VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

Modeliais paremtas testavimas: testavimo įrankių tyrimas

Model-based testing: analysis of the model based testing tools

Magistro baigiamasis darbas

Atliko: Ernestas Adomaitis (parašas)

Darbo vadovas: l. Irmantas Naujikas (parašas)

Recenzentas: Doc. dr. Vladas Tumasonis (parašas)

Vilnius
2011

Santrauka

Ernestas Adomaitis

Modeliais paremtas testavimas: testavimo įrankių tyrimas

Magistrinis darbas

Informatikos programa

Vilniaus universitetas, Matematikos ir informatikos fakultetas, Informatikos katedra

Vilnius, 2011

Modeliais paremtas testavimas tampa vis populiaresnis. Sudėtingos programinės įrangos kokybės užtikrinimas ir modeliais orientuotas sistemų kūrimas yra pagrindinės šio testavimo būdo išpopuliarėjimo priežastys.

Pagrindiniai šio darbo tikslai yra modeliais paremtų testavimo įrankių analizė ir jų tobulinimo galimybės. Darbe naudojama daugiau nei dešimt įvairių įrankių vertinimo kriterijų. Analizuojama testų padengimo kriterijų pritaikymo ir tobulinimo galimybės. Analizuojamos modeliais paremto proceso problemos ir pateikiami sprendimo būdai. Atlikta detali testų kūrimo kriterijų ir modelio perėjimo algoritmų analizė.

Atlikus detalę kriterijų ir modelio perėjimo algoritmų analizę, pateikiama kriterijų apjungimo privalumai ir kaip naudojant įvairius kriterijus galima valdyti testų kūrimo procesą ir siekti 100 procentinio modelio padengimo. Pateikiama ir detaliai išanalizuojama kriterijų kūrimo metodika, kurią galim taikyti testavimo įrankiuose.

Raktiniai žodžiai: Modelis, testavimas, modeliais paremtas testavimas, testavimo įrankiai, testavimo kriterijai

Summary

Ernestas Adomaitis

Model Based Testing: analysis of the model based testing tools

Paper of the Master dagree

Computer science program

Vilnius University, Faculty of Mathematics and Informatics, Department of Computer Science

Vilnius, 2011

Model-based testing has become increasingly popular in recent years. Major reasons include the need for quality assurance for increasingly complex systems and the emerging model-centric development paradigm.

The present thesis aims at analysing model-based testing tools and presents the possibilities for their improvement. More than dozen criteria are applied in analysing selected tools. The paper analyses the application of testing coverage criteria and presents their possible improvements. The problems occurring in the model-based testing process are being analysed and the solutions presented. The study contains an in-depth analysis of test coverage criteria and model traversal algorithms.

A thorough analysis of test criteria and model traversal algorithms being performed, the advantages of the integration of test criteria are presented. Furthermore, a solution for managing the development process of test cases as well as pursuing a hundred percent model coverage is proposed. The paper presents a comprehensive analysis of test criteria development methodology that could be applied in testing tools.

Keywords: Model, testing, model-based testing, testing tools, test coverage criteria

Turinys

Įvadas

Didėjant programinės įrangos patikimumo ir kokybės reikalavimams, testavimas tapo vienas iš pagrindinių programinės įrangos etapų, kuris glaudžiai susijęs su sistemos vystymu. Programinei įrangai testuoti yra naudojama daugybė įvairių testavimo stilių ir kriterijų. Per paskutinius dešimt metų, išpopuliarėjus objektiniam orientavimuisi ir modelių pritaikymui programinės įrangos vystyme, paplito juodos dėžės testavimo metodas, kuris padidino susidomėjimą modeliais paremtu testavimu. Dėl savo paprastumo ir pritaikymo galimybių jis sulaukė didelio pramonininkų bei akademikų pusės dėmesio.

Išpopuliarėjus šiam testavimo būdui, atsirado jo pritaikymui įvairių testavimo įrankių. Nors testavimo būdas nėra senas, bet gausus įrankių. Testavimo būdas pasižymi dideliu testavimo proceso automatizavimu. Atsiranda papildomi iššūkiai: modeliavimas, testų kūrimo automatizavimas ir automatinis testų paleidimas. Šiems žingsniams įgyvendinti reikalingos papildomos žinios ir sugebėjimai. Pasitelkiami įvairūs testų padengimo kriterijai, modelio perėjimo algoritmai ir modeliavimo įrankiai.

Programinei įrangai kurti naudojamos vis naujesnės technologijos. Vartotojo sąsajos tampa vis sudėtingesnės. Programų kūrimo metu atsiranda vis daugiau klaidų. Modeliais paremto testavimo įrankiai padeda spręsti šias problemas.

Tyrimo objektas

Darbe tiriami prieinami testavimo įrankiai. Didžiausias dėmesys skiriamas dvejoms, tik modeliais paremto testavimo procesui būdingoms, dalims – abstraktaus sistemos modelio kūrimui ir automatiniam testų kūrimui. Tyrimai bus atliekami su realiai veikiančiomis Virtualių parodų sistemomis.

Darbo aktualumas

Naujos technologijos ir veiklos reikalauja naujų darbo metodų ar įrankių. Susiduriama su naujomis problemomis ir iššūkiais. Nauji dalykai reikalauja naujų žinių ir tam tikslui įgyvendinti naujų įrankių.

Modeliais paremtam testavimui įgyvendinti yra sukurta daugiau nei 30 įvairių testavimo įrankių. Didžioji dalis yra mokami ir sunkiai prieinami. Neteikiamos net bandomosios versijos. Atviro kodo įrankiai reikalauja papildomų žinių juos pritaikant. Nėra patogios vartojo sąsajos.

Skirtingi įrankiai palaiko skirtingą funkcionalumą. Nėra aiškiai apibrėžta, kaip kokius įrankius galima pritaikyti. Testavime atsiranda naujos sąvokos – modelis ir modeliavimas. Testai kuriami automatiniu būdu. Nėra aiškiai apibrėžta metodikos, kokiais būdais ir kriterijais remiantis galima sukurti geriausius testų rinkinius. Nėra nustatytų kriterijų, pagal kuriuos galima būtų vertinti esamus įrankius.

Tikslai ir siekiami rezultatai

Modeliais paremtiems testavimo įrankiams tirti reikia surinkti įvairią susijusią informaciją; detaliai išnagrinėti modeliais paremto testavimo procesą, pagrindinius jo skirtumus ir sprendžiamas problemas.

Norint pateikti atsirandančių problemų sprendimo būdus ir siūlymus, reikia atlikti detalią esamų modeliais paremto testavimo įrankių analizę. Taip pat būtina išanalizuoti, kokiais būdais įrankiai sprendžia modeliais paremto testavimo problemas: modeliavimą, testų automatinį kūrimą, abstrakčių testų konvertavimą į paleisti tinkamus testų skriptus. Šis darbas turi turėti mokomąją reikšmę. Darbas turėtų pateikti, kaip ir kokiose situacijose galima naudoti įrankius; į kokias įrankio savybes reikia atkreipti dėmesį; kaip tas savybes galima pritaikyti, kad maksimaliai ir greičiausiu būdu būtų pasiekti užsibrėžti testavimo tikslai; kokiais atvejais patartina taikyti šį būdą; kokių žinių ir sugebėjimų turi turėti testuotojų komanda, kuri ruošiasi taikyti šį testavimo būdą. Testuojant dideles ir sudėtingas sistemas, įrankio tobulinimas tampa neišvengiamas. Norint sukurti didelius ir įdomius testus, reikia įvairių programavimo kalbų ir metodų. Darbe bus pateikiami galimi įrankio pagerinimo metodai.

Tikslai: pasiūlyti tinkamiausią ir paprasčiausią kriterijų taikymo būdą, kriterijų tobulinimo ir taikymo metodiką; išanalizavus įvairius įrankius, pateikti galimus testavimo kriterijų ir modeliais paremtame testavime naudojamų algoritmų taikymo būdus bei privalumus; palyginti gautus rezultatus ir pateikti siūlymus, kaip galima pagerinti testų kūrimą.

1. Programinės įrangos testavimas

Skyriuje pateikiami pagrindiniai modeliais paremto testavimo ypatumai ir skirtumai lyginant su įprastais testavimo būdais. Prieš nagrinėjant esamus modeliais paremto testavimo įrankius, bendras supratimas apie modeliais paremtą testavimą yra būtinas, kad geriau būtų suprantamos iškylančios problemos ir sunkumai.

1.1. Klasikiniai testavimo procesai

Testavimas – veikla, atliekama įvertinti produkto kokybę ir produktą pagerinti surandant klaidas ir problemas.

Programinės įrangos testavimo metu iškyla trys pagrindinės užduotys:

Testų kūrimas. Testai kuriami remiantis sistemos reikalavimais atsižvelgiant į testavimo tikslus ir strategiją.

Testų paleidimas ir rezultatų analizė. Testais tikrinama testuojama sistema. Gauti rezultatai tikrinami ir ieškomos rastų klaidų priežastys.

Tikrinama, kiek reikalavimų padengė testai. Testavimo proceso kokybė skaičiuojama remiantis, kiek sistemos reikalavimų padengė testai. Šis procesas atliekamas naudojant perėjimų matricą.

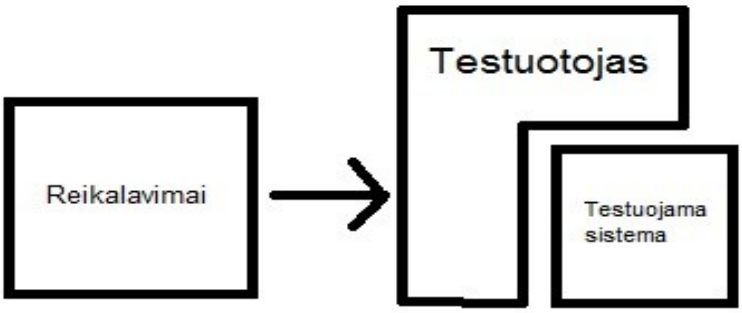
Šioms užduotims spręsti naudojami keturi testavimo būdai: rankinis, kaupimo/kartojimo, paremtas skriptais ir raktiniais žodžiais automatizuotas

1.1.1. Rankinis testavimas

Pats seniausias testavimo būdas, bet vis dar plačiai naudojamas.

Naudojamas testavimo planas, kuris nusako pagrindines testavimo užduotis – kurias testuojamos sistemos dalis testuoti, kokias strategijas pasirinkti, kaip dažnai bus atliekami testai ir kiek jų reikės.

Testai kuriami remiantis sistemos reikalavimais rankiniu būdu. Sukurti testai paleidžiami rankiniu būdu. Visas procesas vykdomas rankiniu būdu. Priklausomai nuo sistemos, testavimas gali užtrukti labai ilgai ir nėra garantijos, kad jo metu bus patikrintas visas sistemos funkcionalumas. Kiekvienas sistemos pasikeitimas reikalauja pakartotino viso proceso vykdymo. Nenaudojamos jokios automatizavimo priemonės.



1 pav. **Paprastas testavimo būdas**

1.1.2. Kaupimo/kartojimo testavimo būdas

Kaupimo/kartojimo testavimo (*angl. capture/replay testing*) metu įrašomi visi veiksmai, atlikti testuojama sistema. Pakartotinio testavimo metu šie veiksmai atliekamai iš naujo. Testai kuriami rankiniu būdu.

Testuoti reikalingas įrankis, surenkantis visą informaciją apie sistema vykdytus veiksmus. Išsaugomos visos įvestys ir išvestys. Išleidus naują sistemą, yra galimybė testus paleisti iš naujo ir gaunamus rezultatus patikrinti su prieš tai buvusiais. Automatizuojamas senų testų paleidimas.

Pagrindinis trūkumas – testavimas labai silpnas. Nedideli programos pasikeitimai gali iššaukti testavimo klaidas (pvz. paprastą lauką užpildant, atsiranda pasirinkimo galimybė). Nedideli vartotojo sąsajos pasikeitimai įrašytus veiksmus padarys nenaudingus.

Pagrindinė problema – per mažas abstrakcijos lygis įrašytuose veiksmuose. Įrašoma labai detali informacija. Šiuo atveju mažas pasikeitimas iššaukia testo klaidą. Automatizavimas tik iš dalies išsprendžia problemą.

1.1.3. Skriptais paremtas testavimas

Testavimo skriptas – skriptas, kuriuo galima paleisti vieną ar daugiau testų. Skripto užduotis – įvesti sistemą į reikiamą būseną, sukurti įvestis, pateikti įvestis testuojamai sistemai, įrašyti gautus rezultatus, gautus rezultatus palyginti su galimais rezultatais ir pranešti, ar testas pavyko. Skriptams rašyti reikalingos programavimo žinios. Testuojamai sistemai kontroliuoti ir stebėti reikalinga aplikacijų programavimo sąsaja (API).

Išsprendžiama testų paleidimo problema ją automatizuojant. Išauga testavimo palaikymo kaina, nes su kiekvienais reikalavimų pasikeitimais ir sistemos versijomis reikalingi skriptų keitimai.

1.1.4. Raktiniais žodžiais paremtas automatizuotas testavimas

Pagrindinė skriptais paremto testavimo problema – abstrakcijos lygis. Žemas abstrakcijos lygis iššaukia palaikymo problemas. Skriptų kiekis tampa labai didelis. Pagrindinis raktiniais žodžiais paremto automatizuoto testavimo (*angl. keyword-driven automated testing*) tikslas – sumažinti abstrakcijos lygį. T. y. transformuoti kiekvieną testą kuo paprasčiau, kad būtų suprantamas testuojamai sistemai.

Tai atliekama naudojant raktinius žodžius, kuriais skriptas gali būti pritaikomas keliems testams. Taip padidinamas abstrakcijos lygis. Šiam testavimo būdui taip pat reikalingos programavimo žinios. Testavimo duomenų rinkimas ir testavimo padengiamumo stebėjimai atliekami rankiniu būdu.

1.1.5. Išspręstos ir likusios problemos

Žemiau lentelėje pateiktos testavimu išspręstos ir likusios problemos.

1 lentelė. **Testavimo procesų palyginimas [UL07]**

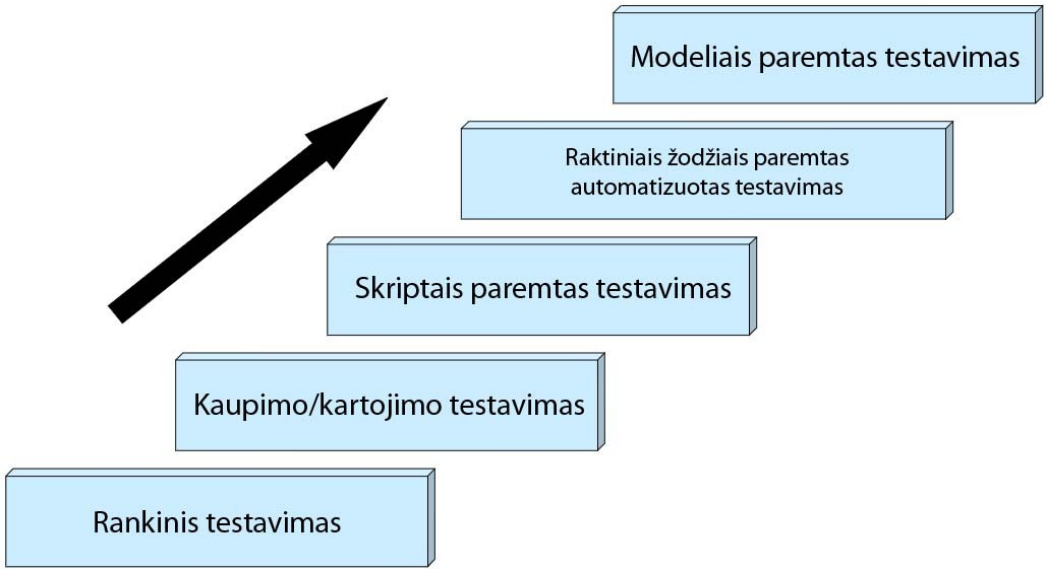
Testavimo procesas	Išspręstos problemos	Likusios problemos
Rankinis testavimas.	Funkcinis testavimas.	Netikslus testuojamos sistemos funkcionalumo padengimas; Neįmanomi regresijos testai; Labai brangus procesas (kiekvienas testas paleidžiamas rankiniu būdu); Nėra efektyvaus testų padengimo skaičiavimo.
Kaupimas/kartojimas.	Galima automatiškai paleisti įrašytus testus.	Netikslus testuojamos sistemos funkcionalumo padengimas; Beveik neįmanomi regresijos testai; Brangus procesas (kiekvienas pasikeitimas iššaukia testo rankinį įrašymą).
Skriptais paremtas testavimas.	Galimybė automatiškai ir iš naujo paleisti testų skriptus.	Netikslus testuojamos sistemos funkcionalumo padengimas; Sudėtinius skriptus sunku rašyti ir prižiūrėti; Reikalavimų perėjimas atliekamas rankiniu būdu.
Raktiniais žodžiais paremtas automatizuotas testavimas.	Lengvesnis skriptų valdymas.	Netikslus testuojamos sistemos funkcionalumo padengimas; Reikalavimų perėjimas atliekamas rankiniu būdu.

Modeliais paremtas testavimas išsprendžia tris pagrindines testavimo problemas:

- Funkcinių testų kūrimo automatizavimas, kuris sumažina kūrimo sąnaudas ir sukuria testų rinkinius, kurie nuosekliai padengia modelį.
- Testavimo rinkinių palaikymo sumažinimas.
- Automatinis perėjimų matricos kūrimas iš testų.

1.2. Modeliais paremtas testavimas

Modeliais paremtas testavimas (MPT) naudojamas kurti funkcinius testus. Kiek rečiau tvirtumo[1] ar vykdymo testavimus. Vartotojų sąsajoms jis retai naudojamas, nes modeliuojant gali atsirasti daug būsenų, kurios padarys testavimą neįmanomu arba nenaudingu. Testams gauti naudojamas tik juodos dėžės metodas.



2 pav. **Programinės įrangos testavimo raida [OP07]**

Ši disciplina yra pakankamai nauja, todėl skirtinguose šaltiniuose galima rasti skirtingą šio testavimo būdo apibūdinimą. Žemiau pateikiama keli MPT apibūdinimai.

Modeliais paremtas testavimas – juodos dėžės metodo funkcinio elgesio testavimas remiantis modeliu, kuris pateikiamas kaip formalus programos kodas, kuris atvaizduoja sistemos veikimą[Tre04].

Modeliais paremtas testavimas – juodos dėžės metodu sukurtų testų automatizavimas[UL07].

Modeliais paremtas testavimas – programinės įrangos testavimas, kurio metu testai iš dalies arba visiškai kuriami naudojant modelį, kuris apibūdina dalį arba visą testuojamą sistemą[OP07].

[1] Testavimo būdas, kai sistemai paduodama labai daug įvesčių ir tikrinama ar sistema nesustos, kokia bus sistemos reakcija.

1.3. Modeliais paremto testavimo procesas

Pagrindinis modeliais paremto testavimo skirtumas yra tas, kad automatizuojamas detalus testų rinkinių kūrimas ir perėjimų matrica. Vietoj rankiniu būdu rašomų testų, testuotojas sukuria abstraktų testuojamos sistemos modelį ir naudojantis modeliais paremtu testų įrankiu, kuria testų rinkinius iš to modelio. Testų kūrimo laikas sumažėja. Dar vienas privalumas, kad bet kas gali kurti testų iš to modelio nurodant norimus testavimo kriterijus.

Modeliais paremto testavimo procesas susideda iš 5 žingsnių:

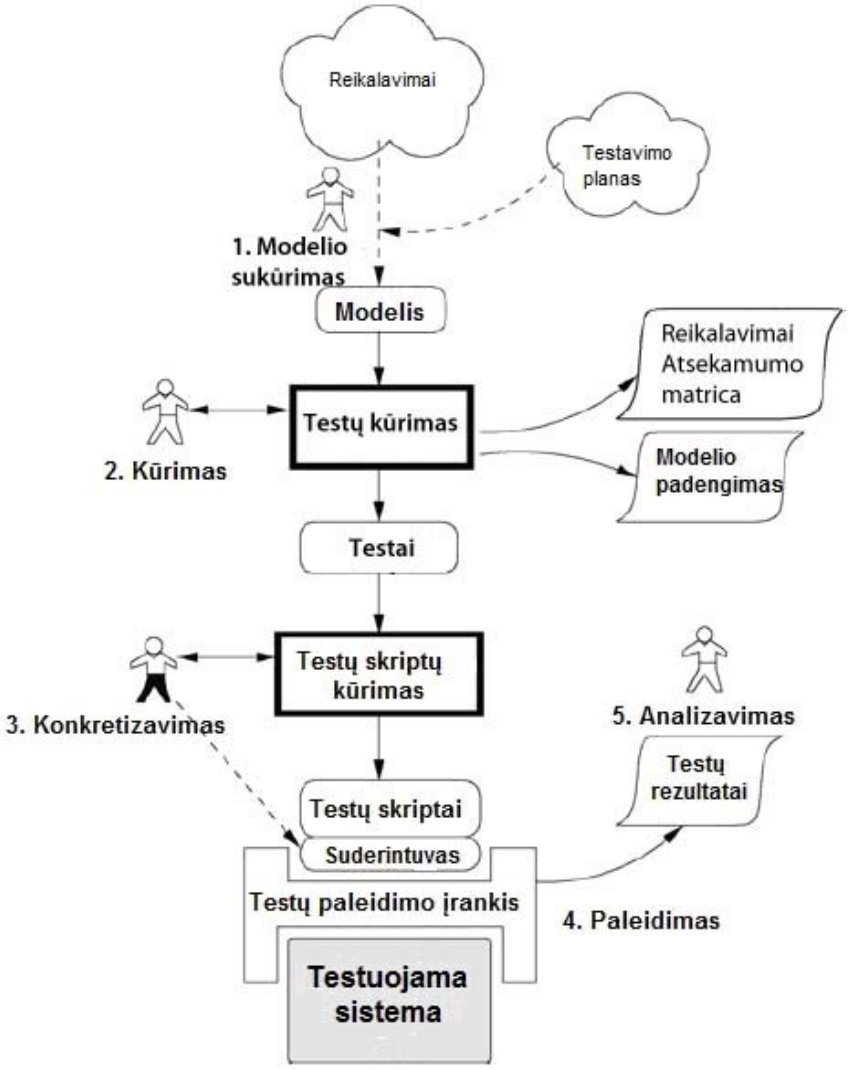
1. Testuojamos sistemos arba jos aplinkos *modelio sukūrimas*.
2. Abstrakčių testų *kūrimas* naudojantis modeliu.
3. Abstrakčių testų *konkretizavimas* padarant juos tinkamus paleisti.
4. Testų *paleidimas*.
5. Testų *rezultatų* analizė.

