairākas MDA definīcijas un pēc autora domām īsi un konstruktīvi MDA būtību atspoguļo šādas definīcijas:

„MDA ir informāciju sistēmu specificēšanas pieeja, kas atdala funkcionalitātes specificēšanu no šīs funkcionalitātes implementēšanas uz specifiskās tehnoloģijas platformas specificēšanās" [MDA].

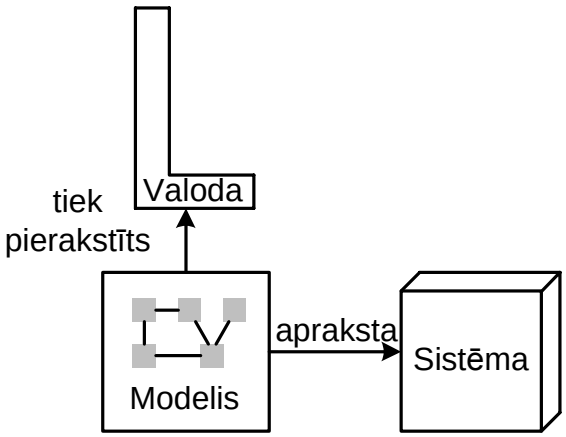
„MDA būtība ir dažādu modeļu veidošana dažādos abstrakcijas līmeņos un tālāka šo modeļu saistīšana ar mērķi implementēt sistēmu. Daži no modeļiem eksistē neatkarīgi no programmatūras platformas, tajā pašā laikā citi modeļi ir specifiski konkrētai platformai. Katrs modelis tiek veidots lietojot tekstu un vairākas papildinošas un savā starpā saistītas diagrammas" [MEL 2004].

Modeļvadāmās arhitektūras princips balstās uz modeļa būtību, kas atspoguļo un specificē sistēmas funkcionēšanu un modeļu transformāciju. Viena no veiksmīgākām modeļa definīcijām ir šāda:

„Modelis ir sistēmas vai tās daļas apraksts, kas ir veidots labi definētājā valodā. Labi definētājai valodai piemīt labi definētas struktūras (sintakse) un skaidrās nozīmes (semantika) īpašības un tā ir derīga datora automātiskai interpretēšanai" [KLE 2003].

Jebkurš modelis ir veidots noteiktajā valodā un MDA nenosaka valodu, kas ir jālieto modeļu veidošanā. Modeļu veidošanā var tikt lietotas dažādas notācijas, piemēram, vienotā modelēšanas valoda (angl. *unified modeling language* – UML), Petri tīkls (angl. *Petri net*), entītiju attiecību diagramma (angl. *entity relationship diagram* – ERD), datu plūsmu diagramma (angl. *data flow diagram* – DFD) un citas

grafiskās un formālās valodas, kas atbilst labi definētas valodas principiem. Zemāk ir attēlota modeļa saistība ar valodu un aprakstāmo sistēmu.



2.1. att. Modeļa attiecības pret valodu un sistēmu [KLE 2003]

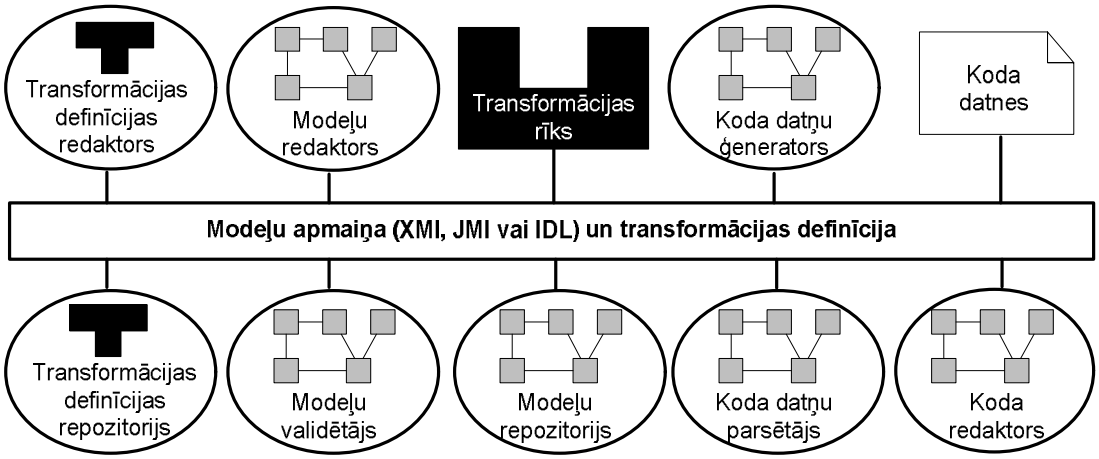
Kā ir redzams no attēla, izvēlētā modelēšanas valoda principā nav atkarīga no sistēmas, savukārt, reālitātē sistēmas īpašības un funkcionēšanas principi ietekmē modelēšanas valodas izvēli. Modelēšanas valodu esamība un piemērotība ir viens no atslēgas momentiem modeļvadāmajā arhitektūrā, līdz ar to tas veicināja modelēšanas valodu un ar to saistīto rīku attīstību un standartizēšanu.

MDA jēdziens un pašlaik zināmais koncepts radās pateicoties objektu vadības grupas (angl. *object management group* – OMG) izstrādātāju grupai, kas nodrošina programmatūras izstrādē svarīgu standartu izstrādi un uzturēšanu (piemēram, UML, CORBA, XMI, MOF un citi [OMG]). Saistībā ar MDA pētījumu svarīgākie OMG standarti ir:

- Meta objekta iespējas (angl. *meta object facility* – MOF) ir OMG standarts, kas definē valodu, kas savukārt definē modelēšanas konstrukciju kopu (jeb modelēšanas valodas sintaksi un semantiku), ko modelētājs var lietot, lai definētu un manipulētu ar spējīgu sadarboties metamodeļu kopu. MOF ir četru līmeņu arhitektūras trešā līmeņa standarta valoda, un visas modelēšanas valodas, tādas kā piemēram UML, CWM un citas, ir MOF eksemplāri. MOF nodrošina meta-metamodelēšanu UML metamodeļiem un definē modelēšanas konstrukciju kopu (piem. pakete, klase, metode, atribūts), kas ļauj definēt un manipulēt ar modeļa metadatiem (dati par datiem). Pats MOF ir definēts UML valodas notācijā [NIK 2007].

- UML – grafiskā valoda programmatūras sistēmas artefaktu attēlošanai, specificēšanai, konstruēšanai un dokumentēšanai. UML pamatlicēji ir Grady Booch, James Rumbaugh un Ivar Jacobson, kas nonākot vienas organizācijas ietvaros, apvienoja savu pieredzi un pētījumus, rezultātā veidojot vienotu modelēšanas valodu UML. UML dod iespēju standartizēti aprakstīt sistēmas projektu, atspoguļojot gan konceptuālās lietas, tādas kā biznesa procesi un sistēmas funkcijas, gan konkretizējot klases, kas var tikt aprakstītas specifiskajā programmēšanas valodā, kā arī dod iespēju definēt datubāzes shēmas un atkārtoti lietojamus programmatūras komponentus [NIK 2007].
- XML metadatu apmaiņa (angl. *XML metadata interchange* – XMI) ir OMG kompānijas standarts priekš informācijas par modeļiem apmaiņas modelēšanas rīku ietvaros. Pēc savas būtības šī ir sintaktisko un semantisko konstrukciju kopas specifikācija, kura var veikt jebkuras metainformācijas nodošanu, gadījumā, ja attiecīgais metamodelis eksistē MOF infrastruktūras ietvaros [NIK 2007].

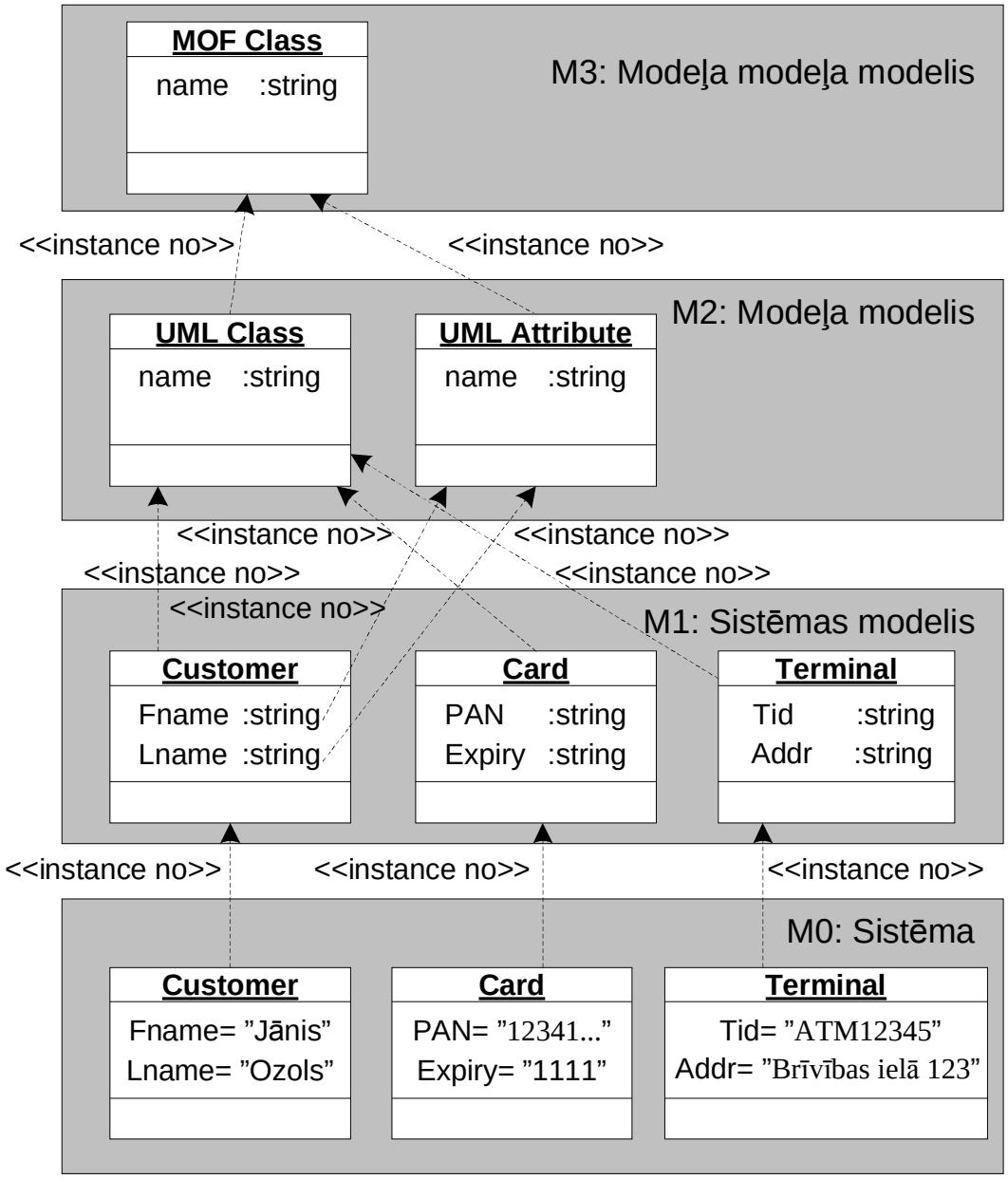
Šie standarti nodrošina vienotu pieeju programmatūras izstrādē, kā arī ir nepieciešami MDA atbalsta rīku izstrādē. Visus aktuālus ar MDA saistītus rīkus var iedalīt vairākās kategorijās, kas ir parādītas 2.2. attēlā.



2.2. att. MDA atbalsta rīku kategorijas [KLE 2003]

Šie rīki balstās uz OMG piedāvātiem standartiem un nodrošina modeļvadāmo programmatūras izstrādi. Pašlaik ir pieejami komerciālie un brīvi izplatītie atbalsta rīki, kas pārsvarā balstās uz sistēmas modeļu izstrādāšanu, to validēšanu un pirmkoda ģenerēšanu, nepievēršot pietiekamu uzmanību testēšanas procesam.

OMG definē modeļvadāmajā arhitektūrā 4 modelēšanas slāņu principu, kur slāņi tiek saukti par M0, M1, M2 un M3. Modeļu veidošanas bāze tiek atspoguļota minētājā MOF standartā, kurš atrodas augstākajā M3 līmenī. Zemāk ir atspoguļots 4 slāņu princips un parādīti slāņu piemēri.



2.3. att. 4 slāņu modeļvadāmās arhitektūras princips ar UML modeli [KLE 2003]

M0 līmenī atrodas pati sistēma ar tās reāliem objektiem. Piemēram, bankomātu vadības sistēmā M0 līmeņa objekti ir bankomāta eksemplārs ATM12345, kas atrodas Brīvības ielā 123 un kurā Jānis Ozols ielika savu karti ar numuru 1234123412341234.

Nākamais augstāk stāvošs līmenis ir M1, kas atspoguļo reālās sistēmas modeli. Šajā slānī noteiktajā modelēšanas valodā tiek definēti reālās sistēmas koncepti. 2.3. attēlā ir parādīts piemērs ar UML modelēšanas valodu, bet tajā pašā laikā M1 modelis varētu būt attēlots arī ar citu modelēšanas valodu. Visi M1 modeļa koncepti ir M0 modeļa elementu klasifikācijas, kā arī visi M0 modeļa elementi ir M1 modeļa elementu eksemplāri.

M2 slānis tiek saukts arī par meta-modeļa slāni.

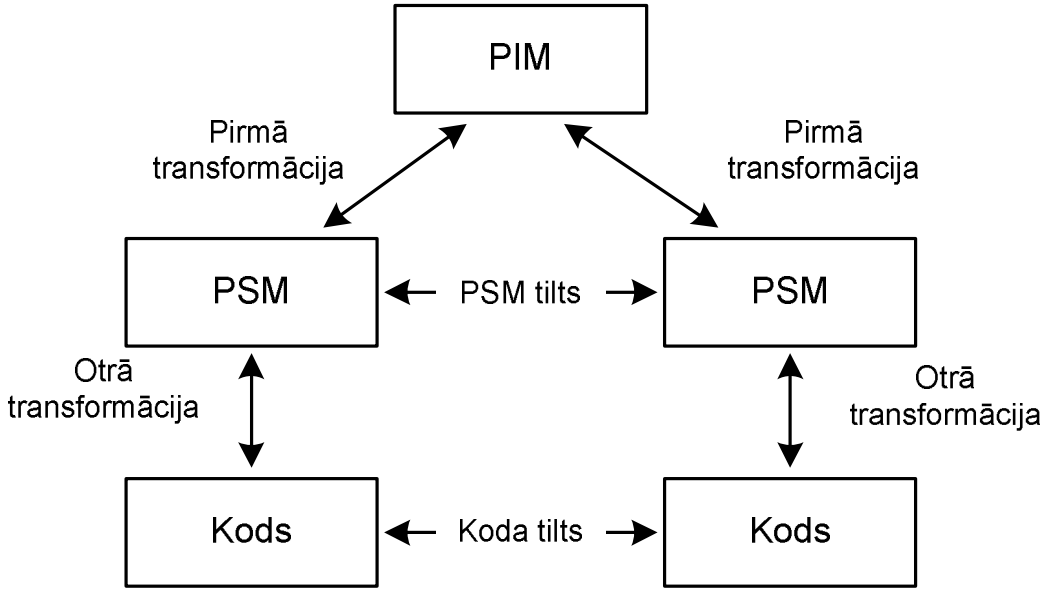
„Meta-modelis ir labi definētas valodas apraksts vai definīcija modeļa formā" [KLE 2003].

Konkrētās sistēmas M1 modeļa elementi (specifiskās klases, atribūti un citi) ir augstāk stāvoša modeļa eksemplāri. Tā piemērām, klase *Card* ir vispārīgās *Class* elementa eksemplārs un klase *Terminal* arī ir tā pati elementa *Class* eksemplārs. Vieni no populārākiem modeļiem M2 līmenī ir UML standarta modeļi. Citas modelēšanas notācijas, piemēram, Petri tīkls, arī atrodas tajā pašā M2 līmenī.

Pēdējais M3 līmenis nodrošina iepriekš minēto modelēšanas notāciju specificēšanu. M2 līmenī tiek definētas stāvokļu, secību, laicīguma un citas diagrammas, kas savukārt ir veidotas balstoties uz M3 modeļa, ko OMG standartizēja un nosauca par MOF.

Slāņu skaits teorētiski var būt arī lielāks un priekš MOF ir iespējams nodefinēt metamodeli, bet praktiskie eksperimenti parādīja, ka tālāka metamodeļu definēšana un M4 slāņa definēšana nav efektīva un nepieciešamie elementi var tikt definēti M3 līmenī [KLE 2003]. Šie četri slāņi nodrošina modeļvadāmās arhitektūras caurspīdīgumu un standartizē pieeju sistēmas modeļu izveidošanā.

MDA definē trīs modeļus: no platformas neatkarīgais modelis (angl. *platform independant model* – PIM), platformai specifiskais modelis (angl. *platform specific model* – PSM) un kods [KLE 2003]. Par sākotnējo modeli tiek uzskatīts PIM, bet visi pārējie modeļi tiek atvasināti no iepriekšējā caur modeļu transformāciju. MDA modeļu atkarības un transformācijas starp tiem ir parādīti 2.4. attēlā.



2.4. att. MDA modeļu atkarības un transformācijas starp tiem [KLE 2003]

PIM modelis apraksta izstrādājamo sistēmu, kas nodrošina kādas biznesa funkcijas. PIM modeļa ietvaros sistēma tiek modelēta ar mērķi parādīt kā sistēma nodrošina nepieciešamās biznesa funkcijas. Šajā abstrakcijas līmenī modelis nefokusējas un vispār neapskata sistēmas realizācijas platformu un neietver realizācijas specifiku. Izstrādes nākamajās fāzēs PIM modelis tiek transformēts vienā vai vairākos PSM modeļos.

PSM modeļi ir ražoti ar mērķi aprakstīt sistēmas tehniskās realizācijas īpašības, kas ir atkarīgas no konkrētās izvēlētās tehnoloģijas. Gadījumos, ja izvēlētā tehnoloģija ir relāciju datu bāze, tad šīs modelis ietvers sevī tādas notācijas kā *table*, *column*, *constraint* un citas. PIM tiek transformēts vairākos PSM modeļos galvenokārt gadījumos, kad katrs PSM modelis specificē izstrādājamo sistēmu izvēlētājā platformā.

Pēdējā transformācija MDA ietvaros ir transformācija no PSM modeļa uz reālo kodu. Katra programmēšanas valoda pieder kādai tehnoloģijai, tāpēc PSM un kods pēc savas būtības ir cieši saistīti un transformācijas no PSM modeļa uz kodu detalizē un papildina PSM modeļa elementus un rezultātā reprezentē tos kā izpildāmo kodu.

2.4. attēlā ir redzams, ka PSM modeļi ir detalizācijas no vairāk abstrakta PIM modeļa un tajā pašā laikā ir vairāk abstrakti modeļi, nekā pirmkods. Katrs PSM modelis ir specifisks konkrētai tehnoloģijai un ietver tai tehnoloģijai specifiskus

elementus un tāpēc PSM modeļi nevar būt taisni saistīti savā starpā. Šādiem nolūkiem MDA definē tiltus starp viena abstrakcijas līmeņa dažādu tehnoloģiju modeļiem, kas varētu būt PSM modeļi vai kods. Savukārt, saite starp dažādu abstrakcijas līmeņu modeļiem tiek nodrošināta ar transformācijām. Runājot par tām, ir nepieciešams sniegt sekojošās definīcijas:

„Transformācija ir beigu modeļa automatizētā ģenerēšana no sākotnējā modeļa, attiecīgi iepriekš definētiem transformācijas likumiem. MDA programmatūras izstrādes cikla ietvaros, transformācijas var būt uzskatītas par tiltiem, kas savieno dažāda tipa modeļus" [NIK 2007].

„Transformācijas definīcija ir likumu kopa, kura definē kādā veidā sākotnējais modelis var būt transformēts beigu modelī" [NIK 2007].

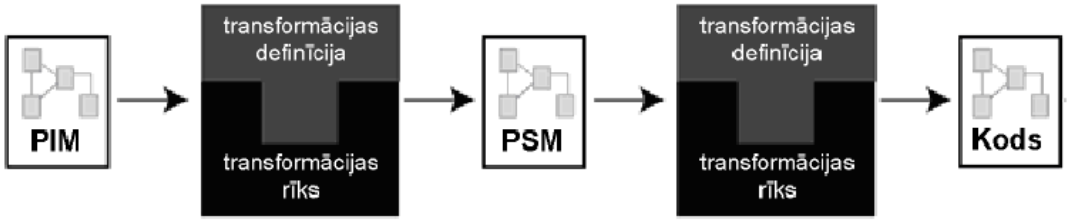
„Transformācijas definīcijas valoda ir valoda, kurā tiek uzdota transformācijas definīcija" [NIK 2007].

Izdala 2 veidu modeļu transformācijas: no modeļa uz modeli un no modeļa uz kodu transformācijas. Abiem veidiem vispārīgie transformācijas principi paliek kopīgi, bet atšķiras pielietojamās metodes. Tā, piemērām, no modeļa uz kodu transformācijā tiek pielietotas sekojošas metodes [STA 2006]:

- veidnes un filtrēšana;
- veidnes un metamodelis;
- ietvaru/kadru apstrāde;
- API bāzētā ģenerācija;
- atsevišķa koda ģenerēšana (angl. *in-line generation*);
- koda atribūtu ģenerēšana;
- koda aušana (angl. *code weaving*).

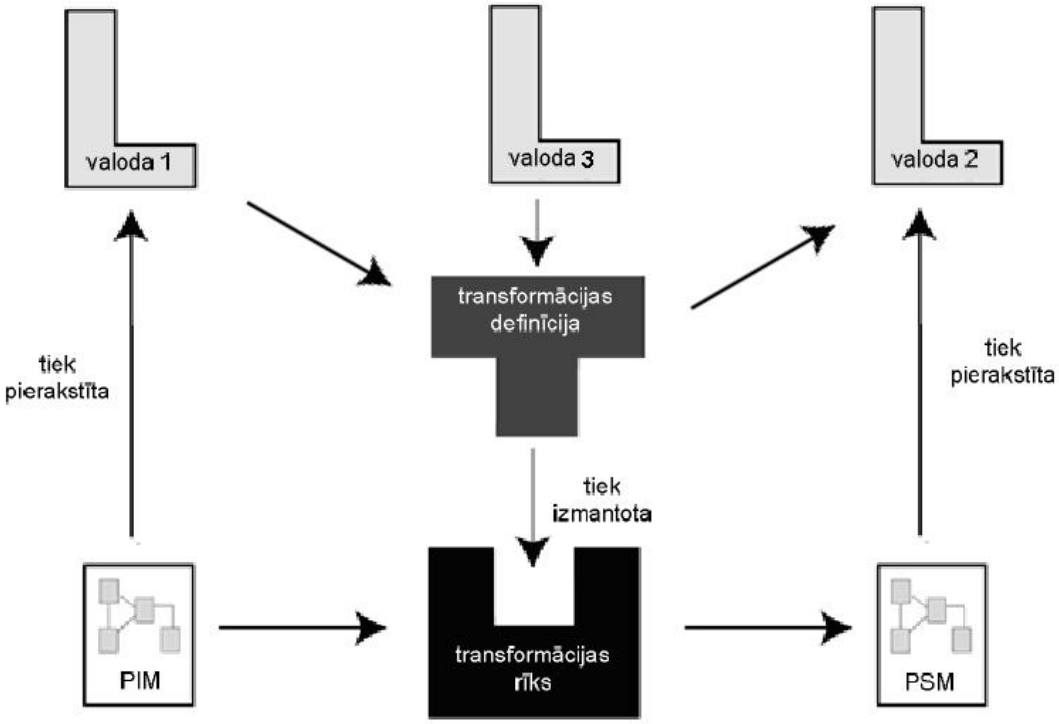
MDA stingri nedefinē, ka transformācijām starp modeļiem ir jābūt automātiskām, tomēr, lai gūtu efektivitāti programmatūras izstrādē, ar MDA līdzekļiem, ir nepieciešama automātiskā modeļu transformācija un rīku atbalsts. Pēc [STA 2006] izstrādājamās aplikācijas kodam ir jābūt iegūtam pilnīgi automatizētā procesā, pretējā gadījumā manuālā programmas modificēšana ir tendēta uz kļūdām un izslēdz vienu no svarīgākām automātiskās koda ģenerācijas īpašībām – strādājošas sistēmas iegūšana no modeļa vienā laidienā. Kaut arī [STA 2006] neizslēdz aplikāciju modificēšanu, bet pieprasa, lai atsevišķās aplikācijas komponentes tiktu pilnībā

ģenerētas automātiski. 2.5. attēlā ir parādīts vispārīgs MDA divu soļu modeļu transformācijas princips.



2.5. att. MDA divu soļu transformācijas princips [NIK 2007]

Kā redzams attēlā, lai nodrošinātu automātisko modeļu transformāciju ir nepieciešams transformācijas rīks un transformācijas definīcija. Transformācijas rīkam jānodrošina kāda definēta transformācijas valoda, kurā tiek specificēta transformācijas definīcija. 2.6. attēlā ir parādītas modeļu un transformācijas valodas atkarības no modelēšanas valodas.



2.6. att. Transformācijas valodas atkarība no modelēšanas valodām [NIK 2007]

No 2.6. attēla ir redzams, ka transformācijas valoda nav ietekmēta no modelēšanas valodām, savukārt transformācijas definīcija ir atkarīga no abu modeļu modelēšanas valodām.

[STA 2006] izdala sekojošas praktiskās transformācijas īpašības:

- Transformāciju valodai jābūt pietiekamai, lai nodrošinātu transformāciju strukturēšanu, piemērām, modularitāti, deleģēšanu, mantošanu, polimorfismu un citu.
- Rīkiem ir jāglabā transformācijas tādā kārtībā, lai tās varētu tikt identificētas ar versijām un lietotas ar citiem rīkiem un citiem cilvēkiem projekta griezumā.
- Izstrādātājiem jāstrukturizē transformācijas saprātīgi, jāsadala moduļos un regulāri jāpārstrādā.

[STA 2006] sniedz sekojošās prasības vispārīgām no modeļa uz modeli transformācijas valodām un to realizācijām:

- Reālistiskas no modeļa uz modeli transformācijas scenārijs pieprasa lai transformācijas likumi spētu analizēt rezultātu, iegūtu no citiem transformācijas likumiem, jo katrs likums parasti apskata tikai vienu aspektu no veselas transformācijas. Šāda tipa analīze ir iespējama, ja transformācijas rīkam ir iespēja glabāt transformācijas atsekošanas pierakstus. Ar atsekošanas pierakstiem saprot transformācijas izpildes laikā saglabātu informāciju, pēc kuras var noteikt transformācijas procesa rezultātu. Šādu pierakstu konkrēts izskats un uztveramība no lietotāja viedokļa var plaši atšķirties starp dažādām transformācijas valodām. Transformācijas atsekošanas aspekts ir līdzīgs parastu aplikāciju izpildes pierakstiem, kas tiek lietoti nekorekto aplikāciju darbību atsekošanai un kļūdu identificēšanai. Līdzīgie pielietojumi attiecas arī uz transformācijas pierakstiem.
- Gadījumos, kad no modeļa uz modeli transformācija tiek laista pirmo reizi, beigu modelis tiek iegūts uz tukšas vietas. Ja transformācija tiek laista atkārtoti, ir svarīgi, lai tā uzģenerētu tikai nepieciešamās izmaiņas mērķa modelī, nevis atkārtoti liktu klāt modelī jau esošus elementus. Šāds mērķa modeļa atsekošanas mehānisms ir iespējams, ja tajā ir realizēta elementu identificēšana. Tas varētu tikt realizēts tādām svarīgām mērķa metamodeļa īpašībām kā modeļa elementa nosaukums, modeļa elementa identifikācijas numurs vai arī unikāli identificējot saiti starp sākotnējā un mērķa modeļa elementiem. Parasti transformācijas

atsekošanas pieraksti satur šādu unikāli identificējamo informāciju. Prasība, lai atkārtotie transformācijas laidieni korekti atjaunotu mērķa modeli, tiek saukta arī par izmaiņu izplatīšanu (angl. *change propagation*).

- Transformācija tieši vai netieši definē saites starp sākotnējo un mērķa modeļiem. Dažos scenārijos abi modeļi eksistē pirms transformācija tiek darbināta, turklāt arī pirms saites starp modeļiem ir nodibinātas. Šādos gadījumos transformācijai var tikt jautāts pārbaudīt, vai saites eksistē un iespējams pamainīt mērķa modeli tik daudz, lai izveidotu saites starp tiem. Šī problēma ir atšķirīga no izmaiņu izplatīšanas gadījuma, jo pēc aprakstīta scenārija var pieņemt atsekošanas pierakstu esamību, kaut arī pati transformācija var arī netikt laista.
- Vispārīgi sākotnējais modelis var būt ārkārtīgi liels. Pēc pirmās transformācijas palaišanas, nākamie laidieni parasti ienes tikai nelielas izmaiņas mērķa modelī, jo sākotnējā modelī netiek taisītas lielas izmaiņas. Šādos gadījumos ir svarīgi apzināties, kādi transformācijas likumi būtu jālaiž atkārtoti un pret kādiem sākotnēja modeļa elementiem transformācija ir jādarbina. Modeļa elementu ietekmes analīze uz transformācijas likumiem, kā arī atsekošanas pierakstu pieejamība var tikt pieprasīta, lai realizētu šādu prasību. Šīs optimizācijas nepieciešamība nevar tikt nepietiekami novērtēta, jo transformācijas lietotāji sagaida adekvāti ātru transformācijas cikla darbību izstrādes laikā. Dažreiz optimizāciju sauc par atjaunošanu pa daļām (angl. *incremental update*).
- Ir zināmi vairāki lietošanas scenāriji, kur mērķa modelis (parasti platformu specifiskais modelis) prasa manuālas izmaiņas. Savukārt, atkārtota transformācijas palaišana teorētiski var nodzēst manuālās izmaiņas, tāpēc transformācijām būtu jānodrošina iespēja, lai pie kārtējās transformācijas palaišanas tikai un vienīgi manuāli izdarītas izmaiņas paliktu bez izmaiņām. Spēja transformācijas rakstītājam veikt aizsargātas manuālas izmaiņas dažreiz tiek saukta par paturēšanas politiku (angl. *retainment policy*).