4 ir 5 testavimo proceso žingsniai yra kiekviename testavimo procese. 3 žingsnis panašus kaip ir raktažodžiais paremto testavimo suderinimo lygis. Pirmi du žingsniai yra unikalūs modeliais paremtame testavime. Juos galima rasti tik modeliais paremtame testavime. „Online“ modeliais paremtuose testavimo įrankiuose 2 ir 4 žingsniai dažniausiai sujungiami į vieną, o „Offline“ modeliais paremtame testavime jie atskiriami. Bet kuriuo atveju juos reikia suprasti atskirai, kad būtų aiškus visas procesas.

Pirmas žingsnis – sukurti abstraktų testuojamos sistemos modelį. Abstraktus todėl, kad ne visada jo veikimas bus lygiai toks pats kaip ir testuojamos sistemos. Gan dažnai ne itin svarbios sistemos dalys bus praleidžiamos arba supaprastintai pavaizduotas tam tikros posistemės veikimas.

Sumodeliavus sistemą patartina patikrinti gauto modelio elgesį. Šis veiksmas svarbus, nes modelis turi idealiai atvaizduoti testuojamą sistemą. Yra įvairių modelio patikrinimo įrankių. Didžioji dalis modelių patikrinimo įrankių yra komerciniai arba labai siauro panaudojimo.

Antras žingsnis – iš modelio sukurti abstrakčius testus. Naudojant testavimo kriterijus, pasakoma, kokius testus reikia kurti iš modelio, nes dažniausiai jų skaičius yra neribotas. Naudojantis testų kūrimo įrankiais, galima nurodyti, kad būtų pereitos visos modelio būsenos. Šie įrankiai taip pat saugo statistiką, kiek procentų modelio padengta ar kiek procentų pasirinkto kriterijaus/ų įvykdyta. Pagrindinis šio žingsnio tikslas – abstraktūs testų rinkiniai. Dauguma įrankių pateikia reikalavimų perėjimo matricas arba kitas padengimo ataskaitas. Reikalavimų perėjimo matrica susieja funkcinius reikalavimus ir sukurtus testus. Padengimo ataskaitos pateikia informaciją, ar gerai vykdomi testai.



3 pav. **Modeliais paremto testavimo procesas [UL07]**

Padengimo ataskaitas galima naudoti tikrinant, ar geri sukurti testai, arba nustatyti, kuri modelio dalis nebuvo patikrinta ir išanalizuoti, kodėl.

Trečias žingsnis – paversti abstrakčius testus į tinkančius panaudoti tekstus. Naudojant įvairius transformacijos įrankius, kurie naudoja šablonus ir žymėjimus, abstraktūs testai paverčiami į testų skriptus. Pagrindinis tikslas – padaryti testus tinkamus testuoti norimą sistemą.

Ketvirtas žingsnis – paleidžiami testai ant testuojamos sistemos. Naudojant „Online“ modeliais paremtą testavimą, testai paleidžiami tik juos sukūrus, todėl testavimo įrankis iškart įrašo ir pateikia gautus rezultatus. „Offline“ modeliais paremto testavimo metu sukuriami testavimo skriptai, kuriuos galima paleisti, kada norima ar reikia.

Penktas žingsnis – gautų rezultatų analizė ir teisingo sprendimo parinkimas. Kiekvienas nesėkmingas testas turi būti patikrintas ir nustatyta jo nesėkmės priežastis. Procesas yra lygiai toks pat, kaip naudojant kitus testavimo būdus. Reikia patikrinti, ar sistema veikia blogai, ar blogas

testas. Modeliais paremtame testavime testo klaida gali būti blogai pritaikius kodą arba modelyje (ar reikalavimuose).

Pradžioje klaidos dažniausiai būna suderinant kodą. Ištaisius šias, apie pusę klaidų randama testuojamoje sistemoje, kita pusė modelyje arba reikalavimuose. Priklausomai nuo testuotojo patirties, sistemos sudėtingumo ir reikalavimų, klaidų skaičius gali kisti.

1.4. Modeliais paremto testavimo privalumai

Modeliais paremto testavimo privalumus galima suskirstyti į šešias dalis: klaidų suradimas, testavimo laiko ir kainos sumažinimas, testavimo kokybės pagerinimas, klaidų suradimas reikalavimuose, atsekamumas ir reikalavimų kitimas[OP07][UL07].

1.4.1. Klaidų suradimas

Pagrindinis testavimo tikslas – testuojamos sistemos klaidų suradimas. IBM ir BMW tyrimuose modeliais paremtas testavimas surado tokius pat klaidų kiekius, kaip ir rankinio testavimo metu. Kompanijos „Microsoft“ atliktuose tyrimuose su savo taikomosiomis programomis surastų klaidų skaičius buvo 10 kartų didesnis[BGL03]. Lustines mokėjimo korteles testuojant modeliais paremtas testavimo būdas naudojamas kiekvieną dieną[UL07].

Modeliais paremtas testavimas jokiais būdais neišsprendžia visų problemų. Klaidų suradimas didžiąja dalimi priklauso nuo testuotojo sugebėjimų ir patirties. Viskas labai susiję su modeliu ir pasirinktais testavimo kriterijais. Didelę reikšmę turi įrankio pasirinkimas. Nemažai tyrimų įrodė, kad modeliais paremtas testavimas randa (tam tikrose srityse) daugiau klaidų[UL07][HN02][AG02].

1.4.2. Testavimo laiko ir kainos sumažinimas

Naudojant modeliais paremtą testavimą, testavimo laiko ir kainos sąnaudos yra sumažinamos tada, kai laikas reikalingas sukurti modelį ir testų generavimas yra mažesni, nei laikas rankiniu būdu kuriant testus.

Įvairūs tyrimai parodė, kad ne tik laiko sąnaudos, bet ir sukurtų testų kiekis pralenkia rankinį testavimą[UL07]. Testavimo laikas gali išaugti dėl testavimo įrankių sudėtingumo[FM00].

Laikas sutaupomas ne tik kuriant testus, bet ir jų analizės metu. Testai yra kuriami nuosekliai automatiniu būdu, tuomet paprasčiau surasti klaidos nuoseklumą. Antra, kai kurie modeliais paremto testavimo įrankiai sugeba sukurti arba surasti trumpiausią testų seką, dėl kurios atsiranda

klaidos. Trečia, testui nepavykus, galima tikrinti ne tik testo kodą, bet ir, kokius veiksmus atliekant, atsirado klaida. Kai kuriais įrankiais galima peržiūrėt testo elgesį einant per modelį. Jei įrankis palaiko reikalavimų perėjimus, tai gaunama informacija, kurie reikalavimai yra patikrinti. Ši papildoma informacija gali ženkliai sutrumpinti testavimo laiką.

1.4.3. Testavimo kokybės pagerinimas

Rankinio testavimo metu testų kokybė visiškai priklauso nuo testuotojo genialumo. Kūrimo procesas neatkartojamas ir testo loginis išdėstymas nėra įrašomas. Testus reikia susieti su sistemos reikalavimais.

Modeliais paremtas testavimas dalį šių problemų išsprendžia. Testų automatiškas kūrimas iš modelio, naudojant tam tikrus algoritmus, yra sistemingas ir atkartojamas. Atsiranda galimybė apskaičiuoti testavimo rinkinių kokybę. Modeliais paremto testavimo metu sukuriama žymiai daugiau testų. Nežymiai keičiant testų parinkimo kriterijus, galima sukurti nemažus testų rinkinius. Testų duomenys ir logika gaunama iš modelio.

1.4.4. Klaidų suradimas reikalavimuose

Vienas iš modeliais paremto testavimo privalumų – klaidų radimas reikalavimuose. Reikalavimai užrašomi žmogui suprantama kalba didžiuliuose dokumentuose, kuriuose gali būti daug prieštaravimų, aplaidumo ir neaiškių reikalavimų. Pirmas modeliais paremto testavimo žingsnis yra testuojamos sistemos veikimo išsiaiškinimas, kad būtų įmanoma sukurti abstraktų testuojamos sistemos modelį. Kuriant sistemos modelį, iškyla įvairių klausimų, kurie atskleidžia įvairias reikalavimų problemas. Modelio kūrimą galima traktuoti, kaip mažo, abstraktaus testuojamos sistemos prototipo kūrimą. Prototipavimas yra puikus būdas surasti reikalavimų klaidas[Bur02].

Dalis nepraėjusių testų atsiranda dėl klaidingo modelio. Įvairių bandymų metu nustatyta, kad pusė gautų klaidų yra dėl modelio arba reikalavimų.

Reikalavimų klaidų suradimas išsprendžia nemažas sistemos problemas. Didžioji dalis klaidų kyla dėl prastų reikalavimų.

Šiuo atveju modeliais paremtas testavimas turi didžiausią įtaką kuriant programinę įrangą. Ankstyvas modeliavimas randa reikalavimų ir projektavimo klaidas, nes modelis pakankamai tikslus. Klaidos randamos pačioje sistemos kūrimo pradžioje. Tokiu būdu klaidos kaina sumažinama iki minimumo.

1.4.5. Atsekamumas

Atsekamumas – galimybė susieti testus su modeliu, testų pasirinkimo kriterijais ir neoficialiais sistemos reikalavimais.

Atsekamumas nusako testus ir, kokia logika vadovaujantis, jis buvo sukurtas. Atsekamumas leidžia optimizuoti testų paleidimus, nes atsiranda galimybė paleidinėti tik tuos testus, kuriuos pakeitė modelio atnaujinimas. Atsekamumą galima naudoti kaip testavimo kriterijų.

Testų kūrimo algoritmai turi pažymėti, kuri modelio dalis naudota kuriant testus ir pateikti informaciją suprantama forma. T. y. sudaryti sąryšį tarp testų ir modelio. Naudojant UML būsenų diagramą, testų kūrimo algoritmas įrašys, kuriuos modelio perėjimus panaudoja testai. Taip atsiranda galimybės:

- Surasti visus testus, kurie pereina per tam tikrą modelio perėjimą.
- Pavaizduoti testą modelyje.

Reikalavimų ir modelio atsekamumas yra žymiai sudėtingesnė užduotis. Problema sprendžiama sudarius ryšius tarp modelio ir reikalavimų. Tokiu būdu galimos tokios užduotys:

- kurie reikalavimai neįtraukti į modelį?
- Kaip reikalavimai keičia modelį?

Paskutinis atsekamumas tarp reikalavimų ir testų. Atsekamumas tarp neoficialių reikalavimų ir sukurtų testų. Galima pavaizduoti perėjimų matrica. Tokiu būdu galimos tokios užduotys:

- nustatyti, kurie reikalavimai nepatikrinti.
- Pateikti visus testus, kurie susiję su pateiktais reikalavimai.
- Pateikti reikalavimus, kurie susiję su tam tikrais testai.

Atsekamumas naudojamas testų kokybės nustatymams, kaip testavimo kriterijus kuriant testus automatiniu būdu arba kaip testų sumažinimo mechanizmas. Visos galimybės visiškai priklauso nuo naudojamo testavimo įrankio.

1.4.6. Reikalavimų pasikeitimai

Naudojant rankinį testavimo būdą, pasikeitus reikalavimams, atsiranda poreikis kurti testus iš naujo. Privaloma atnaujinti testus, kad testai atitiktų reikalavimus.

Naudojant modeliais paremtą testavimą tereikia tik atnaujinti modelį ir kurti testus iš naujo automatiniu būdu. Modelis dažniausiai yra mažesnis nei testas, todėl laikas sugaištas taisant modelį yra žymiai trumpesnis, nei visus testus. Tokiu būdu greičiau reaguojama į reikalavimų pasikeitimus.

Kai kurie modeliais paremto testavimo įrankiai turi galimybę atskirti, kurie testai yra nauji, pasikeitę ar nebuvo pakeisti po modelio pasikeitimo. Tokiu būdu sumažinamas testų tikrinimo laikas.

1.5. Modeliais paremto testavimo trūkumai ir sunkumai

Niekada negalima garantuoti, kad bus surasti visi modelio ir programos skirtumai. Bet toks trūkumas galioja visiems testavimo būdams.

Vienas pagrindinių modeliais paremto testavimo trūkumų – šio testavimo būdo pritaikymo galimybės, nes reikalingos visiškai kitokios žinios. Modelio kūrėjas turi sumąstyti ir sukurti modelius ir tuo pačiu būti taikomųjų programų ekspertas. Tai pareikalauja papildomų mokymų.

Modeliais paremtas testavimas dažniausiai naudojamas funkciniams testams kurti. Ne visus testus galima lengvai automatizuoti.

Būtina suprasti, kad be testavimo automatizavimo patirties modeliais paremtas testavimas negali būti pradėtas. Galima išskirti kelias problemas, su kuriomis susiduriama pradedant modeliais paremtą testavimą:

Pasenę reikalavimai. Vykdant programinės įrangos projektą, kartais atsitinka, kad oficialūs reikalavimai keičiasi. Sukurtas modelis nebebus tikslus ir sukurti testai praneš tik apie testuojamos sistemos klaidas.

Neteisingas modeliais paremto testavimo panaudojimas. Kai kurios testuojamos sistemos dalys gali būti sunkiai modeliuojamos ir visada turi būti apsvarstyta ar nelengviau būtų pasirinkti kitą testavimo būdą. Testuotojas turi turėti pakankamai patirties, kad suprastų, kurias testuojamos sistemos dalis modeliuoti, o kurias tikrinti rankiniu būdu.

Nepavykusių testų analizė. Nepavykus testui, būtina nuspręsti, kurioje vietoje įvyko klaida: testuojamoje sistemoje, adaptuotame kode ar modelyje. Modeliais paremto testavimo testų eilės būna ganėtinai sudėtingos, dėl to atsiranda papildomos problemos juos analizuojant ir ieškant atsiradusios klaidos priežasties.

Modelio dydis. Modeliuojat dideles sistemas, atsiranda būsenų kiekio eksponentinis didėjimas. Net ir mažos taikomosios programos modelis gali turėti didelius kiekius būsenų.

Modeliais paremtas testavimas nėra pats geriausias testavimo būdas. Jis nėra labai plačiai naudojamas, todėl susiduriama su šiam būdui pritaikytų įrankių trūkumu.

1.6. Modeliais paremto testavimo įrankiai

Žemiau pateikta keletas modeliais paremtam testavimui sukurtų įrankių su trumpais jų aprašymais. Keli iš jų bus naudojami šio darbo tikslams pasiekti. Didžioji dalis yra komerciniai. Dalis teikiami moksliniams tikslams nemokamai.

Leirios test generator – testai kuriami iš UML modelių. Neturi vidinio modeliavimo įrankio, bet palaiko trečių šalių modeliavimo įrankius: *Borland Together* arba *IBM Modeler*. Palaiko reikalavimų atsekamumą. Automatiškai tikrina testus su modeliu. Įrankis mokamas.

MaTeLo – testai kuriami iš testuojamos sistemos modelio. Turi vidinį modeliavimo įrankį. Palaiko trečių šalių modeliavimo įrankius. Modelis kuriamas rankiniu būdu remiantis Markovo grandinėlių teorija. Modeliui padengti naudojami 2 kriterijai: ribinės reikšmės, būsenų ir perėjimų padengimas. Modelio perėjimams naudoja labiausiai tikėtino kelio algoritmą ir atsitiktinių perėjimų algoritmą. Įrankis mokamas.

Qtronic – testai automatiškai kuriami iš sistemos modelio. Turi vidinį modeliavimo įrankį, bet yra galimybė importuoti UML modelius. Palaiko 9 modelio padengimo kriterijus ir 3 modelio perėjimo algoritmus. Testai eksportuojami HTML formatu. Įrankis mokamas.

Reactis – testus kuria iš *Simulink*® ir *Stateflow*® modelių. Neturi vidinio modeliavimo įrankio. Modelio perėjimams naudojamas atsitiktinis perėjimo būdas, palaiko 10 modelio padengimo kriterijų. Testai eksportuojami į *Matlab* formatą. Įrankis mokamas.

SpecExplorer – *VisualStudio* priedėlis naudojamas testams kurti. Modeliai kuriami naudojantis *Spec#* arba *AsmL*. Modelio perėjimams naudojamas trumpiausio kelio radimo algoritmas ir atsitiktinio perėjimo algoritmas. Modeliui padengti naudojami perėjimų padengimo kriterijai. Testai rašomi XML formatu. Įrankis nemokamas.

GraphWalker – testai kuriami iš importuotų modelių, kurie kuriami naudojant *graphML*. Neturi vidinio modeliavimo įrankio. Atviro kodo projektas.

ModelJUnit – Java biblioteka, kuri *JUnit* pritaiko modeliais paremtam testavimui. Modeliai kuriami Java programavimo kalba. Modeliai atvaizduojami kaip išplėsti baigtiniai automatai. Testų paleidimui automatizuoti naudojama Java programavimo kalba. Tai atviro kodo projektas.

NModel – modeliais paremto testavimo ir analizės karkasas modelius aprašantis c#. Modelis atvaizduojamas kaip išplėstinis baigtinis automatas. Modelis tikrinamas rankiniu būdu. Testų paleidimui naudojama c# programavimo kalba. Yra galimybė apsirašyti savo kriterijus. Tai atviro kodo projektas.

TestOptimal – visai palaiko visą modeliais paremtą procesą. Turi vidinį modeliavimo įrankį. Palaiko 5 skirtingus testavimo sekų kūrimo algoritmus. Modelio perėjimams naudojamas trumpiausio kelio radimo algoritmas ir atsitiktinio perėjimo algoritmas. Skirtas automatiniams interneto aplikacijų ir Java kalba rašytų aplikacijų testavimams. Papildomais priedais galima testuoti ir kitas programas. Pusiau komercinis įrankis.

GOTCHA-TCBeans – testai kuriami iš baigtinio automato. Java klasės naudojamos testams kurti. Modeliui kurti naudoja GDL. Yra galimybė nustatyti testavimo scenarijų. Įrankis komercinis. Neteikiama bandomoji versija.

AGEDIS – testai kuriami iš UML. Turi vidinį modeliavimo įrankį, sukurtų testų analizavimo įrankį, ataskaitų įrankį. Modeliui kurti naudojama AML modeliavimo kalba. Turi vidinę skriptų programavimo kalbą IF. Modelio perėjimui naudoja kelis grafų perėjimo algoritmus. Įrankis pusiau komercinis.

2. Esamų įrankių analizė

Skyriuje pateikiama esamų įrankių analizė. Testuoti pasirinkta internetinė taikomoji programa – Virtualios parodos, kurią galima rasti šiuo adresu: http://www.muziejai.lt/emuziejai/. Testuoti naudojama kiekvienam vartotojui matoma aplinka ir administravimo sąsaja naudojama informacijai įkelti ir tvarkyti. Sukuriami du skirtingo sudėtingumo modeliai. Nepriklausomai nuo to, kad tirti naudojama internetinė taikomoji programa, testavimo specifika nesikeičia ir visas dėmesys skiriamas esamų įrankių galimybių analizei ir tyrimui.

Modeliais paremto testavimo procesas susideda iš 5 dalių: modelio kūrimas, testų kūrimas, konkretizavimas, paleidimas ir analizė (žiūrėti 3 pav.). Tokia struktūra pasižymi beveik kiekvienas modeliais paremto testavimo įrankis. Priklausomai nuo įrankio, kai kurios dalys gali būti atliekamos šalutiniais įrankiais.

2.1. Modeliavimas

Testuojamos sistemos modelis turi būti kuo paprastesnis ir naudingesnis. Abstrakcijos lygis privalo būti kuo aukštesnis, kad būtų įmanoma kurti naudingus testus. Prieš sistemos modeliavimą reikia nusistatyti gerą abstrakcijos lygį; kurias testuojamos sistemos dalis įtraukti į modelį. Reikia apsispręsti, kurios dalys yra svarbios ir vertos modeliuoti. Keli maži sistemos dalių modeliai yra žymiai naudingesni nei vienas didelis visos sistemos modelis. Galima modeliuoti tik sistemos posistemius ir juos testuoti nepriklausomai nuo visos sistemos. Viskas labai priklauso nuo testavimo plano ir tikslų.

Kitas žingsnis – apsirašyti visus testuoti reikalingus duomenis ir operacijas, kurias reiks atlikti, ir kitas sistemos dalis, su kuriomis modeliuojama dalis sąveikaus. Dažnai naudojamos UML klasių diagramos, apibūdinančios sistemos veikimą. Modeliuoti reikalinga kuo paprastesnė informacija. Abstrakcijos lygis privalo būti kuo aukštesnis, kad, pasikeitus reikalavimams, kuo mažiau modelio reiktų keisti.