- Strīdīgs jautājums ir vai divvirzienu (angl. *bidirectional*) transformācija ir nepieciešama prasība. Šāds efekts var tikt iegūts ar divām vienvirzienu transformācijām vai arī ar vienu transformāciju, kas var tikt darbināta divos virzienos. Reālajā dzīvē divvirzienu transformācija ir reti sastopama, jo tā pieprasa, lai abi, sākotnējais un mērķa modelis, būtu vienā abstrakcijas līmenī un aprakstītu vienu un to pašu apgabalu. No praktiskā viedokļa divvirzienu transformācija pie izmaiņām abos modeļos var vest pie neparedzētām situācijām un konfliktiem.

Iepriekš aprakstītās prasības tiek izvirzītas vispārīgām no modeļa uz modeli transformācijas valodām. 2002. gadā OMG izvirzīja pieprasījumu pēc piedāvājumiem vienotai modeļu transformācijas valodai. Šī pieprasījuma kontekstā tika piedāvātās vairākās transformācijas valodas, kas rezultātā daļēji apvienojās. Balstoties uz tām tika standartizēta QVT (vaicājumi/skati/transformācijas, angl. *queries/views/transformations*) transformācijas valoda un pirmā versija ir publicēta 2008. gada aprīlī (pēdējā aktuālā 1.1. versija tiek izlaista 2011. gadā janvārī) [QVT 2011]. Paralēli QVT valodai parādījās arī citas transformācijas valodas: MTF [MTF], ATL, [JOU 2006], BOTL [MAR 2003], VMTS [LEV 2004], MOLA [LU 2011] un citas. Galvenais pamatojums šo valodu radīšanai ir to fokusēšanās uz konkrēto uzdevumu pildīšanu, kas ļauj nodrošināt šo valodu efektīvu pielietošanu. Tajā pašā laikā neeksistē izstrādes vides, kas pilnībā nodrošina QVT valodas realizāciju [SOS 2010].

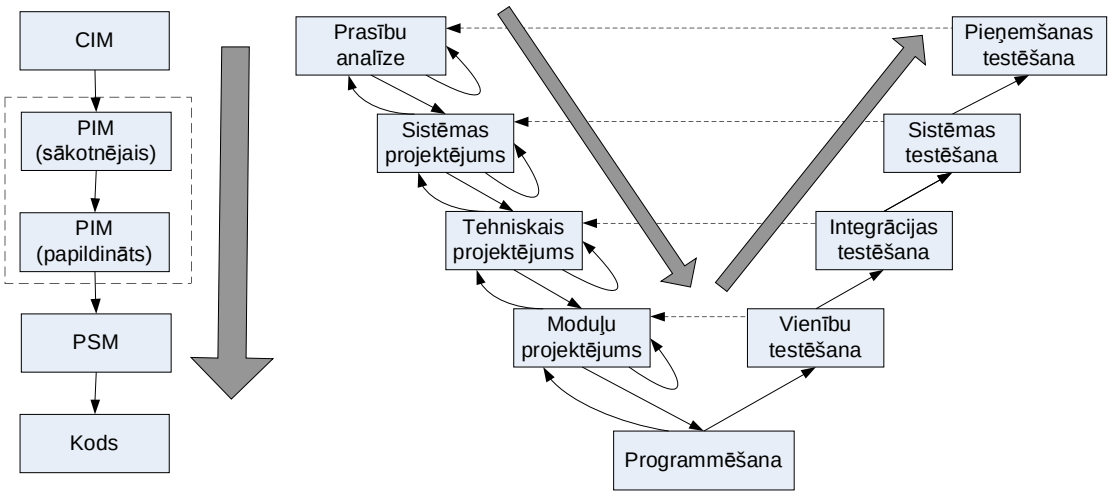
Klasiskie transformācijas pielietojumi ierobežojas ar sistēmas modeļa vai pirmkoda ģenerēšanu. Nākamajā sadaļā MDA transformāciju principi tiek apskatīti testēšanas procesa kontekstā.

2.2 Modeļvadāmās arhitektūras principu pielietošana testēšanā

Modeļvadāmajā arhitektūrā galvenie artefakti ir modeļi, jo tie apraksta gan sistēmas biznesa funkcionalitāti, gan iekšējās tehniskās īpašības un kalpo par avotiem koda ģenerācijas procesā. Klasiskajā testēšanas procesā testu gadījumu veidošanā tiek lietoti sistēmas funkcionālie apraksti, kas modeļvadāmajā arhitektūrā var tikt realizēti modeļu veidā. Šis fakts norāda uz to, ka testpiemēri var tikt veidoti no modeļiem.

Šāds process tiek saukts par uz modeļiem balstītu testēšanu (angl. *model-based testing* vai *model-driven testing*) – programmatūras testēšana, kur testpiemēri tiek pilnīgi vai daļēji iegūti no modeļa, kas apraksta kādus (vai visus) testējamās sistēmas aspektus [ENG 2006].

Uz modeļiem balstītā testēšana apskata tos pašus klasiskās testēšanas veidus, kas bija aprakstīti 1.2 sadaļā. Modeļu abstrakcijas līmeņa dažādība nodrošina un atbalsta atbilstošu testēšanas paveidu. Aplūkojot MDA bāzēto un klasisko programmatūras izstrādes un testēšanas secību, var pamanīt, ka abstrakciju līmeņu artefakti paliek nemainīgie. Testēšanas procesa V modeļa līmeņu analoģija ar modeļvadāmās izstrādes ķēdi ir parādīta 2.7. attēlā.



2.7. att. Modeļvadāmās un klasiskās izstrādes modeļi [GRI 2006]

2.7. attēlā ir parādīta sistēmas specificēšanas un testēšanas veidu abstrakcijas līmeņu atbilstība. Katram testēšanas procesam atbilst noteikts modelēšanas līmenis atbilstoši MDA principiem. Pieņemšanas testi vienmēr balstās uz prasībām sistēmai, kas MDA ietvaros tiek reprezentētas CIM modelī. Sistēmas un integrācijas testēšanai jānotiek balstoties uz sistēmas PIM modeļiem, jo tie apskata vispārīgo sistēmas funkcionēšanu un nedetalizējas līdz konkrētu moduļu realizēšanas specifikai. Pats zemākais modelēšanas līmenis ir PSM modelis, kas definē platformai specifisko realizāciju un komponenšu detalizāciju. Testēšanas process, kas notiek šajā līmenī un kas apskata komponenšu realizācijas aspektus, tiek saukts par vienības testēšanu. Visi minētie testēšanas veidi modeļvadāmajā testēšanā balstās uz sistēmas modeļiem.

Izdala trīs uz modeļiem bāzētus testēšanas veidus:

- Tūlītējā testēšana (angl. *online testing*): testpiemēri automātiski tiek iegūti no sistēmas modeļa un uzreiz dinamiski darbināti, aktivizējot testējamo sistēmu. Piemērs, testēšanas rīks ģenerē testpiemērus un uzreiz darbina sistēmu.
- Atliktā testēšana ar automātiski darbināmo testpiemēru ģenerēšanu (angl. *offline generation of executable tests*): testpiemēri automātiski tiek iegūti no sistēmas modeļa, bet sistēmas darbināšana ir atlikta un tā notiek manuālā kārtībā. Piemērs, testēšanas rīks ģenerē testpiemērus datoram saprotama formā, kas vēlāk var tikt lietoti sistēmas darbināšanai.
- Atliktā testēšana ar manuāli darbināmo testpiemēru ģenerēšanu (angl. *offline generation of manually deployable tests*): testpiemēri automātiski tiek ģenerēti cilvēkam saprotamā valodā, pēc kuriem vēlāk tiek darbināta sistēma. Piemērs, testēšanas rīks no modeļa ģenerē PDF datnes ar testējamiem gadījumiem, kurus vēlāk lieto testēšanas speciālists manuālajā sistēmas darbināšanā.

Lai nodrošinātu vienotu pieeju testēšanas artefaktu vadībai un strukturēšanai OMG ierosināja izstrādāt testēšanas profilu, kas atbalstītu modeļvadāmo testēšanas procesu. 2007. gadā tika publicēta UML testēšanas profila (angl. *UML testing profile* – UTP) versija 1.0, kas ir bāzēta uz vispārīgo UML 2.0 versiju un kas joprojām ir pēdējā un aktuālā versija. UTP definē valodu testējamo sistēmu artefaktu projektēšanai, vizualizēšanai, specificēšanai, analīzei, konstruēšanai un dokumentēšanai [UTP]. UTP var tikt lietots atsevišķi vai arī var tikt integrēts ar sistēmas UML aprakstu, lai aprakstīt sistēmas un testēšanas artefaktus kopā. UTP paplašina vispārīgo UML ar tādiem testēšanas specifiskiem konceptiem kā testu konteksts (angl. *test context*), spriedums (angl. *verdict*), testpiemērs (angl. *test case*), testu uzvedība (angl. *behavior*) un citiem. Visi šie koncepti ir apkopoti četrās pakotnēs: testu arhitektūra, testu dati, testu uzvedība un laiks. Katra komponente satur noteiktu artefaktu komplektu, lai nodrošinātu šīs sfēras primāro aprakstu.

Testu arhitektūras komplekts ir saistīts ar testpiemēru organizēšanu un realizāciju. Testu konteksti sastāv no viena vai vairākiem testpiemēriem, kas savukārt ir realizēti caur testu komponentēm (angl. *test component*) un verdiktu (*verdict*) testpiemēram nosaka arbitrs (*arbiter*).

Testu uzvedības koncepti apraksta testpiemēru uzvedību, kas ir definētas testa kontekstā. Testu mērķi ir saistīti ar testpiemēriem un apraksta validācijas, kas ir jāveic testpiemēra ietvaros. Testpiemērs sastāv no validācijas aktivitātēm (atjauno testpiemēra verdiktu) un aktivitāšu informācijas pierakstīšanas.

Laicīguma koncepti ir nepieciešami, lai pilnīgi un detalizēti aprakstītu testu kontekstu un testu komponentes. UTP laicīgumu definē ar diviem primitīviem tipiem laiku (angl. *time*) un ilgumu (angl. *duration*), kā arī taimeri (angl. *timer*) un laika zonām (angl. *timezone*).

UTP nodrošina testēšanas artefaktus, nepieciešamus testpiemēru definēšanai. Tas nozīmē, ka UTP var tikt pielietots kā mērķa modelēšanas valoda transformācijas procesā, kura mērķis ir ģenerēt testpiemērus no sistēmas modeļiem.

2.3 Hipotēze par iespēju iegulto sistēmu nefunkcionālo īpašību testēšanu balstīt uz MDA principiem

UTP standarta līdzekļi piedāvā funkcionālo īpašību testpiemēru ģenerēšanu no sistēmas modeļiem [ZAN 2009]. Savukārt, nefunkcionālo īpašību testēšana ar MDA principiem joprojām tiek pētīta. Šajā sadaļā autors izvirza **hipotēzi**:

Sistēmas atbilstības testēšanu nefunkcionālajām prasībām var balstīt uz vispārīgiem MDA modeļu transformācijas principiem, kas, savukārt, var tikt lietoti automātiskai testpiemēru ģenerācijai iegulto sistēmu nefunkcionālo īpašību verificēšanai.

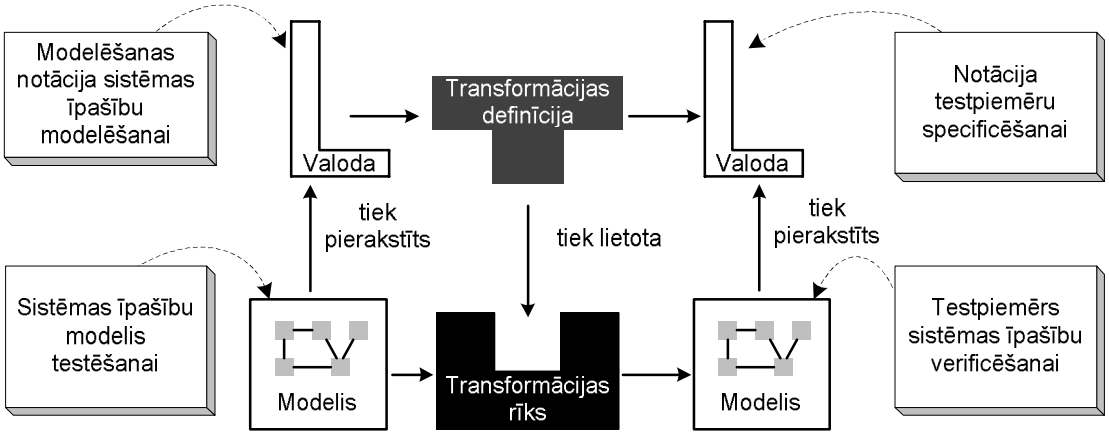
MDA modeļu, metamodeļu un modeļu transformāciju projicējums uz testēšanas procesu ļauj definēt testpiemēru ģenerācijas procesu balstītu uz vispārīgiem modeļu transformācijas principiem. Modeļu implementācija prasa atbilstošu metamodeļu definēšanu, kas būtu pietiekama nepieciešamo īpašību specificēšanai.

Modeļvadāmās programmatūras izstrādes principi ir balstīti uz modeļu transformāciju, kur jaunie modeļi tiek ģenerēti no eksistējošiem modeļiem. Transformācija starp modeļiem tiek nodrošināta ar transformācijas definīciju – transformācijas likumu kopa, kas ir pierakstīta nepārprotamās specifikācijas formā, kur daļējais vai vesels modelis tiek lietots cita vesela vai daļējā modeļa izveidei [KLE 2003]. Transformācijas likumi tiek pierakstīti noteiktajā transformācijas definīcijas valodā. Lai nodrošinātu testēšanas modeļa ģenerēšanu, darbā tiek piedāvāta

transformācijas valodas definīcija, kas ir fokusēta uz testēšanas artefaktu ģenerēšanu un būtu viegli lietojama transformācijas likumu izstrādes laikā.

Automātiskā modeļu transformācija var tikt realizēta ar speciālu rīku, kas ir spējīgs atpazīt un strādāt ar avota un mērķa modeļa objektiem, pielietojot definētus transformācijas likumus avota modelim. Transformācijas rezultātā tiek iegūts mērķa notācijai atbilstošs modelis ar atbilstošiem uzģenerētiem objektiem.

Vispārīgs modeļvadāmās programmatūras izstrādes princips definē vispārīgo shēmu modeļu transformācijai atbilstoši attiecīgam transformācijas shematiskam attēlojumam no [KLE 2003]. 2.8. attēlā ir parādīts izvirzītas hipotēzes koncepts, kas ir projicēts vispārīgajā modeļu transformācijas shēmā.



2.8. att. Autora piedāvātā hipotētiskā risinājuma attēlojums transformācijas koncepcijas shēmā

Par avota modeli autors piedāvā uzskatīt attiecīgās nefunkcionālās īpašības modeli, kas ir veidots UML secību/stāvokļu diagrammas notācijā. Tādējādi UML valoda ir uzskatīta par modelēšanas notāciju, kas specificē avota modeli. Par mērķa modeli autors piedāvā uzskatīt testēšanas piemērus, kas ir izteikti UML testēšanas profilam atbilstošā pieraksta formā. Tādējādi UML testēšanas profila notācija kalpo par mērķa modeļa modelēšanas valodu. Avota modeļa transformācija mērķa modelī tiek veidota izmantojot transformācijas definīciju, kuras pamatā ir abu modelēšanas valodu sintakse un semantika.

Attēlā paradītie principi atspoguļo darbā izvirzīto hipotēzi un reprezentē testēšanas metodes pamata artefaktus ar saitēm starp tiem. Tehniskā hipotēzes realizācija un citas ar metodi saistītas detaļas tiek aprakstītas darba nākošajās sadaļās.

2.4 Nodaļas secinājumi

Sistēmu modeļi aizņem arvien plašāku jomu programmatūras sistēmu izstrādē. Pateicoties UML vienotai modelēšanas valodai, programmu izstrādātājiem parādījās iespējas vispārīgi pieņemtajā veidā specificēt sistēmu statisko struktūru un dinamisko uzvedību. Papildus UML un citiem OMG piedāvātiem standartiem ir pieejams rīku klāsts, kas nodrošina sistēmas modeļu izveidi un to turpmāko lietošanu.

Modeļvadāmās arhitektūras principi paredz sistēmas modeļu lietošanu izstrādes procesā pamata artefaktu lomā, kad balstoties uz tiem tiek iegūta darbināma sistēma un tās testpiemēri. Programmu ģenerēšana no sistēmu modeļiem jau tagad tiek pielietota reālos izstrādes projektos, tomēr testēšanas procesa pilnā automatizācija joprojām tiek pētīta un attīstīta.

Rezumējot nodaļā aprakstītos modeļvadāmās programmatūras principus un testēšanas procesa īpatnības, darba autors var secināt, ka:

- Modeļvadāmā arhitektūra dod iespēju ne tikai risināt programmatūras sistēmas izstrādes uzdevumu, bet tajā definētie formālie modeļu transformācijas principi var tikt pielietoti dažādas specifikas uzdevumu risināšanā un viens no tiem ir iegulto sistēmu testēšana.
- Tas fakts, ka UML tiek lietots sistēmu izstrādē dažādu sistēmas aspektu attēlošanai un ir uzskatīts par standartu sistēmu izstrādē, dod iespēju izmantot eksistējošus UML profilus, rīkus, standartus un pārējus instrumentus savas iegulto sistēmu testēšanas metodes izstrādē.
- Ir iespējams meklēt analoģiju starp modeļvadāmās arhitektūras bāzes elementiem un testēšanas artefaktiem, kas dod iespēju izvirzīt hipotēzi par to, ka modeļvadāmās arhitektūras principi var tikt lietoti iegulto sistēmu nefunkcionālo īpašību testēšanas uzdevuma risināšanā.

3 IEGULTO SISTĒMU ĪPAŠĪBU TESTĒŠANAS METODE

Šajā nodaļā autors detalizēti apraksta piedāvāto metodi iegulto sistēmu testēšanai, kas balstās uz iepriekšējā nodaļā izvirzītās hipotēzes apgalvojumiem un demonstrē metodes lietošanas principus uz abstrakta iegultās sistēmas fragmenta testēšanas piemēra. Kā arī autors analizē piedāvātās metodes priekšrocības un trūkumus un demonstrē efektu, ko dod metodes pielietojums testēšanas uzdevuma izpildē. Tas balstās uz identificēto testpiemēru komplektu analīzi, kas tiek iegūts ar metodes pielietošanu.

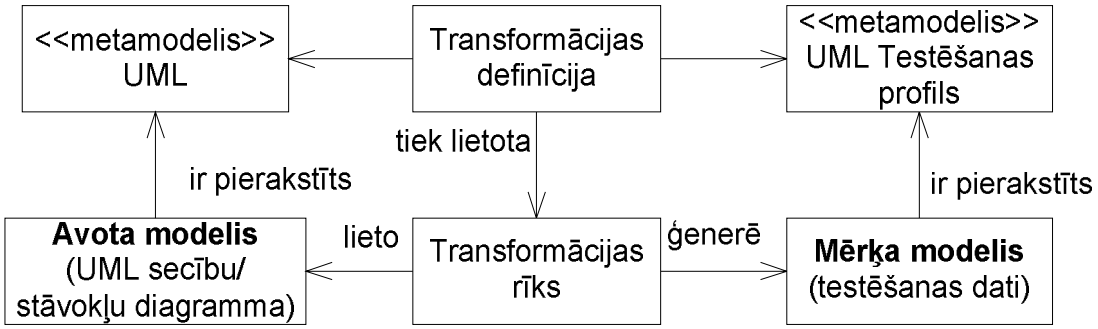
3.1 Avota un mērķa modeļu analīze

Iegulto sistēmu sinhronizācijas, asinhronās darbības un laicīguma īpašības var tikt modelētas ar UML standartizētām diagrammām [GRI 2011a]. UML secību un stāvokļu diagrammas tika izvēlētas par avota modeļiem, balstoties uz šādiem faktoriem:

- UML 2.x versijas ieviesa laika ierobežojumu specificēšanu, nodrošinot to definēšanu vairākos UML modeļos, tajā skaitā secību un stāvokļu diagrammās. UML dinamiskās uzvedības diagrammas ļauj definēt „mīkstos" un „cietos" laika ierobežojumus, kā arī programmu funkcionēšanu atkarībā no laika ierobežojumu sasniegšanas vai pārsniegšanas.
- UML secību diagramma ir viena no sistēmu dinamiskās uzvedības modelēšanas diagrammām, kas tiek pielietota vairāku procesu mijiedarbības specificēšanai. Notācijas principi ļauj modelēt asinhronās darbības un sinhronizācijas procesus, pārklājot darbā aprakstītas šo īpašību modelēšanas prasības.
- UML stāvokļu diagramma ir dinamiskās uzvedības modelēšanas notācija, kas spēj nodrošināt izvēlēta objekta dažādas detalizācijas funkcionēšanas principu modelēšanu, ieskaitot sinhronizēšanu ar darbā definētiem modelēšanas aspektiem.

- UML valoda ir aprobēta un tiek pielietota modeļvadāmās programmatūras izstrādē, ka arī UML tiek uzturēts vairākos MDA atbalsta rīkos [EMF] [EA].
- Ir pieejams XMI standarts, kas definē UML modeļu importēšanas/eksportēšanas datnes formātus plašai modeļu apstrādei ārpus noteiktiem izstrādes rīkiem.

Ņemot vērā aprakstītos faktus, UML secību un stāvokļu diagrammas ir sinhronizācijas, asinhronās darbības un laicīguma īpašību nepieciešamo artefaktu modelēšanas notācija.



3.1. att. Hipotētiskā metode testēšanas modeļa ģenerēšanai no sistēmas modeļa

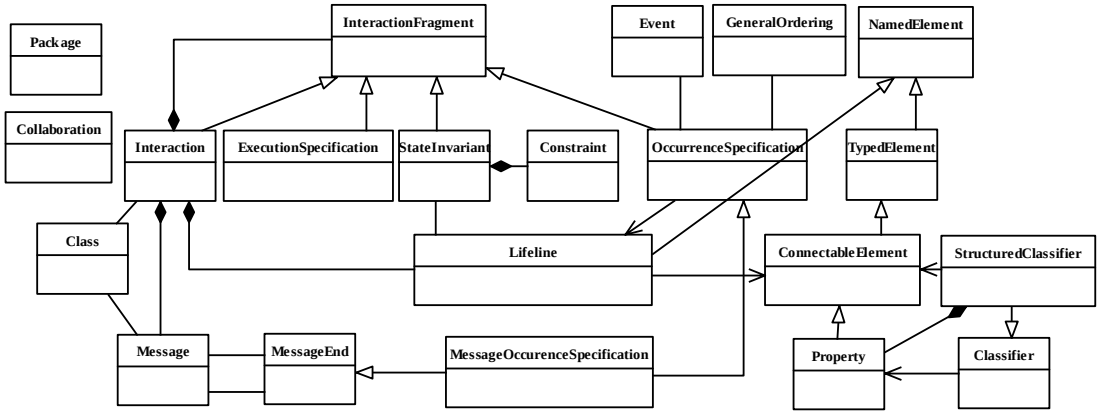
Par mērķa notāciju tika izvēlēts testēšanas profils, ko piedāvā UML. Šī izvēle balstās uz sekojošiem apgalvojumiem:

- UTP ir vienīgais formāli aprakstītais un standartizētais testēšanas artefaktu modelis.
- Vispārīgā UTP struktūra definē tikai galvenos testēšanas modeļa objektus, kas tiek lietoti testēšanas datu organizēšanai, ļaujot papildināt un pielāgot šo modeli no projekta uz projektu, saglabājot pamata modeli.

UTP paredz melnās kastes testēšanas principu [COP 2004] [UTP], kad par testējamās programmatūras uzbūvi nav informācijas un testēšana notiek ārēji, izsaucot testējamo objektu dažādos nosacījumos. Neskatoties uz to, ka šis pētījums balstās uz sistēmu īpašību verificēšanu, kas pēc būtības ir iekšējo procesu analīze un atbilstošu testpiemēru ģenerēšana, UTP tiek izvēlēts par piemērotāko metamodeli testēšanas datu vadībai.

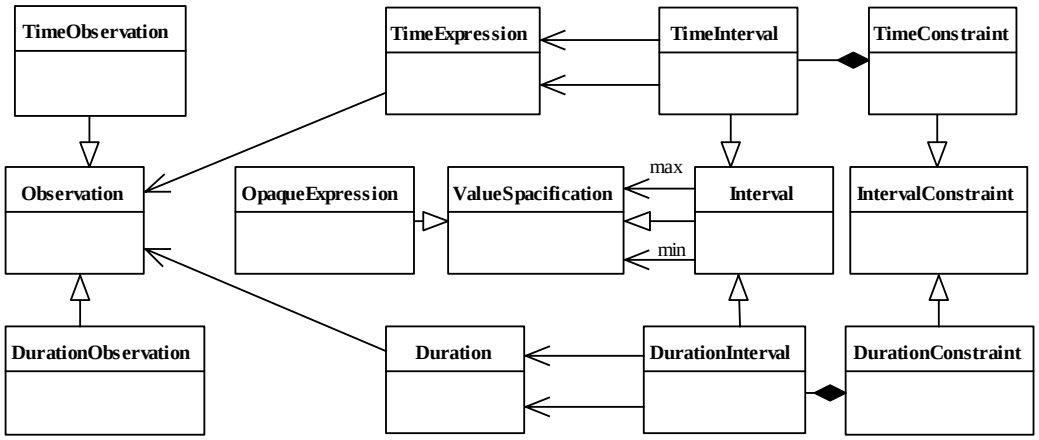
3.1.1 UML secību diagramma

UML secību diagramma tiek lietota sistēmas dinamiskās uzvedības modelēšanai. UML standartizētās secību diagrammas metamodelis apraksta diagrammas objektu tipus un to atribūtus. 3.2. attēlā ir parādīts UML secību diagrammas metamodelis, prezentējot diagrammā attēlotos objektus un atkarības starp tiem.



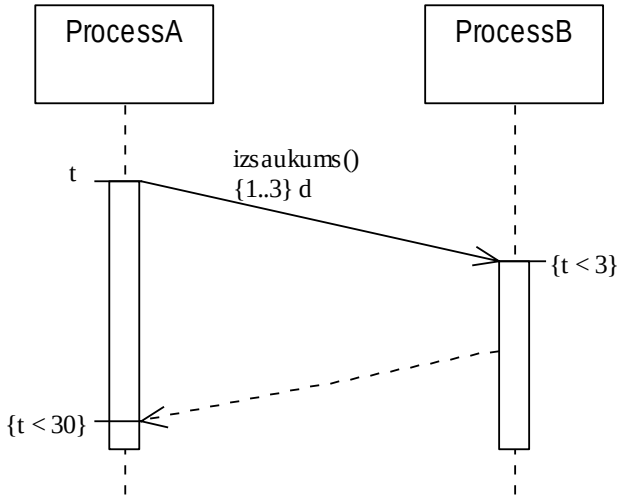
3.2. att. UML secību diagrammas bāzes objektu klašu diagramma

Prezentētais modelis sniedz informāciju par secību diagrammas objektiem, kuri tiek apstrādāti transformācijas laikā. Papildus definētām klasēm, secību diagramma var saturēt arī citus specifisko klašu objektus, piemēram, ar laicīgumu saistītus artefaktus. Laika ierobežojumu attēlošana ir neatņemama modelējamā īpašība un to klašu diagramma ir paradīta 3.3. attēlā.



3.3. att. Laicīguma artefaktu klašu diagramma

Piedāvātās testēšanas metodes viena no testējamām īpašībām ir laicīgums. Sākot ar UML 2.x versijām, secību diagrammās ir iespējams definēt divu veidu laika ierobežojumus: laika un ilguma ierobežojumus. 3.4. attēlā ir parādīts secību diagrammas piemērs ar diviem dažādiem laiku ierobežojumiem.



3.4. att. Secību diagrammas piemērs ar laika ierobežojumiem

Diagramma prezentē divus paralēlus procesus, kur process ProcessA sinhroni izsauc procesu ProcessB un sagaida atbildi. Papildus tam modelī tiek definēts laika mainīgais *t* (angl. *time observation*), kas reprezentē laika vērtību kopš notikuma (izsaukuma sākums), kad tika uzsākta mērīšana, un jebkuru citu tam sekojošo momentu. Atbildes atgriešanas brīdim tiek noteikts laika ierobežojums (angl. *time constraint*) vienāds ar „*t < 30*”. Tas nozīmē, ka diagrammā tika attēlots sinhronais izsaukums ar laika ierobežojumu 30 laika vienības.

Otrais laika ierobežojumu paveids ir ilguma ierobežojums (angl. *duration constraint*), kas reprezentē laika intervālu starp diviem notikumiem. Iepriekš aprakstītājā piemērā tiek attēlots ilguma ierobežojuma mainīgais *d* (angl. *duration observation*) un pats ilguma ierobežojums no 1 līdz 3 laika vienībām starp izsaukuma iniciēšanu un tā apstrādes uzsākšanu.

Laika un ilguma ierobežojumus ir iespējams specificēt dažādos veidos:

- Pozitīvs vesels skaitlis vai nulle (piemēram, „0”, „3”, „30”): biežāk sastopamais ierobežojuma pieraksts, kas norāda laika vienību skaitu starp diviem notikumiem [UML].

- Intervāls starp diviem nenegatīviem skaitļiem (piemēram, „0..1”, „3..5”): norādot laika diapazonu starp minimālo un maksimālo pieņemamo vērtību un tiek apzīmēts „min..max” formā.
- Formula ar mainīgo vērtību, kas reprezentē noteiktu mērījumu, un patvaļīgo konstanta vērtību (piemēram, „t < 30”, „d*2”): ar šādu pierakstu ir iespējams definēt ierobežojumus atkarībā no dinamiskiem mērījumiem.
- Intervāls ar mainīgam vērtībām (piemēram, „d..d+3”, „0..t*2”): šāds pieraksts ir salikums no diviem iepriekšējiem pierakstiem un ļauj definēt pieņemamo laika diapazonu, izmantojot dinamiskus mērījumus.

Papildus laika ierobežojumiem un dažāda veida ziņojumiem, secību diagrammas var saturēt fragmentus (angl. *Fragment)* un vārtejas (angl. *gate*). Fragmenti tiek lietoti, lai aprakstītu specifiskas apstrādes scenārijus. Eksistē sekojošie fragmentu paveidi [BAK 2010] [FAK]:

- alt (angl. *alternatives*) – alternatīvas apstrādes scenāriji, kas atbilst pirmkoda IF..THEN..ELSE konstrukcijām;
- opt (angl. *option*) – neobligātie apstrādes scenāriji, kas var tikt izlaisti. Opt fragments var tikt aizvietots ar *alt* fragmentu, kurā ir aizpildīta tikai viena opcija un pārējās ir tukšas;
- par (angl. *parallel*) – vairāku apstrādes scenāriju potenciāli iespējamā paralēlā apstrāde patvaļīgā secībā;
- loop (angl. *iteration*) – fragments, kas apraksta apstrādes scenāriju n reizes atkārotu apstrādi;
- neg (angl. *negative*) – negatīvs fragments apraksta sistēmas darbības scenāriju, kad ir notikusi kļūda;
- assert (angl. *assertion*) – fragments, pēc kura tiek apgalvots, ka sistēmas turpmākā darbība ir iespējama tikai pie veiksmīgas fragmenta izpildes;
- strict (angl. *strict sequencing*) – fragments norāda stingru tajā definētu operatoru secību, kas nevar tikt izmainīta;
- seq (angl. *weak sequencing*) – apraksta secīgu operatoru izpildi, kas gadījumā ar diviem savā starpā komunicējošiem objektiem ir identiska

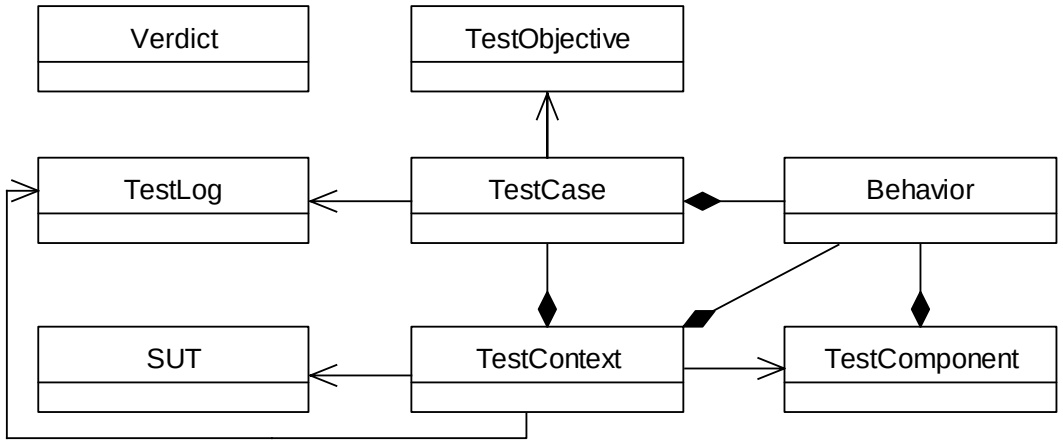
ar strict fragmentu un vairāku objektu gadījuma ir līdzīga *par* fragmentam;

- ignore (angl. *ignore*)– fragmenta operatori var tikt ignorēti apstrādes laikā;
- consider (angl. *consider*) – apraksta operācijas, kas tiek ņemtas vērā, pieņemot, ka pārējas var tikt ignorētas;
- critical (angl. *critical region*) – fragments apraksta kritisko sekciju, kas nevar tikt pārtraukta ar citām operācijām un kurai ir jāizpildās vienā piegājienā;
- break (angl. *break*) – apraksta izņēmuma fragmentu, kad pēc tā izpildes beidzas programmas apstrāde.

Vārtejas var tikt pielietotas secību diagrammās, lai aprakstītu ziņojumu pārsūtīšanu starp fragmentiem, reprezentējot ziņojuma sākumu vai beigas.

3.1.2 UML testēšanas profils

UTP testēšanas profils apraksta būtiskākus konceptus testēšanas datu vadībai, ieskaitot testējamo komponenti, testpiemērus, konkrēto gadījumu ievaddatus un darbināšanas nosacījumus. 3.5. attēlā ir parādīta testēšanas profila klašu diagramma ar šī pētījuma būtiskākiem elementiem [UTP].



3.5. att. UML testēšanas profila pamata elementu klašu diagramma

UTP var tikt pielietots arī automatizētās testēšanas ietvaros. Šajā gadījumā UTP koncepti tiek lietoti kā pamats automatizētās testēšanas vides uzbūvei. 3.5. attēlā

minēto klasifikatoru pielietošana funkcionālajā automatizētajā testēšanā ir detalizēti apskatīta ar atbilstošiem piemēriem [UTP] [BAK 2010].

Pētījuma ietvaros funkcionālā testēšana apzināti netiek apskatīta un fokuss tiek likts uz nefunkcionālām īpašībām. Šī iemesla dēļ, 3.5. attēlā netika attēloti un darbā netiek apskatīti tādi UTP klasifikatori kā *DataPartition*, *DataPool*, *DataSelector*, *InstanceValue* un citi, kas ir saistīti ar testpiemēru ievaddatu specificēšanu. Tajā pašā laikā piedāvātā metode nodrošina manuāli izpildāmo testpiemēru ģenerēšanu bez iespējas automātiski darbināt testējamo sistēmu. Tieši tāpēc darbā netiek apskatīti tādi koncepti kā *Scheduler*, *Timer* un *TestComponent*.

Centrālais klasifikators testēšanas modelī ir *TestContext*, kas reprezentē grupēšanas mehānismu priekš vairākiem testpiemēriem. Testēšanas konteksts apkopo testiem nepieciešamos aspektus, nodrošinot sasaisti ar testējamiem objektiem, to specifisku uzvedību un konkrētiem testpiemēriem.

Klasifikators *TestCase* reprezentē konkrēto testpiemēru ar tam definēto uzvedību. Testpiemēram tiek noteikta konkrēta sistēmas uzvedība, sagaidāmais rezultāts un ar to tiek asociēti apstrādes pieraksti un apstrādes rezultāts. Automatizētās sistēmas darbināšanas gadījumā ar testpiemēru tiek saistītas arī testēšanas komponentes, kas nodrošina verificējamam gadījumam nepieciešamo uzvedību.

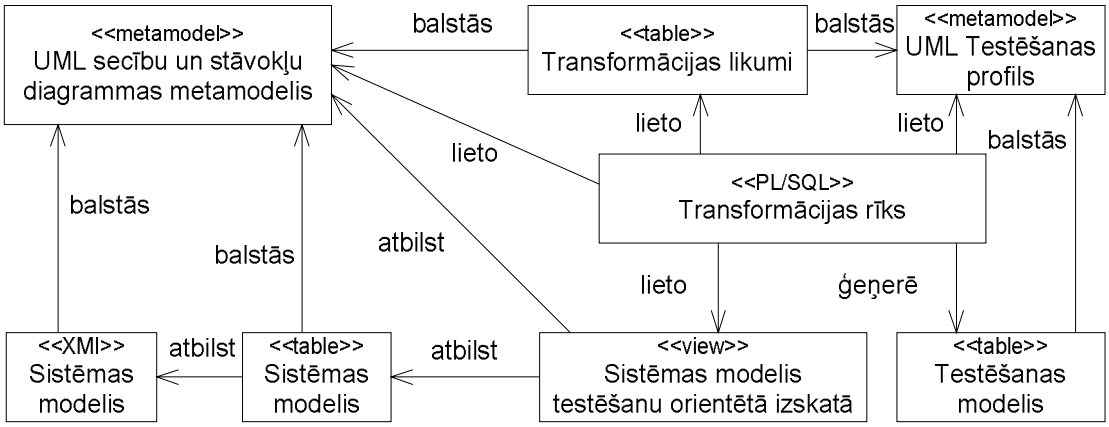
Verificējamās sistēmas uzvedība tiek aprakstīta ar *Behavior* konceptu. Uzvedība var tikt specificēta *TestContext* līmenī, aprakstot testējamās programmatūras visiem izpildāmiem testpiemēriem kopīgo uzvedību. Kā arī uzvedība var tikt attiecināta konkrētiem testpiemēriem. Automatizētājā testēšanā tā tiek attiecināta arī automatizētai testēšanas komponentei, kas nodrošina sistēmas darbināšanu pie noteiktiem apstākļiem.

TestObjective klasifikators nodrošina konkrētu testpiemēru mērķu specificēšanu un ir saistīts ar *TestCase* objektiem, turklāt vienam testpiemēram var tikt attiecināti vairāki testēšanas mērķi. Testēšanas mērķis apraksta sagaidāmo rezultātu un atbilstoši tam tiek uzstādīts testu rezultāts.

Testējamais objekts tiek specificēts ar klasifikatoru *SUT* (System Under Test). Integrācijas un sistēmas testēšanā par testējamo objektu tiek uzskatīta vesela sistēma vai atsevišķas funkcionālas daļas. Savukārt, vienībtestēšana nodrošina konkrēto moduļu un izsaukumu verificēšanu un ar testējamo objektu saprot mazākās komponenšu daļas, pret kurām tiek darbināti testi.

3.2 Metodes detalizēts apraksts

Testpiemēru ģenerēšanas metode balstās uz sadaļas 2.3 izvirzīto hipotēzi par šādas metodes eksistēšanu un potenciālo pielietošanu. Metodes konceptuāls modelis ir parādīts 3.6. attēlā.



3.6. att. Piedāvātās testēšanas metodes konceptuāls modelis

3.6. attēlā prezentētais modelis parāda piedāvātās metodes svarīgākos konceptus un atkarības starp tiem. Izvirzītā metode balstās uz UML modelēšanas notācijas secību un stāvokļu diagrammām un UML testēšanas profilu. Diagrammā abas notācijas ir atzīmētas ar <<*metamodel*>> stereotipu, lai parādītu šo konceptu būtību un atšķirību no pārējiem konceptiem. Abi metamodeļi balstās uz MOF principiem un ir detalizēti aprakstīti 3.1 sadaļā.

Sistēmas modeļi, kas kalpo par metodes sākotnējiem modeļiem, atbilst UML secību un stāvokļu diagrammu modelēšanas principiem. Metode paredz sistēmas modeļu prezentēšanu XMI formātā [XMI], detalizētāk par formātu ir aprakstīts sadaļā 3.3, kas tiek parādīts diagrammā ar atbilstošu <<*XMI*>> stereotipu. XMI formāts plaši tiek pielietots kā koplietošanas standarts UML modeļu apmaiņai starp vairākiem izstrādes procesa atbalsta rīkiem.

Nākamais metodes koncepts ir sistēmas modeļu prezentācija datu bāzes tabulu veidā. Implementējot secību diagrammu kartēšanu no XMI formāta uz tabulām, tiek nodrošināta sistēmas modeļu jaunā tehniskā prezentācija. Tas nozīme, ka sistēmas modeļi tabulās pēc būtības un satura ir tie paši modeļi, kas ir pieejami XMI formātā. Tajā pašā laikā, tas arī nozīmē to, ka tie atbilst arī jau minētajām UML secību un stāvokļu diagrammu metamodelim. Līdzīgi ar <<*XMI*>> stereotipu, <<*table*>> stereotips diagrammā parāda modeļu tehniskās prezentācijas formātu.

Uz testēšanu orientēta sistēmas modeļu reprezentācija ir metodes koncepts, kas nodrošina apstrādājamo modeļu sagatavošanu transformācijas procesam. UML modeļu transformācija uz testpiemēriem ir specifiska ar to, ka, lai uzģenerētu vienu testpiemēru, ir nepieciešams analizēt datus par vairākiem saistītiem objektiem. Pastāv vairākās alternatīvas transformācijas likumu implementācijai šādos gadījumos:

1) Rakstīt komplicētus transformācijas likumus, kuros ir realizēta datu analīze un saistītu objektu piemeklēšana.
2) Izstrādāt vairāku līmeņu transformāciju: sākotnējā transformācija nodrošina saistītu objektu vienotu datu prezentēšanu, iekapsulējot visu saistīto informāciju par konkrētiem objektiem vienā vietā, un nākamie transformācijas cikli analizē jaunus uzģenerētus objektus mērķa modeļa ģenerēšanai.
3) Izveidot apkopotu vai kādu citu datu reprezentāciju ar datu bāžu līdzekļiem, tādējādi vienkāršojot sākotnējo modeli un sagatavojot to transformācijai.

Metodes realizācijā tiek pielietota 3. alternatīva un uz datu bāzes tabulām pārnestie UML modeļi tiek attēloti skatu veidā (angl. *view*), parādot to diagrammā ar atbilstošu <<*view*>> stereotipu. Pie šādas pieejas skati ir vienkāršotais kopsavilkums no komplicētā sistēmas modeļa, prezentējot testēšanai nepieciešamos aspektus. Skati tiek konstruēti balstoties uz sistēmas modeļiem, iekapsulējot nepieciešamos elementus un atribūtus pieprasītājā formātā. Skati neievieš jaunus datus un satur tikai un vienīgi oriģinālo modeļu datus. Balstoties uz šiem apgalvojumiem, var spriest par skatu atbilstību sākotnējo sistēmas modeļu metamodelim.

Transformācijas rīks nodrošina mērķa modeļa ģenerēšanu, t.i. transformācijas likumu nolasīšanu un apstrādi attiecībā pret vienkāršoto sistēmas modeli un testēšanas modeļa ģenerēšanu, kas atbilst UML testēšanas profilam. Rīka implementācija realizē noteiktās transformācijas likumu notācijas apstrādi, kas detalizēti ir apskatīta 3.2.1 sadaļā. Pats rīks ir realizēts PL/SQL programmēšanas valodā, parādot to diagrammā ar <<*PL/SQL*>> stereotipa pielietošanu.

Transformācijas likumu kopa ir koncepts, kas apraksta nosacījumus sistēmas modeļu transformācijai uz testēšanas modeli. Likumi tiek saglabāti atsevišķā tabulā, nodrošinot pieeju to modificēšanai un nolasīšanai transformācijas procesa laikā. Transformācijas likumi atbilst minētai iepriekš definētai notācijai, kas savukārt balstās uz UML secību un stāvokļu diagrammu un UML testēšanas profila metamodeļiem.

Mērķa modelis apraksta testēšanai nepieciešamos aspektus un atbilst UML testēšanas profilam. Līdzīgi sistēmas modeļu tabulu prezentācijai, testēšanas modelis sastāv no savā starpā saistītu tabulu kopas, attēlojot testēšanas profilā aprakstītus klasifikatorus. Katram klasifikatoram tiek sagatavota tabula, kurā transformācijas laikā tiek ierakstīti ģenerētie dati, balstoties uz transformācijas likumiem un modeli.

3.2.1 Transformācijas likumu notācija

Viens no līdzekļiem, ko var izmantot transformāciju definēšanā ir objektu vadības grupas standarts QVT, kas ir minēts 2.1 nodaļā. Savukārt, balstoties uz to, ka sistēmas modeļa transformācija uz testēšanas modeli ir specifiskā uzdevuma transformācija noteiktajā izstrādes vidē, tad QVT valodas pielietošana ir apgrūtināta un ir nepieciešams piedāvāt speciāli veidotu transformācijas valodu, lai būtu iespējams uzdot transformācijas, balstoties uz testēšanas uzdevuma specifiku.

Transformācijas likumu mērķis ir aprakstīt nosacījumus sistēmas modeļu transformācijai uz testēšanas modeli. Citiem vārdiem sākot, tie ir testpiemēru veidošanas principi, kas tiek lietoti arī manuālās testēšanas procesā. Tā, piemēram, viena no populārākajām testpiemēru ģenerēšanas metodēm ir robežvērtību analīze ar sadalīšanu ekvivalences klasēs [SPI 2007]. Balstoties uz šo metodi, populārākie kritēriji testpiemēru ģenerēšanai galējā izpildes termiņa (piemēram, $t < 10$) verificēšanai ir kādas funkcijas darbības pārbaude pie apstrādes laika vienāda ar *T-1*, *T* un *T+1* (*T+1* gadījuma pārbaude ir neobligāta pēc [SPI 2007] norādījumiem un var tikt neapskatīta gadījumos, kad ir jāfokusējas uz testpiemēru kopas minimizēšanu), kur *T* ir konkrētā ierobežojuma vērtība (iedotam piemēram tas ir 10) ar sagaidāmo rezultātu *veiksmīgi*, *neveiksmīgi*, *neveiksmīgi*. Aprakstītais piemērs parāda gadījumus, kādi būtu jānoklāj ar transformācijas likumu starpniecību.

Lai nodrošinātu transformācijas likumu pielietošanu sistēmas modelim un testpiemēru ģenerēšanu, tam ir jāsatur verificējamo elementu izvēles kritēriji un testējamās uzvedības kritēriji. Darbā autors piedāvā sekojošo notāciju transformācijas likumu definēšanai (likums tiek pierakstīts vienā rindā):

```
FOR: <objekta tips> <objekta nosaukums>
WITH: <priekšnosacījums>
DO: <darbība> WITH: <testējamās uzvedības apraksts>
WHICH: <sagaidāmais rezultāts>
```

Zemāk ir aprakstīts definētas notācijas katras sekcijas formāts un iespējamās pieļaujamās vērtības:

`<objekta tips>` – definē objekta tipu, kuram tiks ģenerēti testpiemēri. Piemēram, „*message*" objekta tips specificē ziņojuma tipu.

`<objekta nosaukums>` – objekta nosaukums, kas tiks izvēlēts validācijai un apstrādei. Gadījumos, kad transformācijas likums ir jāpielieto visiem definēta tipa objektiem, tad jādefinē „*" simbols. Savukārt, konkrēto objektu specificēšanai ir jādefinē šī objekta pilns nosaukums (var definēt ar lieliem un maziem burtiem, jo transformācija spēj apstrādāt abus variantus).

`<priekšnosacījums>` – priekšnosacījuma sekcija apstrādājamo objektu piemeklēšanai. Priekšnosacījums tiek pierakstīts SQL vaicājuma WHERE teikumam atbilstošajā formātā [COND]. WHERE teikuma formāts ļauj definēt vienkāršus nosacījumus konkrēto atribūtu validēšanai, vai arī iekļaut EXISTS nosacījumus un aprakstīt loģiskus priekšnosacījumus komplicēto testpiemēru ģenerēšanai.

`<darbība>` – darbība, kas ir jāveic ar piemeklēto objektu. Pašlaik tiek uzturēta „CREATE TC" darbība, kas paredz jauna testpiemēra ģenerēšanu. Šādas struktūras pielietošana ļauj nākotnē definēt arī citas darbības, piemēram, CREATE MULTIPLE TCS/UPDATE TC/DELETE TC (vairāku testpiemēru izveidošana, testpiemēru atjaunošana un dzēšana), dažādu papildus operāciju veikšanai ar testpiemēriem.

`<testējamās uzvedības apraksts>` – uzvedības aprakstīšanas sekcija, kur tiek definēti ģenerētā testpiemēra nosacījumi. Tiek uzturētas 2 alternatīvas uzvedības aprakstīšanai:

- statiskais pieraksts – uzvedības parametri tiek definēti ar statiskām vērtībām. Piemēram, "durationconstraint=0", "timingconstraint=99" vai līdzīgi.
- dinamiskais pieraksts – šāds pieraksts tiek definēts laicīguma ierobežojumu specificēšanai. Dinamiskā uzvedība tiek definēta ar mērķi pielietot konkrētas vērtības no sistēmas modeļa. Laicīguma ierobežojumi (ilguma un laika ierobežojumi) pēc UML specifikācijas var tikt specificētas ar laika intervāliem: kā intervāls starp minimālo un maksimālo pieļaujamo vērtību „min..max" (piemēram, „1..6") vai kā

intervāls starp bezgalību un kādu noteiktu vērtību (piemēram, „t <4” vai „d>=30”). Šādos gadījumos uzvedības definīcijā var tikt lietoti 2 rezervēti vārdi „min” un „max”, reprezentējot minimālo un maksimālo definēto vērtību. Šie vārdi tiek lietoti tikai un vienīgi šo vērtību iegūšanai, lai testējamā uzvedībā automātiski noteikt konkrētas vērtības. Piemēram, var definēt sekojošo uzvedību, “durationconstraint = min +1” vai “timingconstraint = max + 2”.

`<sagaidāmais rezultāts>` – sekcija ir paredzēta manuālai testpiemēru rezultāta definēšanai. Neskatoties uz šo definēto rezultātu, transformācijas procesā rīks nodrošina iepriekšējā sekcijā aprakstītas uzvedības validēšanu attiecībā pret sistēmas specifikāciju un automātisko sagaidāmo rezultātu izskaitļošanu. Manuāla sagaidāmā rezultātā definēšana ir izdalīta kā neobligāta sekcija izņēmuma gadījumu definēšanai, kad automātiski izrēķināt sagaidāmo rezultātu nav iespējams, nepietiekamas specificēšanas vai citu iemeslu dēļ.

Zemāk ir sniegs transformācijas likuma piemērs atbilstoši iepriekš aprakstītai notācijai.

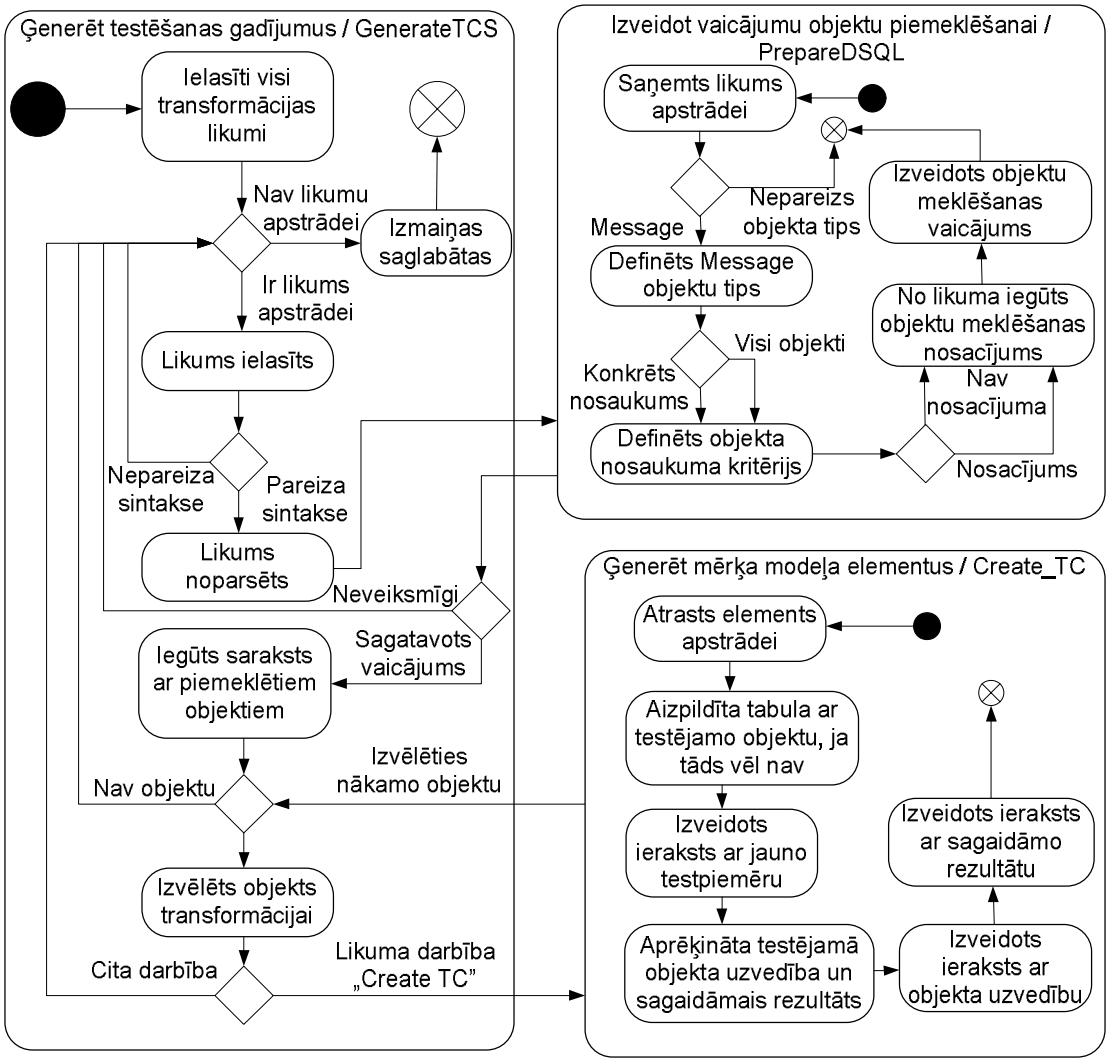
```
FOR: MESSAGE * WITH: durationconstraint is not null DO: CREATE TC
WITH: durationconstraint = max + 1 WHICH: EXPECTED_RESULT = FAILED
```

Aprakstīta transformācijas likuma mērķis ir uzģenerēt visus iespējamos testpiemērus priekš visiem ziņojumiem ar definēto ilguma ierobežojumu, kur ziņojuma apstrādes ilgums pārsniedz specificēto laiku par 1 daļu.

3.2.2 Transformācijas algoritma realizācija ar rīka atbalstu

Rīks nodrošina vienvirziena transformācijas likumu definēšanu. Virziens no testēšanas modeļa uz sistēmas modeli netiek apskatīts šajā metodē. Pirmkārt, sistēmas modeļa papildināšana, balstoties uz jauniem testpiemēriem, ir nekorekts process un tas var novest pie sistēmas modeļa neatbilstības iepriekšējiem sistēmas modeļiem, tajā skaitā prasībām izstrādājamai sistēmai. Otrkārt, pilnvērtīga atpakaļejošā virziena realizācija prasītu sistēmas modeļa atjaunošanu līdz pat XMI standarta datnei, jo tas tiek lietots kā sākotnējais modelis. Treškārt, atpakaļvirziens nav iespējams vienkāršotas sistēmas reprezentācijas dēļ, kas ir realizēta skatu veidā.

Transformācijas rīks ir balstīts uz ciklisko likumu apstrādes principu un tā implementācijas algoritms ir parādīts ar UML stāvokļu diagrammas palīdzību 3.7. attēlā.



3.7. att. Transformācijas rīka UML stāvokļu diagramma

Transformācijas procedūra katram likumam veic sintakses validāciju atbilstoši definētai notācijai un korektas sintakses gadījumā veic apstrādājamo sistēmas modeļa elementu piemeklēšanu, balstoties uz likumā definētiem kritērijiem. Likuma kritēriji ļauj attiecināt to vairākiem sistēmas objektiem, kas savukārt nozīmē, ka transformācija ir jāveic vairākiem piemeklētiem objektiem. Katram šādam piemeklētam elementam tiek ģenerēts testpiemērs ar noteiktu uzvedību un sagaidāmo rezultātu. 3.7. attēlā parādīta stāvokļu diagramma apraksta būtisko funkciju un procedūru apstrādes algoritmus. *GenerateTCS* procedūra ir galvenā programmas

vienība, kas tiek laista un kura veic nepieciešamo funkciju un procedūru izsaukšanu, lai rezultātā iegūtu mērķa modeli. *PrepareDSQL* funkcija nodrošina dinamiskā vaicājuma izveidi, kas tiek lietots piemēroto objektu piemeklēšanai transformācijas procesam. *Create_TC* procedūras uzdevums ir izveidot mērķa modeļa elementus atbilstoši piemeklētam objektam un apskatāmām transformācijas likumam. Transformācijas procedūra modeļa apstrādei pielieto arī citas funkcijas un procedūras, kuru apraksts ir sniegts pielikumā 5.