Abstrakcijos principas taikomas ir modelyje naudojamoms įvestims ir išvestims. Jei įvesties reikšmė nekeičia sistemos būsenos, tai jos neverta įtraukti. Modelis kuriamas atsižvelgiant į testavimo tikslus. Abejotinos įvestys ir išvestys praleidžiamos.

Svarbus žingsnis – modeliavimo sistemos (*angl. Notations of modeling*) pasirinkimas. Koks modelis bus naudojamas, dažniausiai apsprendžia testavimo įrankis. Didžioji dalis įrankių palaiko tik vieną modelio tipą. Sistemos veikimui modeliuoti naudojama begalės modeliavimo sistemų. Jas galima skirstyti į kelias dalis: būsenomis paremtos sistemos (UML OCL, JML, Spec#, VDM),

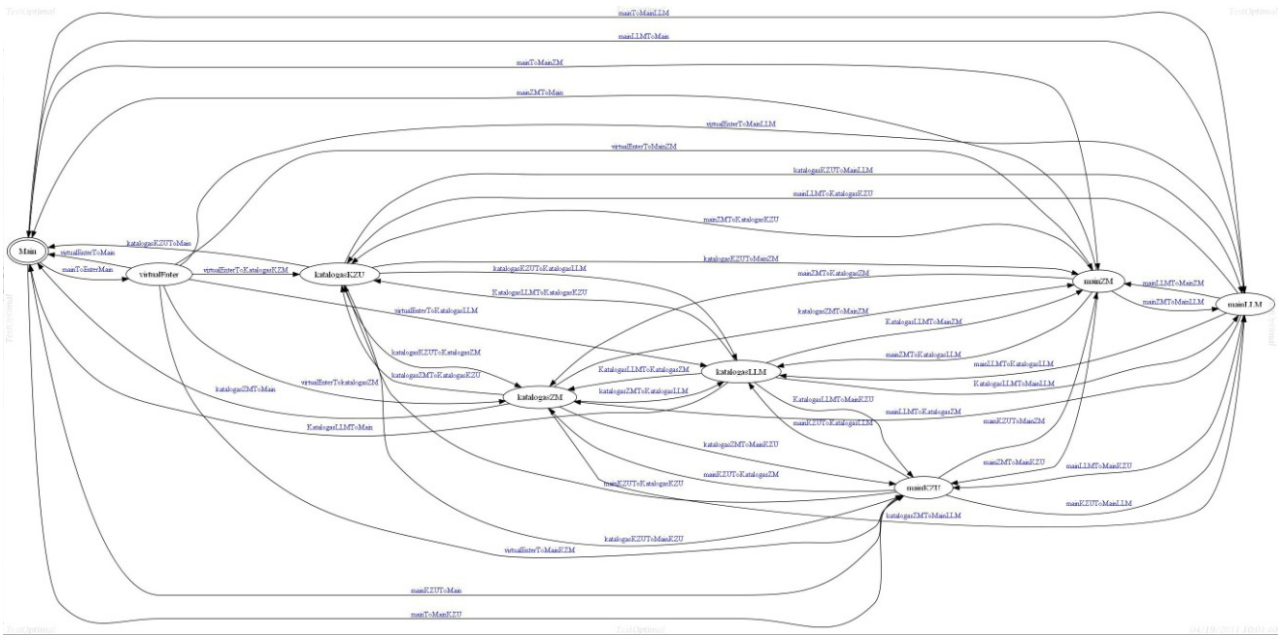
perėjimais paremtos sistemos (baigtiniai automatai), funkcinės sistemos (matematinės funkcijos), statistinės modeliavimo sistemos (Markovo grandinės), Duomenų sekomis paremtos sistemos.

Modeliais paremtame testavime sistemos elgesį lengviausia aprašyti būsenų perėjimo sistemomis ir perėjimais paremtomis sistemomis. Kurią sistemą pasirinkti, priklauso nuo testuojamos sistemos. Pavyzdžiui, *SpecExplorer* savyje turi modeliavimo sistemą Spec#, o *ModelJUnit* naudoja JML (*Java Modeling Language*).

2.1.1. Modeliai

Pateiksime tris dažniausiai naudojamus modelius, su kuriais dažniausiai susiduriama ir yra populiariausi. Labiausiai naudojami modeliai yra baigtinis automatas (*angl. Finite state machines*), vieninga modeliavimo kalba (*angl. Unified Modeling Language*), Markovo grandinė (*angl. Markov chains*).

Baigtinis automatas – elgsenos modelis, susidedantis iš būsenų ir perėjimų. Perėjimai sujungia skirtingas būsenas. 4 pav. pateiktas testuojamos sistemos baigtinis automatas, sukurtas *TestOptimal* įrankio pagalba.



4 pav. **Vartotojo aplinka, sumodeliuota *TestOptimal***

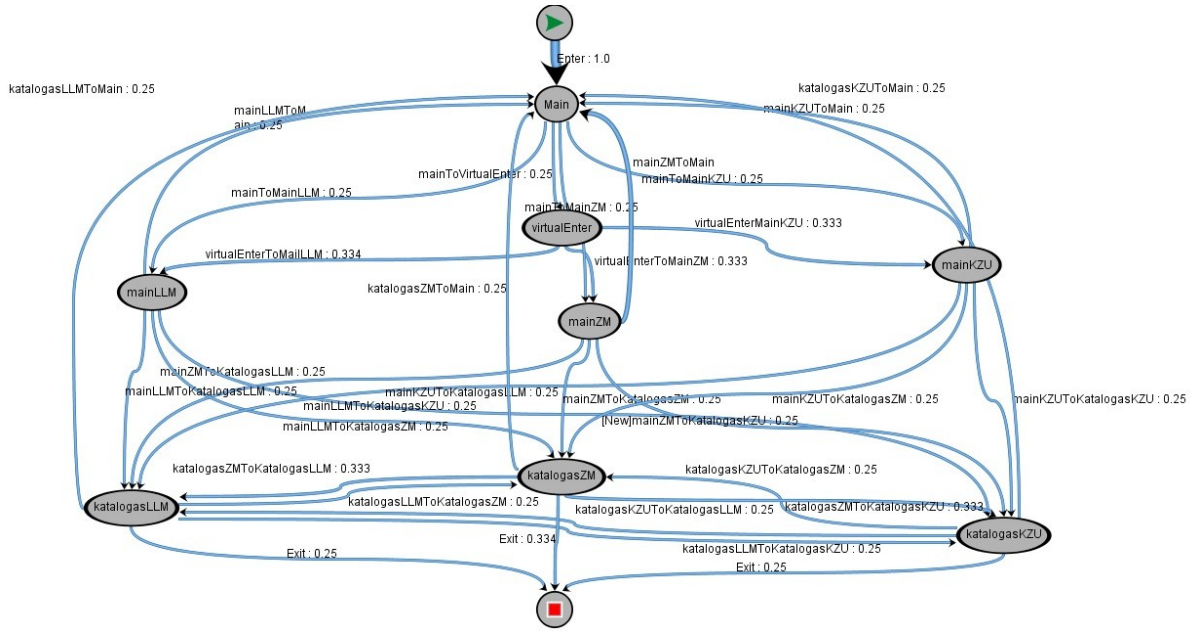
Baigtinis automatas susideda iš penkių dalių (*I, S, T, F, L*):

- *I* - įvesčių rinkinys testuojamai sistemai (arba įvesčių seka).
- *S* - testuojamos sistemos visos būsenos.
- *T* - funkcija, kuri nusako, koks bus perėjimas iš tam tikros būsenos, kai pritaikoma įvestis.

- *F* - galutinė būsena. Būsena, kurioje baigtinis automatas baigs darbą.
- *L* - pradinė būsena. Būsena, kurioje baigtinis automatas pradeda darbą.

Baigtinis automatas vienu metu gali būti tik vienoje būsenoje. Perėjimas iš vienos būsenos į kitą priklauso nuo įvesčių *I*[AOA02].

Markovo grandinės – elgsenos modelis, susidedantis iš būsenų ir perėjimų. Perėjimai sujungia skirtingas būsenas. Kiekvienas perėjimas turi tikimybę, kuria renkamasi, į kurią būseną bus einama kito žingsnio metu. Tikimybėmis galima apriboti tam tikrus perėjimus suteikiant jiems mažesnę tikimybę. Atsiranda galimybė tikrinti programinės įrangos patikimumą.



5 pav. **Vartotojo aplinka sumodeliuota su *MaTeLo – Usage Model Editor***

UML (*angl. Unified Modeling Language*) – vieninga modeliavimo kalba (UML) yra plačiai naudojama modeliais paremtame testavime. UML labai panaši į baigtinius automatus ir naudojama sudėtingam programinės įrangos elgesiui aprašyti. Žemiau pateikiamos UML diagramos, kurios yra naudojamos modeliais paremtame testavime.

1 lentelė. **UML diagramos ir jų panaudojimas modeliais paremtame testavime[UL07]**

Diagrama	Panaudojimas modeliais paremtame testavime
Klasių diagrama	Puikiai tinka apibūdinti statiškai sistemos daliai, duomenims ir sąsajoms tarp klasių.
Objektų diagrama	Tinka pradinių modelio būsenų apibūdinimui kuriant testus.
Panaudos atvejų diagrama	Nusako sistemos reikalavimus, bet nesuteikia pakankamai informacijos testų kūrimui. Galima naudoti kaip testavimo kriterijų

	arba nusakyti testavimo tikslą.
Baiginių automatų diagrama	Beveik identiška baigtiniams automatams, todėl puikiai tinka sistemos testavime.
Veiklos diagrama	Tinka sistemos procesų arba darbo eigos modeliavimui. Galima kurti testus.
Sekų diagrama	Puikiai tinka apibūdinti abstraktiems testams. Galima naudoti kaip testų parinkimo priemonę, kuria nurodomas norimas kelias per modelį.
Bendradarbiavimo diagrama	Toks pats panaudojimas kaip ir sekų diagramose.

2.1.2. Modeliavimo įrankiai

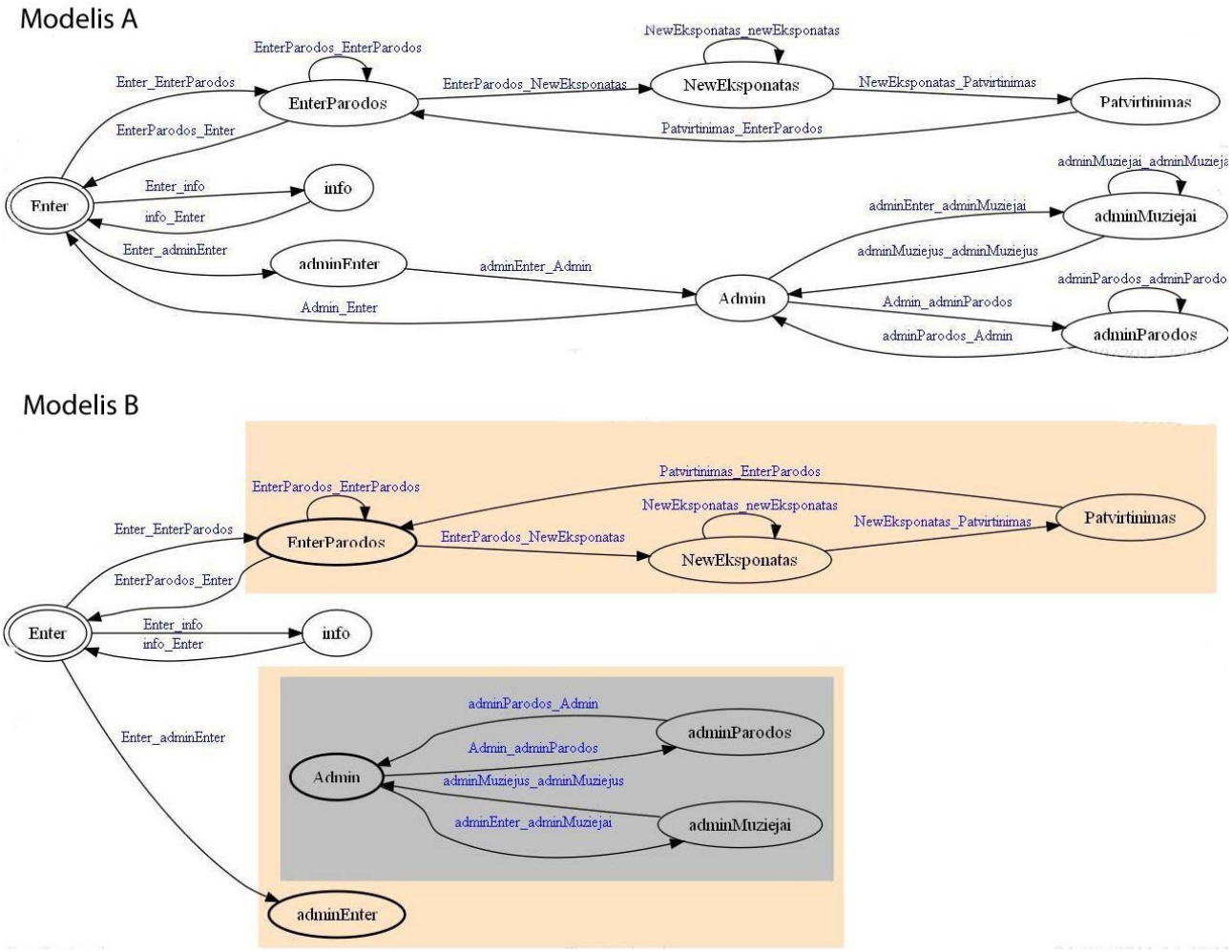
Modelis yra viena pagrindinių modeliais paremto testavimo dalių. Juo remiantis kuriami visi testai. Yra begalės modeliavimo įrankių, skirtų sistemos elgesiui apibūdinti. Šiuo atveju mus domina tik tie modeliavimo įrankiai, kurie tiesiogiai naudojami modeliais paremtame testavime. Kai kurie testavimo įrankiai pateikiami su modeliavimo įrankiu, kiti naudoja esamus modeliavimo įrankius. Iškyla klausimas, kokiais kriterijais remiantis galima lyginti modeliavimo įrankius?

Galima išskirti kelis kriterijus, kuriais remiantis galima analizuoti MPT modeliavimo įrankius: modelio kūrimas, sub-modeliavimas (minimodelis), modelio tikrinimas ir reikalavimų atsekamumas.

Modelio kūrimas – kriterijus nusakantis MPT įrankio savybę modelį kurti pačiam (turi vidinį modeliavimo įrankį) arba trečių šalių modelio importavimas. Sub-modeliavimas – galimybė modelį suskaidyti į dalis, taip sumažinant jo sudėtingumą (žiūrėti 6 pav.). Modelio tikrinimas – prieš kuriant testus, galimybė modelius patikrinti (pavyzdžiui, patikrinti ar nėra nepasiekiamų būsenų arba perėjimų). Reikalavimų atsekamumas – galimybė modelį susieti su reikalavimais ir testais. Atsekamumo privalumai išdėstyti 1.3.5. skyriuje.

Šias veiklas MPT įrankiai patys palaiko arba naudoja trečių šalių įrankius. Bet kuriuo atveju, visos šios veiklos (išskyrus modelio kūrimą ir sub-modeliavimą, nes visuose įrankiuose modelio kūrimas vyksta rankiniu būdu ir jo visas automatizavimas nėra įmanomas) turi būti kuo labiau automatizuotos. Didelė dalis įrankių palaiko modelių importavimą. Šiuo atveju gali iškilti suderinamumo problema, kurią tenka spręsti rankiniu būdu.

2 lentelėje pateiktas kelių MPT įrankių palyginimas remiantis aukščiau išvardintais ir dar keliais papildomais kriterijais.



6 pav. **Modelis A ir modelis su sub-modeliais B**

Iš lentelės matyti, kad didžioji dalis MPT įrankių naudoja išorinius modeliavimo įrankius. Tik keturi čia paminėti įrankiai turi vidinius modeliavimo įrankius. *SpecExplorer*, *AGEDIS* ir *Test Designer* turi integruotą vientisą išorinį modeliavimo įrankį. Tai reiškia, kad sukurtam modeliui importuoti nereikia papildomo testuotojo įsikišimo, o kiti paminėti įrankiai tiesiog importuoja modelius. *GOTCHA-TCBeans* reikalauja, kad importuojamas modelis būtų paverstas į skriptą. *SpecExplorer* išplečia *VisualStudio*, kad būtų galimybė kurti modelius. *ModelJUnit* modelius užrašo Java programavimo kalba (8 pav.), yra galimybe modelį pavaizduoti grafiškai.

Daugiau nei pusė lentelėje pateiktų įrankių palaiko sub-modeliavimą. Tai labai svarbus kriterijus, į kurį būtina atsižvelgti, jei modeliuoti naudojamas baigtinis automatas. 6 pav. pateikta sistemos dalis sumodeliuota naudojant paprastą modeliavimo būdą (modelis A) ir sub-modelius (modelis B). Modelis B padalintas į papildomas dvi dalis. Testų kūrimo metu atsiranda galimybė testus kurti tik iš sub-modelio. Kitas sprendimo būdas – modeliuojant testuojamą sistemą uždėti

griežtus modelio perėjimo saugiklius arba žymėti norimas būsenas ir tik iš jų kurti testus. Viskas priklauso nuo testuotojo profesionalumo, nes didžioji dalis įrankių palaiko tiek saugiklius, tiek būsenų žymėjimą.

2 lentelė. **MPT modeliavimo įrankių palyginimas**

Įrankio pavadinimas	Modelio kūrimas	Modelio tipas	Sub-modeliavimas	Modelio tikrinimas	Reikalavimų atsekamumas
MaTeLo	Vidinis	MG[2]	Palaiko	Palaiko	Iš dalies palaiko
Qtronic	Vidinis	UML	Nepalaiko	Palaiko	Iš dalies palaiko
SpecExplorer	Išorinis	BA[3]	Palaiko	Palaiko	Iš dalies palaiko
GraphWalker	Išorinis	BA/IBA[4]	Palaiko	Nepalaiko	Iš dalies palaiko
TestOptimal	Vidinis	BA	Palaiko	Palaiko	Nepalaiko
GOTCHA-TCBeans	Išorinis	BA	Nepalaiko	Nepalaiko	Nepalaiko
AGEDIS	Išorinis	UML(AML)	Nepalaiko	Nepalaiko	Nepalaiko
Test Designer	Išorinis	UML	Palaiko	Palaiko	Palaiko
ModelJUnit	Vidinis	BA	Palaiko	Palaiko	Nepalaiko

Modelio tikrinimą ne visi įrankiai palaiko. Esant dideliam modeliui gali atsirasti nepasiekiamų būsenų arba perėjimų. Žinoma, modeliuojant visada vertėtų apsiriboti testuojamos sistemos dalių modeliavimu, o ne visa sistema. Kai turimas modelis nėra didelis, užtenka modelio grafinio vaizdo, kad jį patikrinti ar peržiūrėti rankiniu būdu.

Dėl reikalavimų susiejimo įrankio (*DOORS*) tik *Test Designer* palaiko visišką reikalavimų atsekamumą. *MaTeLo*, *Qtronic*, *GraphWalker* ir *SpecExplorer* palaiko reikalavimų padengiamumą, bet tik iš dalies atsekamumą. *ModelJUnit* yra galimybė pačiam testuotojui apsirašyti norimus susiejimus, bet tiesioginio susiejimo nepalaiko.

Čia matomi tik bendri MPT įrankio modeliavimo kriterijai ir vien jais remiantis negalima vertinti įrankio kokybės. Kitame žingsnyje modelis naudojamas testams kurti. Kokie testai bus kuriami, priklauso nuo kitų kriterijų, kurie nusako MPT įrankio testų kūrimo ypatybes.

2.2. Testų kūrimas

Turint modelį, galima pradėti kurti abstrakčius testus. Viena modeliais paremto testavimo savybių – testai kuriami pereinant modelį automatiniu būdu. Dauguma įrankių suteikia galimybę testus kurti ir rankiniu būdu.

[2] Markovo grandinėles modelis.
[3] Baigtinis automatas.
[4] Išplėstas baigtinis automatas.

Kokiu būdu bus pereinamas modelis ir kuriami testai, priklauso nuo testų kūrimo kriterijų. MPT galimybių tyrimuose yra apibrėžta įvairių testų kūrimo kriterijų [UL07][PJ08], bet ne visus juos palaiko testavimo įrankiai. Didelė dalis kriterijų yra pasiskolinti iš kitų testavimo būdų. Kuo daugiau testų kūrimo kriterijų palaiko įrankis, tuo jis pranašesnis. Tokiu būdu testavimo kriterijus galima tiesiogiai naudoti, kaip MPT įrankio vertinimo kriterijų.

2.2.1. Testavimo kriterijai

Testavimo kriterijai – užduotys ir taisyklės, kuriomis remiantis kuriami testų rinkiniai [UL07]. Taikant testavimo kriterijus, testavimo įrankiams nurodoma, kokius testus reikia kurti. Modeliais paremtame testavime naudojami kriterijai, susiję su modeliais ir reikalavimais. Naudojantis kai kuriais kriterijais, galima paskaičiuoti testų padengimą testuojamai sistemai. Modeliais paremtame testavime naudojami kriterijai netikrina testuojamos sistemos kodo, todėl juos galima naudoti įvairiais būdais. Aukščiau pateiktame modeliais paremto testavimo procese testavimo kriterijai yra reikalingi antrame žingsnyje. Naudojantis testavimo įrankiais, nurodoma, kokius kriterijus naudoti.

Kriterijų pasirinkimas nurodo algoritmus, kuriuos testavimo įrankiai naudos kurdami testų rinkinius, taip pat tai, kokie testai bus kuriami, kaip ilgai bus vykdomas testų kūrimas ir kurią modelio dalį reikės testuoti. Tai vadinama testavimo kriterijumi, nes juo kontroliuojamas testų rinkinių kūrimas[HKO06].