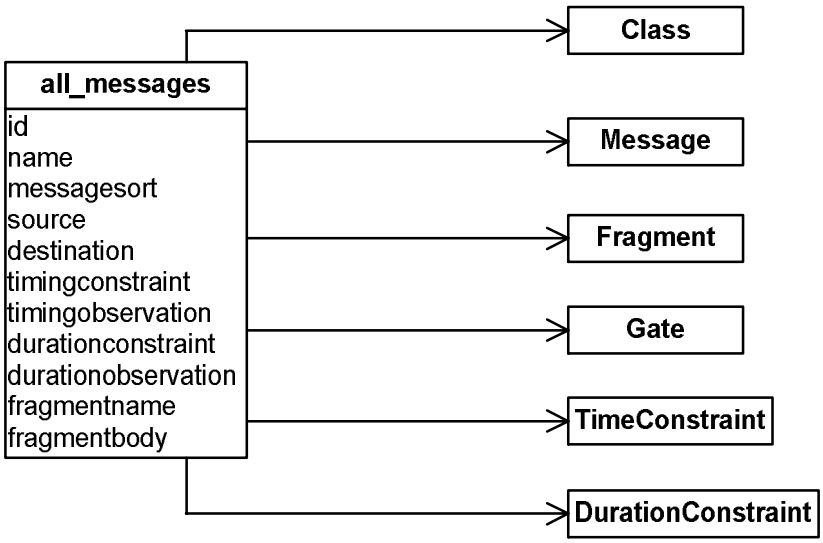
3.7. attēlā prezentētais algoritms sākās ar procedūras *GenerateTCS* izsaukšanu. Transformācijas procedūra paredz, ka transformācijas likumi ir iepriekš sagatavoti un atrodas *TRANSFORMATION_RULES* tabulā, kā arī datu bāzē glabājas sistēmas modelis. Pēc procedūras palaišanas notiek transformācijas likumu ielasīšana, ko nodrošina statisks *SELECT* vaicājums, definētais *GenerateTCS* procedūrā. Gadījumā, ja tabulā nav neviena likuma, procedūra beidz darbības. Ja no tabulas tika ielasīti transformācijas likumi, tad secīgi tiek ielasīts un apstrādāts katrs no tiem. Likuma apstrāde sākas ar tā sintakses pārbaudi atbilstoši 3.2.1 sadaļā aprakstītai notācijai. Likumi, kuriem tiek konstatētas neatbilstības, tiek izlaisti no turpmākās apstrādes un algoritms ielasa nākamo, ja tādi vēl ir. Atbilstošās sintakses gadījumā, likums tiek parsēts un tiek identificētas transformācijas likumu notācijai atbilstošās sastāvdaļas. Kad likums ir noparsēts, tiek izsaukta *PrepareDSQL* funkcija, kas sagatavo dinamisko vaicājumu piemēroto objektu piemeklēšanai. Dinamiskais vaicājums tiek izveidots katram transformācijas likumam un ir balstīts uz likumā definētu objektu tipa, tā nosaukumu un uzvedības nosacījumiem. Saņemot likumu kā parametru, funkcija veic attiecināma objekta tipa pārbaudi. Secību diagrammas ir balstītas uz ziņojumu pārsūtīšanu starp vairākiem objektiem un to atribūti, ieskaitot ziņojuma avota un mērķa objektus, laicīguma un citas īpašības ir saistītas ar pašiem ziņojumiem. 3.2. sadaļā piedāvātais koncepts paredz vienkāršota modeļa pielietošanu un secību diagrammām šāds modelis ir balstīts uz ziņojuma objektiem un aprakstīts nākamajā sadaļā. Funkcija *PrepareDSQL* validē transformācijas likumu pēc tajā definēta objektu tipa un sakritības gadījumā konstruējamam vaicājumam pieliek vienkāršota modeļa nosaukumu (šajā gadījumā skats *all_messages*). Tālāk, ja likumā ir definēts konkrēts objekta nosaukums, tad pieliek objekta nosaukumu vaicājumam, bet ja ir definēts „*„ simbols, tad vaicājums paredz visu norādītā tipa objektu piemeklēšanu. Gadījumā, ja priekšnosacījuma daļa tiek definēta transformācijas likumā, tad tā arī tiek pielikta vaicājumam. Funkcijas izpildes rezultātā tiek izveidots

SQL valodas *SELECT* vaicājuma sintaksei atbilstošs teikums. Kļūdas gadījumā funkcija atgriež tukšo lauku, paziņojot par neveiksmīgu vaicājuma izveidi. Veiksmīga rezultāta gadījumā *GenerateTCS* funkcija palaiž vaicājumu un iegūst sarakstu ar objektiem, kuriem jāpiemēro transformācija. Ja vaicājumu neizdevās izveidot, tad procedūra sāk apstrādāt nākamo likumu. Ja vaicājuma izpildes rezultātā netiek atrasti objekti, tad procedūra analizē nākamo likumu. Gadījumā, ja tiek piemeklēti objekti, procedūra cikliski apstrādā katru no tiem. Pirms sākt piemeklēta objekta apstrādi, procedūra pārbauda likuma definētu darbību un, ja darbība ir „Create TC", tad tiek izsaukta *Create_TC* procedūra. Testēšanas modeļa aizpilde notiek secīgi, sākot ar testējamā objekta identificēšanu un pieglabāšanu (ja tāds vēl nav testējamo objektu tabulā *SUT*). Tad notiek testpiemēra unikālā ieraksta izveide *TESTCASE* un *TESTCONTEXT* tabulās. Lai nodrošinātu testpiemēru un citu elementu unikalitāti, datu bāzē ir izveidotas un tiek pielietotas secības. 3.2.1 sadaļā ir aprakstīts uzvedības definēšanas algoritms, ar kuru automātiski var tikt izrēķināts konkrētā testpiemēra sagaidāmais rezultāts. Testpiemēra rezultāta izrēķināšana notiek *PREPARE_BEHAVIOR* funkcijā, kas tālāk tiek pieglabāts *TESTOBJECTIVE* tabulā. Testpiemēra uzvedība tiek izrēķināta no sistēmas modeļa un transformācijas likuma nosacījuma un tiek ierakstīta *TESTOBJECTIVE* tabulā. Procedūras izpildes laikā tiek aizpildītas mērķa modeļa tabulas, kas ir parādītas 3.9. attēlā. Apstrādājot vienu objektu, procedūra sāk apstrādāt nākamo, ja tādi vēl ir. Gadījumos, ja procedūru un funkciju izpildes laikā notiek neparedzēta kļūda, tiek parādīts kļūdas ziņojums un izmaiņas netiek saglabātas.

3.2.3 Sistēmas modeļa vienkāršotais prezentējums

Metodes realizācijā tiek pielietota sistēmas modeļa vienkāršošana un uz RDB pārnestie UML modeļi tiek attēloti skatu veidā, nodrošinot oriģinālo datu lietošanu bez papildus kopiju starpniecības. Skatu pielietošana modeļu vienkāršošanā ir efektīvs līdzeklis kā apvienot savā starpā saistītus datus vienotā vietā un formātā. Pateicoties šādai reprezentācijai, transformācijas likumi var fokusēties uz loģiskiem nosacījumiem un nepievērst uzmanību tehniskām detaļām. Aprakstītā pieeja nodrošina transformācijas notācijas atbilstību testēšanas uzdevumam un to uztveršanu.

Laicīguma īpašības verificēšanai ar secību diagrammu tika izveidots *all_messages* skats, kas iekapsulē testiem nepieciešamo informāciju par testējamo sistēmu un reprezentē to viegli pieejamā formā. 3.8. attēlā ir parādīta klašu diagramma ar saitēm starp oriģinālām klasēm un vienkāršotu *all_messages* klasi (pēdējo var uzskatīt par klasi, jo skatu formāts apraksta visus skatā piemeklētos elementus).



3.8. att. Secību diagrammas ziņojumu vienkāršotais prezentējums

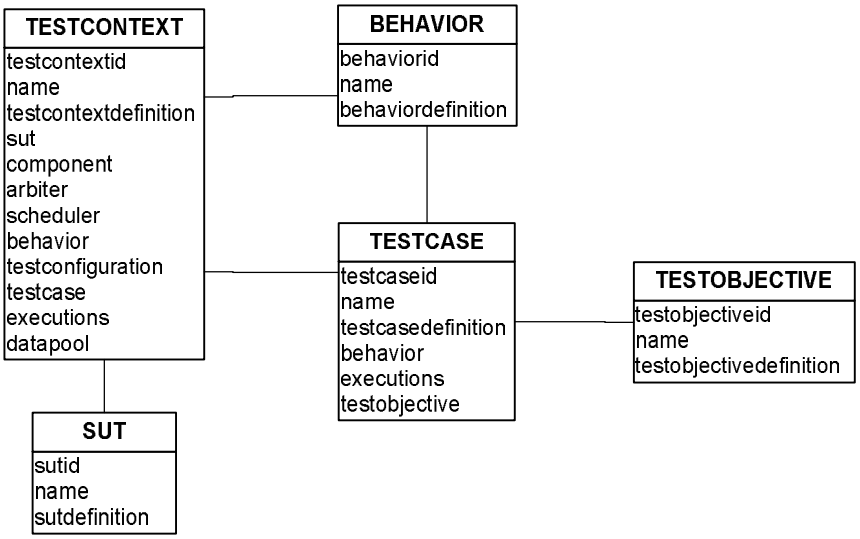
3.8. attēlā ir parādītas tikai būtiskākas secību diagrammas klases, kuru dati apstrādātā vai oriģinālā formā tiek prezentēti *all_messages* skatā. Skatā iekapsulētie dati tiek parādīti attēlā kā *all_messages* klases atribūti. Lai nodrošinātu aprakstīto datu šādu interpretēšanu, tiek lietoti papildus skati sākotnēja modeļa sarežģītās struktūras dēļ.

Skatu pielietošana modeļa sagatavošanai pirms transformācijas, ļauj definēt jaunus un papildināt eksistējošos skatus ar jauniem nepieciešamiem datiem no sākotnējā modeļa. Šāda iespēja nodrošina uzražoto transformācijas likumu aktualitāti un neprasa to atjaunošanu pie vienkāršota modeļa izmaiņām.

3.2.4 Testēšanas modeļa struktūra

Transformācijas rezultātā iegūtais testēšanas modelis atbilst darbā pieminētam un aprakstītam UML testēšanas profilam. UTP definē būtiskākos konceptus testēšanas modeļa organizēšanai un konkrēto atribūtu specificēšanas un implementēšanas

īpašības tiek atstātas konkrēto izstrādes projektu pārziņā, piedāvājot vispārīgas atribūtu definīcijas. 3.9. attēlā ir parādīta implementēta testēšanas modeļa klašu diagramma ar darbā pielietotiem atribūtiem, kas ir daļa no UTP profila.

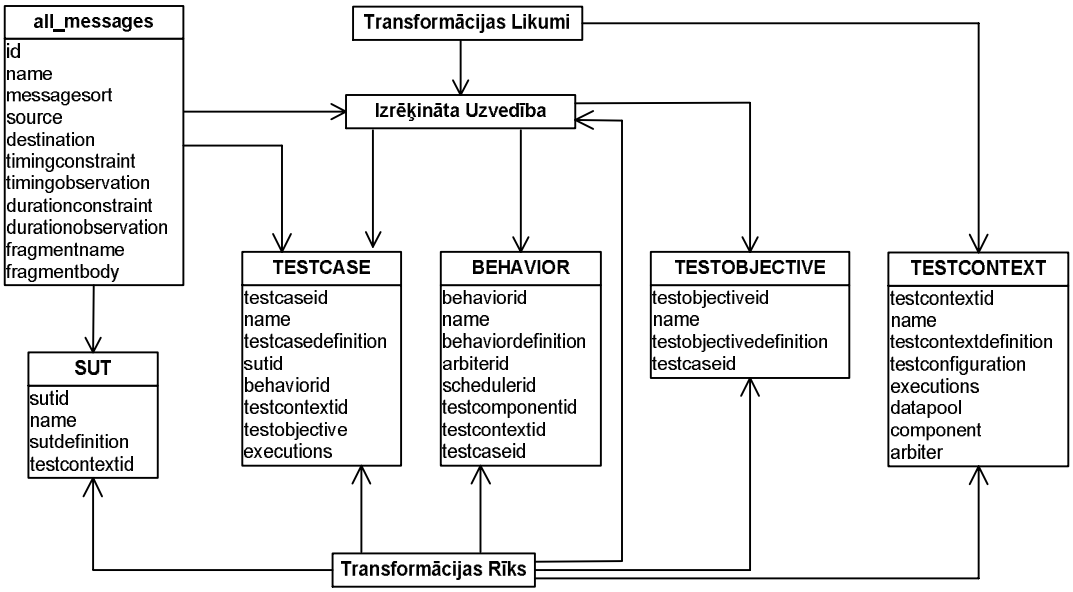


3.9. att. Mērķa modeļa klašu diagramma ar atribūtiem

Attēls tajā pašā laikā prezentē aprakstīto tabulu atribūtus un mērķa tabulu struktūru. Diagrammā parādītas klases un to atribūti pilnīgi atbilst UTP definētai XMI prezentācijai, kas apraksta arī klašu atribūtus. Pateicoties *testcontextdefinition*, *behaviordefinition*, *testcasedefinition*, *testobjectivedefinition* un *sutdefinition* atribūtu vispārīgam formātam, ir iespējams aizpildīt tos ar nepieciešamiem datiem un pielietot nefunkcionālo īpašību testēšanai. Ņemot vērā to, ka darbā netiek pielietotas parējās UTP profila klases, daļa no aprakstītie atribūtiem arī netiek apskatīta (*component*, *arbiter*, *scheduler*, *testconfiguration*, *executions*, *datapool*), jo tie nodrošina sasaisti ar neapskatītām klasēm.

3.2.5 Datu kartēšana starp avota un mērķa modeļiem

Datu ģenerēšana mērķa modelī notiek transformācijas gaitā atbilstoši definētiem transformācijas likumiem. Mērķa modeļa datu ģenerēšana balstās uz trīs avotiem: sistēmas modeļiem, transformācijas likumiem un transformācijas rīku. 3.10. attēlā ir parādīti mērķa modeļa datu kartēšanas un ģenerēšanas principi.

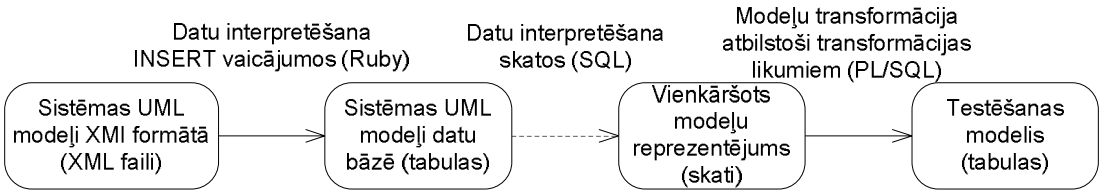


3.10. att. Mērķa modeļa datu kartēšana un ģenerēšana

Lai uzģenerētu konkrēto testpiemēru uzvedību, transformācijas rīks analizē transformācijas likumus attiecībā pret sistēmas modeli un iegūst nepieciešamos testpiemērus un to uzvedību. Kā arī, analizējot sistēmas ierobežojumus, transformācijas rīks izskaitļo katram testpiemēram sagaidāmo rezultātu. Balstoties uz informāciju par ziņojumiem, to avotiem un galamērķiem, tiek aizpildīta informācija SUT tabulā. Savukārt, transformācijas rīks visām mērķa tabulām ģenerē tehnisko informāciju. Par tehnisko informāciju tiek uzskatīti ierakstu identifikācijas lauki, kas tiek ģenerēti unikalitātes nodrošināšanai no speciāli sagatavotām datu bāzes vadības sistēmas (turpmāk DBVS) secībām, un sasaistes nodrošināšanas lauki starp mērķa tabulām.

3.3 Metodes realizācijas principi

Lai nodrošinātu metodes pielietošanu dažādās izstrādes vidēs un izvairītos no papildus rīku atkarībām, piedāvātā metode balstās uz eksistējošiem modeļvadāmās programmatūras principiem un standartiem. 3.11. attēlā ir parādīti tehniskās metodes realizācijas principi.



3.11. att. Metodes tehniskās realizācijas principi

3.11. attēls prezentē metodes izpildāmos soļus no sākotnējā UML sistēmas modeļa līdz UML testēšanas profilam. Metode paredz transformācijas implementēšanu divos soļos. Pirmajā solī notiek sākotnējo UML modeļu pārveidošana no XMI standarta datnes uz datu bāzes tabulām ar speciāli izstrādāto kartēšanas rīku. Ar iepriekš definētiem skatiem uz UML modeļiem, tiek iegūta vienkāršotā sistēmas modeļu prezentācija, kas tālāk tiek lietota transformācijās. Otrajā solī notiek pati transformācija, lasot un apstrādājot vienkāršotus UML modeļus ar definētiem transformācijas likumiem. Transformācijas rezultātā tiek iegūts testēšanas modelis ar uzģenerētiem testpiemēriem. Zemāk detalizēti tiek apskatīta metodes realizācija un tiek pamatotas un aprakstītas pielietotās tehnoloģijas.

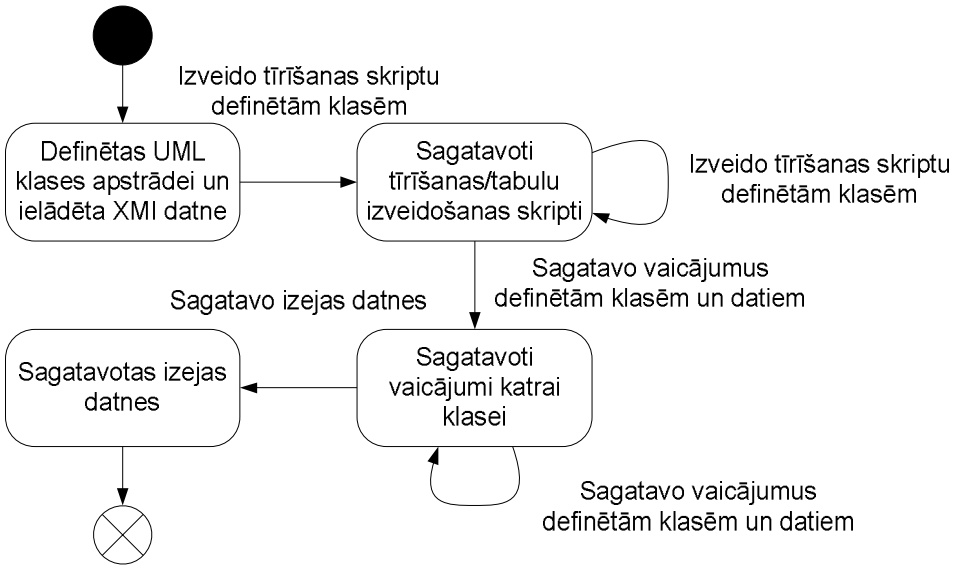
3.3.1 XMI datu kartēšana uz datu bāzes tabulām

UML modeļu prezentēšanai var tikt lietots XMI standarts – integrētais ietvars XML datu un objektu definēšanai, apmaiņai, manipulēšanai un integrācijai. Uz XMI bāzētie standarti tiek lietoti aplikāciju, rīku un datu noliktavu integrācijai. XMI nodrošina likumus, ar kuriem XML shēma var tikt ģenerēta no jebkura XMI transformācija derīga MOF bāzēta metamodeļa. XMI nodrošina kartēšanu no MOF uz XML, kas atjaunojās līdz ar MOF un XML tehnoloģijām, nodrošinot aktuālo versiju atbilstību [XMI]. XMI standarta lietošana nodrošina neatkarību no UML modeļu izstrādes rīkiem. Tā, piemēram, modeļi var tikt izveidoti Enterprise Architect [EA], UModel [ALT], Rational Rose [RAT 2003] vai citos rīkos, kas nodrošina UML modeļu definēšanu un to eksportēšanu XMI formātā (INSERT vaicājumos datu interpretēšanas programmas tekošā realizācija nodrošina XMI 2.1 versija datņu apstrādi). Tālāk izveidotie modeļi var tikt lietoti testpiemēru ģenerācijā.

Sistēmas modeļi pēc būtības sastāv no savā starpā saistītiem objektiem, kas ir noteiktu objektu tipu (klase, ziņojums, atribūts un citi) vairāku eksemplāru kopa, kas salikumā dod sistēmas modeli. Tajā paša laikā relāciju datu bāzes (angl. *relational*

database jeb RDB) ļauj definēt tabulas un ierakstīt tajās atbilstošās struktūras datus. Šāds skats uz sistēmas UML modeļiem ļauj secināt, ka relāciju datu bāzes ir piemērotas UML modeļu glabāšanai. Šāda pieeja nav jauna un eksistē vairāki darbi, kas paredz UML modeļu glabāšanu relāciju datu bāzēs [WU 2002] [BES]. RDB ir viena no populārākām tehnoloģijām strukturētās informācijas glabāšanai, koplietošanai, apmaiņai un apstrādei. Kaut arī minēto pētījumu argumentācija par RDB pielietošanu pārsvarā balstās uz informācijas par UML modeļiem attālināto koplietošanu un vadību, no tiem var secināt par izvēlēto tehnoloģiju aktualitāti un pielietošanu UML modeļu glabāšanai.

UML modeļu kartēšana no XMI formāta uz tabulu INSERT vaicājumiem tiek nodrošināta ar speciāli tam izstrādāto kartēšanas rīku. Eksistē vairāki gatavi rīki, kas ir spējīgi transformēt datus no XML datnēm uz datu bāzes tabulām. Šie rīki pārsvarā nodrošina triviālo struktūru apstrādi ar vienkāršiem XML tagiem. UML modeļu implementācijas XMI formātā padara XML datni par sarežģītu un prasa īpašu pieeju datu apstrādē. XMI formātā plaši tiek pielietoti tagu atribūti, nodrošinot nepieciešamo datu iekapsulēšanu XML tagos. Tieši tāpēc izstrādātājā rīkā ir implementēta vajadzīgo datu apstrāde un interpretācijas INSERT vaicājumos. Otrais iemesls jaunā rīka izstrādei ir rīka turpmāka uzlabošana un atbilstība jaunām XMI un UML versijām, kas ienes XML formātā jaunu sintaksi un jaunus elementus. 3.12. attēlā ir paradīta UML stāvokļu diagramma aprakstītam kartēšanas rīkam.



3.12. att. XMI uz RDB kartēšanas rīka stāvokļu diagramma

Rīka izstrādei tika izvēlēta Ruby programmēšanas valoda un par pamatu šādai izvēlei tika ņemti vērā sekojošie apgalvojumi:

- valoda ir viegli un efektīvi pielietojama nelieliem konkrētiem uzdevumiem;
- ir pieejamas vairākas bibliotēkas, tajā skaitā REXML, nodrošinot manipulācijas ar XML datnēm, kas ir šī uzdevuma viens no galvenajiem aspektiem;
- valoda ir pietiekama, lai izstrādātu XML datņu apstrādi saskaņā ar noteiktiem loģiskiem nosacījumiem un jauno datņu veidošanu atbilstoši pieprasītai sintaksei.

Rīks ir izstrādāts saskaņā ar principu, ka XMI datnes apstrādes rezultātā tiek izveidota SQL vaicājumu datne ar datiem iespraušanai tukšās tabulās. Tas nozīmē, ka datu bāzē no jauna tiks ielasīti UML modeļi, nodrošinot XMI datnes reprezentāciju RDB formātā.

3.3.2 Mērķa modeļa prezentācija

Līdzīgi ar UML sistēmas modeļiem, UTP arī ir balstīts uz UML un MOF un tāpēc UTP modeļi pēc analoģijas var tikt glabāti relāciju datu bāzes tabulās. [BES] aprakstīta priekšrocība par RDB elastīgu un ērtu lietošanu UML modeļiem ir pielietojama UTP modelim. Testēšanas dati bieži tiek sasaistīti ar vairākiem izstrādes rīkiem, nodrošinot trasējamību starp tiem. Piemēram, starp prasībām, uzdevumu un kļūdu pārvaldības rīkiem, pirmkodu, atskaitēm un citiem. Tas viss prasa attālināto pieeju pie testēšanas datiem, kas var tikt nodrošināta, glabājot tos datu bāzēs.

Līdzīgi UML modeļiem, UTP profilam atbilstošā struktūra tiek izveidota datu bāzē. Izveidotās tabulas ir spējīgas glabāt UML testēšanas profilam atbilstošus artefaktus, tādejādi nodrošinot testēšanas modeļu prezentāciju. Papildus minētajām tabulām, tiek sagatavotas datu bāzes rindas, kas nodrošina saglabājamo elementu identificēšanu un unikalitāti. Transformācijas gaitā UTP tabulas tiek aizpildītas ar ģenerētiem datiem, nodrošinot testēšanas artefaktus manuālo testu veikšanai.

3.3.3 Modeļu transformācijas realizācija

UML modeļu transformācija notiek ar transformācijas rīku, kas ir realizēts PL/SQL valodā un pieprasa Oracle DBVS pielietošanu. DBVS šī pētījuma ietvaros jānodrošina sekojošās iespējas:

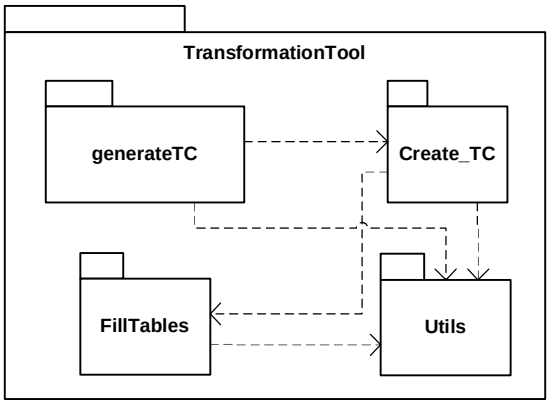
- tabulu veidošanu un SQL vaicājumu apstrādi;
- PL/SQL procedūru un funkciju uzturēšanu;
- skatu veidošanu uz regulārām tabulām;
- secību (angl. *sequences*) veidošana un pielietošana.

DBVS izvēles ierobežojošs kritērijs ir PL/SQL procedūru uzturēšana. Transformācijas rīka izstrādē tiek lietoti dinamiskie SQL vaicājumi (programmēšanas tehnika, kas ļauj konstruēt SQL vaicājumus izpildes laikā [DSQL]), ka arī papildus atbalsta procedūras un funkcijas. Tekošā transformācijas realizācija tiek nodrošināta ar Oracle DBVS, bet nepieciešamības gadījumā tā var tikt pārnesta uz kādu citu DBVS, kas nodrošina dinamiskos SQL vaicājumus, piemēram, MS SQL Server, MySQL vai citu. Transformācijas rīka izstrādei tika izvēlēta Oracle DBVS un PL/SQL programmēšanas valoda balstoties uz sekojošiem apgalvojumiem:

- komplekso, savā starpā saistītu datu apstrādei ir jālieto šiem mērķiem izstrādātas aplikācijas – DBVS;
- Oracle DBVS nodrošina nepieciešamo funkcionalitāti transformācijas pakotnes implementācijas;
- Oracle ir populārākā DBVS un Oracle korporācija ir līderis DBVS tirgū [GAR 2011].

Transformācijas likumi tiek glabāti atsevišķajā tabulā ar definēto notāciju. Transformācijas procedūra secīgi nolasa likumus un pielieto tos sākotnējām modelim, ģenerējot jaunus ierakstus UTP tabulās. Secības tiek pielietotas testēšanas artefaktu unikālai identificēšanai un sasaistei savā starpā.

Transformācijas rīka konceptuāls dalījums tehniskajās komponentēs ir prezentēts ar pakotņu diagrammu 3.13. attēlā.

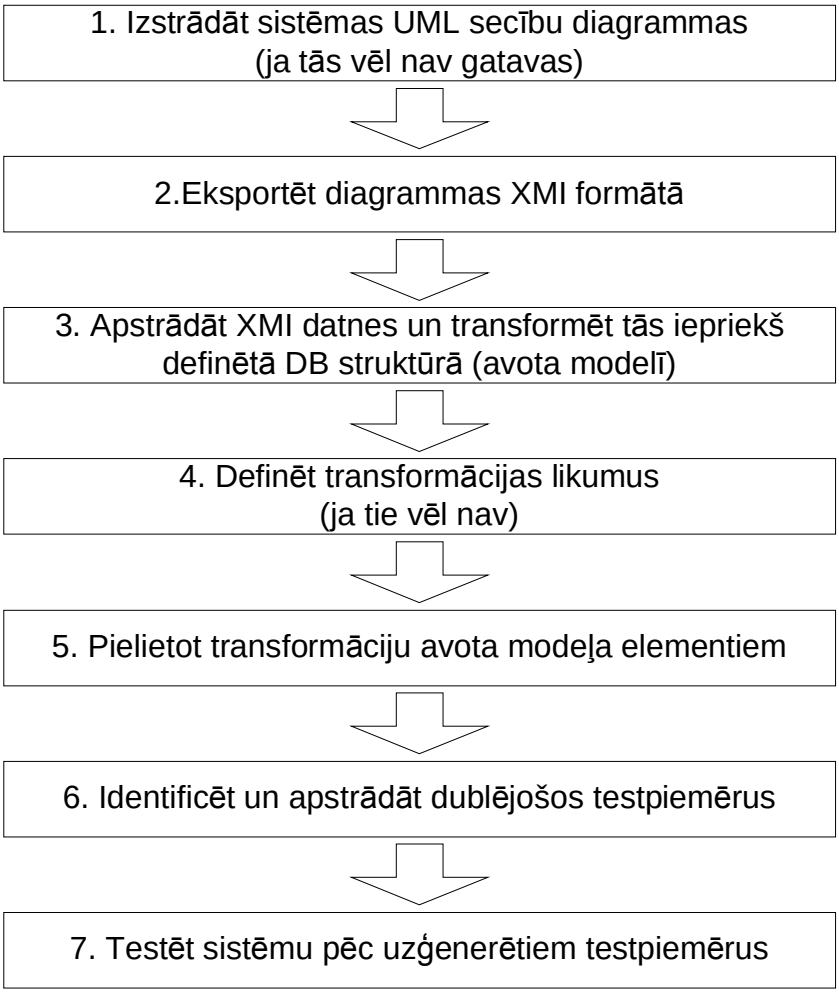


3.13. att. Transformācijas rīka UML pakotņu diagramma

Attēlā ir parādīts transformācijas rīka sadalījums četrās pakotnēs. Centrālā pakotne ir *generateTC*, kas nodrošina transformācijas ķēdes izpildīšanu un nepieciešamo papildus pakotņu izsaukšanu. Pakotne *Create_TC* nodrošina nepieciešamo tabulu aizpildīšanu konkrētām testpiemēram. Gadījumos, kad transformācijas likumu nosacījumi paredz to pielietošanu vairākiem sistēmas objektiem, *Create_TC* pakotnes izsaukšana notiek cikliski, nodrošinot vairāku testpiemēru ģenerēšanu transformācijas likuma ietvaros un visiem likumiem. Pakotne *FillTables* sastāv no vairākām funkcijām, kas nodrošina *TESTCONTEXT*, *TESTCASE*, *SUT*, *BEHAVIOR* un *TESTOBJECTIVE* tabulu aizpildīšanu ar atbilstošiem datiem. Visas trīs aprakstītas pakotnes pielieto atsevišķās funkcijas no *Utils* pakotnes, kur ir apkopotas apstrādei nepieciešamās funkcijas un procedūras. Tajā ir iekapsulētas operācijas, kas nodrošina: transformācijas likumu pārsēšanu, dinamisko SQL vaicājumu sagatavošanu, tabulu tīrīšanu un citas. Detalizēti apstrādes algoritms ir aprakstīts sadaļā 3.2.2.

3.4 Metodes pielietošanas soļu secība

Metodes pielietošana sastāv no vairākiem soļiem, daļa no kuriem var tikt izlaista atkarībā no projekta specifikas. 3.14. attēlā ir parādīta vispārīgā metodes pielietošanas soļu secība, kas definē iepriekš aprakstītas testēšanas metodes nepieciešamās aktivitātes.



3.14. att. Piedāvātās metodes soļu secība

Attēlā parādītā shēma prezentē nepieciešamos soļus, ieskaitot manuāli darbināmus un automatizētus. Ar automatizētiem soļiem tiek saprastas darbības, kuru rezultātu sagatavošanu nodrošina atbilstošs atbalsta rīks. Pie manuāliem soļiem var attiecināt 1., 4. un 7. darbības, savukārt visas pārējās aktivitātes tiek nodrošinātas ar standarta vai speciāliem, ka arī pētījuma ietvaros papildus izstrādātiem rīkiem.