Vykdant testavimą, patartina vesti sistemos būsenų ir perėjimų statistiką [HKO06]. Remiantis šia statistika, galima pasakyti, ar gerai atliktas sistemos testavimas. Dauguma testavimo kriterijų yra pasiskolinti iš kitų testavimo stilių.

Testavimo kriterijai naudojami dviem pagrindiniams tikslams:

- apskaičiuoti testų rinkinių atitikimą.
- nustatyti, kada testavimą galima sustabdyti.

Naudojant testavimo kriterijus, nereikėtų pervertinti jų galimybių. Ne visada MPT įrankiai sugebės 100 procentų patikrinti sistemą. Tam tikros būsenos dėl modeliavimo klaidų ar griežtų saugiklių gali būti nepasiekiamos. Gali būti, kad automatinių testų kūrimo algoritmas nesugebės surasti reikalingo kelio. Dideliam modeliui patikrinti gali prireikti nesuskaičiuojamos daugybės testų, kuriems paleisti nebus laiko. Testavimo kriterijai suteikia sprendimus tokioms problemoms. Labai retai testuojama sistema patikrinama 100 procentų.

Remiantis [UL07], kriterijus galima skirstyti į 6 dalis:

Struktūrinis modelių kriterijus – nusako veiksmų seką modelyje ir kaip atvaizduojama veiksmų seka programoje. Kiekvienas modelis gali turėti tik jam būdingą struktūrą, todėl šio kriterijaus pasirinkimas priklauso nuo situacijos.

Duomenų padengiamumo kriterijus – naudojantis šiuo kriterijumi, tikrinama, ar visos galimos įvestys ir išvestys yra patikrintos.

Klaidomis paremtas kriterijus – gamina testų rinkinius, kurie tinka aptikti tam tikras modelio klaidas. Teigiama, kad testuojama sistema turi tam tikras klaidas. Šis kriterijus jas aptinka.

Reikalavimais paremti kriterijai – pagrindinis tikslas užtikrinti, kad visi neformalūs reikalavimai yra patikrinti. Galimybė automatizuoti visą procesą vadovaujantis reikalavimais.

Tikslus testų rinkinių detalizavimas – tai leidžia inžinieriams tiksliai pasakyti, kurie testai ar testų rinkiniai turi būti sukurti naudojant modelius. Yra nurodomos įvairios testų detalizavimo kalbos, kurios gali būti puikus įrankis testams generuoti.

Statistinis testų kūrimo metodas – atsitiktinis testų generavimas yra vienas iš paprastesnių būdų testuoti sistemos elgesį.

Analizei pasirinkti testavimo įrankiai ne visus kriterijus palaiko, dėl to šiame darbe kriterijus suskirstysime šiek tiek kitaip[5]:

Modelio perėjimo kriterijai – į juos įeina būsenų, perėjimų, perėjimų poromis, kelio, lygiagrečių perėjimų ir scenarijaus kriterijai. Šie kriterijai naudojami testams kurti. Pirmi 4 gerai žinomi iš kitų testavimo būdų ir dažnai naudojami. Lygiagrečių perėjimų kriterijus pasižymi tuo, kad lygiagrečiai vykdomi du modelio perėjimai skirtingais keliais. Scenarijaus kriterijus užtikrina, kad visos testuotojo nurodytos būsenos ir perėjimai būtų patikrinti.

Skriptais perėjimo kriterijus (jį galima vadinti tiksliu testų rinkinių detalizavimu) – į šitą kriterijų įeina funkciniai, išdėstymo, sprendimų/šakų, sąlygų, pakoreguotų sprendimų/sprendimų ir vienetinių sprendimų kriterijai. Nemažai testavimo įrankių turi papildomus mechanizmus testuojamai sistemai apibūdinti. Naudojama ne tik baigtiniai automatai. Arba perėjimams uždedami saugikliai. Kiti įrankiai naudoja skriptų programavimo kalbas. Plačiai naudojama pre- ir post-sąlygų modeliavimo stilius. Pre- sąlyga yra išraiška, kuri turi būti teisinga operacijos vykdymo pradžios momentu. Post- sąlyga yra išraiška, kuri turi būti teisinga operacijos vykdymo pabaigos momentu. Paskutinius kriterijus galima naudoti kaip saugiklius, kurie (teisingai naudojat) supaprastina modelį ir jo perėjimą.

[5] Žemiau pateiktos 4 testų kriterijų grupės, naudojamos daugiau statistikai rinkti (kiek modelio buvo padengta, kokios viršūnės pereitos ir t. t.). Įvairioje literatūroje jie vadinami arba modelio padengimo, arba testų padengimo, arba testų pasirinkimo kriterijai. Šiame darbe jie bus vadinami testų kriterijais.

Duomenų padengimo kriterijus – šis kriterijus apima vienos reikšmės, visų reikšmių, kraštutinių reikšmių ir porinių reikšmių pasirinkimo kriterijus. Naudojant šiuos kriterijus testui, nurodoma, kokias įvesčių aibes pasirinkti. Pasirinkus vienos reikšmės kriterijų, bus pasirinkta tik viena įvestis ir klaidos radimo tikimybė sumažėja. Pasirinkus visas reikšmes, testuojama bus su visomis galimomis reikšmėmis, bet tai gali neapsimokėti dėl didelio testų skaičiaus ir laiko sąnaudų. Kraštutinių reikšmių kriterijus pasirinks tik pačias kraštines įvestis, su kuriomis dažniausiai atsiranda klaidos.

Reikalavimais paremtas kriterijus – šis kriterijus trumpai apibūdintas ankstesniame puslapyje.

Remiantis šiais kriterijais, galima analizuoti, kokie testavimo įrankiai, kokias turi savybes. Kaip aukščiau minėta – kuo daugiau kriterijų įrankis palaiko, tuo jis pranašesnis, ir turi daugiau galimybių kurti įvairesnius testus, atlikti įvairesnius uždavinius.

2.2.2. Testų kriterijai modelias paremto testavimo įrankiuose

Žemiau pateiktose lentelėse matyti pasirinktų testavimo įrankių analizė pagal testų pasirinkimo kriterijus. Kai kurių komercinių įrankių (*GOTCHATCBeans, Qtronic, Test Designer*) išbandyti nepavyko vien dėl to, kad jie net neteikia bandomosios versijos. Todėl žinios apie juos surinktos remiantis oficialiuose puslapiuose pateikta informacija, vartotojų vadovais ar pavyzdžiais. Jų nemažas funkcionalumas leidžia juos įtraukti į analizę kaip pavyzdinius.

3 lentelėje pateiktas modelio perėjimo kriterijaus palaikymas MPT įrankiuose. Išskyrus *SpecExplorer,* visi įrankiai palaiko būsenų perėjimo ir perėjimų kriterijus. Tai yra patys paprasčiausi kriterijai, kuriais pereinamas abstraktus sistemos modelis ir sukuriami testai. *GOTCHA-TCBeans* įrankis pažymėtas iš dalies vien dėl to, kad būtina nurodyti pradinę ir galutinę perėjimo būsenas. Perėjimų poromis kriterijus tampa labai naudingas, kai abstraktus sistemos modelis yra pakankamai didelis. Naudojant šį kriterijų, reikia mažiau testų, kad būtų pasiektas perėjimų padengimas. Šį kriterijų palaiko 3 pateikti įrankiai. Scenarijaus pasirinkimą galima nurodyti daugelyje įrankių. Sistemos modelyje pažymimos norimos būsenos arba perėjimai. Tokiu būdu modelio perėjimo algoritmas pirmiausia stengsis pereiti pažymėtas arba tik pažymėtas dalis. *MaTeLo* ir *TestOptimal* įrankiuose yra galimybė modelio (grafo) perėjimo algoritmui nurodyti, kad testavai būtų kuriami tik iš pažymėtų dalių.

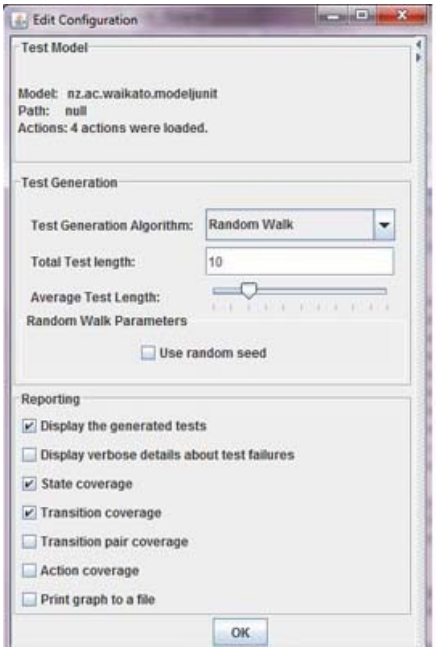
3 lentelė. **Modelio perėjimo kriterijai MPT įrankiuose**

Įrankio pavadinimas	Būsenų perėjimas	Perėjimais	Perėjimai poromis	Pasirinkto kelio	Lygiagretūs perėjimai	Scenarijus
MaTeLo	Palaiko	Palaiko	Nepalaiko	Nepalaiko	Nepalaiko	Palaiko
Qtronic	Palaiko	Palaiko	Palaiko	Palaiko	Palaiko	Nepalaiko
SpecExplorer	Nepalaiko	Palaiko	Nepalaiko	Nepalaiko	Nepalaiko	Nepalaiko
GraphWalker	Palaiko	Palaiko	Nepalaiko	Nepalaiko	Nepalaiko	Iš dalies palaiko
TestOptimal	Palaiko	Palaiko	Nepalaiko	Nepalaiko	Nepalaiko	Palaiko
GOTCHA-TCBeans	Iš dalies palaiko	Iš dalies palaiko	Nepalaiko	Nepalaiko	Nepalaiko	Iš dalies palaiko
AGEDIS	Palaiko	Palaiko	Nepalaiko	Nepalaiko	Nepalaiko	Iš dalies palaiko
Test Designer	Palaiko	Palaiko	Palaiko	Nepalaiko	Nepalaiko	Nepalaiko
ModelJUnit	Palaiko	palaiko	Palaiko	Nepalaiko	Nepalaiko	Nepalaiko

Funkcijų kriterijaus padengimas suteikia testavimo įrankiui galimybę patikrinti, ar tam tikri metodai testuojamoje sistemoje, ar abstrakčiame sistemos modelyje buvo įtraukti į testus. Testavimo įrankis, neturintis skriptų programavimo kalbos, tokio kriterijaus neatitiks. *GraphWalker* neturi savyje skriptų programavimo kalbos, dėl to nepalaiko nei vieno iš 4 lentelėje išvardintų testų kūrimo kriterijų. *SpecExplorer* šį uždavinį sprendžia sukurdamas perėjimus baigtiniame automate. Tokiu būdu padengiamos visos funkcijos. Visi kiti MPT kriterijų palaikantys įrankiai turi savyje mechanizmą, kuriuo ne tik padengiamos visos funkcijos, bet ir įrašoma, kurie testai padengia jas.

Vidinės skriptų programavimo kalbos *GDL* (*GOTCHA-TCBeans*), *mScript* (*TestOptimal*) ir *IF* (*AGEDIS*) nesprendžia skriptais paremtų kriterijų uždavinių. Jie naudojami tik testaams kurti. Norint palaikyti 4 lentelėje išvardintus kriterijus, reikalinga stipri skriptų programavimo kalba. Jas turi du komerciniai testavimo įrankiai: *Qtronic* ir *Test Designer*. *SpecExplorer* testuotojo importuotą modelį transformuoja beveik be jokių saugiklių ir veiksmų.

Skriptais paremti testų kūrimo kriterijai nėra lengvai įgyvendinami. Nes įrankis turi turėti nuosavą skriptų programavimo kalbą. Net ir toks reikalavimas nesuteikia testavimo įrankiui galimybės kurti tokius testus. Didžioji dalis atviro kodo ir nemokamų įrankių šito kriterijaus nepalaiko. Šiuos trūkumus gali išspręsti testuotojo profesionalumas ir patirtis. Vienas iš sprendimų būtų paprastesnių testavimo kriterijų jungimas arba paprastesnio modelio kūrimas. *ModelJUnit* pasitelkia Java programavimo kalbą, kuria galima spręsti tam tiktas problemas, bet tada testuotojas privalo turėti neblogus programavimo įgūdžius. 7 pav. pateikti du būdai, kaip galima nustatyti padengimo kriterijus. Testuotojas gali programuoti arba naudotis paprasta *ModelJUnit* grafine vartotojo sąsaja.




```
CoverageMetric stateCoverage = new StateCoverage();
tester.addCoverageMetric(stateCoverage);

CoverageMetric transitionCoverage = new TransitionCoverage();
tester.addCoverageMetric(transitionCoverage);

CoverageMetric transitionPairCoverage = new TransitionPairCoverage();
tester.addCoverageMetric(transitionPairCoverage);

tester.addListener(new VerboseListener());
tester.generate(10);

System.out.println("State coverage = "+stateCoverage.toString());
System.out.println("Transition coverage = "+transitionCoverage.toString());
System.out.println("Transition pair coverage = "+transitionPairCoverage.toString()
```

7 pav. ***ModelJUnit* įrankio kriterijų aprašymas**

Panaši situacija ir su duomenų padengimo kriterijais. Labai maža dalis testavimo įrankių juos palaiko. Tai turi didelę įtaką testuojant programinę įrangą, kurioje vyksta dideli duomenų apsikeitimai.

Dalis MPT įrankių kūrėjų vienos ir visų reikšmių kriterijus neįtraukia arba nerekomenduoja naudoti. Naudojant vienos reikšmės kriterijų, nebus įmanoma surasti visų klaidų. Visų reikšmių kriterijus gali sukurti daugybę testų ir testavimas taps per brangus.

Remiantis keliais testavimo tyrimais pramonėje [UL07][Bur02] ir akademinėje [PJ08] srityse, galima teigti, kad kraštutinių ir porinių reikšmių kriterijai žymiai naudingesni. Klaidos suradimo kaina žymiai lenkia pirmuosius du. *AGEDIS* ir *GOTCHA-TCBeans* nepalaiko šių kriterijų, bet leidžia testuotojui pačiam apsirašyti, kad testai būtų su reikiamomis įvestimis. Ši dalis paliekama pačiam testuotojui apsispręsti. *MaTeLo*, *ModelJUnit*, *GraphWalker* ir *TestOptimal* yra galimybė pačiam pateikti norimas įvestis kuriant modelį, bet atskiro funkcionalumo įrankiai neturi.

Testų kūrimo kriterijai yra labai svarbūs kuriant testus. Įrankių kriterijų palaikymo bendrą suvestinę galima matyti 6 lentelėje. Ar ja remiantis, galima teigti, kad du testavimo įrankiai yra žymiai pranašesni už kitus? Ir taip, ir ne.

4 lentelė. **Skriptai pereinami kriterijai MPT įrankiuose**

Įrankio pavadinimas	Funkcinis	Išdėstymo	Sprendimų/ šakų	Sąlygų	Koreguotų sprendimų/ būsenų	Vienetinių sprendimų
MaTeLo	Palaiko	Nepalaiko	Nepalaiko	Nepalaiko	Nepalaiko	Nepalaiko
Qtronic	Palaiko	Palaiko	Palaiko	Palaiko	Nepalaiko	Palaiko
SpecExplorer	Palaiko	Nepalaiko	Nepalaiko	Nepalaiko	Nepalaiko	Nepalaiko
GraphWalker	Nepalaiko	Nepalaiko	Nepalaiko	Nepalaiko	Nepalaiko	Nepalaiko
TestOptimal	Nepalaiko	Nepalaiko	Nepalaiko	Nepalaiko	Nepalaiko	Nepalaiko
GOTCHA-TCBeans	Nepalaiko	Nepalaiko	Nepalaiko	Nepalaiko	Nepalaiko	Nepalaiko
AGEDIS	Palaiko	Nepalaiko	Nepalaiko	Nepalaiko	Nepalaiko	Nepalaiko
Test Designer	Palaiko	Palaiko	Palaiko	Nepalaiko	Palaiko	Nepalaiko
ModelJUnit	Nepalaiko	Nepalaiko	Nepalaiko	Palaiko	Nepalaiko	Nepalaiko

Dalis įrankių, neturinčių daug testų kūrimo kriterijų, turi mechanizmą, leidžiantį modeliavimo metu aprašyti būsenas taip, kad būtų pasiektas reikiamas modelio padengimo lygis. Yra galimybės aprašyti saugiklius arba pažymėti norimas būsenas (šio funkcionalumo neturi *SpecExplorer*). Kriterijai žymiai palengvina ir išplečia testų kūrimą. Bet kai kurie įrankiai (*MaTeLo*, *TestOptimal*) skriptais leidžia praplėsti modelio perėjimo galimybes. *ModelJUnit* ir *GraphWalker* naudoja gerai žinomas programavimo kalbas, kuriomis galima apsibrėžti norimus tikslus.

5 lentelė. **Duomenų padengimo kriterijai ir reikalavimo kriterijus MPT įrankiuose**

Įrankio pavadinimas	Viena reikšmės	Visos reikšmės	Kraštutinės reikšmes	Porinės reikšmės	Reikalavimai
MaTeLo	Nepalaiko	Nepalaiko	Nepalaiko	Nepalaiko	Palaiko
Qtronic	Nepalaiko	Nepalaiko	Palaiko	Nepalaiko	Palaiko
SpecExplorer	Palaiko	Palaiko	Palaiko	Palaiko	Palaiko
GraphWalker	Nepalaiko	Nepalaiko	Nepalaiko	Nepalaiko	Palaiko
TestOptimal	Nepalaiko	Nepalaiko	Nepalaiko	Palaiko	Nepalaiko
GOTCHA-TCBeans	Nepalaiko	Nepalaiko	Nepalaiko	Nepalaiko	Nepalaiko
AGEDIS	Nepalaiko	Nepalaiko	Nepalaiko	Nepalaiko	Nepalaiko
Test Designer	Palaiko	Palaiko	Palaiko	Palaiko	Palaiko
ModelJUnit	Nepalaiko	Nepalaiko	Nepalaiko	Nepalaiko	Nepalaiko

Visi išvardinti įrankiai testuojamą sistemą atvaizduoja kaip baigtinį automatą. Šiuo atveju modelio perėjimo kriterijus yra gan svarbus. *Qtronic* ir *TestDesigner* vidinė skriptų programavimo kalba leidžia pirmauti skriptais paremtuose kriterijuose. Tokiu būdu atsiranda nemažos galimybės

manipuliuoti testų kūrimą. Testuotojui suteikiamos vidinės (nebūtina naudotis papildomais įrankiai) galimybės valdyti testavimo procesą.

Duomenų padengimo kriterijais išsiskiria *SpecExplorer*, kuris yra nemokamas įrankis. Šis kriterijus suteikia pranašumą, kai testuojamose taikomosiose programose vyksta duomenų apsikeitimai (pildomos įvairios formos ir t. t.). Kiti testavimo įrankiai išsprendžia šias problemas leisdami patiems apsirašyti testavimui reikalingas įvestis. Įvestys aprašomos kuriant abstraktų sistemos modelį. Tokiu būdu gali atsirasti dideli būsenų kiekiai. *TestOptimal* išsprendžia šią problemą vidine skriptų programavimo kalba. *MaTeLo* modeliavimo įrankyje yra galimybė įvestis skirstyti į aibes ir aprašyti jų tipus. Didelio būsenų skaičiaus problema išsprendžiama naudojant sub-modelius. *ModelJUnit* irgi leidžia apsirašyti įvestis modeliavimo etape. Tokiu būdu testuotojas turi spręsti, kurias įvesti reikia naudoti. Taigi nei vienas paminėtas testavimo įrankis nenusižengia modeliais paremto testavimo įdėjai, kad testai kuriami automatiškai iš abstraktaus sistemos modelio.

Nemaža dalis įrankių palaiko reikalavimais paremtus kriterijus. Reikalavimų padengimas dažniausiai vykdomas kiekvienam reikalavimui suteikiant unikalų numerį ir jį priskiriant kažkokiai būsenai arba perėjimui. Reikalavimų padengimas skaičiuojamas remiantis ta pačia metodika kaip ir būsenų ar perėjimų padengimas. Atsiranda galimybė pereidinėti tas būsenas ar perėjimus, kurie pažymėti kaip reikalavimo įgyvendinimas. Testai kuriami, kol nepadengiami visi reikalavimai.

6 lentelė. **Testų kūrimo kriterijai MPT įrankiuose**

	MaTeLo	***Qtronic***	***SpecExplorer***	***GraphWalker***	***TestOptimal***	***GOTCHA-TCBeans***	***AGEDIS***	***Test Designer***	***ModelJUnit***
Modelio perėjimo kriterijai	3	5	1	3	3	3	3	3	3
Skriptais perėjimo kriterijai	1	5	1	0	0	0	1	4	1
Duomenų padengimo kriterijai	0	1	4	0	1	0	0	4	0
Reikalavimais paremtas kriterijus	1	1	1	1	0	0	0	1	0
Iš viso	5	12	7	4	4	3	4	12	4

Naudojant testų kriterijus, galima susidaryti bendrą testų kūrimo įrankiuose vaizdą. Vien šiais kriterijais remiantis negalima rinktis įrankio. Kai kurie testavimo įrankiai gali pareikalauti nemažų programavimo žinių pačioje pradžioje (*ModelJUnit*, *GraphWalker*, *SpecExplorer*). Kitiems užtenka tik nedidelių programavimo žinių, nes yra sukurta gera grafinė vartotojo sąsaja (*TestOptimal*, *MaTeLo*). Tokiu būdu modelio ir testų kūrimas vykdomas „drag-n-drop“ būdu. Nepriklausomai nuo modelio dydžio, testuotojas visada bus viso proceso centre. Įrankiai yra tik pagalbininkas testavimo procese, bet pakeisti žmogaus neįmanoma. Šio darbo 3 skyriuje pateikta siūlymai, kokius sprendimus įvykdžius galima pagerinti MPT procesą.