Pirmais solis paredz UML diagrammu sagatavošanu rīkos, kas nodrošina tās eksportēšanu XMI formātā. Gadījumos, ja izstrādes procesā jau tiek pielietota UML modeļu izstrāde, tad šis posms var tikt izlaists vai arī eksistējošās diagrammas var tikt papildinātas ar testējamo īpašību artefaktiem.

Nākamajā solī sagatavotais modelis tiek eksportēts XMI formātā un šo darbību nodrošina modeļu izstrādes rīks. Ir pieejami vairāki atbalsta rīki, kas nodrošina šo funkcionālo iespēju (piemērām, Enterprise Architect, UModel un citi), tāpēc šis solis ir pilnībā automatizēts.

3. solī tiek pielietota šajā pētījuma ietvaros izstrādātā programma, kas nodrošina XMI formāta datnes apstrādi un tās satura ielasīšanu datu bāzē, kur dati tiek saglabāti datnei līdzīgajā struktūrā. Izstrādātā programma nodrošina SQL vaicājumu datnes veidošanu, kas nodrošina nepieciešamo tabulu izveidi un datu ielasīšanu tajās.

Specifisko transformācijas likumu izstrāde notiek 4. solī, kas var tikt izlaists gadījumā, ja tiek pielietoti vispārīgie transformācijas likumi. Vispārīgie transformācijas likumi ir veidoti pēc principa analizēt kopīgās sistēmas īpašības un ģenerēt likumiem atbilstošus testpiemērus. Par šādu var tikt uzskatīts likums, kas paredz noteiktā laika ierobežojuma (piemēram, galējā izpildes termiņa) pārsniegšanu par vienu vienību. Transformācijas likumiem tiek izstrādāti atbilstoši 3.2.1. sadaļā aprakstītai notācijai.

5. solī notiek iedarbināta transformācija, kas nodrošina ielādējamo modeļu analīzi atbilstoši definētiem transformācijas likumiem un automātisko testpiemēru ģenerēšanu. Transformācijas rezultātā tiek aizpildītas UTP profilam atbilstošās tabulās, tādējādi ģenerējot testpiemēriem nepieciešamus artefaktus, kas apraksta tos.

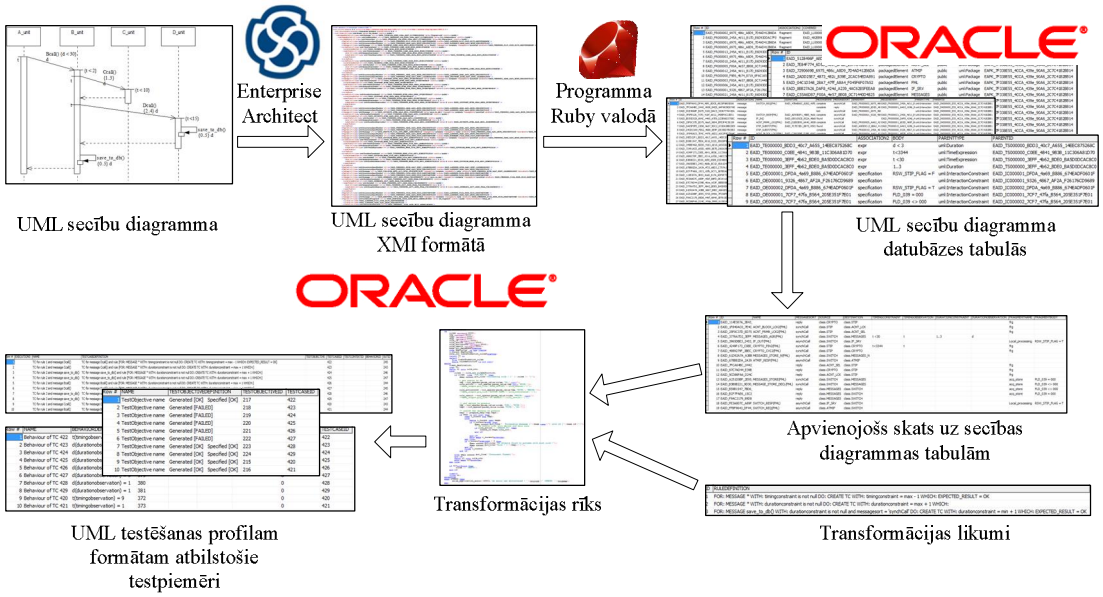
Transformācijas likumi var tikt definēti tā, ka divi transformācijas likumi var novest pie vienādu testpiemēru ģenerēšanas. Transformācijas rezultātā izveidotu testpiemēru pārbaude uz atkārtojamību tiek nodrošināta 6. solī, kur papildus izstrādātie skripti identificē vienādus testpiemērus. Piedāvātie skripti analīzē testējamos objektus, kas tiek pierakstīti UTP profilam atbilstošā formātā, un nosacījumus to darbināšanai un atkārtošanas gadījumā tie var tikt dzēsti, atstājot tikai unikālus gadījumus. Šāds solis ir nepieciešams, lai mazinātu nākamajā solī darbināmus testpiemērus.

7. solī notiek sistēmas manuāla testēšana pēc automātiski uzģenerētiem testpiemēriem. Sistēmas darbināšanas īpatnības ir specifiskas katrai sistēmai un ir ārpus šī pētījuma ietvariem.

Augstāk aprakstītie soļi apraksta vispārīgu darbību secību metodes pielietošanā. Metodes detalizēts pielietošanas apraksts ar tehniskam detaļām ir sniegts nākamajā sadaļā.

3.5 Metodes pielietošanas piemērs

Aprakstītas metodes demonstrācijai tiek piedāvāts tās pielietošanas piemērs testpiemēru ģenerēšanai abstraktai programmai ar sinhroniem izsaukumiem ar laika ierobežojumiem. Promocijas darbā piedāvātais risinājums paredz vairāku soļu realizāciju, kur katrā solī ir nepieciešams izmantot attiecīgās aktivitātes atbalsta rīku. Tādējādi izstrādātā algoritma pielietošanai ir definēta rīku ķēde (angl. *tool chain*), kuras vispārīga struktūra ir parādīta 3.15. attēlā.

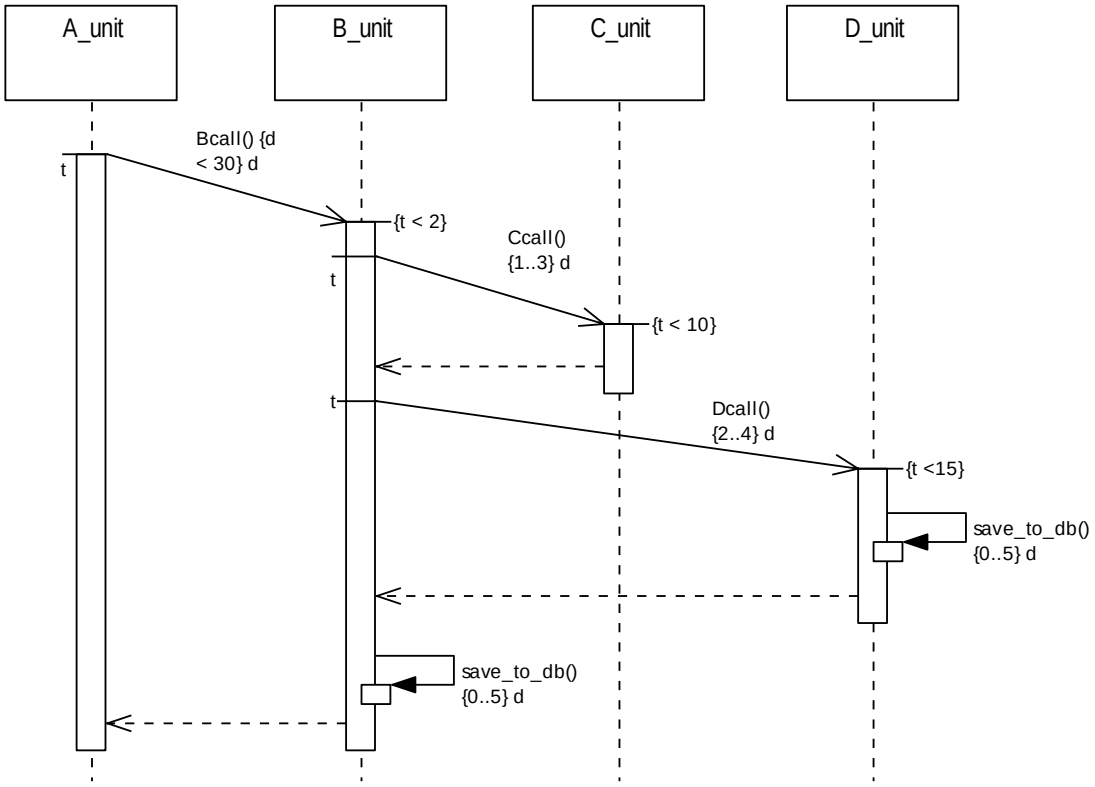


3.15. att. Piedāvātās testēšanas metodes rīku ķēde

Lai nodrošinātu sistēmas modeli transformācijai, metodes demonstrācija paredz programmas specificēšanu UML secību diagrammas veidā ar Enterprise Architect rīku. Šis komerciālais rīks nodrošina sistēmu modelēšanu, modeļu transformāciju uz kodu, testēšanas skriptu veidošanu un izstrādāto modeļu eksportēšanu XMI formātā. Pielietojot šo funkciju, rīkā izveidotā UML secību diagramma tiek eksportēta XMI formātā, nodrošinot sākotnējus datus aprakstītai testēšanas metodei. Atbilstoši 3.3.1 nodaļā aprakstītam XMI datņu apstrādes rīkam, tiek iegūta datne ar INSERT vaicājumiem datu ielādēšanai datubāzes tabulās. Pēc datu ielādes tabulās, tās satur pilnīgu sistēmas modeli, kas saturiski atbilst apstrādātai XMI datnei. Tālāk ielādētais modelis tiek reprezentēts skata veidā, attēlojot transformācijai nepieciešamus datus vienkāršotā un viegli apstrādājamā formātā. Transformācijas likumi tiek aprakstīti un saglabāti atbilstošajā *transformation_rules*

tabulā. Aktivizējot transformācijas rīku ar definētiem un ielādētiem datiem, tiek ģenerēts testēšanas modelis atbilstoši UML testēšanas profilam. Sistēmas modeļa apstrāde un testpiemēru ģenerēšana atbilst 3. nodaļā aprakstītiem principiem. Tālāk tekstā detalizēti tiek aprakstīts metodes pielietošanas piemērs ar definēto secību diagrammu, transformācijas likumiem un uzģenerētu testēšanas modeli.

Piedāvātās metodes demonstrācijai tiek izstrādāts sistēmas modelis UML secību diagrammas veidā, kas tiek parādīts 3.16. attēlā.



3.16. att. Secību diagramma programmai ar laika ierobežojumiem

3.16. attēlā ir parādīta programma, kurā ir implementēta ar 5 sinhroniem izsaukumiem ar laika ierobežojumiem. Aprakstītie laika ierobežojumi iekļauj laika un ilguma ierobežojumus. Eksportējot sistēmas modeli uz XMI formātu un translējot to uz dati bāzi, tiek aizpildīts *all_messages* skats. Skata dati ir aplūkojami 3.1. tabulā.

3.1. tabula

all_messages skata ieraksti ar būtiskākām kolonām

name	**sort**	**source**	**destination**	**tconstr**	**tobs**	**dconstr**	**dobs**
Bcall()	synchCall	class.A_unit	class.B_unit	t < 2	t	d < 30	d
Ccall()	synchCall	class.B_unit	class.C_unit	t < 10	t	1..3	d
save_to_db()	synchCall	class.B_unit	class.B_unit			0..5	d

name	sort	source	destination	tconstr	tobs	dconstr	dobs
Dcall()	synchCall	class.B_unit	class.D_unit	t <15	t	2..4	d
	reply	class.B_unit	class.A_unit				
	reply	class.C_unit	class.B_unit				
save_to_db()	synchCall	class.D_unit	class.D_unit			0..5	d
	reply	class.D_unit	class.E_unit				

Lai nodrošinātu testpiemērus laika ierobežojumu verificēšanai tabulā *transformation_rules* tiek definēti transformācija likumi, kuru mērķis ir ģenerēt testpiemērus veiksmīgai un neveiksmīgai sistēmas apstrādei. Tabulā 3.2. ir parādīti transformācijas likumi šo īpašību pārbaudei.

3.2. tabula

Transformation_rules tabulas saturs ar definētiem transformācijas likumiem

id	name	ruledefinition
1	Verify max boundary value for timecontraint	FOR: MESSAGE * WITH: timingconstraint is not null DO: CREATE TC WITH: timingconstraint = max - 1 WHICH: EXPECTED_RESULT = OK
2	Verify max boundary value for duration constraint	FOR: MESSAGE * WITH: durationconstraint is not null DO: CREATE TC WITH: durationconstraint = max + 1 WHICH:
3	Verify min boundary value for save_to_db	FOR: MESSAGE save_to_db() WITH: durationconstraint is not null and messagesort = 'synchCall' DO: CREATE TC WITH: durationconstraint = min + 1 WHICH: EXPECTED_RESULT = OK

Pirmais transformācijas likums paredz testpiemēru ģenerēšanu visiem ziņojumiem ar specificēto laika ierobežojumu ar nosacījumu, ka novērojamā laika mainīgā vērtība būs par vienu vienību mazāka par maksimāli pieļaujamo un sagaidāmais rezultāts šim pie šī nosacījuma ir veiksmīgs.

Otrais transformācijas likums ir veltīts ilguma ierobežojumam, bet atšķirībā no pirmā likuma, tas ir fokusēts uz ierobežojuma pārsniegšanu, turklāt manuāli netiek definēts sagaidāmais rezultāts.

Trešais transformācijas likums ir veltīts visiem sinhroniem *save_to_db* izsaukumiem ar ilguma ierobežojumiem, lai ģenerētu testpiemērus ar programmas uzvedību, kas pārsniedz šī ziņojuma definēto ilguma ierobežojumu par vienu vienību.

Transformācijas apstrādes rezultātā tiek iegūti 10 testpiemēri, kas tiek saglabāti mērķa modeļa tabulās. Lai nodrošinātu apkopotu testpiemēru prezentēšanu, tika izstrādāts *all_testcases* skats, kas iekapsulē būtiskāko ar testpiemēriem saistīto informāciju. Skats ir balstīts uz 3.9. attēlu, kas prezentē mērķa modeļa klašu

diagrammu ar pielietotiem atribūtiem. Skats ir ērts līdzeklis kopsavilkuma par ģenerētiem testpiemēriem prezentēšanai. Skatā attēlojamie dati ir pietiekami, lai iegūtu priekšstatu par uzģenerētiem testpiemēriem, to būtību un atbilstību transformācijas likumam. Balstoties uz skata datiem, ir iespējams iegūt nepieciešamo informāciju programmas darbināšanai un aprakstīt testpiemēru verificēšanai. Tabulā 3.3. ir prezentēti uzģenerētie testēšanas gadījumu *all_testcases* skata formātā.

3.3. tabula

All_testcases skata ieraksti ar uzģenerētiem testpiemēriem

tcid	**tcname**	**sut**	**behavior**	**objective**
390	TC for rule 1 and message Ccall()	Message Ccall() from class class.B_unit to class.C_unit	t(timingobservation) = 9	Generated [OK] Specified [OK]
391	TC for rule 1 and message Bcall()	Message Bcall() from class class.A_unit to class.B_unit	t(timingobservation) = 1	Generated [OK] Specified [OK]
392	TC for rule 1 and message Dcall()	Message Dcall() from class class.B_unit to class.D_unit	t(timingobservation) = 14	Generated [OK] Specified [OK]
393	TC for rule 2 and message Ccall()	Message Ccall() from class class.B_unit to class.C_unit	d(durationobservation) = 4	Generated [FAILED]
394	TC for rule 2 and message save_to_db()	Message save_to_db() from class class.B_unit to class.B_unit	d(durationobservation) = 6	Generated [FAILED]
395	TC for rule 2 and message save_to_db()	Message save_to_db() from class class.D_unit to class.D_unit	d(durationobservation) = 6	Generated [FAILED]
396	TC for rule 2 and message Bcall()	Message Bcall() from class class.A_unit to class.B_unit	d(durationobservation) = 31	Generated [FAILED]
397	TC for rule 2 and message Dcall()	Message Dcall() from class class.B_unit to class.D_unit	d(durationobservation) = 5	Generated [FAILED]
398	TC for rule 3 and message save_to_db()	Message save_to_db() from class class.B_unit to class.B_unit	d(durationobservation) = 1	Generated [OK] Specified [OK]
399	TC for rule 3 and message save_to_db()	Message save_to_db() from class class.D_unit to class.D_unit	d(durationobservation) = 1	Generated [OK] Specified [OK]

Uzģenerētie testpiemēri ļauj spriest par sistēmas modeļa un transformāciju likumu izpildīto analīzi un rezultātā automātiski iegūto sagaidāmo rezultātu, kas tiek prezentēts kopā ar manuāli definēto vērtību, gadījumos, ja tas tika definēts. 383.-397. testpiemēros parādās tikai automātiski uzģenerētais sagaidāmais rezultāts, jo otrajā

transformācijas likumā tas netika specificēts. Uzģenerētie testpiemēri var tikt lietoti manuālas programmas darbināšanai un reālā rezultāta salīdzināšanai ar definēto sagaidāmo rezultātu. Metodes pielietošana ļauj izvairīties no manuālas sistēmas specifikācijas analīzes un pie eksistējošās secību diagrammas uzreiz iegūt testpiemēru kopu laicīguma īpašību verificēšanai.

3.6 Nodaļas secinājumi

Modeļvadāmās arhitektūras principi paredz modeļu ģenerēšanu no eksistējošiem modeļiem. Testpiemēru formalizācijai OMG grupa piedāvāja UML testēšanas profilu, kas nodrošina testēšanas procesam nepieciešamo elementu prezentāciju noteiktajā formā. Tas, savukārt, nozīmē, ka modeļu transformācija var tikt pielietota testēšanas uzdevuma risināšanas kontekstā, lai ģenerētu testpiemērus. Analizējot iegulto sistēmu nefunkcionālās īpašības, tiek konstatēts, ka UML versija 2.x nodrošina to specificēšanu ar formāliem UML valodas modeļiem. Šie secinājumi ļāva piedāvāt iegulto sistēmu nefunkcionālo īpašību testēšanas metodi, kas paredz automātisko testpiemēru ģenerēšanu no sistēmas modeļiem.

Pateicoties objektu vadības grupas piedāvātajam XMI standartam, kas tika izstrādāts modeļu pārnesamības nodrošināšanai, darbā izstrādāta testēšanas metode ļauj veidot sistēmas modeļus dažādos modelēšanas rīkos un lietot to attēlošanu XMI formātā testpiemēru ģenerēšanai.

Lai nodrošinātu transformācijas likumu uztveramu un skaidru pierakstu, kā arī ar tiem aprakstīt dažādu testēšanas metožu būtību, likumiem jāfokusējas uz testēšanas procesa artefaktiem. Tajā pašā laikā automātizēta sistēmas modeļu analīzes vaicājumu realizācija pieprasa komplicētus pierakstus, kas rada problēmas transformācijas likumu definēšanai. Šo iemeslu dēļ šī darba autora piedāvātā metode paredz savu transformācijas likumu sintaksi un vienkāršotu sistēmas modeļu prezentāciju, kas tiek iegūta ar datu bāzes vadības sistēmu iespēju pielietošanu.

Balstoties uz nodaļā sniegto analīzi un iegūtiem rezultātiem darba autors secina, ka:

- Modeļu transformācijas principu izmantošana testēšanas uzdevuma risināšanā, ņemot par avota modeli attiecīgas sistēmas īpašības attēlojumu un uzdodot attiecīgās transformācijas, dod iespēju

automātiski ģenerēt testpiemēru komplektu modelētās nefunkcionālās īpašības verificēšanai.

- UML 2.0 versijas bagātināta apzīmējuma sistēma dot iespēju attēlot nefunkcionālās sistēmas īpašības, piemēram, laika ierobežojumus, kaut gan iegulto sistēmu nefunkcionālo īpašību attēlošanai notācijā eksistējošie elementi pagaidām vēl nav pilnībā implementēti modelēšanas rīkos.
- Objektu vadības grupa definē UML testēšanas profilu, kas ir lietojams testēšanas uzdevumu risināšanā, un, neskatoties uz to, ka tas ir vispārīgs, tas dot iespēju uzdot testēšanas piemēra pieraksta formu arī nefunkcionālo īpašību verificēšanai.
- QVT valodas pielietošanas testēšanas uzdevuma kontekstā ir apgrūtināta nepieciešamās vides atbalsta dēļ un vispārīgo pieeju transformāciju likumu definēšanai, nefokusējoties uz testēšanas uzdevumam nepieciešamo specifiku. Šī iemesla dēļ algoritma realizācijas transformāciju definēšanas posmā autors izmantoja savu transformācijas likumu pieraksta sintaksi.
- Modeļu transformācijas atbalsta rīku klāsts nodrošina vairāku specifisku modeļu apstrādes darbību realizāciju, trūkstošus modeļu apstrādes mehānismus ir iespējams rīkos implementēt, turklāt eksistējošie standarti un iespējas modeļu apmaiņai dod iespēju definēt attiecīgu rīku, kas ir izmantojams katrā autora definētajā algoritma solī. Tādējādi ir iespējams uzdot tā saucamo rīku ķēdi autora piedāvātā algoritma realizācijai, kas sastāv gan no jau eksistējošiem rīkiem, gan iekļauj autora izveidotus spraudņus (angl. *plug-in*).

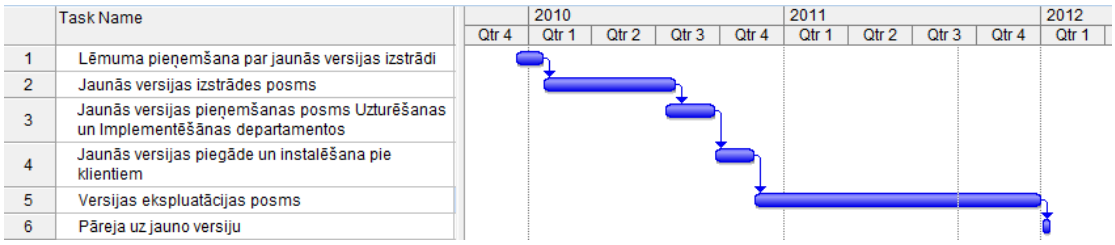
4 PIEDĀVĀTĀS TESTĒŠANAS METODES PIELIETOJUMS REĀLĀ LAIKA SISTĒMAS VERIFICĒŠANĀ

Šajā nodaļā autors veic piedāvātās metodes un izstrādāto rīku aprobāciju uz reālā laika maksājuma karšu sistēmas Card Suite laicīguma īpašības testēšanā. Laicīguma īpašības ir kopīgas iegultām un reāla laika sistēmām. Tieši šī iemesla dēļ par testējamo sistēmu tika izvēlēta Card Suite sistēma, kas satur ar iegultām sistēmām līdzīgus laika ierobežojumus un to apstrādes principus. Par testējamo objektu ir paņemts minētās sistēmas pamata datu plūsmas apstrādes scenārijs, kas iekļauj vairāku laika ierobežojumu veidu nosacījumus. Balstoties uz iegūtiem aprobācijas rezultātiem, autors analizē metodes priekšrocības un tās ierobežojumus.

4.1 Projekta apraksts

Piedāvātā metode tika pielietota reālā laika maksājuma karšu sistēmas Card Suite RTPS (angl. *Real-Time Processing System*) versijas 8.0 testēšanā uzņēmumā SIA Tieto Latvia. Produkta versijas izstrāde un pārbaude tika pabeigta 2010 gada 4. ceturksnī un pašlaik tā tiek ekspluatēta pie vairākiem esošiem klientiem. Šīs versijas galvenie uzlabojumi ir vērsti uz sistēmas drošuma īpašību uzlabošanu un atbilstību maksājuma aplikāciju datu drošības standarta prasībām (angl. *Payment Application Data Security Standard, PA DSS*) [PADSS].

Card Suite produktu dzīves cikli sastāv no vairākiem posmiem, ieskaitot pašas versijas izstrādi un sagatavošanu, ieviešanu pie klienta un citiem. 4.1. attēlā ir parādīts Card Suite RTPS 8.0 versijas plānotais dzīves cikls.

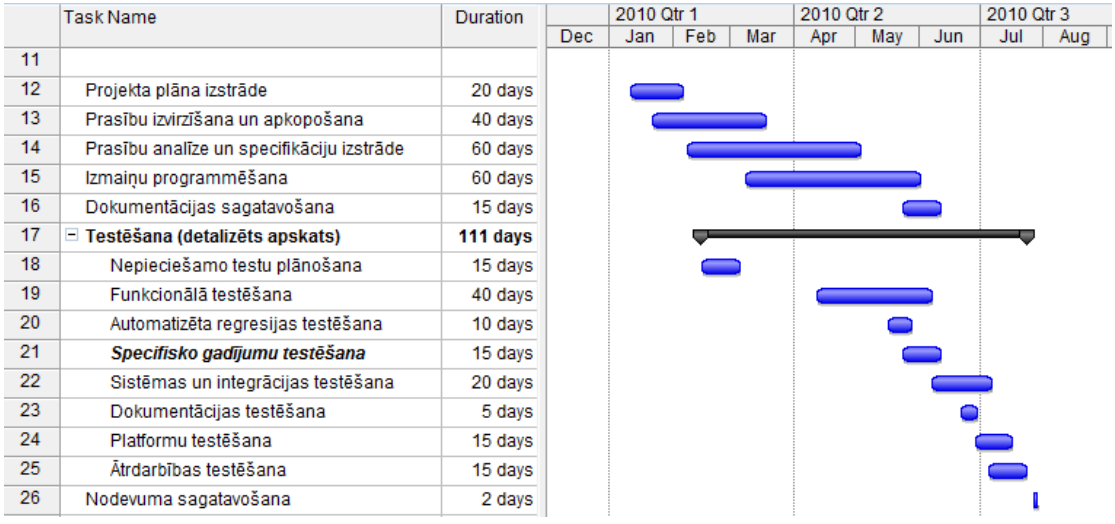


4.1. att. Card Suite RTPS 8.0 versijas dzīves cikla pārskats

Augstāk aprakstītais RTPS versijas dzīves cikls atbilst vispārīgam Card Suite produktu attīstīšanas modelim un prezentē būtiskākos posmus, kas ir nepieciešami

ilgstošai un kvalitatīvai Card Suite produktu saimes nodrošināšanai klientu vajadzībām. Programmatūras izstrāde notiek 2. posmā, kur tiek vāktas un analizētas jaunās prasības, tiek izstrādātas jaunas un uzlabotas esošās komponentes, kā arī šajā posmā notiek detalizētas programmatūras testēšanas, ieskaitot funkcionālus, sistēmas, integrācijas, platformu, ātrdarbības un citus testus. 2. posma rezultātā tiek iegūta funkcionējošā produkta versija, kas ir pārbaudīta ar standartu konfigurācijas komplektu. Tā kā Card Suite produkti pašlaik tiek darbināti pie vairāk nekā 40 klientiem no 20 valstīm, tad arī to izvietošanas vide un apstrādājamo datu klāsts atšķīrās starp instalācijām. Lai minimizētu risku jaunās versijas instalācijai pie šiem klientiem, uzņēmumā ir ieviests 3. posms, kur notiek izstrādātās versijas testēšana uz lielāku un svarīgāku klientu vidēm. 4. posmā notiek jaunās versijas piegāde klientiem, tās instalācija uz klientu testa vidēm un pieņemšanas testēšana no klienta puses. Pēc veiksmīgi izietiem pieņemšanas testiem, programmatūra tiek instalēta reālās ekspluatācijas vidē, kur tiek uzturēta ar piegādājamām nelielām izmaiņām un kļūdu labojumiem līdz nākamās versijas pieejamībai.

Darbā piedāvātā metode tika iekļauta produkta izstrādes posmā, ar kuras palīdzību un citām plānotām testēšanas aktivitātēm tika iegūta augsta RTPS versijas kvalitāte (sk. pielikumu 1.). 4.2. attēlā detalizētāk ir apskatīts RTPS 8.0 versijas izstrādes posms ar tajā iekļautiem uzdevumiem.



4.2. att. RTPS 8.0 versijas izstrādes posma plāns ar testēšanas procesa detalizēšanu

Produkta izstrādes procesu nevar tieši attiecināt pie kādas vienas izstrādes metodoloģijas: ūdenskrituma (angl. *waterfall*), veiklām metodēm (angl. *agile methods*) vai racionāli unificēta procesa (angl. *Rational Unified Process* vai *RUP*).

Tomēr pēc savas būtības un koncepcijas tas ir līdzīgs ūdenskrituma modelim un paredz pamata izstrādes procesu (prasību sagatavošana, analīze, programmēšana un testēšana) secīgu plānošanu un izpildi ar to pārklāšanu savā starpā.

4.2. attēlā prezentētais plāns parāda būtiskākus posmus produkta izstrādes fāzē un apzināti neapskata nelielas aktivitātes, kas arī ir neatņemama izstrādes projekta sastāvdaļa. Par šādām neuzskatītām aktivitātēm tiek uzskatītas kļūdu labošana visa projekta garumā, regulārās sapulces par projekta gaitu un statusu atsekošanu, caurskates sapulces un citas. Prezentētais plāns apraksta sekojošas testēšanas aktivitātes:

- Testu plānošana – tiek analizētas izvirzītās prasības un tiek sagatavots testēšanas plāna dokuments, kas apraksta nepieciešamos testēšanas veidus, testējamo funkcionalitāti un konfigurāciju, testēšanas ierobežojumus, nepieciešamos rīkus un aparatūru, nosaka testu pabeigšanas kritērijus un citus testēšanas procesam būtiskus aspektus.
- Funkcionālā testēšana – aktivitāte nodrošina pieprasīto izmaiņu verificēšanu atbilstoši izstrādātajai specifikācijai un prasību dokumentam.
- Automatizētā regresijas testēšana – balstoties uz iepriekš sagatavotiem automatizētās testēšanas skriptiem un rīkiem, tiek nodrošināta piegādājamā produkta regresijas testēšana.
- Specifisko gadījumu testēšana – aktivitāte nodrošina nefunkcionālo īpašību verificēšanu, ieskaitot laicīguma, sinhronizācijas, asinhronās darbības un citus testus. Šajā posmā notika arī grafiskās lietotnes lietojamības testi.
- Sistēmas un integrācijas testēšana – RTPS sistēma tiek integrēta ar citiem Card Suite produktiem un tiek verificēta kompleksa sistēmas funkcionēšana ar noteiktām biznesa funkcijām.
- Dokumentācijas testēšana – tiek testētas sagatavotas un uzlabotas lietotāju instrukcijas, kā arī cita ar RTPS 8.0 produkta versiju piegādājama dokumentācija.
- Platformu testēšana – ar instalācijas un dūmu testiem tiek verificēta produkta darbība uz dažādām izvietošanas platformām, piemēram, dažādas Unix platformas un to versijas, Oracle datu bāzes vadības

sistēmas versijas, operatora operētājsistēmas un citas trešo pušu programmatūras.

- Ātrdarbības testēšana – izstrādājamā programmatūra tiek darbināta pie lielām slodzēm, tiek iegūti darbināšanas parametri (piemēram, transakciju apstrādes laiki, aparatūras resursu patērēšanas apjomi uz vienu transakciju, atmiņas noplūdes atsekošana un citi) un tie tiek salīdzināti ar iepriekšējās versijas datiem, tādejādi pārbaudot sistēmas ātrdarbības parametrus.

Darbā piedāvātā metode tika pielietota specifisku gadījumu testēšanas posmā, kur notika sistēmas nefunkcionālo īpašību testēšana. Šāda veida testi tiek iekļauti projektā, balstoties uz veiktām izmaiņām un tā specifiku. Jaunā RTPS produkta versija ir plānota ilgstošai ekspluatēšanai pie vairākiem Card Suite sistēmas klientiem. Šī aspekta dēļ papildus klasiskiem un neatņemamiem testiem versijas izstrādes posmā tika ieplānoti dažādi nefunkcionālie testi. Detalizēti laicīguma testi tika iekļauti projektā, lai nodrošinātu šīs kritiskās nefunkcionālās īpašības verificēšanu pēc veiktām programmatūras izmaiņām un pāreju uz iegulto Oracle Tuxedo [TUX] transakciju monitoru, kas iepriekš tika piegādāts kā neatkarīga blakus instalējama trešo pušu programmatūra. Minētās izmaiņas skāra iekšējās komponenšu saskarnes un kopīgu sistēmas kodolu un prasīja detalizēto verificēšanu, kas tika nodrošināta ar šajā darbā aprakstītu metodi (sk. pielikumu 1.). Papildus laicīguma testiem, projektā tika iekļauti drošuma un lietojamības testi, kas neskar dotā pētījuma piedāvāto metodi un sīkāk darbā netiek apskatīti.

Šī darba autors bija aktīvi iesaistīts aprakstītajā projektā un bija atbildīgs par vairākām testēšanas aktivitātēm. Darba autors izstrādāja testēšanas plānu visām RTPS 8.0 projekta testēšanas aktivitātēm un veica visu testēšanas darbu, t.sk. funkcionālu, sistēmas, integrācijas, platformu un ātrdarbības testu uzraudzību un atsekošanu. Tajā pašā laikā nodrošināja arī specifisko gadījumu verificēšanu, izņemot lietojamības testus. Projekta ietvaros notika reālā laika sistēmas laicīguma īpašību testēšana, balstoties uz šī darba 3. nodaļā aprakstītu metodi. Tā kā modeļvadāmās programmatūras izstrādes process tikai daļēji ir ieviests uzņēmumā un ar modeļiem tiek specificēta neliela daļa no sistēmas funkcionēšanas, laicīguma īpašību testēšanas posmā darba autors izstrādāja UML secību diagrammas sistēmas komponentēm Sparx Enterprise Architect lietojuma programmā un turpmāk tās pielietoja testēšanas procesā. Izstrādātās UML secību diagrammas specificē laicīguma ierobežojumus.

Gadījumā, ja sistēmai būtu pieejamas šādas diagrammas, papildus diagrammu izstrāde nebūtu nepieciešama un tiktu lietotas eksistējošās UML secību diagrammas ar laicīguma ierobežojumiem. Pēc modeļu sagatavošanas notika šajā darbā piedāvātas metodes pielietošana, kuras rezultātā tika uzģenerēti testpiemēri. Ar uzģenerētiem testpiemēriem darba autors veica RTPS sistēmas laicīguma ierobežojumu testēšanu. To darbināšanai bija nepieciešams simulēt kļūdainas situācijas ar sistēmas komponenšu izslēgšanu, Oracle tabulu bloķēšanu un komunikāciju pārtraukšanu. Darba autors pilnībā veica šos testus atbilstoši automātiski izveidotiem testpiemēriem. To darbināšanas rezultātā detalizēti verificēti sistēmas laicīguma ierobežojumi un atrasti nepareizas konfigurācijas parametri, kas tika uzstādīti sākotnējās instalācijas laikā, tādējādi varēja tikt piegādāti kopā ar maksājuma karšu sistēmu. RTPS sistēmas un Tuxedo transakciju monitora programmatūrā kļūdas netika atrastas.

RTPS 8.0 versijas izstrādes projekts ir kārtējās reālā laika sistēmas versija, kuras atbilstību PA DSS standartam verificēja ārējā auditoru kompānija, veicot drošuma testus. Audita rezultātā tika iegūts sertifikāts, apliecinot auditējamās sistēmas un izstrādes procesa atbilstību minētām standartam.

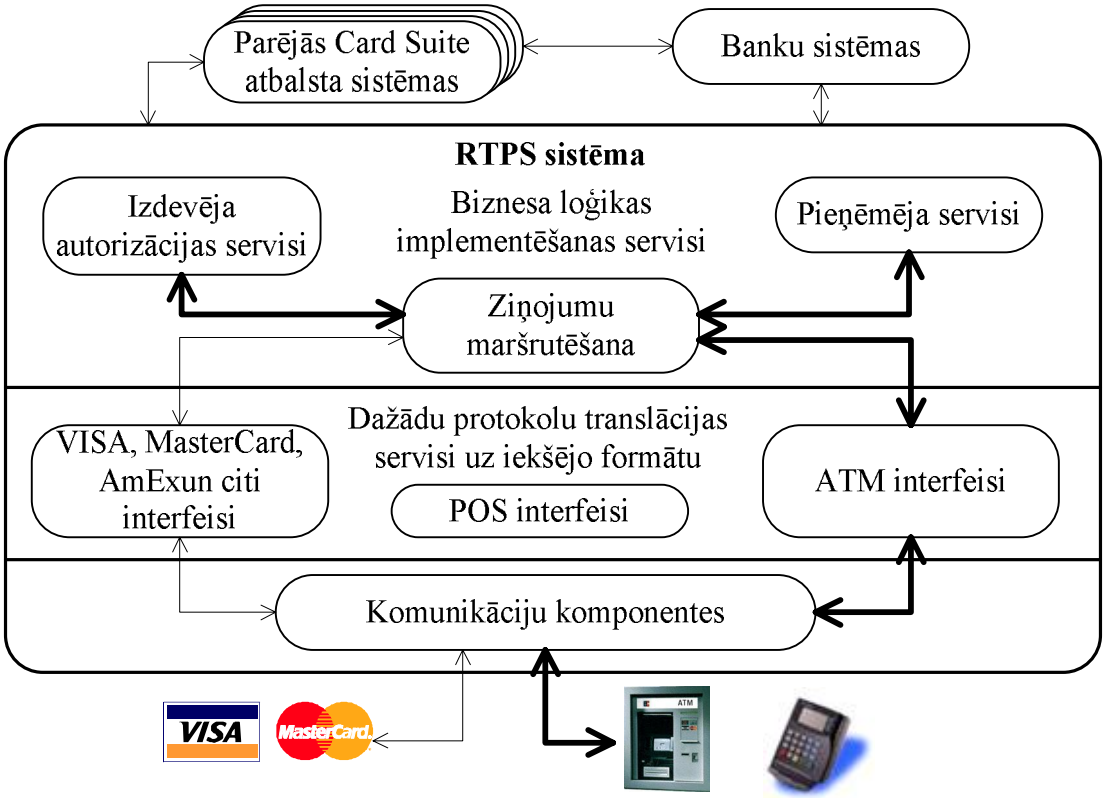
Projektā rezultātā tika izstrādāta RTPS produkta versija, kas veiksmīgi tika ieviesta pie 8 ārzemju klientiem (dati uz 11.09.2011), ar to apliecinot augstu produkta kvalitāti sasniegtu tā izstrādes gaitā (sk. pielikumu 1.).

4.2 Testējamās sistēmas apraksts

Metodes aprobācija laika ierobežojumu testēšanai tiek veikta uz reālā laika maksājuma karšu sistēmas Card Suite. Card Suite ir produktu kopa, kas nodrošina maksājuma karšu izdošanu (angl. *issuing*), pieņemšanu (angl. *acquiring*) ar visām populārākām metodēm (piemēram, tīmeklī, bankomātos (angl. *automated teller machine* – ATM), pārdošanas vietu termināļos (angl. *point of sale* – POS) un citās ierīcēs), autorizāciju apstrādi reālā laikā, kā arī vairākas citas atbalsta operācijas, kas ir saistītas ar maksājuma karšu apstrādi (piemēram, krāpšanas atsekošanas sistēma, izņēmuma gadījumu apstrādes sistēma, norēķinu un citas sistēmas). Metode tiek pielietota RTPS (angl. *real time processing system*) sistēmai, kas nodrošina dažāda veida operāciju apstrādi reālā laikā.

4.3. attēlā ir parādīta RTPS sistēmas atrašanās vieta maksājuma karšu operāciju apstrādes ciklā. Attēlā prezentētā diagramma parāda RTPS sistēmas

piederību finanšu operāciju apstrādei ar maksājuma kartēm. RTPS atbild tikai par reāla laika operācijām un turpmākā finansu operācijas apstrāde (finanšu transakcijas norakstīšana un nodošana citām sistēmām) notiek ārpus tās. Ar bultām ir paradītas pamata ziņojumu plūsmas starp ziņojumu avotiem, būtiskākām sistēmas komponenšu grupām un citām ārējām sistēmām, kas var notikt vienas finanšu operācijas ietvaros.



4.3. att. RTPS sistēmas atrašanas vieta maksājuma karšu operāciju ciklā

Diagrammā ar biezām bultām ir parādīts autorizācijas pieprasījuma[2] apstrādes cikls, kura verificēšanai tiks izmantota darbā aprakstītā metode. Apstrādes scenārijs sākas ar autorizācijas pieprasījuma iniciēšanu bankomātā un caur vienu no uzturamiem bankomātu protokoliem caur komunikāciju komponenti ziņojums tiek nogādāts līdz ATM interfeisam. ATM komponente veic saņemtā ziņojuma pārsēšanu un sagatavo iekšējo ziņojumu turpmākai apstrādei, turklāt šajā brīdī tiek uzsākta globāla transakcija, kas rūpēsies par veicamo operāciju veselumu. Tālāk ziņojums tiek

[2] Lai izvairīties no pārpratumiem terminos, darba 4. nodaļā par autorizācijas pieprasījumu jeb autorizāciju tiek saukts finansu ziņojums, kura mērķis ir pierezervēt konkrēto summu uz konta, bet par autorizācijas procesu – autorizācijas pieprasījuma verifikācijas process ar mērķi pārbaudīt izmantojamās kartes un kartes turētāja identitāti, kā arī iespēju pierezervēt pieprasītu summu no kartes turētāja līdzekļiem. Savukārt, par transakciju tiek saukta operāciju kopa, kura izpildās kā viens veselums un izjūk, ja kaut viena operācija ir neveiksmīga.

nogādāts uz maršrutēšanas komponenti, kas veic ziņojuma validāciju, izsauc pieņēmēja servisus un atbilstoši ziņojuma izrēķinātam galamērķim nogādā to nepieciešamajam servisam. Gadījumos, kad maksājuma karte ir lokālā karte un RTPS sistēmā ir visi nepieciešamie dati par šo karti, ziņojuma autorizācijas procesu veic atbilstošie izdevēja autorizācijas servisi, kas atrodas RTPS sastāvā. Autorizācijas servisi veic scenārijam definētas pārbaudes (pārbauda derīguma termiņu, ievadītu PIN (angl. *personai identification number*) kodu, definētus limitus, piemeklē pieejamo summu un citas) un veiksmīgas pārbaudes gadījumā finansu autorizācijas pieprasījumam veic summas bloķēšanu. Ziņojuma atbilde tiek nogādāta autorizācijas pieprasījuma iniciēšanas ierīcei caur tām pašām komponentēm, kas piedalījās autorizācijas pieprasījuma apstrādē.

RTPS sistēma pieder pie reālā laika sistēmām ar „cietiem" laika ierobežojumiem, jo to pārsniegšana nozīmē transakcijas atteikšanu. Sistēmā ir realizēti vairāku veidu laika ierobežojumi. Tā, piemērām, katra operācija tiek apstrādātā ar vienu vai vairākām globālām transakcijām, kurām tiek noteikts maksimāli pieļaujamais laika ierobežojums. Laika ierobežojumi tiek definēti konkrēto servisu izsaukšanai, pieprasījumu pārsūtīšanai uz ārējo sistēmu un citām darbībām. Detalizētāk aprakstītas operācijas apstrādes cikls ir aprakstīts nākamajā sadaļā.

4.3 Reālā laika autorizācijas apstrādes scenārijs

Autorizācijas apstrādē, atkarībā no sistēmas konfigurācijas un ziņojuma datiem, parasti piedalās no 10 līdz 20 un vairāk atsevišķi servisi. RTPS sistēmā iekšējā struktūra atbilst komponēšu bāzētai arhitektūrai, turklāt visas infrastruktūras pārvaldībai un veseluma nodrošināšanai tiek lietots Oracle Tuxedo transakciju monitors [TUX]. Tas nodrošina izstrādātās programmatūras sadalīšanu servisos, elastīgi konfigurējamo servisu eksemplāru vadību, efektīvu apmaiņas protokolu starp tiem, transakciju definēšanu un vadību apstrādes operācijām un vairākas citas tehnoloģiskās operācijas, kas ir nepieciešamas kā atbalsta funkcijas maksājuma karšu un cita veida sistēmām.

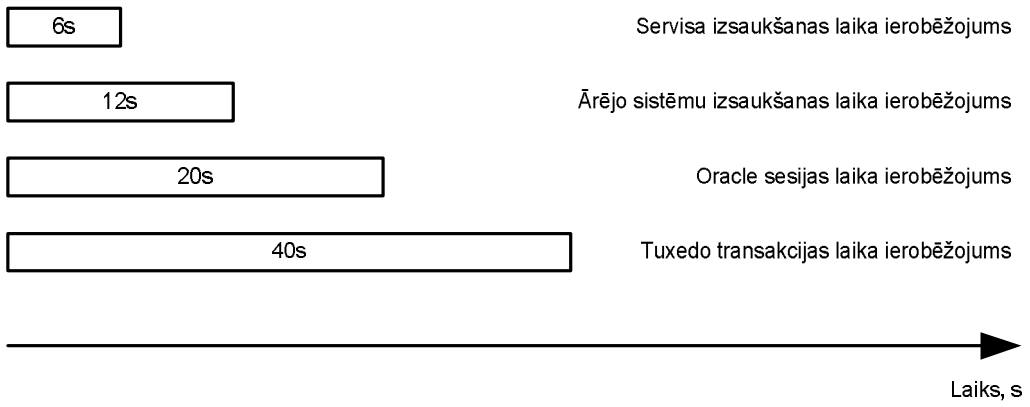
Sistēmas darbībā tiek lietoti četru veidu laika ierobežojumi:

- Globālās Tuxedo transakcijas laika ierobežojums – maksimāli pieļaujamais laiks kopš transakcijas iniciēšanas kādā no servisiem līdz visu operāciju pabeigšanas, kas notiek pirms atbildes ziņojuma

pārsūtīšanas kādam citam servisam. Transakciju var uzsākt viens serviss un pabeigt to var cits. Globālās transakcijas iniciēšanas servisi tiek definēti sistēmas konfigurācijā. Vienas finansu operācijas apstrādē var tikt iniciētas vairākas globālās Tuxedo transakcijas.

- Oracle sesijas laika ierobežojums – maksimāli pieļaujamais laiks, kas tiek izdalīts uz visām sesijas ietvaros iekļautām operācijām, ieskaitot visu izmaiņu saglabāšanu. Oracle sesija sākās ar globālās transakcijas iniciēšanu un beidzās ar *commit* operāciju.
- Ilguma ierobežojums uz Tuxedo servisa izsaukšanu – laika intervāls starp izsaukuma iniciēšanu un vadības nodošanu saucamajam servisam.
- Laika ierobežojums uz ārējo sistēmu izsaukumiem – maksimāli pieļaujamais laika intervāls starp pieprasījuma aizsūtīšanu uz ārējo sistēmu un atbildes saņemšanu no tās.

Lai nodrošinātu korektu sistēmas darbību, laika ierobežojumiem ir jābūt konfigurētiem ar atbilstošām vērtībām. Tas ir nepieciešams, lai izvairītos no gadījumiem, kad turpinās ziņojuma apstrāde, bet globālā transakcija jau tiek atcelta. 4.4. attēlā ir parādīts minēto laika ierobežojumu konfigurācijas piemērs, kas atbilst korektai savstarpējai ierobežojumu proporcijai.



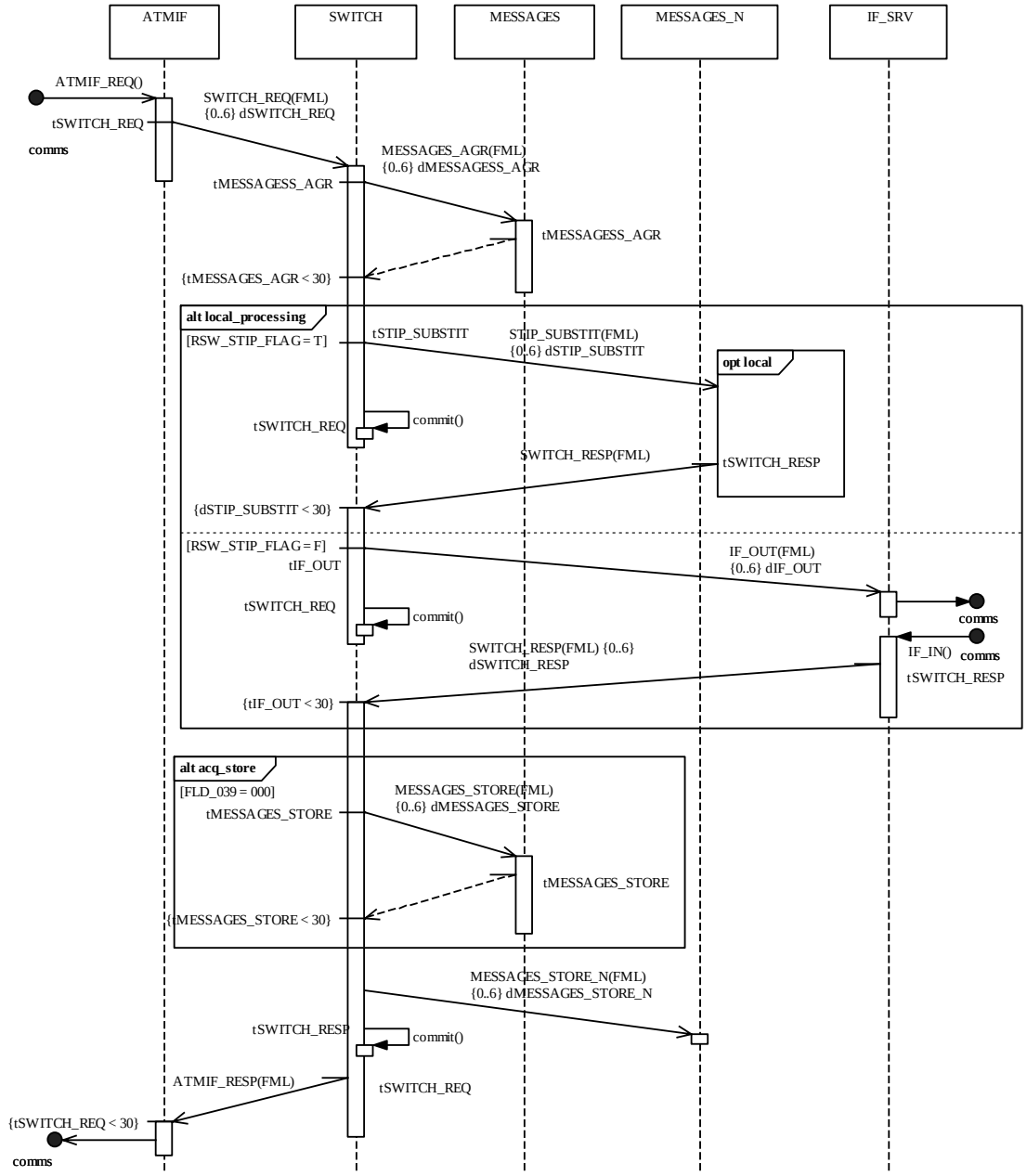
4.4. att. Laika ierobežojumu attiecības RTPS sistēmas konfigurācijā

4.4. attēlā prezentētie laika ierobežojumi tiek uzskatīti par piemēru laika ierobežojuma vērtībām, kas tiek lietotas kā vērtības pēc noklusēšanas. Atkarībā no pilnās sistēmas konfigurācijas, apstrādājamiem ziņojumu apjomiem, kā arī no aparatūras līdzekļu vides, aprakstīto ierobežojumu vērtības var atšķirties no attēlā minētām.

Iepriekšējā sadaļā konceptuāli tika aprakstīts autorizācijas apstrādes cikls sākot ar iniciēšanu līdz atbildes saņemšanai uz gala ierīces. Zemāk detalizēti tiek apskatīts tās pašas autorizācijas apstrādes cikls RTPS sistēmā. 4.5. un 4.6. attēlos tiek parādīts bankomātā iniciētās finansu autorizācijas apstrādes scenārijs būtiskajos sistēmas servisos (angl. *services*). Autorizācijas apstrādes aprakstā tiek lietots servisa jēdziens, kas ir paņemts no Tuxedo definīcijām. Ar to tiek saprasta autonomā operāciju kopa, kas tiek apvienota, balstoties uz noteiktiem loģiskiem principiem, nodrošinot servisam konkrēto biznesa vai funkcionālo mērķi. Ziņojumu apmaiņa sistēmā notiek starp servisiem, kas tiek nodrošināti ar Tuxedo līdzekļiem. Ziņojumu apmaiņa var tikt nodrošināta ar sinhroniem un asinhroniem izsaukumiem. Asinhrono izsaukumu gadījumos, izsaucamais serviss negaida atbildi no saucamā servisa un ir gatavs apstrādāt kārtējo ziņojumu. Savukārt, sinhronā izsaukuma gadījumā saucējserviss gaida atbildi no izsaucamā servisa. Attēlos sinhronie izsaukumi tiek paradīti ar biezām/aizpildītām bultām, bet asinhronie ar šaurām bultām.

Vairāki pēc savas būtības saistīti servisi var tikt un parasti tiek apvienoti aplikācijas serveros. Tā, piemērām, 4.5. attēlā ir parādīti 5 aplikāciju serveri: *ATMIF*, *SWITCH*, *MESSAGES*, *MESSAGES_N* un *IF_SRV*. Turpat tiek attēlots *MESSAGES* serveris, kurš nodrošina divus servisus – *MESSAGES_AGR* un *MESSAGES_STORE*. Abu servisu izsaukšana notiek ar atsevišķiem ziņojumiem dažādos laika momentos un pie dažādiem nosacījumiem.

Lai nodrošinātu secību diagrammas uzskatāmību un neveidotu to pārāk sarežģītu, diagrammā tiek izdalīts atsevišķs apstrādes fragments *local*, kas detalizētāk tiek aprakstīts 4.6. attēlā. Pārējie diagrammā attēlotie fragmenti ar tipu *alt*, *local_processing* un *acq_store*, apraksta atšķirīgus apstrādes scenārijus atkarībā no apstrādājamā ziņojuma lauku vērtībām.



4.5. att. Finansu autorizāciju apstrādes scenārijs RTPS sistēmā

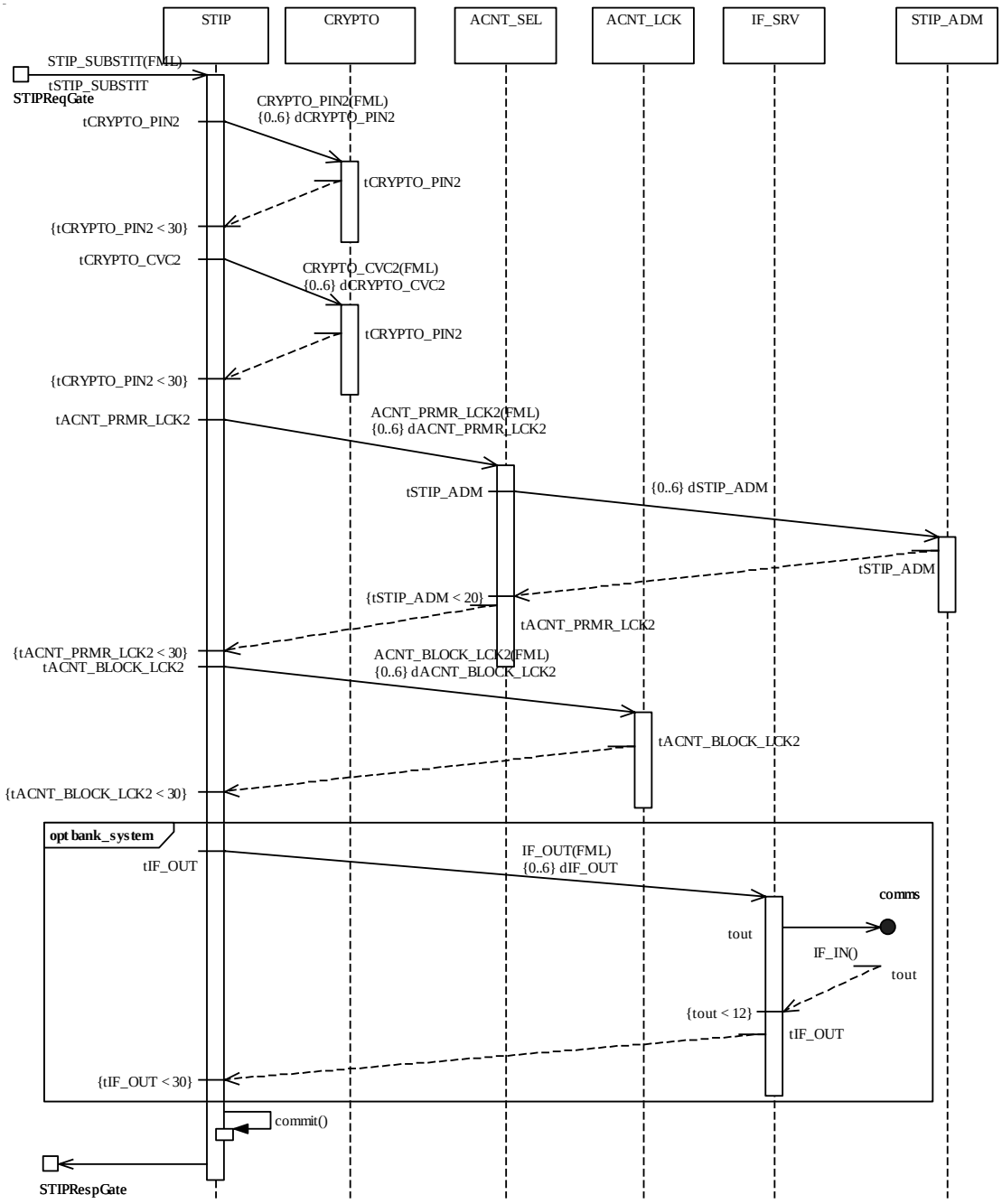
4.5. attēlā parādītā diagramma apraksta finansu autorizācijas apstrādi RTPS sistēmas servisos sākot ar sistēmas komponentēm, kas atbild par komunikāciju nodrošināšanu un datu pārsūtīšanu uz ārējiem avotiem – starptautiskiem tīkliem un dažāda veida ierīcēm. Tā kā komunikāciju slānis ir izstrādāts citās, no Tuxedo servisu atšķirīgās tehnoloģijās, tad tas tiek attēlots diagrammās ar gala punktu (angl. *endpoint*) palīdzību. 4.5. attēlā ir parādīti četri tādi galapunkti – aprakstot komunikāciju slāni uz bankomātu iekārtām un uz ārējiem tīkliem. Minētais komunikāciju slānis paredz servisu izsaukšanu un datu saņemšanu no tiem.