2.2.3. Modelio perėjimo algoritmai

Sukurti testai ne visada bus pakankamai geri, net ir naudojant testavimo kriterijus. Vieni testai bus labai trumpi ir neatitiks reikalavimų. Kiti labai ilgi, per daug išsamūs ir sunkiai panaudojami. Vien testavimo kriterijaus neužtenka, tad tenka pasitelkti grafų perėjimo algoritmus. Kadangi sistemos modelis atvaizduojamas kaip baigtinis automatas, tai atsiranda galimybė taikyti visus grafų teorijoje žinomus grafo perėjimo algoritmus.

Jų nauda akivaizdi. Atsitiktinis perėjimų algoritmas su kriterijumi pereiti visas būsenas iš 4 pav. pavaizduoto paprasto vartotojo sąsajos modelio sukuria testą, susidedantį iš 480-500 perėjimų. Toks testas jokiais būdais negali duoti naudos, nes atsiras problemos jį paleidžiant ir tikrinant.

Taikant būsenų perėjimo kriterijų 8 pav. pateiktam modeliui su atsitiktinio perėjimo algoritmu, išeina toks testas:

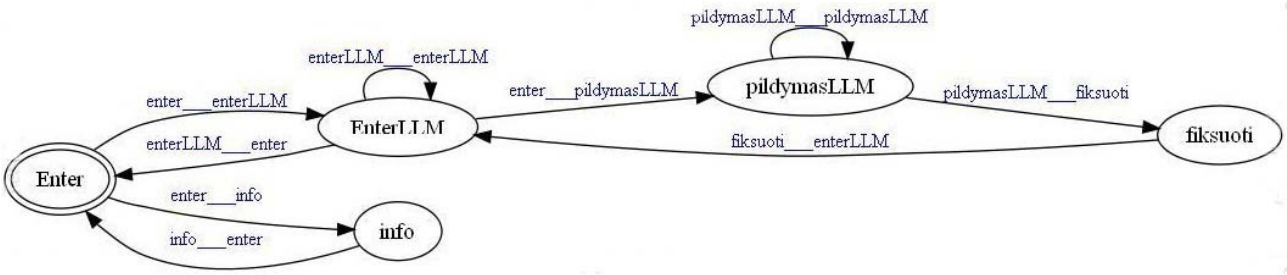
Enter -> EnterLLM -> Enter -> info -> Enter -> info -> Enter -> EnterLLM -> pildymasLLM -> fiksuoti -> EnterLLM -> Enter

Testas įvykdo sąlygą pereiti visas viršūnes. Bet dalis viršūnių apeinama kelis kartus, todėl testas akivaizdžiai per ilgas. Kai kuriais atvejais testas išėjo dar ilgesnis. Tik iš 6 karto buvo gautas testas, kuris buvo gautas iš pirmo karto panaudojus optimalų modelio perėjimo algoritmą:

Enter -> EnterLLM -> pildymasLLM -> fiksuoti -> EnterLLM -> Enter -> Info -> Enter

Grafų teorijoje žinoma jūreivio kelionės problema – algoritmas, kuris pereina visas grafo viršūnes bent vieną kartą. Deja, šio algoritmo pritaikymo MPT įrankiuose neteko rasti.

Modelio perėjimo algoritmų palaikymas testavimo įrankiuose duoda dar daugiau naudos, kai testams kurti naudojami perėjimų kriterijai. Visi perėjimai turi būti pereiti nors vieną kartą. Perėjimų kriterijus yra aukščiau būsenų kriterijaus, nes užduotis pereiti visus perėjimus nors vieną kartą pereis ir visas viršūnes. Tokiu atveju bus įvykdytas ir būsenų padengimo uždavinys.



8 pav. **Virtualių parodų sistemos parodų tvarkymo dalis**[6]

Atsitiktinis perėjimo algoritmas ir perėjimų kriterijus iš 8 pav. sukuria tokį testą[7]:

enter_enterLLM, enterLLM_enterLLM, enterLLM_enter, enter_enterLLM, enter_pildymasLLM, pildymasLLM_pildymasLLM, pildymasLLM_pildymasLLM, pildymasLLM_pildymasLLM, pildymasLLM_fiksuoti, fiksuoti_enterLLM, enterLLM_enter, enter_enterLLM, ... , info_enter

Šiuo atveju išeina 26 perėjimai. Modelio perėjimai padengti 100 procentų. Modelis labai nedidelis. Testuojant visą Virtualių parodų sistemos modelį, testas gali išeiti žymiai didesnis ir visai nepanaudojamas.

Geriausias būdas kurti trumpiausius testus ir pasiekti šimtaprocentinį perėjimų padengimą yra naudojant gudresnius perėjimų algoritmus. Algoritmas randa trumpiausią kelią pereiti visus perėjimus bent vienąkart. Geriausiai žinomas ir dažniausiai naudojamas yra kinų paštininko algoritmas, kuris buvo išrastas Kinijos matematiko Guan Mei Gu [Thi03]. Šį algoritmą naudoja *TestOptimal* ir *SpecExplorer*. Juo sukuriamas toks testas:

enter_enterLLM, enter_pildymasLLM, pildymasLLM_pildymasLLM, pildymasLLM_fiksuoti, fiksuoti_enterLLM, enterLLM_enterLLM, enterLLM_enter, enter_info, info_enter

Modelis pakankamai mažas, bet akivaizdžiai matomas skirtumas. Kiti įrankiai naudoja kitus algoritmus perėjimams per modelį optimizuoti. Tokiu būdu padengiamos ne tik visos viršūnės, bet ir apsirašytos įvestys.

Gali pasirodyti, kad paprasti grafo perėjimo algoritmai gali sukurti gerus testus ir nėra reikalingos testuotojo geros žinios, sugebėjimai ir patirtis. Testuotojas visada lieka viso proceso centre. Kurdamas modelį, testuotojas turi nuspręsti, kuriuos algoritmus naudoti, kokias žinias taikyti, privalo atlikti testavimo rezultatų analizę (rastos klaidos, kodo padengimo lygis, modelio padengimo lygis ir t. t.) ir nuspręsti, kiek testų dar reikės vykdyti.

---

[6] Paveiksle pavaizduota maža Virtualių parodų sistemos parodų tvarkymo dalis. Pereinant iš būsenos „Enter“ į „EnterLLM“, reikia pasirinkti parodą ir įvesti slaptažodį. Būsenoje „pildymasLLM“ yra virš 50 laukų, iš kurių yra 7 privalomi. Šio pavyzdžio metu laukai bus nepildomi arba pildomi tik privalomi. „Enter“ būsena yra ir pradinė, ir galutinė.

[7] Kiekvienu atveju sukuria vis skirtingą testą. Čia pateiktas tik vienas iš variantų. Būsenos nevaizduojamos.

Algoritmai ir kriterijai padeda testuotojui lengviau pasirinkti, kurie testai bus naudingesni. Tai yra tik darbo palengvinimo įrankis. Bet vien tik juo pasitikėti negalima. Įrankiai padeda pasirinkti įdomius testus, svarbesnes sistemos dalis. Vienas iš būdų pagerinti testavimą – nusistatyti aiškią testų specifikaciją, t. y. nusistatyti, kuriuos testus vertėtų atlikti. Tokiu atveju testavimo įrankis tampa puikus pagalbininkas šiam tikslui pasiekti.

Žinant savo galimybes ir poreikius, reikia žinoti, kaip testai bus konvertuojami į testų skriptus, kaip pateikiami sistemai ir tikrinami gauti rezultatai. Šis klausimas nagrinėjamas kitame skyriuje.

```
import nz.ac.waikato.modeljunit.Action;
import nz.ac.waikato.modeljunit.FsmModel;
import nz.ac.waikato.modeljunit.GreedyTester;
import nz.ac.waikato.modeljunit.Tester;
import nz.ac.waikato.modeljunit.VerboseListener;

class ParoduTvarkymas implements FsmModel {
            //modelis susideda iš 5 būsenų
            protected boolean Enter, EnterLLM, PildymasLLM, fiksuoti, info
            protected String pass = "pass", muziejus = "LDM"; enteredInfo = "info"
            //atstatom į pradinę būseną
            void reset(boolean testing) {Enter = true, EnterLLM = false, PildymasLLM = false,
fiksuoti = false, info = false}
            //aprašomi visi perėjimai
            @Action public void enter_enterLLM(){
                        //įvedamas slaptažodis ir pasirenkama paroda
                        if(pass == enteredPass && muziejus == enteredMuziejus){
                                    Enter = false; EnterLLM = true;
                        }else {
                                    Enter = true;
                        }
                        };
            @Action public void enterLLM_enter(){EnterLLM = false; Enter = true;}
            @Action public void enter_info(){Enter = false; info = true;}
            @Action public void info_enter(){info = false; Enter = true;}
            @Action public void enterLLM_enterLLM(){EnterLLM = true;}
            @Action public void enter_pildymasLLM(){EnterLLM = false; EnterPildymas = true;}
            @Action public void pildymasLLM_pildymasLLM(){if(enteredInfo) PildymasLLM = true;}
            @Action public void pildymasLLM_fiksuoti(){
                        if(enteredInfo) {PildymasLLM = false; fiksuoti = true;}
                        }
            @Action public void fiksuoti_enterLLM(){fiksuoti = false; EnterLLM = true;}
            //Naudojant godųjį algoritmą bus sukurta 10 testo atvejų
            public static void main(String[] args)
              {
                Tester tester = new GreedyTester(new ParoduTvarkymas());
                tester.addListener(new VerboseListener());
                tester.generate(10);
              }
}
```

9 pav. **8 pav. pateiktas modelis aprašytas *ModelJUnit***

2.3. Sistemos testavimas

Šiame skyriuje didžiausias dėmesys skirtas testų konvertavimui į paleisti tinkamus testus. Pagrindinė problema – kaip testai bus konvertuojami ir paleidžiami. Kai kuriais atvejais testų konvertavimas ir paleidimas užima nemažą testavimo laiko tarpą. Laiko sąnaudos tiesiogiai priklauso nuo MPT įrankio galimybių.

Visada yra noras testus paleidinėti automatiškai. Kuo didesnė proceso dalis bus automatizuota, tuo daugiau bus paleista testų ir tokiu būdu sumažės testavimo kaina, sutrumpės testavimo laikas.

Testai yra pakankamai abstraktūs, nes jie gaunami iš abstraktaus testuojamos sistemos modelio. Sukurti testai neturi pakankamai detalios informacijos, kad būtų galimybė juos paleidinėti tiesiogiai. Sistemos modelis ir testuojama sistema dažniausiai neatitinka vienas kito dėl skirtingo abstrakcijos lygio. Aukščiau pateikta rekomendacija atsisuka prieš mus, kai norima paleisti testus.

2.4. Testų adaptavimas ir transformavimas

Norint paleisti sukurtus testus, būtina paruošti testuojamą sistemą testavimui, pridėti trūkstamą informaciją ir sutvarkyti visus galimus testų ir testuojamos sistemos nesutapimus. Būtina susieti visas abstrakčių testų reikšmes ir objektus su testuojama sistema. Atsiranda reikšmių praplėtimo poreikis. Gautas išvestis irgi būtina pakelti į tinkamą abstrakcijos lygį.

Testavimo metu atsiradusius naujus objektus būtina sekti. Atsiranda susiejimo matrica, reikalinga objektams ir reikšmėms palaikyti.

Šis procesas nėra naujas. Su tokiomis pat problemomis susiduriama skriptais paremtame testavime. Naudojami trys abstrakcijos sumažino būdai, pateikti 10 pav.

Adaptavimas – rankiniu būdu parašomas kodas, kuris sistemos veikimą atvaizduoja abstraktesniame lygyje. Tokiu būdu iš modelio sukurti testai gali būti paleisti testuojamai sistemai.

Transformavimas – abstrakčius testus transformuojame į paprastus paleidimo skriptus.

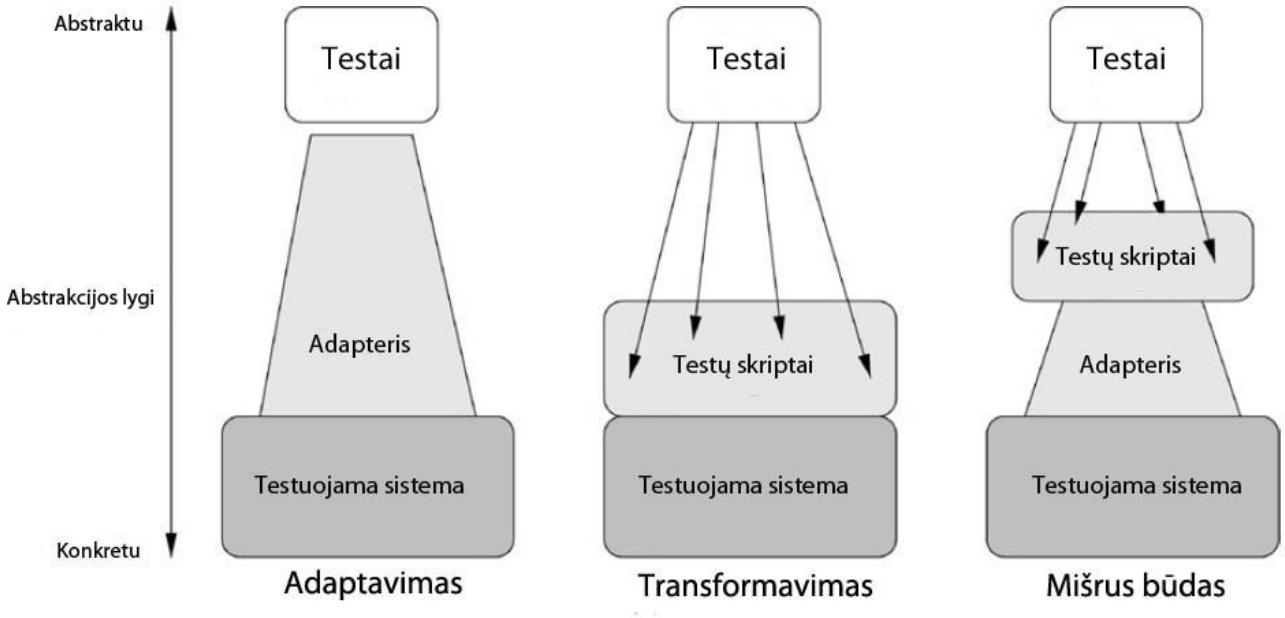
Miršus būdas – pirmų dviejų būdų apjungimas. Adapteris sistemos veikimą atvaizduoja abstraktesniame lygyje. Nebereikia kurti detalių testavimo skriptų. Transformacija žymiai palengvėja, nes sistema atvaizduojama paprasčiau ir sukurti testai yra paprastesni. Atsiranda galimybė naudoti kelis skirtingus modelius ir testavimo scenarijus.

Žemiau trumpai aprašomi visi abstrakcijos problemos sprendimo būdai.

Adaptavimo metu parašomas kodas, kuris tampa tarpininku tarp testuojamos sistemos ir testo. Adapterio kodas tampa abstrakčių komandų interpretatoriumi.

Adapteris atsakingas už tokius veiksmus:

Nustatymai. Testuojamos sistemos paruošimas testavimui. Pagal nustatytą scenarijų sistema paruošiama ir konfigūruojama.



10 pav. **Trys abstrakcijos problemos sprendimo būdai [Pou08]**

Konkretizavimas. Kiekvieną abstraktaus testo kreipinį ir įvestį paverčia sistemai suprantamu formatu.

Išgavimas. Surenka visus sistemos gautus rezultatus ir pateikia atgal tinkamu formatu tinkamame abstrakcijos lygyje. Gauti rezultatai siunčiami patikrinti.

Atjungimas. Po kiekvienos testų sekos sistema išjungiama.

Tai yra keturi pagrindiniai adapterio veiksmai. Kiekvienas adapteris juos atlieka. Kai kurie testavimo įrankiai naudoja išorinius testų paleidimo įrankius (*ModelJUnit*, *MaTeLo*, *Test Designer*). Yra daugybė būdų automatiškai sujungti testuojamą sistemą ir rankiniu būdu sukurtus testus. Tuos pačius būdus galima naudoti ir MPT procese. Didelė dalis įrankių skolinasi jau esamus įrankius juos patobulindama arba kuria naujus naudodami esamų įrankių architektūrą. Priklausomai nuo testuojamos sistemos sudėtingumo, gali tekti adapterį pritaikyti rankiniu būdu. Tokiu būdu reiks programavimo sugebėjimų. Naudojant *SpecExplorer* įrankį prireiks papildomo testuotojo įsikišimo, nes įrankis teikia tik šabloną. Adaptavimo būdą galima taikyti įvairioms sistemoms. Visuose paminėtuose MPT įrankiuose būtinas papildomas testuotojo įsikišimas jungiant adapterį su testuojama sistema.

Transformavimo metodas kiekvieną abstraktų testą paverčia į paleidimui tinkamą skriptą. Naudojamos įvairios skriptams programuoti kalbos:

- tradicinės programavimo kalbos kaip *Java* (*GraphWalker, ModelJUnit*) ar *C*.
- Skriptų programavimo kalbos kaip *TCL, JavaScript* ar *VBScript*.
- Tradicinės testų notacijos kaip *TTCN-3*.

- Vidinės testų notacijos kaip *TSL*.

Transformavimo procesas vykdomas tam tikromis transformavimo kalbomis, kurios sukuria paleidžiamus skirptus arba naudojami tam tikri šablonai, kuriais susiejami visi objektai ir parametrai. Pavyzdžiui, galima apibrėžti *TCL* šabloną kiekvienai abstrakčiai operacijai. Kiekvienas *TCL* kreipinys turės testuojamai sistemai reikalingą operaciją ir ją paleis reikiamu momentu. Tokiu būdu bus sukurtas testo skriptas, kuris pasirinks tinkamą operaciją ir ją pateiks testuojamai sistemai. Gali atrodyti, kad visas procesas yra ganėtinai paprastas, nes dalis jo automatizuota, ir reikalingas nedidelis testuotojo įsikišimas. Tačiau kiekviena sistema yra skirtinga ir esant sudėtingesniems veiksmams, testuotojo įsikišimas būtinas. Taikant šį būdą, reikia atkreipti dėmesį į kelis dalykus:

- reikalingas testuojamos sistemos nustatymas ir atjungimas. Kodas dažniausiai rašomas rankiniu būdu ir integruojamas automatiškai. Šie du žingsniai lygiai tokie pat, kaip ir adaptavimo metode.
- Kiekvienos operacijos šablonas gali būti gan sudėtingas, nes abstraktaus testo ir testuojamos sistemos susiejimas ne visada bus vienas su vienu. Gali tekti iššaukti kelias testuojamos sistemos operacijas, kad būtų įvykdyta viena abstrakti operacija. Testo skriptas privalės sukurti trūkstamas įvestis. Testo skriptas turės „sugauti“ visas galimas išimtis ir patikrinti kitus galimus variantus. Gali atsirasti poreikis patikrinti, ar gauti rezultatai tinka modeliui. Gali atsirasti poreikis patikrinti tam tikrus galimus veiksmus.
- Modelis naudoja abstrakčias konstantas ir reikšmes. Jas būtina paversti konkrečiomis testuojamos sistemos reikšmėmis. Abstrakčias reikšmes galima susieti su galimų reikšmių aibe ir tada kurti galimus variantus. Pavyzdžiui, paveiksle Nr. 8 būsenoje „Enter“, norint patekti į parodos taisymą, reikia suvesti slaptažodį, pasirinkti muziejų ir parodą. Modeliuojant su *MaTeLo* apsirašomi trys kintamieji: „muziejus“, „paroda“, „slaptažodis“. Kuriant testų skriptus, būtina kintamiesiems priskirti reikšmių aibes. Šiuo atveju susiejimas vykdomas modeliavimo metu. Naudojant *SpecExplorer*, būtini reikšmių susiejimai.
- Testai gali turėti medžio struktūrą. Tokiu atveju reikalingas gudrus transformavimo variklis, kuris gali nustatyti, kokią būseną pasiekė testo skriptas.
- Būtinas atsekamumas tarp testo skripto ir testo, kad būtų žinoma, kuris testas atliktas ar kuris testas nepavyko. Tokiu būdu atsiranda galimybė gauti ataskaitas, kiek modelio padengė skriptai. Šios savybės MPT įrankiai dažniausiai neturi.

Taigi transformacijos metodas sukuria testų skriptus, kurie atitinka esamus testavimo valdumo principus, tą pačią kalbą, struktūrą ir užvardinimo tvarką, kaip ir rankiniu būdu sukurti testai.

Iškyla klausimas, kuris būdas geresnis.

„Online“ testavimui beveik visada geriau yra adaptavimo metodas, nes būtinas glaudus integravimas. Reikalinga pastovi jungtis tarp MPT įrankio ir testuojamos sistemos. Šis tikslas lengviausiai pasiekiamas, kai MPT metu adapterio aplikacijų programavimo sąsaja tiesiogiai sujungta su testuojamos sistemos sąsaja.

„Offline“ testavimui naudojami visi trys paveiksle Nr. 8 pavaizduoti būdai. Naudojant transformavimo metodą, gauti testų skriptų rinkiniai paleidžiami tiesiogiai testuojamai sistemai. Adaptavimo metodo metu testų rinkiniai tampa iš karto efektyvūs, nes adapteris veikia kaip interpretatorius, kuris susieja kiekvieną abstraktų veiksmą su testuojama sistema ir gautus rezultatus grąžina pavertęs į reikiamą abstraktų lygį. Mišraus metodo metu abstraktūs testai paverčiami testų skriptais, pateikiami adapterio lygiui, kuriame detalizuojami.