Komunikāciju komponente asinhroni izsauc *ATMIF_REQ* servisu, nodrošinot to ar apstrādājamā ziņojuma datiem un pārliekot autorizācijas apstrādes vadību tam. Tālāk *ATMIF_REQ* serviss asinhroni izsauc *SWITCH_REQ* servisu ar ilguma ierobežojumu *0..6* sekundes. Servisa izsaukšanas brīdī tiek definēts laika novērošanas mainīgais *tSWITCH_REQ*, kas tiek lietots turpmākajos laika ierobežojumos. Apstrādes gaitā *SWITCH_REQ* izsauc *MESSAGES_AGR* servisu ar līdzīgu ilguma ierobežojumu uz servisa izsaukšanu, analoģiski definējot laika novērošanas mainīgo *tMESSAGES_AGR*, kas tiek lietots kā laika ierobežojums atbildes saņemšanai – *tMESSAGES_AGR < 30*. Tālāk tiek definēts alternatīvās apstrādes fragments, kas apraksta ziņojuma apstrādes scenārijus atkarībā no noteikta ziņojuma lauka vērtības. Gadījumā, ja ziņojums ir jāapstrādā lokālajos servisos (*RSW_STIP_FLAG = T*), tad asinhroni tiek izsaukts *STIP_SUBSTIT*, nododot šim servisam turpmāko autorizācijas apstrādes vadību, un tiek saglabātas visas izmaiņas tekošajā Oracle sesijā. Izmaiņu pieglabāšana notiek ar *commit* operāciju, kurai tiek definēts laika ierobežojums *tSWITCH_REQ < 20* (Enterprise Architect modelēšanas rīks neattēlo šo ierobežojumu izsaukumiem, kuriem saucējs un saucamais objekts ir viens un tas pats). Alternatīvā gadījumā, kad autorizācijas ziņojums ir jānosūta uz ārējo sistēmu apstrādei (*RSW_STIP_FLAG = F*), ar ilguma ierobežojumu asinhroni tiek izsaukts *IF_OUT* serviss, kas tālāk nokonvertē ziņojumu uz ārējo formātu un pārsūta tālāk. Atbildes ziņojums tiek apstrādāts *IF_IN* servisā un tiek nodots tālāk *SWITCH_RESP* servisam ar ilguma ierobežojumu uz servisa izsaukumu. *SWITCH_RESP* servisa apstrādes scenārijs paredz alternatīvu fragmentu *acq_store*, kas tiek aktivizēts, ja autorizācijas ziņojums tiek apstiprināts (*FLD_039 = 000*). Šajā fragmentā sinhroni tiek izsaukts *MESSAGES_STORE* serviss ar ilguma ierobežojumu uz servisa izsaukšanu (*0..6*) un laika ierobežojumu uz atbildes saņemšanu no servisa (*tMESSAGES_STORE < 30*). Apstrādes scenārija noslēgumā *SWITCH_RESP* serviss asinhroni ar ilguma ierobežojumu uz servisa izsaukumu izsauc *MESSAGES_STORE_N* servisu, lai pieglabātu apstrādājamo autorizācijas ziņojumu vēstures tabulās. Atšķirībā no iepriekšējiem servisu izsaukumiem, asinhronais izsaukums speciāli tiek pielietots šeit, lai saīsinātu kopējo autorizācijas apstrādes laiku, jo tās pieglabāšana vēstures tabulās nevar ietekmēt kopējo autorizācijas rezultātu. Kopējais apstrādes scenārijs beidzās ar ziņojuma atgriešanu *ATMIF* serverim, asinhroni izsaucot *ATMIF_RESP* servisu ar ilguma ierobežojumu uz servisa izsaukšanu un laika ierobežojumu uz kopējo *SWITCH*

servera apstrādes laiku (*tSWITCH_REQ < 40*). Ziņojuma nodošana uz komunikāciju slāni notiek ārpus transakcijām un bez laika ierobežojumiem.

4.6. attēlā tiek parādīts fragmenta *local* apstrādes scenārijs, kas prezentē pamata serveru mijiedarbību, kuru mērķis ir nodrošināt kartes izdevēja pārbaudes kartes un tā lietotāja identificēšanai, pieejamā atlikuma pietiekamībai un autorizācijas summas bloķēšanu, kā arī atbildes ziņojuma sagatavošanu un nodošanu turpmākai apstrādei. Saišu nodrošināšanai starp secību diagrammām un to fragmentiem tiek lietotas vārtejas (angl. *gate*). 4.6. attēlā ir parādītas 2 vārtejas: *STIPReqGate* – pieprasījuma avota modelēšanai un *STIPRespGate* – atbildes saņēmēja prezentēšanai. Serviss *SWITCH_REQ* asinhroni izsauc *STIP_SUBSTIT* servisu ar ilguma ierobežojumu uz servisa izsaukšanu un laika ierobežojumu uz kopējo servisa *STIP_SUBSTIT* apstrādes laiku. Lai sistēmas modelī attēlotu šos laika ierobežojumus, autors izvēlējās 4.5. attēlā definēt laika ierobežojumus *local* fragmenta izsaukšanai un 4.6. attēlā atstāt *STIP_SUBSTIT* servisa izsaukšanu bez laika ierobežojumiem.

4.6. attēlā prezentētā secību diagramma parāda *STIP_SUBSTIT* servisa darbības scenāriju, iekļaujot secīgu sinhronu 4 servisu izsaukšanu ar līdzīgiem ilguma ierobežojumiem uz izsaukšanu (*0..6*) un laika ierobežojumiem uz šo servisu izsaukšanu. Kā arī scenārijs paredz neobligāto *bank_system* fragmentu, kura piederība apstrādes scenārijam tiek konfigurēta sistēmā. Šis fragments izpildās, ja sistēmas konfigurācija paredz banku sistēmas izsaukšanu gala lēmuma pieņemšanai par autorizācijas apstiprināšanu vai noraidīšanu. Interfeisa servisos tiek definēts laika ierobežojums atbildes gaidīšanai (*tout < 12*) un pārējie laika ierobežojumi ir līdzīgi ar citu servisu izsaukšanu. *STIP_SUBSTIT* serviss ziņojumus apstrādā atsevišķajā Tuxedo transakcijā, tāpēc servisa apstrādes beigās notiek *commit* operācija ar laika ierobežojumu (*tSTIP_SUBSTIT < 20*) uz visu izmaiņu saglabāšanu datubāzē.



4.6. att. Fragmenta *local* apstrādes scenārijs finansu autorizācijas laikā

4.5. un 4.6. attēlos prezentētas UML secību diagrammas apraksta autorizācijas ziņojuma apstrādes scenārijus būtiskākajos RTPS sistēmas servisos. Sistēmas sarežģītības dēļ, nav iespējams darbā aprakstīt pilnu apstrādes scenāriju ar visiem iespējamiem servisiem un apstrādes alternatīvām. Tajā pašā laikā prezentētais modelis atbilst sistēmas darbības scenārijam un prezentē sistēmā implementētus laika ierobežojums, kas ir atslēgas moments dotajā pētījumā.

4.4 Transformācijas likumu definēšana

Transformācijas likumi apraksta testpiemēru ģenerēšanas principus. Manuālajā testēšanā šādi principi balstās uz plaši pielietojamām metodēm, kas kļuva par testēšanas standarta metodēm. Darba 1.2 nodaļā tiek pieminētas šādas standarta metodes, kas plašāk ir aprakstītas [SPI 2007] [MYE 2004].

Laika ierobežojumi ir sistēmas apstrādes aspekti, pie kuriem mainās tās apstrādes rezultāts. Laika ierobežojumiem tiek definētas konkrētās pieļaujamās vērtības starp visām iespējamām. Tas nozīmē, ka laika ierobežojumiem var pielietot tādas testēšanas metodes kā sadalīšana ekvivalences klasēs, robežvērtību analīze un kombinācija starp šīm metodēm [SPI 2007]. Transformācijas likumiem šādā gadījumā ir jāimplementē šo testēšanas metožu principi un jāspēj ģenerēt testpiemēri pēc minētām metodēm.

Kā piemērs tiek apskatīts abstrakts laika ierobežojums laika vērtībai *t* vienāds ar „*3..5*”, pie kura pieļaujamās vērtībās ir no 3 līdz 5, neieskaitot robežvērtības. Ņemot vērā to, ka laika vērtība nevar būt mazāka par nulli, parādās vēl viens dabisks ierobežojums ($t >= 0$). Savukārt nulle nevar būt reāla pieļaujamā vērtība, jo pēc laika mainīga iniciēšanas līdz jebkurai nākamai operācijai paiet kāds laiks, kas ir lielāks par nulli (dotajā apgalvojumā netiek apskatīti gadījumi, kad laika mainīga formāta dēļ notiek noapaļošana un tiek iegūta nulle un tiek uzskatīts, ka laika mainīgais var apstrādāt minimālās laika izmaiņas). Tas nozīmē, ka $t = 0$ ir tikpat nereālā vērtība kā $t < 0$, un kopējais laika mainīgo ierobežojums izskatās $t > 0$ (testos tiks apskatītas tikai šīs vērtības). Ņemot vērā šo ierobežojumu, minētam piemēram var izdalīt sekojošas ekvivalences klases:

- (0 ; 3] – laika vērtības, pie kurām sistēmas apstrādes rezultāts ir aplams.
- (3 ; 5) – pieļaujamo vērtību intervāls.
- [5 ; MAX_VAR] – vērtības, pie kurām apstrādes rezultāts ir aplams (ar MAX_VAR tiek saprasta maksimāli apstrādājamā laika mainīgā vērtība).

Robežvērtību pieeja paredz minimālo vienību definēšanu apstrādājamam mainīgajam, lai nodrošinātu vērtības, kas minimāli atšķīrās no robežvērtībām (piemēram, veseliem skaitļiem tā minimālā vērtība ir 1, savukārt mainīgajam ar divām

zīmēm aiz komata minimālā vērtība ir 0,01). Dotajam abstrakta piemēram pieņemsim, ka minimālā novērojama laika vērtība ir 0,0001 un tas nozīmē, ka ekvivalences klases metodes ar robežvērtību analīzes kombinācijas rezultātā ir jātestē situācijas pie šādiem nosacījumiem un šādiem sagaidāmiem rezultātiem:

- 0,0001 – aplams rezultāts.
- 3 – aplams rezultāts.
- 3,0001 – veiksmīgs rezultāts.
- 4,9999 – veiksmīgs rezultāts.
- 5 – aplams rezultāts.
- MAX_VAR – aplams rezultāts, turklāt pēc [SPI 2007] norādījumiem, lai mazinātu izpildāmo testpiemēru skaitu, šo gadījumu var neapskatīt.

Lai nodrošinātu minēto testpiemēru ģenerēšanu tiek izstrādāti sekojošie transformācijas likumi:

```
FOR: MESSAGE SOME_MESSAGE WITH: timingconstraint is not null
DO: CREATE TC WITH: timingconstraint = 0.0001 WHICH:

FOR: MESSAGE SOME_MESSAGE WITH: timingconstraint is not null
DO: CREATE TC WITH: timingconstraint = min WHICH:

FOR: MESSAGE SOME_MESSAGE WITH: timingconstraint is not null
DO: CREATE TC WITH: timingconstraint = min + 0.0001 WHICH:

FOR: MESSAGE SOME_MESSAGE WITH: timingconstraint is not null
DO: CREATE TC WITH: timingconstraint = max WHICH:

FOR: MESSAGE SOME_MESSAGE WITH: timingconstraint is not null
DO: CREATE TC WITH: timingconstraint = max – 0.0001 WHICH:
```

Šo likumu transformācijas rezultātā tiek iegūti iepriekš minētie testpiemēri un automātiski tiek izrēķināts sagaidāmais rezultāts. Minētie likumi prezentē standarta testēšanas metožu pielietošanu transformācijas procesā un ir paredzēti laika ierobežojumu testēšanai. Analoģiskie transformācija likumi tiek izstrādāti arī ilguma ierobežojuma verificēšanai.

Aprakstītam finansu autorizācijas apstrādes procesam tiek definēti sekojošie testēšanas nosacījumi laika ierobežojumu verificēšanai:

1. pārbaudīt visu servisu *commit* operācijas darbību laika ierobežojumu pārsniegšanas un nepārsniegšanas gadījumos;
2. pārbaudīt laika ierobežojumu pārsniegšanu un nepārsniegšanu konkrētajā *local* fragmentā;
3. pārbaudīt gadījumus, kad tiek un netiek pārsniegti laika ierobežojumi *local_processing* fragmenta izsaukumiem pie nosacījuma, ka *RSW_STIP_FLAG = F*;
4. pārbaudīt laika ierobežojumus visiem izsaukumiem no konkrētā *STIP* servera;
5. pārbaudīt laika ierobežojumus visiem sinhroniem izsaukumiem;
6. pārbaudīt laika ierobežojumus visiem asinhroniem izsaukumiem.

Minētie testēšanas nosacījumi apraksta iespējamos verificēšanas aspektus, kas var tikt pielietoti kopā vai jebkādā citā kombinācijā. Pielietojot šos nosacījumus kopā, acīmredzami, ka tiks iegūti arī dublējošie testpiemēri, jo 5. un 6. nosacījumi pārklājas ar iepriekšējiem nosacījumiem. Šāds nosacījumu klāsts tiek izvēlēts lai parādītu iespējamus un dzīvē pielietojamus verificējamus gadījumus.

Pielikumā 2 ir parādīti transformācijas likumi, kas atbilst iepriekš minētiem testēšanas nosacījumiem (60 transformācijas likumi atbilst sešiem nosacījumiem). Definētie transformācijas likumi paredz laika un ilguma ierobežojumu analīzi un ir fokusēti uz testpiemēru ģenerēšanu atbilstoši aprakstītām testēšanas metodēm – sadalīšana ekvivalences klasēs un robežvērtību analīze.

4.5 Transformācijas process un iegūtais modelis

Transformācijas rīks tiek darbināts ar definētiem transformācijas likumiem un ar sistēmas modeli, kas tika aprakstīts 4.3. nodaļā. Transformācijas rezultātā tiek iegūti 180 testpiemēri, kas ir aprakstīti 3. pielikumā. 4.1. tabulā ir sniegts ģenerēto testpiemēru sadalījums pa transformācijas likumiem.

4.1. tabula

Ģenerēto testpiemēru sadalījums pa transformācijas likumiem

transformācijas likumi	**ģenerēto testpiemēru skaits katram no pārskaitītiem likumiem**
46, 47, 48, 49, 50	7
36, 37, 38, 39, 40	5
11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 56, 57, 58, 59, 60	4

transformācijas likumi	ģenerēto testpiemēru skaits katram no pārskaitītiem likumiem
1, 2, 3, 4, 5, 41, 42, 43, 44, 45	3
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 51, 52, 53, 54, 55	2
6, 7, 8, 9, 10, 31, 32, 33, 34, 35	0

No 4.1. tabulas ir redzams, ka daļa no transformācijas likumiem neveicināja testpiemēru ģenerēšanu, savukārt, balstoties uz citiem likumiem, tiek ģenerēti vairāki testpiemēri.

Iepriekšējā nodaļā minētie testēšanas nosacījumi paredz dublējošo testpiemēru ģenerēšanu. Testpiemēru unikalitāti nodrošina konkrētā verificējamā gadījuma sistēmas uzvedība – komplekts no *behavior* un *sut* laukiem. Pielikumā 4 ir parādīti dublējošie uzģenerētie testpiemēri un to atkārtošanas skaits. Tajā pašā laika tabulā nav parādīts 21 unikāls testpiemērs. Neņemot vērā dublējošus testpiemērus, transformācijas laikā tika uzģenerēti 72 unikālie testpiemēri.

4.6 Metodes novērtēšana

Darbā aprakstītā metode ļauj definēt testpiemēru ģenerēšanas likumus un ar transformācijas rīku veidot testēšanas datus, nepieciešamus laika ierobežojumu verificēšanai. Tekošā rīka realizācija apzināti neparedz dublējošo testpiemēru dzēšanu vai neģenerēšanu. Tas ir darīts ar mērķi veidot visus gadījumus lai, balstoties uz tiem, varētu spriest par rīka funkcionēšanu un analizēt tā apstrādes rezultātu.

Lai nodrošinātu pielietotās metodes salīdzinājumu ar manuālo testēšanu, papildus automātiski uzģenerētiem testpiemēriem manuāli tiek izstrādāti testpiemēri vienu un to pašu īpašību verificēšanai. Testpiemēru izstrāde pēc klasiskās robežvērtību analīzes testēšanas metodes, balstoties uz sistēmas specifikāciju (šajā gadījuma uz sistēmas UML secību diagrammu), notika pēc automātiskās metodes pielietošanas un pirms sistēmas darbināšanas. 4.4. sadaļā tika definēti 6 nosacījumi automātisko testpiemēru ģenerēšanai, kas tika pielietoti arī manuālajā testpiemēru izstrādes procesā. Izstrādes process iekļāva sekojošas aktivitātes:

1. sistēmas modeļa analīze – tiek apskatīts un analizēts sistēmas funkcionēšanas modelis ar definētiem laicīguma ierobežojumiem;
2. robežvērtību nosacījumu izvirzīšana – balstoties uz klasisko metodes formulējumu tiek izvirzīti kritēriji testpiemēru ģenerēšanai (piemēram,

maksimāli pieļaujamā apstrādes laika sasniegšana vai minimālā pieļaujamā apstrādes laika pārsniegšana).

3. testpiemēru pierakstīšana testpiemēru vadības sistēmā (angl. *test case management system*) – tiek analizēts katrs laika ierobežojums un balstoties uz 2. punktā izvirzītiem kritērijiem speciālā formātā tiek pierakstīti testpiemēri, kas ir jāveic, lai nodrošinātu apskatāmo laika ierobežojumu verificēšanu.
4. izveidotu testpiemēru caurskate – pēc manuālās testpiemēru izveides tie tiek apskatīti kopumā ar mērķi pamanīt trūkstošus vai dublējošus gadījumus.

Lai iegūtu neatkarīgu manuālo testpiemēru sagatavošanu, to bija nepieciešams veikt neatkarīgam cilvēkam, kas nepiedalījās sistēmas modeļa izveidē. Savukārt, nepieejamo cilvēku resursu dēļ, manuālo testpiemēru izveidi veica šī darba autors, cenšoties minimizēt atkarību no pārāk lielām zināšanām par sistēmas modeli. Šī procesa rezultātā tika izstrādāti 72 testpiemēri, kas pilnīgi atbilst automātiski uzģenerētiem gadījumiem.

Lai novērtētu piedāvātās metodes efektivitāti un salīdzinātu automātisko un manuālo testpiemēru ģenerēšanu, tika izvirzīti 3 kritēji, kas, pēc autora domām, apraksta metodes kvalitāti, nepieciešamo laiku un resursus:

- unikālo testpiemēru skaits (mērāms un salīdzināms skaitļos);
- testpiemēru izstrādei patērētais laiks (mērāms un salīdzināms 8 stundu darba dienās, neiekļauj sistēmas darbināšanas laiku);
- prasības cilvēku resursiem, lai izveidotu testpiemērus (nav iespējams viennozīmīgi prezentēt ar skaitļiem, jo nevar salīdzināt dažādas zināšanas viens pret vienu).

Balstoties uz manuāli izveikto procesu un piedāvātās metodes pielietošanu 4.2. tabulā ir parādīts divu metožu salīdzinājums pēc augstāk minētiem kritērijiem.

No 4.2. tabulas var secināt, ka abas pieejas pie vieniem un tiem pašiem kritērijiem vienam sistēmas modelim ļauj ģenerēt vienādu testpiemēru skaitu. Šāds novērojums ir loģisks, jo automatizācijas rīki ir spējīgi uztaisīt visu to, ko var izdarīt manuāli, tikai ievērojami ātrākā laikā. Turklāt, palielinoties sistēmas modelim vai kļūstot arvien sarežģītākam, automatizācija šī uzdevuma kontekstā kļūst arvien efektīvāka no patērēta laika aspekta. To rāda tabulā prezentētais laiks, kas bija

nepieciešams abu metožu pielietošanai. Jaunās metodes gadījumā tas ir ievērojami mazāks. Savukārt, ir jāņem vērā, ka minētie skaitļi apraksta tendenci starp divām metodēm un nepietiekamo eksperimentu dēļ nevar tikt lietoti kā apstiprinātie un pieradītie efektivitātes salīdzināšanas koeficienti.

4.2. tabula

Piedāvātās metodes salīdzināšana ar klasisko testpiemēru ģenerēšanu

Kritērijs	**Manuālā pieeja**	**Piedāvātā metode**
testpiemēru skaits	72	72
patērētais laiks	5 dienas	1 diena
prasības cilvēku resursiem	• UML modelēšanas notācijas zināšanas; • sistēmas funkcionēšanas principu pārzināšana; • zināšanas par sistēmas uzvedību laika pārsniegšanas gadījumos.	Ieskaitot specifisku transformācijas likumu izveidi: • UML modelēšanas notācijas zināšanas; • sistēmas funkcionēšanas principu pārzināšana; • zināšanas par sistēmas uzvedību laika pārsniegšanas gadījumos; • transformācijas notācijas pārzināšana; • zināšanas par UML modeļu klasēm un atribūtiem.

No 4.2. tabulas var redzēt, ka piedāvātās metodes pielietošanai ar jauno transformācijas likumu definēšanu ir nepieciešamas zināšanas par transformācijas notāciju, UML klasēm un to atribūtiem. UML popularitātes un plašās pielietošanas, kā arī transformācijas notācijas vienkāršās struktūras dēļ, darba autors uzskata, ka minētās papildus prasības jaunās metodes pielietošanai nav būtiskas un nevar ietekmēt metodes pielietošanu.

Tajā pašā laikā viens no būtiskākajiem pozitīviem metodes aspektiem ir metodes automātiskā testpiemēru ģenerēšana. Tas nozīme, ka tiek samazināts cilvēciskais faktors kļūdu pieļaušanai testpiemēru ģenerēšanas ķēdē. Pateicoties iepriekš nodefinētiem transformācijas likumiem, sistēmas modelis tiek analizēts un transformēts pa tiešo uz testēšanas modeli cilvēkam neiejaucoties. Automātiska testpiemēru ģenerēšana nodrošina ātrāku testa datu iegūšanu, netērējot laiku uz manuālo testu veidošanu. UML testēšanas profila lietošana par metamodeli testēšanas datiem ļauj pielietot tos arī citās sistēmās. Atbilstība OMG standartiem ļauj izvairīties no posmiem, kas varētu būt nepieciešami testēšanas datu pārnešanai kādā no testēšanas vadības rīkiem, kas atbalsta šo standartu.

Piedāvātai transformācijas valodai ir divas būtiskās priekšrocības. No vienas puses valoda ir vienkārša un ļauj fokusēties uz sistēmas funkcionalitāti un testēšanas artefaktiem, bet no otrās puses – piedāvā iespējas rakstīt triviālus un komplicētus transformācijas likumus. Darbā iepriekšējā nodaļā tika definēti vienkāršie transformācijas likumi, kas nodrošina testpiemēru ģenerēšanu, balstoties uz standarta testēšanas metodēm. SQL WHERE nosacījumu definēšanas bloks dod plašas iespējas transformācijas likumu izstrādātājiem. Tā, piemēram, var uzrakstīt transformācijas likumu, kurš analizēs ne tikai vienu konkrētu ziņojumu, bet arī meklēs citus ziņojumus, kuriem avota serveris ir tekošā ziņojuma galamērķis. Šāds likums izskatās šādi:

```
FOR: MESSAGE * WITH:  durationconstraint is not null and destination
in
      (select source from
            (select source, count(source) as cnt from all_messages
            where messagesort <> 'reply' and source like 'class%' and
            destination like 'class%' group by source)
      where cnt = 1 )
DO: CREATE TC WITH: durationconstraint = 0.0001 WHICH:
```

Šis transformācijas likums meklē visus ziņojumus uz kādu noteiktu serveri, kurš izsauc tikai vienu citu serveri. Apstrādājot šo likumu, tiek piemeklēts izsaukums no STIP servera uz ACNT_SEL (kurš tālāk izsauc STIP_ADM) un tiek ģenerēts viens testpiemērs ar sekojošu uzvedību – *„dACNT_PRMR_LCK2(durationobservation) = ,0001”*. Aprakstītais piemērs parāda transformācijas valodas pielietošanu vienkāršiem, kā arī komplicētiem transformācijas likumiem.

Vēl viena piedāvātās metodes priekšrocība ir UML un XMI standartu pielietošana avota modeļu definēšanai, kas ļauj viegli integrēt metodi dažādās izstrādes vidēs. UML modelēšanas valoda ir atzīta visā pasaulē un tiek plaši pielietota. Savukārt, XMI standartu nodrošina lielākā daļa modelēšanas rīku un tas ļauj sagatavot XMI datni ar sistēmas modeli turpmākai transformācijai uz testēšanas modeli.

Metode ir realizēta ar vairākiem rīkiem, ievērojot komponenšu bāzēto projektēšanu. Šāda pieeja ļauj uzlabot atsevišķas komponentes pēc atbilstošās nepieciešamības. Tekošā rīku realizācija atbalsta XMI 2.1 versiju un gadījumā, kad būs nepieciešams pāriet uz jaunāko XMI versiju ieviesto izmaiņu dēļ, izmaiņas būs

nepieciešamas tikai Ruby programmā, kas nodrošina datu translēšanu uz datubāzes tabulām. Šajā gadījumā, transformācijas rīka izmaiņas nav nepieciešamās.

Piedāvātai metodei ir arī ierobežojumi. Pirmkārt, metode neparedz sistēmas automātisko darbināšanu. Lai nodrošinātu universālo metodi un tās pielietošanu dažādām izstrādes vidēm un sistēmām, automātiskā sistēmas darbināšana apzināti netiek veidota promocijas darba ietvaros veiktā pētījuma gaitā. Metodes uzdevums ir ģenerēt testpiemērus, kurus nepieciešams veikt, lai verificētu nefunkcionālās īpašības.

Vēl viens metodes ierobežojums ir transformācijas rīka implementēšana Oracle DBVS vidē, kas ir komerciāls produkts, un līdz ar to metodes pielietošana var prasīt noteiktas izmaksas Oracle licencēm. Oracle tika izvēlēta kā populārākā DBVS, kas ļauj nodrošināt metodei nepieciešamās funkcijas. Transformācijas rīka izstrādē tiek lietoti PL/SQL bloki, kas tiek uzturēti tikai Oracle vidē. Savukārt, šo trūkumu nevar attiecināt pie kritiskajiem, jo programmatūras izstrādes uzņēmumos un mācību iestādēs DBVS pielietošana neprasa papildus licences un līdz ar to neierobežo metodes pielietošanu. Tajā pašā laikā, komponenšu bāzētā projektēšana, nepieciešamības gadījumā, pieļauj transformācijas komponentes pārrakstīšanu citā tehnoloģijā.

4.7 Nodaļas secinājumi

Piedāvātā metode tika pielietota izstrādes projektā, kurā notika reāla laika maksājuma karšu sistēmas Card Suite RTPS uzlabošana un oficiālās versijas sagatavošana piegādei esošiem un jauniem klientiem. Projekta rezultātā tika izstrādāta esošā produkta jaunā versija, kas veiksmīgi ieguva PA DSS sertifikātu, notika pieņēmšanas testēšana klienta pusē un pašlaik sistēma tiek darbināta pie vairākiem Card Suite klientiem. Metodes pielietošana tika saskaņota un iepriekš ieplānota ar SIA Tieto Latvia uzņēmuma izstrādes departamenta vadītāju un metodei atbilstoši darbi notika pēc sākotnēji sagatavota plāna.

Darba autors projekta ietvaros nodrošināja visu testēšanas aktivitāšu plānošanu un uzraudzību, ieskaitot testēšanas darbu novērtēšanu, testēšanas resursu izdalīšanu, darbu atsekošanu un testēšanas progresa ziņošanu projekta vadībai. Sistēmas laicīguma ierobežojumu testēšana notika, balstoties uz šī darba 3. nodaļā aprakstītu metodi. Lietojot Sparx Enterprise Architect modelēšanas rīku, darba autors izstrādāja

UML secību diagrammas ar laika ierobežojumiem sistēmas funkcionēšanas specificēšanai. Balstoties uz metodes pielietošanu, darba autors var secināt, ka:

- Piedāvātā metode var tikt integrēta dažādos izstrādes procesos, kur notiek sistēmas darbības modelēšana ar UML diagrammām, kas var tikt eksportētas standarta XMI formātā. Ja šādu diagrammu nav, metode prasa, lai tās būtu izstrādātas metodes pielietošanas laikā. Tas nav būtisks ierobežojums, jo sistēmas dinamikas modelēšanā tieši UML secību diagrammas tiek plaši pielietotas.
- Metode ļāva automātiski uzģenerēt laicīguma īpašību testpiemērus no sistēmas modeļiem, tādejādi samazinot cilvēka faktora ietekmi. Kā arī par papildus efektu var nosaukt uzģenerētas testpiemēru kopas atkārtotu lietošanu.
- Metode paredz izstrādāt standarta transformācijas likumu kopu, kas var tikt pielietota dažādu sistēmu testēšanai un kas var paātrināt testpiemēru izstrādi. Vienīgais, kas nepieciešams metodes lietošanai, ir UML secību diagrammas XMI formātā. To nodrošina vairāki mūsdienas UML atbalsta rīki.
- Metodes pielietošana ļāva pārliecināties par RTPS sistēmas korekto funkcionēšanu pie dažādiem ar laika ierobežojumiem saistītiem gadījumiem un atrast nepareizo konfigurāciju, kas varēja tikt piegādāta klientiem un novest pie sistēmas kļūdainas darbības.

  Balstoties uz metodes novērtēšanu var secināt, ka tā ļauj ātrāk iegūt testējamus gadījumus un metodes efektivitāte var palielināties arvien sarežģītāku sistēmu testēšanas procesā.

NOBEIGUMS

Datorizētās sistēmas arvien vairāk parādās mūsu ikdienā, pildot arvien plašākas funkcijas darba automatizēšanai. Šādas sistēmas nodrošina cilvēku transportēšanu, viņu īpašumu aizsardzību, ražošanas procesa automatizēšanu un optimizēšanu, kā arī ļauj cilvēkam iekarot jaunas kosmosa robežas. Visas šīs sistēmas tiek vadītās ar speciālo iegulto programmatūru, kurai jāpilda pieprasītas funkcijas specifiskos apstākļos. Papildus funkcionālajām prasībām, iegulto sistēmu programmatūrai tiek izvirzīta virkne ar nefunkcionālām īpašībām, ieskaitot laika ierobežojumus, sinhronizēšanu, asinhrono darbību un citas. Lai nodrošinātu minēto sistēmu funkcionēšanu, programmatūra kļūst arvien sarežģītāka un prasības pret tās kvalitāti paliek maksimāli augstas.