Transformacijos metodo privalumas, kad jo metu sukurti testų skriptai sukurti ta pačia kalba, taip pat užvadinti ir turi tokią pat struktūrą, kaip rankiniu būdu kuriami skriptai. Tokiu būdu atsiranda galimybė taikyti kituose testavimo procesuose naudojamus testų skriptų paleidimo įrankius. Testavimo valdymas ir testų paleidimo platforma nesikeičia. MPT keičia tik testų ir testų skriptų kūrimo procesą. Visas kitas testavimo procesas lieka tas pats. Tokiu būdu MPT procesą tampa lengviau pritaikyti, nes dalis įrankių lieka nepakitę. Pavyzdžiui, *ModelJUnit* įrankis naudojamas modelio, testams ir testų skriptams kurti. Kiti veiksmai atliekama su *JUnit*.

„Online“ testavimui naudojamas adaptavimo metodas. „Offline“ testavimui naudojamas transformacijos metodas ir kartais mišrus.

2.4.1. Testų skirptų kūrimas ir paleidimas įrankiuose

Visi aukščiau išvardinti metodai naudojami modeliais paremtame testavimo procese. Žinoma, ne kiekvienas įrankis palaiko visus metodus. Nemaža dalis įrankių pasitelkia jau esamus ir taip supaprastina MPT įsisavinimą. Kitais įrankiais galima patiems kurti ir paleisti testus testuojamai sistemai.

Galima išskirti kelis kriterijus, pagal kuriuos būtų galimybė analizuoti esamus MPT įrankius. Kriterijai yra pakankamai bendri, bet suteikiantys pakankamai aiškų supratimą, kaip vienas ar kitas įrankis vykdo testų paleidimus.

Pagal aukščiau pateiktą informaciją, matoma, kad visi MPT įrankiai patys kuria testus. Testų skriptus kuriant ir paleidžiant situacija šiek tiek kitokia.

2.3.1. skyriuje aptarėme pagrindinę problemą ir jos sprendimą, kai kuriami testų skriptai. Testuojamos sistemos modelis yra gan abstraktus. Iš jo sukurti testai irgi abstraktūs. Reikalingi papildomi įrankiai spręsti šias problemas.

Galima išskirti 4 kriterijus, pagal kuriuos būtų galimybė analizuoti ir lyginti esamus MPT įrankius. Pirmiausia – testų skriptų kūrimas ir adaptavimas. Šiame žingsnyje sukuriami nedideli kodai, kurie paleidžia testus. Visas procesas aprašytas 2.3.1. skyriuje.

Didžioji dalis įrankių iš dalies atitinka kriterijų, nes vieni testų skriptus sukuria, o paleidimas vykdomas išoriniais įrankiais (*MaTeLo, GraphWalker, ModelJUnit*). *Test Designer* naudoja išorinius įrankius testams paleisti. Visi iš dalies kriterijų palaikantys įrankiai automatiškai sukuria adapterio skeletą. Visas kitas funkcijas turi apsirašyti pats testuotojas. Vienas įrankis (*ParTeG*), kuris čia nėra paminėtas, automatiškai sukuria visą adapterį. Bet jis nėra galutinai sukurtas ir su dideliais modeliais ir sudėtingomis sistemomis automatiškai visko nesukuria.

Atviro kodo įrankiai reikalauja daugiausia programavimo žinių, bet jie yra pakankamai paprasti. Pavyzdžiui, *ModelJUnit* įrankiu sukurtas modelis pridėjus papildomą informaciją sukurs pakankamai gerus testus, kuriuos bus galima pateikti testuojamai sistemai. Testuotojas, kuris programavęs *Java* programavimo kalba, neturės didelių problemų. Šis įrankis yra *JUnit* plėtinys ir modeliai, ir gaunami testai yra parašyti *Java* kalba. *GraphWalker* įrankio sukurti testai irgi *Java* programavimo kalba. Kadangi įrankis vis dar vystomas, nedaug apie jį galima rasti informacijos. Vartotojo vadovai arba nepabaigti, arba pasenę. Yra labai prasta vartotojo sąsaja. Įrankis lengviau suprantamas programuotojui, o ne testuotojui. Skriptų paleidimas vykdomas su pačiu įrankiu.

SpecExplorer nemokamas *VisualStudio* priedas, kuriuo galima vykdyti testavimą. Testuotojui, dirbusiam su *VisualStudio*, nebus jokių problemų, nes tiek modelis, tiek gauti testai yra užrašomi c# programavimo kalba. Adaptavimas irgi atliekamas tik iš dalies. Būtinas testuotojo įsikišimas. Nors testavimo įrankis yra nemokamas, *SpecExplorer* reikalinga mokama *VisualStudio* versija.

Daugiau ar mažiau, bet paleidžiant testus testuojamai sistemai, reikalingas papildomas testuotojo darbas. Tai visai nestebina, nes kiekviena testuojama programa yra skirtinga. Testuojant, Virtualių parodų sistema nesukėlė daug problemų, nes vartotojo sąsaja nėra labai sudėtinga ir tik įvedant naują eksponatą reikalingi dideli duomenų kiekiai. *TestOptimal* testų paleidimams naudoja *Selenium IDE*[8]. Yra galimybė testus *kurti* Java programavimo kalba. Beveik visi įrankiai teikia galimybes testus eksportuoti į tam tikras programavimo kalbas. Taip atsiranda galimybė naudoti papildomus įrankius.

[8] Daugiau informacijos galima rasti adresu http://seleniumhq.org/projects/ide/ .

Kita gan svarbi testavimo įrankio savybė – testų tikrinimas. Kaip ir testų paleidimo, taip ir testų tikrinimo MPT įrankiai visai nepalaiko. Testų tikrinimas dažnai vykdomas testų skriptų kūrimo metu. Testuotojas rankiniu būdu turi nurodyti, kaip bus vykdomas testų tikrinimas. Kai kurie įrankiai turi papildomą funkcionalumą, kuriuo galima gauti testų vykdymo ataskaitą. *TestOptimal* turi testų tikrinimo ataskaitas. *ModelJUnit* testų kūrimo metu būtina nurodyti, kad tikrintų, ar testas pavyko, ar ne.

Ši savybė labai svarbi, nes nežinant, ar testas pavyko, ar ne, prarandamas visas testavimo proceso tikslas. Tam tikslui galima pasitelkti ir kitus įrankius. *MaTeLo* įrankiu sukuriami testai, kuriuos galima pateikti *XML* formatu arba kaip *Java* programavimo kalbos kodo gabalą. Testams tikrinti naudojama išorinė programa, su kuria vykdomas sistemos tikrinimas. Šiuo atveju su *Selenium IDE* arba *JUnit*. *Test Designer* savyje neturi net testų kūrimo mechanizmo. XML formatu gauti testai teikiami kitam testavimo įrankiui, kuriuo vykdomi visi reikalingi veiksmai.

7 lentelė. **Testų paleidimas ir tikrinimas**

Įrankio pavadinimas	**Testų skriptų kūrimas ir adaptavimas**	**Testų tikrinimas**	**„Online“ testavimas**	**„Offline“ testavimas**
MaTeLo	Dalinai	Išoriniai įrankiai	Ne	Taip
Qtronic	Ne	Ne	Išoriniai įrankiai	Išoriniai įrankiai
SpecExplorer	Iš dalies	Iš dalies	Ne	Taip
GraphWalker	Iš dalies	Ne	Iš dalies	Išoriniai įrankiai
TestOptimal	Iš dalies	Iš dalies	Išoriniai įrankiai	Išoriniai įrankiai
GOTCHA-TCBeans	Iš dalies	Iš dalies	Taip	Taip
AGEDIS	Iš dalies	Iš dalies	Ne	Taip
Test Designer	Išorinis įrankis	Išoriniai įrankiai	Ne	Išoriniai įrankiai
ModelJUnit	Iš dalies	Iš dalies	Išorinis	Išorinis

Kitas svarbus kriterijus – „Online“ ir „Offline“ testavimas. Pagrindinis šių testavimo būdų skirtumas yra tas, kad „Offline“ testavimo metu testai pirmiausia sukuriami ir testuotojas gali pasirinkti, ar pradėti testavimą, ar palikti kitam laikui. „Online“ testavimo metu įrankis tiesiogiai bendrauja su testuojama sistema ir sukurti testai iškart pateikiami testuojamai sistemai.

Visi pateikti įrankiai (lentelė Nr. 7) palaiko „Offline“ metodą. Dalis jų naudoja išorinius testavimo įrankius. „Online“ testavimo metodas palaikomas tik daugiau nei pusės testavimo įrankių. Naudojant *GraphWalker*, reikalingas papildomas testuotojo įsikišimas.

3. Modelias paremto testavimo įrankių patobulinimo galimybės

Šiame skyriuje apžvelgiama MPT įrankių analizė. Nagrinėjami tyrimo rezultatai ir pateikiami siūlymai, kaip galima patobulinti testavimo įrankius. Pateikiamos atsirandančios problemos ir sprendimo būdai taikant modeliais paremtą testavimo procesą. Pateikiami galimi sprendimo būdai, kaip galima patobulinti testavimo įrankius.

3.1. Modeliais paremtos testavimo proceso adaptavimas

Prieš nagrinėjant testų įrankių analizę ir teikiant siūlymus, būtina žinoti ir suprasti, kokių žinių ir veiksmų būtina imtis norint pritaikyti modeliais paremtą testavimą. Didžioji dalis informacijos gaunama iš testų įrankių analizės, nes būtent jos metu puikiai matyti, su kokiomis problemomis susiduriama naudojant testavimo įrankius.

Kaip minėta 2 skyriuje, visi tyrimui naudoti įrankiai daugiau ar mažiau palaiko visą MPT procesą (paveikslas Nr. 3). Norint naudoti šį testavimo būdą, yra būtinos kelios sąlygos, kurios išplaukia iš testavimo įrankių analizės.

Žmogiškasis faktorius yra vienas svarbiausių veiksnių adaptuojant bet kokią naują praktiką. Gera komanda būtina, nes šis testavimo būdas reikalauja įvairių sugebėjimų ir patirties.

Aukštas testų paleidimo brandos lygis būtinas. Modeliais paremto testavimo pagrindinė idėja – testai kuriami automatiniu būdu. Iš to išplaukia, kad ir testų paleidimas yra automatizuotas. Tai reiškia, kad testuotojas turi turėti testavimo automatizavimo patirties. Modeliais paremtą testavimo procesą patartina taikyti testuotojų komandai, kuri yra susidūrusi su skriptais paremtu testavimu arba raktiniais žodžiais paremtu automatizuotu testavimo būdu.

Testavimui reikalingas modelis. Didelė modeliavimo praktika nėra būtina. Pakanka pradinių arba vidutinių modeliavimo žinių, nes nemaža dalis įrankių naudoja vidinius modeliavimo įrankius, kurie nemaža dalimi skiriasi nuo įprastų modeliavimo įrankių. Vartotojo sąsajos kūrimo patirtis ir sugebėjimas pasirinkti gerą abstrakcijos lygį yra labai naudinga modeliuojant. Su šiuo uždaviniu programuotojai ir sistemų analitikai susiduria kiekvieną dieną, todėl ši dalis jiems nebūna problemiška.

UML diagramų modeliavimas suteikia nemažai patirties, nes nemaža dalis įrankių turi galimybę modelius importuoti. Modelyje tenka aprašyti nemažą dalį detalaus testuojamos sistemos funkcionalumo, todėl prie UML tenka žinoti ir OCL. Šiuo atveju būtinos programavimo žinios. Pavyzdžiui, *MaTeLo* testavimo įrankis turi labai ištobulintą vidinį modeliavimo įrankį su detalia

grafine vartotojo sąsaja. Tačiau norint aprašyti detalų sistemos veikimą, būtina apsirašyti funkcijas, kurioms rašyti naudojama *Java* programavimo kalba.

Priklausomai nuo modeliavimo sistemos, gali reikti papildomų mokymų. Kiekvienas įrankis turi tam tikrų savybių, kurioms reikia laiko įsisavinti. Galima naudoti vartotojų vadovus, bet nemokomų įrankių vartotojų vadovai ir pavyzdžiai dažnai būna pasenę arba seniai atnaujinti, o komercinių įrankių – gana paviršutiniški ir reikalingi papildomi mokymai iš šalies. Galima pirkti mokymus, kurių metu būsimas testuotojas bus supažindintas su modeliavimo ypatumais. Bet mokymai dažniausiai būna bendri ir orientuoti į populiariausius įrankius. Efektyviausias ir veiksmingiausias būdas – turėti komandoje patyrusį MPT žinovą, kuris, atsiradus poreikiui, galėtų pakonsultuoti ar patarti, kokiais būdais galima pasiekti maksimalų sistemos padengimą išlaikant kuo aukštesnį abstrakcijos lygį.

Ne visos aplikacijos tinka modeliais paremtam testavimui. Būtina apsispręsti ir išsiaiškinti, ar verta taikyti modeliais paremtą testavimą. Jei testuojant būtina glaudi testuotojo sąveika su sistema, tai patartina rinktis rankinį testavimo būdą. Viskas priklauso nuo testuojamos sistemos. Šiame darbe didesnis dėmesys orientuotas į testavimo įrankius.

Pagal aukščiau pateiktus teiginius galima išskirti 4 pagrindines užduotis, su kuriomis susiduriama taikant modeliais paremtą testavimą:

1. Modelio kūrimas. Būtinas geras taikomųjų programų supratimas ir geros modeliavimo ir modeliavimo įrankių žinios.
2. Automatinis testų kūrimas. Testavimo tikslų pavertimas į testų rinkinius ir jų pavertimas į testavimui tinkamus testų skriptus. Šiai užduočiai reikalingas geras taikomųjų programų suvokimas. Reikia suprasti, kurias dalis testuoti, o kurias palikti.
3. Adapterio kūrimas. Kaip minėta 2.3. skyriuje, nei vienas MPT įrankis pilnai nesukuria adapterio, kuris pilnai susieja testų skriptus su testuojama sistema. Reikalingas testuotojo įsikišimas. Testuotojui būtina testavimo automatizavimo patirtis.
4. Testų analizavimas. Įprasta kiekvieno testavimo proceso užduotis, kuri reikalauja puikaus testuojamos sistemos funkcionalumo supratimo. Įrankiais galima palengvinti testų tikrinimą, bet testuotojas pats turi nuspręsti, kurioje vietoje įvyko klaida: modelyje, testavimo skripte, adapterio problemos ar testuojamoje sistemoje.

Šias visas užduotis padeda spręsti analizės metu pateikti testavimo įrankiai. Kituose skyriuose pateikiami sprendimai, susiję su aukščiau pateiktomis problemomis, ir siūlymai, kokiais būdais galima būtų pagerinti esamus testavimo įrankius.

3.2. Modeliavimo problematikos ir sprendimo būdai

Kaip minėta aukščiau, modeliavimas – viena pagrindinių ir išskirtinių modeliais paremto testavimo dalių. Priklausomai nuo sistemos, modeliavimas gali užtrukti nemažą testavimo laiko dalį. Būtina paminėti keturias pagrindiniais problemas, su kuriomis susidurta modeliuojant:

- Abstrakcijos lygio nustatymas.
- Didelis būsenų skaičius.
- Sudėtingų dalių modeliavimas.
- Modelio tikrinimas.

3.2.1. Abstrakcijos lygio nustatymas

Pirmas ir svarbiausias modeliavimo žingsnis yra gero abstrakcijos lygio nustatymas. Apie tai jau kalbėta šio darbo 2 skyriuje. Šis žingsnis visiškai priklauso nuo testuotojo sugebėjimų ir patirties. Reikalingas geras vartotojo sąsajos išmanymas. Šią užduotį daugiausia nulemia testavimo planas ir testuojamos sistemos specifikacija. Šiame žingsnyje testavimo įrankis niekaip negali padėti.

3.2.2. Didelis būsenų skaičius

Didelis būsenų skaičius yra viena pagrindinių modeliais paremto testavimo problemų. Įvairūs šaltiniai siūlo nemodeliuoti visos sistemos, pasirinkti tik svarbiausias dalis.

Vienas iš sprendimo būdų – sub-modelių kūrimas (paveikslas Nr. 6). Ne visi įrankiai palaiko šią savybę, bet jos nauda akivaizdi. Modeliavimo įrankis į pateiktą modelį gali žiūrėti kaip į vieną didelį arba kaip į tris atskirus. Atsiranda galimybė kurti testus tik iš norimo modelio.

Saugikliai, būsenų ir perėjimų žymėjimas yra dar vienas funkcionalumas, kuris puikiai padeda spręsti šią problemą. Beveik visi MPT įrankiai juos palaiko. Tai yra viena svarbesnių funkcionalumo dalių, kurias turėtų turėti kiekvienas įrankis. Nereikia persistengti su saugikliais, nes 6 pav. pateiktame A modelyje, uždėjus saugiklį ant „info“ būsenos, ji niekada nebuvo pasiekta. Šiuo atveju pats testuotojas turi apsispręsti ir apmąstyti, kokias vietas ir kaip galima būtų apriboti.

Būsenų ir perėjimų žymėjimas pasižymi tuo, kad atsiranda galimybė kurti testus, kuriuose būtų tik šios būsenos ar perėjimai. *TestOptimal* turi puikų funkcionalumą, kuriuo galima išimti modelio dalis, kurios yra pažymėtos. Tokiu būdu testai bus kuriami tik iš svarbiausių sistemos modelio dalių.

Jei modelis didelis, tai saugikliais, būsenomis ir perėjimais galima testavimą sukoncentruoti į norimas modelio dalis, taip pat atkurti galimus vartotojo veiksmus. *MaTeLo* įrankis modeliavimui naudoja Markovo grandines, kurios kiekvieną perėjimą pažymi tikimybėmis. Tokiu būdu atsiranda papildomas funkcionalumas, kuris padeda atkartoti galimus sistemos naudotojo veiksmus. Uždėjus perėjimui į būseną „info“ mažesnę tikimybę nei vienas testas šios būsenos nepasiekė. Iš dalies buvo atkartoti sistemos naudotojo veiksmai, bet testavimo tikslas, pereiti visas būsenas, nebuvo pasiektas. Šiuo atveju modelis yra labai mažas. Dideliuose modeliuose gali atsirasti didelės grupės nepasiekiamų būsenų. Niekada nereikia pervertinti savo ir įrankio galimybių.

Modelis turi atitikti testavimo tikslus, t. y. nebūtina modeliuoti visos sistemos arba kurti didelį visos sistemos modelį. Keli maži sistemos dalių modeliai pasitarnaus žymiai geriau nei vienas didelis. Šiame darbe visa Virtualių parodų sistema nebuvo modeliuojama. Modeliuojamos tik svarbiausios dalys. Taip pat dalys, kurios turėjo sudėtingesnį funkcionalumą.

Kaip matoma, ne viskas priklauso nuo įrankio. Šiai problemai spręsti iš įrankio galima reikalauti: sub-modelių kūrimo galimybių, perėjimų ir būsenų žymėjimo, saugiklių sistemos. Paskutiniai bus nesunkiai įgyvendinami, jei MPT įrankis turės vidinę skriptų programavimo kalbą arba naudos plačiai naudojamas modeliavimo kalbas (pavyzdžiui, *OCL*).

3.2.3. Sudėtingų dalių modeliavimas

Viena iš problematiškesnių modeliavimo dalių. Galima mėginti vengti jas modeliuoti, nes kai kuriais atvejais rankiniu būdu jos bus ištestuotos žymiai greičiau. Šiuo atveju įrankiai pateikia sprendimą: skriptų programavimo kalbos arba tradicinės programavimo kalbos (*Java*, *C#*).

Problema iškyla, kai perėjimams iš būsenų reikalingos tam tikros reikšmės (pavyzdžiui, slaptažodžio įvedimas, laukų užpildymas). Visi 2 skyriuje pateikti MPT įrankiai naudoja kažkokius modeliavimo įrankius. Vieną modelį pateikia *Java* programavimo kalba (*ModelJUnit*, *GraphWalker*), kiti *C#* (*Spec Explorer*), dar kiti XML formatu. *AGEDIS* įrankis yra sukūręs *AML* modeliavimo kalbą, kuria sprendžiamos visos iškilusios problemos.

Šiuo klausimu įrankiai teikia įvairiausių pasiūlymų. Pagrindiniai sprendimai ir reikalavimai MPT įrankiams: vidinė skriptų programavimo kalba, tradicinių programavimo kalbų įtraukimas į modelio aprašymą. *ModelJUnit* yra *JUnit* plėtinys, todėl, kaip modelis bus aprašytas, priklauso nuo testuotojo programavimo įgūdžių. *MaTeLo* leidžia apsirašyti įvairias funkcijas *Java* programavimo kalba.

3.2.4. Modelio tikrinimas

Modelio tikrinimas – galimybė patikrinti, ar modelis neturi nepasiekiamų būsenų ar viršūnių. Ne visi įrankiai turi tokį funkcionalumą. Modeliuojant Virtualią parodų sistemą, trūko papildomo funkcionalumo – modelio tikrinimo modeliavimo metu. Šis klausimas išsprendžiamas, kai yra galimybė iškart tikrinti modelį. Atlikus nors vieną žingsnį, yra galimybė peržiūrėti, kaip modelis atvaizduojamas grafiškai. Tokia savybe pasižymi *TestOptimal* ir *MATeLo* testavimo įrankiai. *SpecExplorer* galima įdiegti papildomus priedus, kuriais atvaizduojamas modelis. *ModelJUnit* gali atvaizduoti tik galutinį modelio variantą. *GraphWalker* naudoja išorinį modeliavimo įrankį ir patikrinimo galimybės neturi.