Tajā pašā laikā nemitīgi attīstās programmatūras izstrādes process, ieviešot jaunās metodes sistēmu modelēšanā un pirmkoda ražošanā. Modernajā programmatūras izstrādē automatizācijas aktivitātes kļūst populārākas, nodrošinot ātrāku programmatūras izstrādi un minimizējot manuālās darbības posmus, tādejādi ierobežojot cilvēciska faktora ietekmi uz programmas kvalitāti. Modeļvadāmās arhitektūras principi atbalsta programmatūras izstrādes procesa automatizēšanu. Modeļvadāmā programmatūras izstrāde kļūst arvien populārāka un tiek bieži pielietota dažāda veida sistēmu izstrādē. Šāda virzība uz modeļvadāmās arhitektūras principiem tiek nodrošināta, pateicoties izstrādātiem standartiem un atbalsta rīkiem. Piedāvātie standarti nodrošina vienotu pieeju sistēmu modelēšanai, modeļu transportēšanai, pirmkoda ģenerēšanai un citas aktivitātes. Savukārt, balstoties uz šiem standartiem, tiek izstrādāti jaunie rīki, kas atbalsta programmatūras izstrādes procesu. Līdzīgās aktivitātes notiek arī testēšanas jomā, kur 2005. gadā tika izstrādāts UML testēšanas profils, kas apraksta vienotu pieeju testēšanas procesa artefaktu vadībai. Kaut arī kopš tā laika ir pagājuši vairāki gadi, joprojām neeksistē vispārīgi principi pieņemto metožu testēšanas procesa automatizācijai un testpiemēru ģenerēšanai no sistēmas modeļa.

Dotais pētījums ir vērsts uz modeļvadāmās programmatūras principu pielietošanu iegulto sistēmu nefunkcionālo īpašību testēšanai. Pētījuma rezultātā ir izstrādāta metode testpiemēru ģenerēšanai no iepriekš sagatavota sistēmas modeļa, kas ir veidots ar UML modelēšanas valodu. Piedāvātās metodes realizēšanai darba

ietvaros, balstoties uz eksistējošiem OMG standartiem, autors izstrādāja nepieciešamus rīkus un nodefinēja rīku ķēdi, kas nodrošina sistēmas modeļu apstrādi un testpiemēru ģenerēšanu. Lai pārbaudītu metodes pielietošanu, tā tika aprobēta laika ierobežojumu verificēšanā reālā laika maksājuma karšu sistēmai, kurai šīs nefunkcionālās īpašības ir kritiskas standarta darbību veikšanai. Izvirzītā mērķa sasniegšanai tika izpildīti šādi uzdevumi:

- apskatītas un izanalizētas eksistējošās testēšanas metodes, kā arī tās tika izskatītas attiecībā uz pielietošanu iegulto sistēmu testēšanai;
- izpētītas iegulto sistēmu nefunkcionālās īpašības un to eksistējošās testēšanas metodes un modelēšanas paņēmieni;
- veikta MDA principu analīze un ir izvērtētas iespējas pielietot tos testpiemēru ģenerēšanā;
- definēta testēšanas metode, kas nodrošina testpiemēru ģenerēšanu no UML modeļiem;
- piedāvātā metode tika pielietota reālā laika maksājuma sistēmas laicīguma īpašības piemērā un ir secināts par metodes pielietošanas iespējām.

Promocijas darba **galvenais rezultāts** ir piedāvātā testēšanas metode iegulto sistēmu nefunkcionālo īpašību testēšanai un izstrādāto rīku kopa, kas nodrošina UML modeļu transformāciju un testpiemēru ģenerēšanu. Piedāvātā metode balstās uz modeļvadāmās programmatūras izstrādes pamatiem un vispārīgiem modeļu transformācijas principiem. Izstrādātais rīks tika pielietots reālās maksājuma karšu sistēmas laicīguma īpašības testēšanā.

Promocijas darba **svarīgākie rezultāti** ir:

1. Sistematizēta informācija par iegulto sistēmu nefunkcionālām īpašībām, kā arī ir aprakstīti analizēto īpašību modelēšanas un testēšanas paņēmieni.
2. Balstoties uz to, ka iegulto sistēmu testēšanai izmanto arī klasiskās testēšanas metodes, ir aprakstīts vispārīgais testēšanas process un tā metodes.
3. Definēti modeļvadāmās programmatūras izstrādes procesa principi, kā arī detalizēti ir aprakstīti tā artefakti.
4. Piedāvāts atbilstības modelis starp modeļvadāmās arhitektūras principiem un vispārīgo testēšanas V modeli.

5. Definēta uz modeļiem balstīta testēšanas metode, kas ir pielietojama nefunkcionālo īpašību testēšanai.
6. Balstoties uz testēšanas specifiku, ir izstrādāta transformācijas notācija, kas ir pielietota likumu definēšanai lai transformētu sistēmas modeļus uz testēšanas modeli.
7. Balstoties uz piedāvāto testēšanas metodi, ir izstrādāti rīki, tajā skaitā transformācijas rīks, kas nodrošina UML modeļu transformāciju uz testpiemēriem, pierakstītiem UML testēšanas profilam atbilstošajā formā.
8. Piedāvātā metode un izstrādātie rīki ir aprobēti uz reālā laika maksājuma karšu sistēmas laicīguma īpašību verificēšanai.

Balstoties uz promocijas darba ietvaros veiktiem pētījumiem un iegūtiem rezultātiem, ir veikti **sekojošie secinājumi**:

1. Iegulto sistēmu funkcionālajā testēšanā pielieto klasiskās testēšanas metodes, verificējot pēc iespējas vairāk testpiemērus un sasniedzot augstāku pārklājumu pirmkoda elementiem.
2. Eksistējošās iegulto sistēmu nefunkcionālo īpašību testēšanas metodes balstās uz iepriekšējo paaudžu modelēšanas notācijām un manuālo testpiemēru ģenerēšanu.
3. Neeksistē vispāratzītās modeļvadāmās testēšanas pieejas funkcionālo un nefunkcionālo īpašību verificēšanai.
4. Testpiemēri var tikt prezentēti noteiktajā notācijā un tie var tikt automātiski iegūti no sistēmas modeļiem, pielietojot modeļvadāmās programmatūras izstrādes principus.
5. Lai nodrošinātu transformācijas likumu fokusēšanos uz loģiskajām darbībām un izvairītos no komplicētās analizējamās struktūras aprakstiem, testēšanas metodē ir paredzēta sākotnējā sistēmas modeļa vienkāršota prezentācija, kas tiek apstrādāta ar transformācijas likumiem.
6. Balstoties uz to, ka piedāvātā metode tika veiksmīgi pielietota reālās sistēmas laicīguma īpašības verificēšanai, piedāvātā metode var tikt pielietota arī pārējo nefunkcionālo īpašību testēšanā.

Turpmākais virziens jaunajam pētījumam var būt saistīts ar metodes attīstīšanu, lai nodrošinātu pilnīgi automatizētu sistēmas testēšanu, ieskaitot testpiemēru ģenerēšanu, to darbināšanu un rezultāta pārbaudi.

Izstrādāto metodi un rīkus ir ieteikts pielietot laicīguma, sinhronizācijas un asinhronās darbības īpašību testēšanai. Metode var tikt pielietota ne tikai iegultām sistēmām, bet arī citu veidu sistēmām, kurās ir implementētas minētās īpašības. Metode var tikt viegli integrēta izstrādes procesā, kas jau paredz sistēmas darbības specificēšanu ar UML modeļiem, jo tā neparedz papildus soļus sākotnējo datu sagatavošanai un apstrādā UML modeļus standarta XMI formātā.

Izstrādāto metodi un rīkus autors iesaka pielietot arī citu nefunkcionālu īpašību testēšanā, jo rīku implementēšanā ir pielietots komponenšu bāzēts projektējums, kas nodrošina to neatkarību savā starpā un nepieciešamības gadījumā tie var tikt elastīgi uzlaboti. Piedāvāto testēšanas metodi, kas balstās uz vienkāršotu sistēmas modeļa prezentāciju, darba autors iesaka pielietot arī funkcionālo īpašību testēšanā, jo metodē aprakstītie principi ir vienlaicīgi derīgi gan priekš nefunkcionālām, gan arī funkcionālām prasībām.

LITERATŪRAS SARAKSTS

[ACK 2008] Ackermann C., Cleaveland R., Ray A., Shelton C., Martin C. Integrated Functional and Non-Functional Design Verification for Embedded Software Systems // In proceedings of SAE World Congress & Exhibition, Paper Number 09AE-0202, USA – 2008. / Internets: http://www.cs.umd.edu/Grad/scholarlypapers/papers/Ackermann.pdf, pieejams 2012. gada 19. martā.

[ACRO 2011] Aircraft Crashes Record Office. Statistics – 2011. / Internets: http://www.baaa-acro.com/Statistiques%20diverses.htm, pieejams 2012. gada 19. martā.

[AGR 2001] Agrawal S., Bhatt P. Real-time Embedded Software Systems – 2001. / Internets: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.202.2819&rep=rep1&type=pdf, pieejams 2012. gada 19. martā.

[ALT] Altova. UModel - UML tool for software modeling and application development. / Internets: http://www.altova.com/umodel.html, pieejams 2012. gada 19. martā.

[ATI 2009] Atig M. F, Habermehl P. On Yen's Path Logic for Petri Nets. In Proceedings of 3rd International Workshop Reachability Problems 2009, Olivier Bournez,Igor Potapov (Eds.). Springer Verlag, Berlin, Germany – 2009. – pp. 51-63.

[BAK 2010] Baker P., Dai Z. R., Grabowski J. u.c. Model-Driven Testing: Using the UML Testing Profile, Springer-Verlag, Berlin, Germany – 2010. – 200 p.

[BAT 2008] Bath G., McKay J. The Software Test Engineer's Handbook: A Study Guide for the ISTQB Test Analyst and Technical Analyst Advanced Level Certificates. Rocky Nook – 2008. – 416 p.

[BBC 2009] BBC. Yemen jet crashes in Indian Ocean – 2009. / Internets: http://news.bbc.co.uk/2/hi/8125664.stm, pieejams 2012. gada 19. martā.

[BEA 2011] Bureau d'Enquêtes et d'Analyses. Interim report on the accident on 1st June 2009 – 2011. / Internets: http://www.bea.aero/docspa/2009/f-cp090601e1.en/pdf/f-cp090601e1.en.pdf, pieejams 2012. gada 19. martā.

[BES] Besser O., Dorotska C., Rudolf G. Exchange of UML Models in a Design Chain Using a Barkley Database. / Internets: http://www.informatik.tu-freiberg.de/prof2/publikationen/CADDD07_EUMUDBD.pdf, pieejams 2012. gada 19. martā.

[BRO 2003] Broekman B., Notenboom E. Testing embedded software. Pearson Education – 2003. – 348 pages.

[BUR 2003] Burnstein I. Practical software testing, New York: Springer-Verlag New York, Inc. – 2003. – 709 p.

[COE 1997] Coen-Porisini A., Ghezzi C., Kemmerer R. Specification of real-time systems using ASTRAL, IEEE Transactions on Software Engineering, Vol. 23, No. 9 - 1997. – pp. 572-598

[COND] Oracle. Conditions. / Internets: http://download.oracle.com/docs/cd/B19306_01/server.102/b14200/conditions.htm#g1077361, pieejams 2012. gada 19. martā.

[COP 2004] Copeland L. A Practitioner's Guide to Software Test Design. Artech House Publishers, London, England. – 2004. – 294 p.

[COR 2000] Cortés L. A., Eles P., Peng Z. Verification of Embedded Systems using a Petri Net based Representation // 13th International Symposium on System Synthesis, ISSS'2000, Madrid, Spain. – 2000. / Internets: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.34.9479&rank=1, pieejams 2012. gada 19. martā.

[CSYS] The Free Dictionary. Complex System. / Internets: http://encyclopedia2.thefreedictionary.com/Complex+System, pieejams 2012. gada 19. martā.

[DOU 2004] Douglass B. P. Real Time UML: Advances in the UML for Real-Time Systems. Addison-Wesley Professional – 2004. – 752 p.

[DSQL] Oracle. Coding Dynamic SQL Statements. / Internets: http://download.oracle.com/docs/cd/B10500_01/appdev.920/a96590/adg09dyn.htm, pieejams 2012. gada 19. martā.

[EA] Sparx Systems. Enterprise Architect./ Internets: http://www.sparxsystems.com/products/ea/index.html, pieejams 2012. gada 19. martā.

[EMF] Eclipse Foundation. Eclipse Modeling Framework Project (EMF). / Internets: http://www.eclipse.org/modeling/emf/, pieejams 2012. gada 19. martā.

[ENC] The Free Dictionary. Data synchronization. / Internets: http://encyclopedia.thefreedictionary.com/Synchronization+%28computer+science%29, pieejams 2012. gada 19. martā.

[ENG 2006] Engels G., Guldali B., Lohmann M. Towards Model-Driven Unit Testing // In Proceedings of the 9th International Conference on Models in Software Engineering MoDELS'06, Genova, Italy. – 2006. – pp. 182-192. / Internets: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.62.200, , pieejams 2012. gada 19. martā.

[FAK] Fakhroutdinov K. Sequence diagrams. Parallel. /Internets: http://www.uml-diagrams.org/sequence-diagrams.html#operator-par, pieejams 2012. gada 19. martā.

[FSS] Formal Systems. Software. / Internets: http://www.fsel.com/software.html , pieejams 2012. gada 19. martā.

[GAR 2011] Gartner. Magic Quadrant for Data Warehouse Database Management Systems. / Internets: http://www.sybase.com/files/White_Papers/Gartner_MagicQuad_forDataWarehouseDMS.pdf, pieejams 2012. gada 19. martā.

[GHE 1989] Ghezzi C., Mandrioli D., Morasca S. u.c. A General Way To Put Time in Petri Nets // In Proceedings of the 5th International Workshop on Software Specification and Design, Pittsburgh, USA. – 1989. – pp. 60-67

[GIR 2002] Girault C., Valk R. Petri Nets for System Engineering. Springer-Verlag, Berlin, Germany – 2002. – 607 p.

[GRI 2005] Grigorjevs J., Nikiforova O. Testing Process Adjustment for Real Time Systems // In Proceedings of the 46th Scientific Conference of Riga Technical University, 5th Series, Vol. 22, Computer Science, Applied Computer Systems, Riga, Latvia: RTU Publishing. – 2005. – pp. 229-241

[GRI 2006] Grigorjevs J., Nikiforova O. Unit Testing for Real Time Systems // Scientific Journal of Riga Technical University, Computer Science, Applied Computer Systems, 5th series, Vol. 26, Riga, Latvia: RTU Publishing. – 2006. – pp. 67-77

[GRI 2007] Grigorjevs J., Nikiforova O. Features of embedded systems that require specific testing approaches // In Proceedings of the 48th Scientific Conference of Riga Technical University, 5th Series, Vol. 30, Computer Science, Applied Computer Systems, Vol. 26, Riga, Latvia. – 2007. – pp. 47-56

[GRI 2008a] Grigorjevs J., Nikiforova O. Modeling of Non-Functional Requirements of Embedded Systems // In Scientific Proceedings of 42nd Spring International Conference MOSIS2008, Ostrava, Czech Republic. – 2008. – pp. 13-20

[GRI 2008b] Grigorjevs J., Nikiforova O. Compliance of Popular Modeling Notations to Non-functional Requirements of Embedded Systems // In Proceedings of the International Scientific Conference Informatics in the Scientific Knowledge 2008, Varna, Bulgaria. – 2008. – pp. 139-149

[GRI 2008c] Grigorjevs J. Testing of Embedded System's Non-functional Requirements // In Scientific Proceedings of the 8th International Baltic Conference Baltic DB&IS 2008, Tallinn, Estonia. – 2008.

[GRI 2008d] Grigorjevs J. Testa gadījumu vadības sistēmas izvēle un ieviešana // 9. Latvijas ikgadējās „Testēšanas teorija un prakse" konferences materiālos, Rīga, Latvija. – 2008.

[GRI 2009a] Grigorjevs J., Nikiforova O. Several Outlines on Model-Driven Approach for Testing of Embedded Systems // Scientific Journal of RTU, 5th series, Computer Science, 38. vol. – 2009. – pp. 96-107

[GRI 2009b] Grigorjevs J. Testing automation framework in combined technology environment // In Proceedings of 10th Annual Software Testing Conference TAPOST2009, Riga, Latvia. – 2009.

[GRI 2010] Grigorjevs J. Several practical approaches in testing automation // In Proceedings of 11th Annual Software Testing Conference TAPOST2010, Riga, Latvia. – 2010.

[GRI 2011a] Grigorjevs J. Model-Driven Testing Approach for Embedded Systems Specifics Verification Based on UML Model Transformation // In Proceedings of 3rd International Workshop on Model-Driven Architecture and Modeling-Driven Software Development, Beijing, China. – 2011. – pp. 26-35

[GRI 2011b] Grigorjevs J. Model-Driven Testing Approach Based on UML Sequence Diagram // In Scientific Journal of Riga Technical University, 5th series, Computer Science, Applied Computer Systems, Riga, Latvia. – 2011. – pp. 85-90

[GRI 2011c] Grigorjevs J. Card Suite Testing Toolbox – product launch case study // In Proceedings of 12th Annual Software Testing Conference TAPOST2011, Riga, Latvia. – 2011.

[HAL 2006] Halbwachs N., Mandel L. Simulation and Verification of Asynchronous Systems by means of a Synchronous Model // In Proceedings of 6th International Conference on Application of Concurrency to System Design, Turku, Finland. – 2006. – pp. 3-14. / Internets: http://hal.archives-ouvertes.fr/docs/00/18/95/67/PDF/main.pdf, pieejams 2012. gada 19. martā.

[HOA 2004] Hoare C. A. R. Communicating Sequential Processes – 2004. / Internets: http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.85.2760, pieejams 2012. gada 19. martā.

[HOL 1995] Holvoet T., Verbaeten P. Petri Charts: an Alternative Technique for Hierarchical Net Construction // In Proceedings of the 1995 IEEE Conference on Systems, Man and Cybernetics, Vancouver, Canada. – 1995. / Internets: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.55.9629, pieejams 2012. gada 19. martā.

[IEEE 829] The Institute of Electrical and Electronics Engineers, Inc., IEEE Standard for Software Test Documentation: IEEE 829-1998. – 1998. – 52 p.

[JEN 1997] Jensen K. An Introduction to the Practical Use of Coloured Petri Nets, Lecture Notes in Computer Science, Volume 1492/1998. – 1998. – pp. 237-292

[JEN 2011] Jensen E. D. Real-Time Overview – 2011. / Internets: http://www.real-time.org/realtimeoverview.htm, pieejams 2012. gada 19. martā.

[JOU 2006] Jouault F., Kurtev I. Transforming Models with ATL // In Proceedings of the MoDELS'06 Conference, Montego Bay, Jamaica. – 2006. – pp. 128-138. / Internets: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.99.6117, pieejams 2012. gada 19. martā.

[KAN 1999] Kaner C., Falk J., Nguyen H.Q. Testing Computer Software, Second Edition. Wiley. – 1999. – 480 pages.

[KLE 2003] Kleppe A., Warmer J., Bast W. MDA Explained: The Model Driven Architecture: Practice and Promise. Addison Wesley – 2003. – 167 p.

[KOS 1982] Kosaraju S. R. Decidability of reachability in vector addition systems. In: STOC, ACM, New York. – 1982. – pp. 267-281.

[KRS 2002] Krstic A., Lai W.-C., Chen L. u.c. Embedded Software-Based Self-Testing For SoC Design // 39th Design Automation Conference, DAC 2002, New Orleans, USA. – 2002. – pp. 355-360. / Internets: http://esdat.ucsd.edu/~lichen/esdat/paper/dac02_krstic.pdf, pieejams 2012. gada 19. martā.

[KSH 1998] Kshemkalyani A. D. Testing of Synchronization Conditions for Distributed Real-time Applications // In Proceedings of IPPS/SPDP Workshops'1998, Orlando, Florida, USA. – 1998. – pp. 1140-1152. / Internets: http://www.springerlink.com/content/rn17r38q117q6676/, pieejams 2012. gada 19. martā.

[KUG 2008] Kugele S., Haberl W., Tautschnig M., Wechs M. Optimizing Automatic Deployment Using Non-Functional Requirement Annotations // In Proceedings of the 3rd International Symphosium ISOLA 2008, Communications in Computer and Information Science, Leveraging Applications of Formal Methods, Verification and Validation, Margaria T., Steffen B. (Eds.), Springer, Greece – 2008. – pp. 400-414.

[LAM 1992] Lambert J. L. A structure to decide reachability in Petri nets. Theoretical Computer Science, Volume 99, Issue 1. – 1992. – pp. 79-104.

[LEV 2004] Levendovszky T., Lengyel L., Mezei G. u.c. A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS // In Proceedings of the 2nd International Workshop on Graph Based Tools (GraBaTs), Rome, Italy. – 2004. / Internets: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.128.3820, pieejams 2012. gada 19. martā.

[LEY 2010] Leyden J. Trojan-ridden warning system implicated in Spanair crash – 2010. / Internets: http://www.theregister.co.uk/2010/08/20/spanair_malware/, pieejams 2012. gada 19. martā.

[LIV 2005] Liversidge E.The Death of the V-Model – 2005./ Internets: http://www.harmonicss.co.uk/index.php/hss-downloads/doc_download/12-death-of-the-v-model , pieejams 2012. gada 19. martā.

[LU 2011] University of Latvia. MOLA – Model Transformation Language. – 2011. / Internets: http://mola.mii.lu.lv/, pieejams 2012. gada 19. martā.

[MAR 2003] Marschall, F., Braun, P. Model Transformations for the MDA with BOTL // In Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications, Enschede, The Netherlands. – 2003. – pp. 25-36. / Internets: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.2.7991, pieejams 2012. gada 19. martā.

[MAT 1995] Mattai J. Real-Time Systems: Specification, Verification, and Analysis. Prentice Hall, PTR Upper Saddle River, New Jersey, USA. – 1995. – 278 p.

[MDA] Object Management Group (OMG). Model Driven Architecture. / Internets: http://www.omg.org/mda, pieejams 2012. gada 19. martā.

[MEL 2004] Mellor S. J., Scott K., Uhl A. u.c. MDA Distilled: Principles of Model-Driven Architecture, Addison Wesley Professional – 2004. – 176 p.

[MTF] IBM. Model Transformation Framework (MTF). / Internets: http://www.alphaworks.ibm.com/tech/mtf , pieejams 2012. gada 19. martā.

[MUR 1989] Murata T. Petri Nets: Properties, Analysis and Applications. Proceedings of the IEEE, Vol. 77, No 4. – 1989. - pp. 541-580.

[MYE 2004] Myers G. J. The art of software testing, Second Edition. New Jersey: John Wiley & Sons, Inc. – 2004. – 255 pages.

[NIK 2007] Ņikiforova O., Ņikuļšins V., Buzdins D. u.c. Modeļvadamas programmatūras izstrādes pamati (II daļa), Projektā Studiju moduļa izstrāde modeļvadāmai programmatūras attīstības tehnoloģijai datorsistēmas programmā ietvaros izstrādāts mācību metodiskais materiāls mācību kursam „Programmatūras attīstības tehnoloģijas" datorzinātnes un informācijas tehnoloģijas akadēmiskās bakalaurantūras 3. kursa studentiem , RTU – 2007. – 24 lpp.

[NIK 2008] Nikiforova O., Pavlova N., Grigorjevs J. Several Facilities of Class Diagram Generation from Two-Hemisphere Model in the Framework of MDA. 23rd International Symposium on Computer and Information Sciences, ISCIS 2008 : Proceedings. - Piscataway, NJ : IEEE, 2008. 6 p.

[OMG] Object Management Group (OMG). About OMG®. / Internets: http://www.omg.org/gettingstarted/gettingstartedindex.htm, pieejams 2012. gada 19. martā.

[PADSS] PCI Security Standards Council. Documents Library. PCI Standards Documents. / Internets: https://www.pcisecuritystandards.org/security_standards/documents.php?association=PA-DSS , pieejams 2012. gada 19. martā.

[PER 2010] Peraldi-Frati M., Mallet F., Deantoni J. MARTE for time modeling and verification of real-time embedded system // In proceedings of 1st Inter. Colloque. RUNSUD, Sophia Antipolis, France – 2010. – pp. 480-489, Internets: http://www.i3s.unice.fr/~map/map_fichiers_publi/RUNSUD2010.pdf, pieejams 2012. gada 19. martā.

[PET 1966] Petri Carl A. Communication with automata: COMMUNICATION WITH AUTOMATA: Volume 1 Supplement 1. APPLIED DATA RESEARCH INC PRINCETON NJ. – 1966. / Internets: http://handle.dtic.mil/100.2/AD630125

[PIL 2005] Pilone D., Pitman N. UML 2.0 in a Nutshell. First Edition. O'Reilly Media Inc., Sebastopol, USA. – 2005. – 240 p.

[QVT 2011] Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Query/View/Transformation – 2011. / Internets: http://www.omg.org/spec/QVT/ , pieejams 2012. gada 19. martā.

[RAT 2003] Rational. Using Rose Data Modeler. Rational Rose. – 2003. – 100 p. / Internets: ftp://ftp.software.ibm.com/software/rational/docs/v2003/win_solutions/rational_rose/rose_dm.pdf, pieejams 2012. gada 19. martā.

[REI 1985] Reisig W. Petri Nets, An Introduction, EATCS, Monographs on Theoretical Computer Science, W.Brauer, G. Rozenberg, A. Salomaa (Eds.). Springer Verlag. Berlin. – 1985.

[SOS 2010] Sostaks A. Implementation of Model Transformation Languages, doctoral thesis at University of Latvia, Riga, Latvia. – 2010.

[SPI 2007] Spillner A., Linz T., Schaefer H. Software Testing Foundations: A Study Guide for the Certified Tester Exam, 2nd Edition, Rocky Nook. – 2007. – 288 p.

[SPT] Object Management Group (OMG). UML Profile for Schedulability, Performance, and Time. / Internets: http://www.omg.org/spec/SPTP/ , pieejams 2012. gada 19. martā.

[STA 2006] Stahl T., Voelter M. Model-Driven Software Development: Technology, Engineering, Management, Wiley – 2006. – 444 p.

[STH 2001] Sthamer H., Baresel A., Wegener J. Evolutionary Testing of Embedded Systems // 14th International Internet & Software Quality Week, San Francisco, USA. – 2001. / Internets: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.86.1798&rep=rep1&type=pdf , pieejams 2012. gada 19. martā.

[STI 2008] Stiles G. S., Rice D. D., Doupnik J. R. Design, Verification, and Testing of Synchronization and Communication Protocols with Java – 2008. / Internets: http://www.neng.usu.edu/ece/research/rtpc/projects/JavaCSP/Synch_Comms.pdf, pieejams 2012. gada 19. martā.

[SVO 2011] Свободная Пресса. Неудачные запуски спутников в 2010 году – 2011. / Internets: http://svpressa.ru/world/photo/36421/, pieejams 2012. gada 19. martā.

[TER] Lielā terminu vārdnīca. / Internets: http://termini.lza.lv/term.php?term=synchronization&list=&lang=EN, pieejams 2012. gada 19. martā.

[THA 2004] Thati P., Viswanathan M. Verification of Asynchronous Systems with Unbounded and Unordered Message Buffers – 2004. / Internets: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.129.4683, pieejams 2012. gada 19. martā.

[TIE] Tieto. Card Suite. / Internets: http://www.tieto.com/industries/financial-services/transaction-banking/card-suite, pieejams 2012. gada 19. martā.

[TUX] Oracle. Tuxedo. / Internets: http://www.oracle.com/technetwork/middleware/tuxedo/overview/index.html, pieejams 2012. gada 19. martā.

[UML] Object Management Group (OMG). Unified Modeling Language™ (UML®). / Internets: http://www.omg.org/spec/UML/, pieejams 2012. gada 19. martā.

[UTP] Object Management Group (OMG). UML Testing Profile. / Internets: http://www.omg.org/utp, pieejams 2012. gada 19. martā.

[VW] Volkswagen. The Group. / Internets: http://www.volkswagenag.com/vwag/vwcorp/content/en/the_group.html, pieejams 2012. gada 19. martā.

[VWREC] Volkswagen. Recall Compaigns./ Internets: http://www.volkswagen.co.nz/nz/en/service_and_parts/maintenance_care/mcare_recalls.html, pieejams 2012. gada 19. martā.

[WAN 1998] Wang J. Timed Petri nets: theory and application. Springer – 1998. – 296 p.

[WU 2002] Wu I.-C., Hsieh S.-H. An UML-XML-RDB Model Mapping Solution for Facilitating Information Standardization and Sharing in Construction Industry // In Proceedings of International Symposium on Automation and Robotics in Construction, Maryland, USA. – 2002. – pp. 317-321. / Internets: http://www.fire.nist.gov/bfrlpubs/build02/PDF/b02158.pdf , pieejams 2012. gada 19. martā.

[WU 2007] Wu X., Li J.J., Weiss D.M., Lee Y. Coverage-Based Testing on Embedded Systems // In Proceedings of AST, Minneapolis, USA. – 2007. – pp. 31-36. / Internets: http://arnetminer.org/viewpub.do?pid=2798644, pieejams 2012. gada 19. martā.

[XMI] Object Management Group (OMG). XML Metadata Interchange (XMI®). / Internets: http://www.omg.org/spec/XMI/, pieejams 2012. gada 19. martā.

[YAK] Yakovlev A., Petrov A., Lavagno L. A Low Latency Arbitration Circuit. / Internets: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.9762, pieejams 2012. gada 19. martā.

[YAK 1996] Yakovlev A. V., Koelmans A. M., Semenov A. Modelling, Analysis and Synthesis of Asynchronous Control Circuits Using Petri Nets – 1996. / Internets: http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.7496, pieejams 2012. gada 19. martā.

[ZAN 2009] Zander-Nowicka J. Model-based Testing of Real-Time Embedded Systems in the Automotive Domain, doctoral thesis at Technical University Berlin – 2009. / Internets: http://opus.kobv.de/tuberlin/volltexte/2009/2186/pdf/zandernowicka_justyna.pdf, pieejams 2012. gada 19. martā.

[ZHI 1999] Zhiming L., Mathai J. Specification and Verification of Fault-tolerance, Timing and Scheduling – 1999. / Internets: http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.2264, pieejams 2012. gada 19. martā.

[ZIM 2009] Zimmer U. R. Real-Time and Embedded Systems – 2009. / Internets: http://cs.anu.edu.au/student/comp4330/Level-0/Contents.html, pieejams 2012. gada 19. martā.