Šią problemą visiškai išsprendžia tie įrankiai, kurie turi gerai išvystytą vidinį modeliavimo įrankį su grafinių modelio atvaizdavimu arba naudoja gerai žinomus išorinius modeliavimo įrankius (pavyzdžiui, *UML*, *OCL*). Modelio netikslumai gali atsiliepti testų kūrimo metu, tuomet nepavyks pasiekti užsibrėžtų tikslų. Nemaža dalis klaidų būda dėl netinkamai sukurto modelio. Šioje vietoje įrankiai suteikia puikius sprendimo būdus. Kaip gerai bus sukurtas modelis, priklauso nuo testuotojo. Modeliais paremto testavimo įrankis yra tik pagalbininkas.

3.3. Modelio padengimo kriterijai ir testai

Padengimo kriterijai naudojami dviem pagrindiniams tikslams pasiekti: testų tinkamumui nustatyti ir nuspręsti, kada testavimą reikia baigti. Kuo daugiau MPT įrankis testavimo kriterijų palaiko, tuo jis yra geresnis.

Skyriuje daugiausia dėmesio skiriama testavimo kriterijų kūrimui ir tobulinimui, kriterijų panaudojimui ir gautų testų analizei.

3.3.1. Kriterijų kūrimo metodika

Šiame skyriuje bus pateikti metodai, kaip galima įgyvendinti padengimo kriterijus savo įrankyje ir į ką daugiausia atkreipti dėmesį. Čia nebus minimas nei vienas 2 skyriuje minėtas įrankis. Ši metodika pakankamai bendra. Idėja gan paprasta, lengvai pritaikoma ir labai veiksminga.

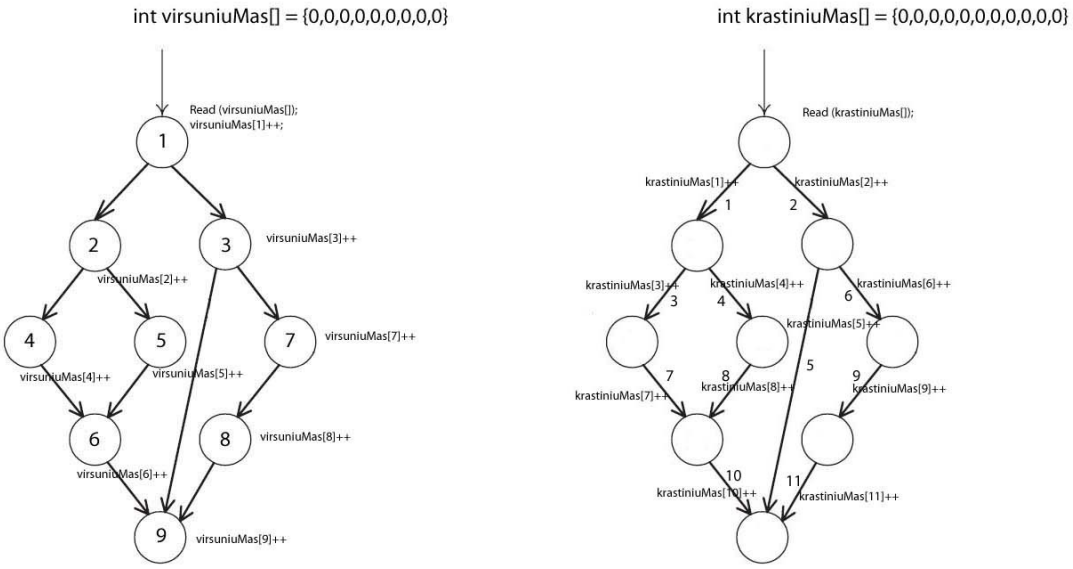
Šis įrankis neturėtų keisti pačios programos. Jo tikslas – rinkti reikiamą informaciją. Kiekvienoje viršūnėje arba perėjime turėtų atsirasti papildoma kodo eilutė, kuri praneštų, kad tam tikra modelio dalis buvo pasiekta.

3.3.1.1. Viršūnių ir perėjimų padengimas

Pats paprasčiausias ir lengviausiai įgyvendinamas kriterijus – viršūnių padengimas. Galima kiekvienoje būsenoje pridėti kelias kodo eilutes, kurios reaguos, jei būsena bus pasiekta. Kadangi į modelį galima žiūrėt kaip į grafą, tai galima naudoti kitą būdą – kiekvienai būsenai (viršūnei) suteikti unikalų numerį. Informaciją apie pereitas viršūnes galima laikyti kažkokiame masyve: *virsuniuMas[]*.

Tokiu būdu viršūnių padengimo masyvas turėtų būti nuolat atnaujinamas, kai tik sukuriamas testas. Testui perėjus viršūnę, tą viršūnę atvaizduojantis laukas bus pakeistas. Jei po testų sukūrimo nors vienas laukas bus nekeistas, reiškia ta viršūnė nebuvo pasiekta. Tokiu būdu reikia kurti papildomus testus arba tikrinti modelį. Jei viršūnė jau buvo pasiekta, tai galima pridėti dar papildomos informacijos. Tokiu būdu galima sekti, kiek kartų viršūnė buvo pasiekta. Paveiksle Nr. 11 (kairėje) pateiktas grafinis vaizdas, kaip tai galėtų atrodyti. Remiantis pavyzdžiu, bus skaičiuojama, kiek kartų viršūnė buvo aplankyta. Galima sukurti viršūnės objektą, kuriame bus laikoma įvairi informacija apie ją (pavyzdžiui, kiek kartų koks testas aplankė viršūnę).

Pati idėja labai paprasta. Informaciją apie viršūnę galima įrašyti prieš ją pasiekiant arba ją pasiekus. Grafo perėjimo algoritmas ir pats modelis visai nesikeičia. Atsiranda tik papildoma funkcija/metodas, kuris atlieka stebėjimus.



11 pav. **Viršūnių ir kraštinių padengimo metodas**

Kraštinių padengimo metodas šiek tiek sudėtingesnis, bet idėja lygiai tokia pati. Kiekvienam perėjimui suteikiamas unikalus numeris (paveikslas Nr. 11, dešinėje). Sukuriamas kraštinių masyvas (*krastiniuMas[]*), kuriame saugoma visa informacija apie jas. Visa informacija turėtų būti

įrašoma automatiškai. Papildomos kodo eilutės, reikalingos surinkti informaciją, irgi turėtų būti įrašomos automatiniu būdu. Šis reikalavimas privalomas ir viršūnių padengimo metodui.

Perėjimų padengimas ypatingas tuo, kad gali tekti nuskaityti visą masyvą. Jei testas pereis 6 kraštinę, tai jis pereis 9 ir 11[9]. Jei modelis didelis, viso kraštinių masyvo užkrovimas nėra logiškas ir gali kainuoti daug kompiuterio resursų. Kad būtų sumažintos sąnaudos, galima kraštinių masyve laikyti papildomą informaciją, dėl to bus užkraunama tik reikalinga masyvo dalys. Kraštinių padengimo metodas tampa sudėtingesnis, nes atsiranda poreikis pereiti modelį.

Šią metodiką galima taikyti kiekvienam MPT įrankiui. Kokia papildoma informacija bus renkama ir kaip ji pateikiama, priklauso nuo įrankio kūrėjų.

3.3.1.2. Duomenų padengimas

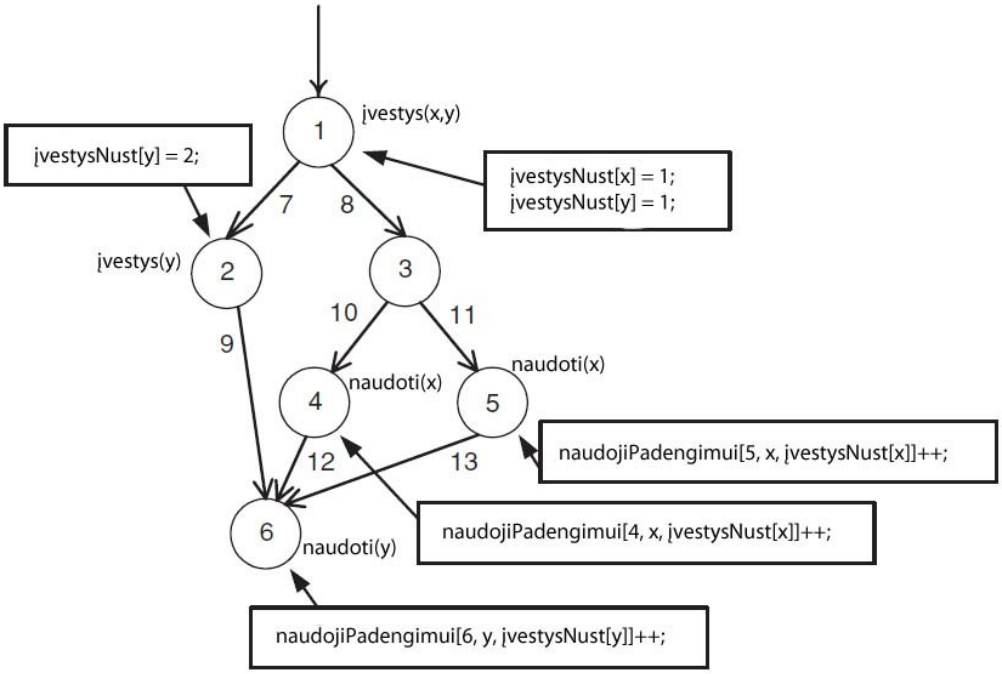
Duomenų padengimo kriterijus žymiai sudėtingiau įgyvendinamas. Tai yra 2.2.1 skyriuje paminėtas duomenų padengimo kriterijus, kuris tikrina ar visos norimos įvestys yra panaudotos. Duomenų judėjimui sekti reikia stebėti du masyvus: laiko įvesties priskyrimo būsenas/perėjimus ir kur įvestys buvo panaudotos. Kiekviena būsena ir perėjimas turi turėti unikalų numerį. Naudojami du masyvai (paveikslas Nr. 12).

Masyvas *įvestysNus[]* laiko informaciją, kurioje būsenoje ar perėjime tam tikra įvestis buvo priskirta. Šis masyvas laikys informaciją, kurioje modelio vietoje įvestis buvo priskirta. Kitas masyvas laiko informaciją, kur esamos įvestys yra panaudojamos (*naudojiPadengimui[i, x, įvestusNust[x]*), *i* – būsena, *x* – kokia įvestis, *įvestisNust[x]* – kokioje būsenoje įvestis priskirta.

Paveiksle Nr. 12 įvestis *y* atsiranda 2 būsenoje ir panaudojama 6 būsenoje. Jei testas pereis 1, 3, 5 ir 6 būsenas, tai įvestys priskiriamos 1 būsenoje, o panaudojamos 5 ir 6 būsenose. Tokiu būdu padengiamos visos įvestys. Tai vienas iš paprastesnių būdų sekti duomenų padengimą. Didesniam duomenų kiekiui arba duomenų padengimui poromis gali reikti daugiau informacijos. Bet pagrindinė idėja išlieka ta pati.

Galimos įvestys gali būti laikomos suskirstytos į įvesčių aibes. Tokiu būdu nelieka masyvo *įvestysNust[]*. Kaip viskas bus įgyvendinta, gali keistis. Skirtingi įrankiai skirtingai apsirašo galimas įvestis, bet įvesčių padengimo mechanizmas lieka toks pats.

[9] Pavyzdiniame grafe laikome, kad viršūnė *1* yra pradinė, o viršūnė *9* galutinė. Tokiu atveju visi testai turės pasibaigti *9* viršūnėje.



12 pav. **Duomenų padengimo metodika**

Sudėtingesniems duomenų padengimo kriterijams gali prireikti ir daugiau informacijos. Pagrindinis šios metodikos tikslas – nekeisti esamo modelio. Pridedama tik papildoma informacija, kuri visiškai nekeičia modelio.

Įrašyta informacija vėliau gali būti naudojama testavimui patikrinti ar įvairioms ataskaitoms.

Aukščiau išvardintų metodų įgyvendinti komerciniuose įrankiuose tikrai nepavyks, bet atviro kodo įrankiuose juos galima taikyti, nes testuotojas gali matyti visą įrankio kodą. Šiais metodais galima pasinaudoti sukuriant savo papildomą įrankį, kuris atliktų modelio padengimo skaičiavimus.

3.3.2. Testų kūrimas

3.3.1. skyriuje pateikta metodika, kokias būdais galima kurti testų padengimo kriterijų, jei jo nepalaiko testavimo įrankis. Testuotojui atsiranda galimybė pačiam susikurti kriterijų arba modifikuoti esamą, kad būtų lengviau pasiekiamas norimas testavimo tikslas. Visi testavimo įrankiai turi tam tikrus testų padengimo kriterijus. Kuo jų daugiau palaiko, tuo testavimo įrankis yra geresnis. Tai galioja ne tik MPT įrankiams.

Testų kriterijai tiesiogiai veikia testų kūrimą. Juos naudojant, modelio perėjimo algoritmui iškeliamos norimos užduotys, kurias jis turi padaryti kurdamas testus. Jei modelio perėjimo algoritmui bus pasakyta, kad 12 paveikslo modeliui sukurti 5 testus, tai dar nereiškia, kad jis juos kurdamas pereis visas viršūnes ar perėjimus. Šiuo atveju gerai tinka testų kriterijai.

Kaip jie bus panaudoti, priklauso nuo testuotojo ir testavimo plano, kuriame nurodyti testavimo tikslai. Bet testų kriterijus yra pakankamai galingas įrankis, kuris ne tik palengvina testų kūrimą, bet ir padeda lengviau pasiekti testavimo tikslus.

Galima išskirti tris būdus, kaip galima pagerinti testų kūrimą.

Tiksli testų specifikacija – testų kūrimo valdymas. Tai formalus dokumentas, kuriame nurodoma, kokius testus reikia kurti. Pavyzdžiui, aprašoma, kuriuos modelio perėjimus reikia blokuoti, kad visas dėmesys būtų skirtas norimai daliai arba nurodoma, kuriuos perėjimus reikia būtinai pereiti.

Pavaizduoti norimiems tikslams galima naudoti tuos pačius modeliavimo įrankius, kaip ir kuriant abstraktų sistemos modelį. Galima naudoti UML ar kokią kitą modeliavimo kalbą. Tikslus galima atvaizduoti baigtiniu automatu, paprastomis išraiškomis, sekų diagramomis, logikos formulėmis arba Markovo grandines.

Tiksli testų specifikacija suteikia aiškią testų kūrimo kontrolę. Tokia specifikacija gali būti labai varginanti ir sukelti nemažai nepatogumų, nes reikalingas papildomas laikas jai nagrinėti ir tikrinti.

Šiam uždaviniui spręsti įrankiai siūlo saugiklių sistemą arba modelio žymėjimo sistemą. Sužymimos testavimo tikslui reikalingos viršūnės ar perėjimai ir leidžiama testus kurti tik iš pažymėtų dalių. Kai kurie įrankiai turi galimybę grafiškai atvaizduoti sužymėtas modelio dalis. Tokiu būdu palengvinamas testuotojo darbas.

Statistinis testų kūrimo metodas – testų kūrimas pasitelkus statistinius metodus. Modeliais paremtame testavime statistika paremtas testų kūrimas vykdomas iš modelio, kuris turi apribojimus, dėl kurių modelis atvaizduoja galimą sistemos naudotojo elgesį. Šis būdas lengvai pritaikomas modeliavimui naudojant Markovo grandinės (*MaTeLo* įrankis sistemą atvaizduoja Markovo grandinėmis). Tokiu būdu testams kurti galima naudoti atsitiktinių perėjimų algoritmą, kuris kiekvieną žingsnį rinksis pagal nustatytas tikimybes. Tokiu būdu mažiausias tikimybes turintys perėjimai bus rečiausiai pereiti, o turintys didžiausias tikimybes bus pereiti pirmiausia. Taip kuriami testai atvaizduos galimą sistemos panaudojimą.

Galima naudoti du modelius: pirmas atvaizduoja statistinį sistemos perėjimo būdą, antras atvaizduos paprastą sistemos perėjimą. Šiuo atveju pirmas modelis tampa testų pasirinkimo kriterijumi. Naudojant šitą būdą, atsiranda galimybė analizuoti tikėtinas galimybes (*angl. test oracle*).

Šitą metodą galima pritaikyti ir duomenų padengimo kriterijui. Kiekvienai įvesčiai galima suteikti norimą tikimybę. Testo kūrimo metu pirmiausia bus pasirenkamos reikšmės su didžiausia tikimybe. Šis metodas tampa geru testų kūrimo įrankiu.

Jei sistemos modeliavimui nenaudojamos Markovo grandinės, tai galima pasitelkti skriptų programavimo kalbas, kuriomis galima apsirašyti norimas tikimybes arba sąlygas, kurioms esant, galima daryti norimus veiksmus.

Testų kriterijų apjungimas – kelių testų kūrimo kriterijų naudojimas. Daugumą kriterijų galima apjunginėti. Visi kriterijus palaikantys įrankiai turi tokį funkcionalumą. Jų įgyvendinimas per daug problemų nesukelia, nes automatiniam testų kūrimo įrankiui liepiama vykdyti kelias užduotis vienu metu. Remiantis 3.3.1. pateikta kriterijų kūrimo metodika, galima nesunkiai apjungti visų viršūnių ir duomenų padengimą. Šie du kriterijai veiks lygiagrečiai.

Norint padengti visus kriterijų reikalavimus, gali tekti sukurti daug daugiau testų. Kadangi testai kuriami automatiniu būdu, tai nesukelia didelių problemų.

Kitas svarbus testų kūrimo aspektas – modelio perėjimo algoritmai. Galima pridėti, kad tai yra tiek pat svarbus testavimo įrankio funkcionalumas, kiek ir testavimo kriterijai. Visi paminėti modeliais paremtos testavimo įrankiai palaiko atsitiktinį modelio perėjimą. Jo įgyvendinimas nėra sudėtingas. Į modelį reikia žiūrėti kaip į grafą. Tokiu būdu atsiranda galimybė taikyti visus grafų teorijoje naudojamus grafo perėjimo algoritmus. Populiariausias ir geriausiai žinomas grafo perėjimo algoritmas – kinų paštininko problema.

Modeliui pereiti įrankis turėtų pateikti bent du galimus algoritmus. Vienas – atsitiktinis modelio perėjimas, kitas – koks nors „protingas“ algoritmas, kuris pasirenka trumpiausią arba realiausią kelią.

Jei testuojamoje sistemoje būtinas aukštas saugumo lygis, tai visi galimi variantai yra būtinas reikalavimas. Tokiam testavimui būtinas atsitiktinis modelio perėjimo algoritmas. Tokiu būdu sukuriama daugybė testų. Daugybei testų optimizuoti būtinas testuotojo įsikišimas, kad nepradėtų kartotis tos pačios būsenos su tomis pačiomis įvestimis.

Kitame skyriuje praktiškai nagrinėjami algoritmų ir kriterijų privalumai.

3.4. Klaidų suradimas

Automatinis testų kūrimas iš abstraktaus sistemos modelio yra išskirtinė modeliais paremto testavimo savybė, kurios neturi kiti testavimo procesai. Kokiu būdu bus kuriami testai (t. y. kaip pereinamas modelis), pasako testavimo kriterijai ir modelio perėjimo algoritmai. Kiekvienas geras testavimo įrankis turi palaikyti kelis testavimo kriterijus ir modelio perėjimo algoritmus.

Nedidelis modelių perėjimo algoritmų ir testavimo kriterijų tyrimas buvo atliktas 2.2.3. skyriuje. Paprasto pavyzdžiu parodyti modelio algoritmų privalumai. Šiame skyriuje pateikiamas

didesnis kriterijų ir modelio perėjimo algoritmų tyrimas. Dėl laiko stokos tyrimui pasirinkti 4 įrankiai. Du komerciniai, du nemokami.

3.4.1. Sistemos modelis ir duomenys

Tyrimui atlikti pasirinkta Virtualių parodų sistemos tvarkymo dalis. Tai yra praplėstas aukščiau pateiktas abstraktus sistemos modelis (6 pav.). Modelis detalizuotas. Įtrauktos papildomos svarbos būsenos, perėjimai ir duomenys.

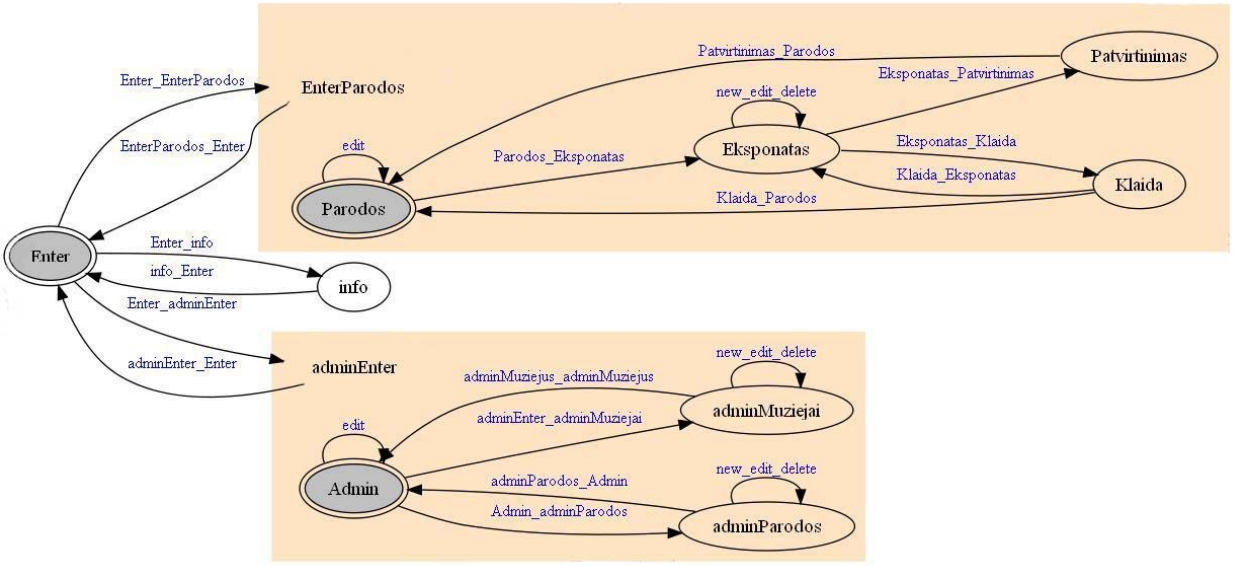
Testavimui atlikti pasirinkti 4 duomenų rinkiniai. Jie panaudojami būsenoje „*Eksponatas*“. Vienas duomenų rinkinys yra klaidingas.

Visi įrankiai, išskyrus *GraphWalker*, turi vidinius modeliavimo įrankius. *GraphWalker* leidžia importuoti modelį, sukurtą su yEd[10] modeliavimo įrankiu.

Sistemai modeliuoti panaudoti sub-modeliai. Testavimo metu galima testuoti tik vieną dalį arba visą modelį. Kuriant testus, į sistemą žiūrėta kaip į vieną modelį.

Pilka spalva ir dviguba linija apibrėžtos būsenos yra pradinės ir galutinės. Kurioje būsenoje testas pradėtas kurti, toje jis turi pasibaigti.

Modeliuojant sistemą specialiai įveltos 4 klaidos: viename duomenų rinkinyje ir būsenose „*info*“, „*adminMuziejai*“, „*adminParodos*“.



13 pav. **Virtualių parodų sistemos tvarkymo abstraktus modelis**

[10] Daugiau informacijos adresu http://www.yworks.com/en/products_yed_about.htm

Šio tyrimo metu modeliavimas užėmė daugiausia laiko. Daugiau nei pusė testavimo laiko sugaišta kuriant, tikrinant ir tobulinant abstraktų sistemos modelį.

3.4.2. Testų kūrimas ir paleidimas

Yra keli būdai kaip galima kurti testus. Galima per modelį pereidinėti atsitiktiniu būdu arba naudoti „gudrius“ algoritmus. Visi pasirinkti įrankiai turi galimybę pereiti modelį atsitiktiniu būdu arba naudoja kokį nors algoritmą. Pavyzdžiui, *TestOptimal* naudoja kinų paštininko algoritmą. *MaTeLo* sistemą atvaizduoja Markovo grandinės principu, taip pat turi įvairius modelio perėjimo algoritmus.

Visi įrankiai palaiko būsenų ir perėjimų padengimo kriterijų. Modelio perėjimo algoritmui suteikiama papildoma užduotis. Didžiausia problema yra su duomenų padengimu. Nei vienas įrankis nepalaiko duomenų padengimo kriterijaus. Ši problema sprendžiama modeliavimo metu. *MaTeLo* ir *ModelJUnit* testavimo įrankiai leidžia duomenis apsirašyti modeliavimo metu. *TestOptimal* reikia apsirašyti tris papildomas būsenas, kurios savyje laiko reikalingus duomenis (modelyje atsiranda keturios „Eksponatas“ būsenos). *GraphWalker* įrankiu šios užduoties įvykdyti nepavyko.

Sukurtus testų rinkinius įrankiai pateikia XML formatu arba *Java* programavimo kalba. Kadangi testams paleisti naudojamas *Selenium IDE Firefox* priedėlis[11], tai visi testai konvertuojami šiam įrankiui tinkamu formatu. Paleidimas vyksta automatiniu būdu.

3.4.3. Klaidų radimo rezultatai

Pateikti trys rodikliai, kuriais remiantis atliekamas tyrimas: testų skaičius, modelio padengimas (skliausteliuose – duomenų padengimas) ir klaidų suradimas. 3.4.1. skyriuje paminėta, kad sistemoje specialiai paliktos 4 klaidos. Testų skaičiaus ir padengimo rodikliai tiesiogiai susiję, nes skaičiuojama, kiek testų reikia padengti visą modelį ar esamus duomenis.

100 procentų padengimas pakankamai daug pastangų ir išteklių reikalaujantis tikslas. 100 procentų padengimo nepavyko visur pasiekti. Pats paprasčiausias tikslas (būsenų padengimas ir atsitiktinis modelio perėjimas) ne visur pasiekė 100 procentų. Tik vienas testavimo įrankis padengė visus duomenų rinkinius. Priežastis gana paprasta – tik 2–3 testai iš 7 ar 8 pasiekia reikiamą būseną. Testų kūrimo metu duomenų rinkiniai buvo papildomas reikalavimas. Duomenų rinkiniai buvo aprašyti pačiame modelyje ir testų kūrimo mechanizmas neturėjo funkcionalumo padengti visus

[11] Kitas būdas yra naudoti *Selenium Webdriver*: http://code.google.com/p/selenium/.

duomenis. Parašytas paprastas skriptas buvo atsakingas, kad duomenų rinkinys nebūtų pateikiamas antrąkart, kol visi rinkiniai nepatikrinti bent vieną kart.

MaTeLo ir *GraphWalker* neperėjo visų būsenų, nes vienai būsenai pasiekti uždėti apribojimai. Tai yra modeliavimo klaida. *GraphWalker* – nepalaiko modelio patikrinimo, kas apsunkina modeliavimo procesą. *MaTeLo* – uždėta maža būsenos pasiekimo tikimybė.

Visų kraštinių padengimo kriterijus yra sudėtingesnis. Visoms kraštinėms padengti su atsitiktiniu modelio perėjimo algoritmu reikia žymiai daugiau testų.

TestOptimal testavimo įrankiui reikėjo daugiausia testų. Bet modelis, kurtas su *TestOptimal*, turėjo daugiau būsenų. Net ir 112 testų nepadengė visų kraštinių. Nors ir nebuvo jokių apribojimų, bet dvi kraštinės nebuvo pasiektos. Kitų dviejų įrankių problema ta pati, kaip su būsenų padengimu: per dideli apribojimai ir per maža perėjimo tikimybė.

Naudojant „gudrų“ algoritmą, gaunami visai kitokie rezultatai. Reikia nedidelio skaičiaus testų, kad būtų pasiektas norimas tikslas. Tik vienas įrankis įvykdė visus testavimo tikslus. *TestOptimal* įrankis turėjo papildomas būsenas su duomenų rinkiniais, todėl visus kartus pasiekiama šimtaprocentinis duomenų padengimas. Su kitais įrankiai sukūrus daugiau testų, duomenų padengimo uždavinys irgi buvo įvykdytas. Viskas priklauso nuo to, kokie kriterijai ir algoritmai pasirenkami. Duomenims padengti pasitelktos vidinės skriptų programavimo kalbos ir tradicinių programavimo kalbų palaikymas įrankyje.

8 lentelė. **Virtualių parodų sistemų testavimo rezultatų suvestinė**

Įrankis	Rodiklis	Būsenos ir Atsitiktinis perėjimas	Kraštinės ir atsitiktinis perėjimas	Kraštinės ir „Gudrus“ algoritmas
MaTeLo	Testai	8	36	10
	Padengimas	89 % (75 %)	91 % (100 %)	100 % (100 %)
	Surastos klaidos	3	4	3
TestOptimal	Testai	12	112	12
	Padengimas	100 % (100 %)	94 % (100 %)	100 % (100 %)
	Surastos klaidos	4	**5**	4
ModelJUnit	Testai	7	35	10
	Padengimas	100 % (50 %)	100 % (100 %)	100 % (100 %)
	Surastos klaidos	3	4	4
GraphWalker	Testai	7	26	9
	Padengimas	89 % (nėra)	91 % (nėra)	91 % (nėra)
	Surastos klaidos	2	3	3

Pasiektas padengimo rezultatas pakankamai geras. Truputį pataisius modelius ir įdėjus papildomą informaciją, galima pasiekti visišką sistemos padengimą. Didelę įtaką turėjo geras sistemos funkcionalumo išmanymas.

Kitas rodiklis – surastos klaidos. Mažiausiai klaidų rasta naudojantis *GraphWalker* sukurtais testais. Tai nereiškia, kad šitas įrankis yra prastesnis už kitus. Tai atviro kodo projektas su gana minimalia vartotojo sąsaja. Bet yra galimybė pačiam jį tobulinti. Remiantis 3.3.1. skyriuje pateiktais metodais, galima įrankį pritaikyti savo reikmėms. Didžiausia įrankio problema – neįmanomas modelio tikrinimas. Dėl to buvo palikta viena didelė klaida modeliavimo metu. Įrankis netikrino duomenų padengimo, todėl su kitomis sąlygomis nerado visų klaidų.

MaTeLo ir *ModelJUnit* su pirmomis sąlygomis nerado visų 4 klaidų. Testai būseną „Eksponatas“ pasiekė atitinkamai 3 ir 2 kartus. Nesurasta klaida duomenų rinkiniuose. Tyrimo metu, duomenų padengimas priskirtas kaip šalutinis uždavinys, todėl įrankis nesistengė pasiekti 100 procentų padengimo.

Naudojant kitas sąlygas, gauti rezultatai nenustebino. Su „gudriu“ algoritmų, įrankiai *MaTeLo* ir *ModelJUnit*, perėjimus 100 procentų padengia su 7 testais. Bet nepasiekiamas duomenų padengimas. Tam tikslui reikalingi dar 3 papildomi testai. Gera įrankio ataskaitų sistema padeda lengviau siekti reikiamų rezultatų.

TestOptimal su kraštinių padengimu ir atsitiktiniu perėjimu, surado 5 klaidas. Paleidinėjant didelius kiekius testų iš eilės, buvo pastebėta, kad testuojamos sistemos veikimas sulėtėja ir po kiek laiko ji sustoja veikusi. Panagrinėjus sistemos kodą atidžiau, buvo rasta sena sistemos klaida. Sistema sukurdavo daug objektų ir jų neuždarydavo. Su mažesniais testų kiekiais ši problema neatsiranda. Kartais dideli testų kiekiai išeina į naudą.

3.5. Siūlymai ir rezultatai

Modeliais paremtas testavimo procesas nuo kitų testavimo būdų skiriasi dviem etapais: modeliavimas ir automatinis testų kūrimas (paveikslėlis nr. 3). Didžiausias dėmesys skirtas šių dviejų žingsnių įgyvendinimui ir iškilusių problemų sprendimams.

9 lentelėje pateikiama šio darbo metu iškilusios problemos ir trumpas jų sprendimo būdas. Didžiąją dalį problemų sprendžia MPT įrankiai. 3.4 skyriuje pateikto tyrimo metu atliktas realios sistemos testavimas naudojant kelis įrankius, pateikiama galimi sprendimo būdai.

Didelis būsenų skaičius sprendžiamas panaudojus sub-modelius, saugiklius, būsenų žymėjimą ir skriptų programavimo kalbas. Skriptų programavimo kalbos turi nemažą testų kūrimo reikšmę. Jomis galima modeliuoti sudėtingą funkcionalumą.

9 lentelė. **MPT iškylančios problemos ir sprendimo būdai**

Iškilę sunkumai	Sprendimo būdas
Didelis būsenų skaičius	Sub-modeliai Saugikliai Būsenų žymėjimas Skriptų programavimo kalbos
Sudėtingas sistemos funkcionalumas	Skriptų programavimo kalbos Tradicinių programavimo kalbų taikymas
Modeliavimo klaidos	Modelio tikrinimas
Modelio padengimas	Būsenų/perėjimų padengimo kriterijai Modelio perėjimo algoritmai
Testų kūrimas	Tiksli testų specifikacija Statistinių testų kūrimo metodas Testų kriterijų apjungimas Modelio perėjimo algoritmai

Kai kuriuose įrankiuose trūko modelio tikrinimo galimybės, tai vėliau iššaukė modelio klaidas ir neteisingus testus.

Yra gausybė padengimo kriterijų ir jų pritaikymo galimybių su realia sistema. Kiekvienas įrankis turi turėti kelis padengimo kriterijus ir modelio perėjimo algoritmus. Jų nauda matoma lentelėje Nr. 8. Kuo įrankis turi daugiau kriterijų, tuo jis naudingesnis. Atsiranda galimybė kurti įvairesnius testus. Sumažėja testavimo laikas, nes testuotojui nebereikia papildomai apsirašyti skriptų.

Aprašyta metodika, kokiais būdais galima pagerinti turimus testavimo įrankius juose įdiegiant arba patobulinant padengimo kriterijus. Šią metodiką galima taikyti kiekvienam MPT įrankiui. Jos įgyvendinimas yra labai paprastas. Papildoma programa į modelį įtraukiama reikalinga kodo eilutė, kuri renka informaciją (paveikslas Nr. 13).

```
public String state (x, y)
{
        String z;
        if (x = y){
                return z;
        }
        return x+y;
}
```

```
public String state (x, y)
{
        String z;
        if (x = y){
                vursuniuMas[nr]++
                return z;
        }
        return x+y;
}
```

13 pav. **Padengimo metodo taikymas**

Papildoma kodo eilutė visiškai nekeičia esamo kodo. Modelis lieka nepakitęs. Kriterijai vienas su kiti nesąveikauja, todėl atsiranda galimybė juos apjungti.

Teisingais algoritmais ir kriterijais galima sukurti įvairius testus ir pasiekti modelio šimtaprocentinį padengimą.

Aukščiau pateiktas tyrimas parodo, kad testavimo kriterijai, modelio padengimo algoritmai, skriptų arba tradicinės programavimo kalbos ir visų metodų taikymas gali sukurti labai įdomius ir naudingus testus. Tokį funkcionalumą turintis MPT įrankis tampa puikiu pagalbininku siekiant testavimo tikslų.

Išvados ir rezultatai

Darbe pateikiamas detalus modeliais paremto testavimo apibūdinimas, privalumai ir trūkumai. Palyginama su tradiciniais testavimo būdais. Modeliais paremtas testavimo procesas išsiskiria dviejomis dalimis – sistemos modelio ir automatiniu testų kūrimu.

Testavimo įrankių tyrimo metu pateikiama detali 9 įrankių analizė. Testavimo įrankiams įvertinti pasirinkta daugiau nei 10 vertinimo kriterijų – įvertinamos įrankių galimybės nuo modeliavimo iki testų paleidimo. Vertinimo kriterijai suskirstyti į tris dideles aibes: modeliavimo, testų kūrimo ir testų paleidimo kriterijai. Darbo metu išbandyta didžioji dalis įrankių funkcionalumo. Kiekvienas įrankis turi savų privalumų ir trūkumų.

Modeliuoti dauguma įrankių naudoja išorinius modeliavimo įrankius. Didžioji dalis palaiko sub-modelius. Beveik visi palaiko modelio tikrinimą. Tai yra gan svarbūs kriterijai kuriant abstraktų sistemos modelį.

Pateikiamas testavimo kriterijų palaikymas įrankiuose (6 lentelė). Šie kriterijai ženkliai lemia, kaip bus kuriami testai. Kuo daugiau kriterijų įrankis palaiko, tuo jis geresnis. Visi įrankiai palaiko du ar daugiau modelio perėjimo algoritmų. Algoritmai turi didžiulę įtaką testų kūrimo kokybei. Šiems uždaviniams spręsti dalis įrankių siūlo vidines skriptų programavimo kalbas arba tradicinių programavimo kalbų pritaikymą.

Testų skriptams kurti ir paleisti didžioji dalis įrankių naudoja išorinius įrankius. Ši modeliais paremto testavimo proceso dalis yra lygiai tokia pati, kaip ir tradiciniuose testavimo procesuose, todėl dauguma įrankių naudoja gerai žinomus testų skriptų paleidimo įrankius. Visi įrankiai palaiko „Offline“ testavimo būdą.

Darbe pateikiamos pagrindinės testavimo metu iškylančios problemos ir jų sprendimo būdai, kokių žinių ir sugebėjimų reikia adaptuojant šį testavimo būdą. Atlikus detalią kriterijų ir modelio perėjimo algoritmų analizę, pateikiami kriterijų apjungimo privalumai. Naudojant įvairius kriterijus, galima valdyti testų kūrimo procesą ir siekti šimtaprocentinio modelio padengimo. Šiam tikslui pasiekti būtina puikiai suprasti testuojamą sistemą ir gerai žinoti modeliais paremto testavimo pritaikymo galimybes.

Pateikiama ir detaliai išanalizuojama kriterijų kūrimo metodika, kurią galima taikyti testavimo įrankiuose. Ši metodika suteikia bendrą supratimą, kaip galima kurti testavimo kriterijus taip pagerinant testavimo įrankį. Jos pagrindinis privalumas – testų padengimo kriterijaus atsiradimas visai nekeičia esamo modelio, atsiranda galimybė taikyti kelis kriterijus lygiagrečiai.

Yra daugybė modeliais paremtam testavimui skirtų įrankių, kurie palaiko visą procesą. Nemaža dalis problemų atsiranda testo skriptų paleidimo metu, kai yra reikalingas adapteris. Darbas daugiausia orientuotas į modeliavimą ir testų kūrimą, todėl ši dalis į darbo tiklus neįtraukta. Tai nėra tik modeliais paremto testavimo problema. Su šia problema susiduria visi testavimo būdai. Kaip visiškai automatizuoti šį procesą, nėra pateikiama. Dėl skirtingo sistemų sudėtingumo ir įvairovės modeliavimas tampa gan sudėtingas procesas, kuris atliekamas rankiniu būdu. Šio proceso nedidelės dalies automatizavimas padėtų spręsti šią problemą. Tam tikrų šablonų atsiradimas labai palengvintų ir sutrumpintų šį darbą. Kol kas nei vienas įrankis ar atliktas tyrimas nesiūlo jokių įmanomų dalinio automatizavimo sprendimo būdų.

Literatūros sąrašas

[Bur02] Ilene Burnstein. Practical Software Testing, A Process-Oriented Approach, 2002, p. 24–326.

[UL07] Mark Utting, Bruno Legeard. Practical Model-Based Testing: A Tools Approach. 2007, p. 455.

[RR02] Steven Rosaria and Harry Robinson. Applying Models in your Testing Process. URL: www.geocities.com/harry_robinson_testing/ApplyingModels.pdf 121 KB, 2002.

[BRN04] Mark Blackburn, Robert Busser, Aaron Nauman. Why Model-Based Test Automation is Different and What You Should Know to Get Started. 2004, p. 16.

[BGL03] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, M. Utting. A subset of precise UML for Model-based Testing. URL: portal.acm.org/citation.cfm?id=1291545, 210 KB. 2003.

[FM00] Marcelo Fantinato, Mario Jino. Applying Extended Finite State Machine and Scenario in Software Testing. URL: www.springerlink.com/index/YJX91QCV63MK6GTK.pdf 101 KB. 2000.

[HN02] A. Harman, K. Nagin. The AGEDIS Tools for Model Based Testing. 15 psl. 167 KB. 2002.

[AG02] AGEDIS consortium. Model Based Testing Generetion Tools. 20 psl. 584 KB. 2002.

[HN06] A. Harman, K. Nagin. Model Based Software Testing. GOTCHA-TCBeans AGEGIS. 20 psl. 358 KB. 2006.

[PJ08] Paul Ammann, Jeff Offutt. Introduction to Software Testing. 2008, p. 346.

[PSA04] A. Pretschner1, O. Slotosch, E. Aiglstorfer, S. Kriebel. Model-based testing for real. 2004, p. 18.

[DJK99] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton. Model-Besed Testing in Practise. 1999.

[AOA02] Anneliese A. Andrews, Jeff Offutt, Roger T. Alexander. Testing Web Applications by Modeling with FSMs. URL: cs.gmu.edu/~offutt/rsrch/papers/webtest.pdf 225 KB. 2002.

[Die05] Reinhard Diestel. Graph Theory. 94-168 psl. 2005.

[Kat07] Mika Katara. Model-based GUI testing of Symbian S60. URL: http://www.tol.oulu.fi/projects/wg4/minutes/WG4_0507_TTY.pdf 522 KB. 2007.

[Puo08] Olli-Pekka Puolitaival. Model-based testing tools. URL: www.cs.tut.fi/tapahtumat/testaus08/Olli-Pekka.pdf 615 KB. 2008

[Kul09] Andres Kull. Model-Based Testing of Reactive Systems. 3,93 MB. 101 psl. 2008.

[MP09] Janne Merilinna, Olli-Pekka Puolitaival. Using Model-Based Testing for Testing Application Models in the Context of Domain-Specific Modelling. URL: www.dsmforum.org/events/DSM09/Papers/Merilinna.pdf 172 KB. 200.

[UPL06] Utting, M., Pretschner, A. and Legeard, B. Ataxonomy of model-based testing. URL: www.cs.waikato.ac.nz/pubs/wp/2006/uow-cs-wp-2006-04.pdf 208 KB. 2006.

[Thi03] Thimbleby, H. The directed Chinese Postman Problem. Software – Practice and Experience. 2003, p. 1081–1096.

[AW02] Mohammed Al-Ghafees, James A. Whittaker. Markov Chain-based Test Data Adequacy Criteria: a Complete Family. URL: www.kaner.com/pdfs/GoodTest.pdf, 148 KB. 2002.

Sutrumpinimai

MPT – modeliais paremtas testavimas (*angl. Model-based testing*)

BA – baigtinis automatas (*angl. Finite state machine*)

IBA – išplėstas baigtinis automatas (*angl. Extended finite state machine*)

MG – Markovo grandinėlės modelis (*angl. Markov chain usage model*)

UML – vieninga modeliavimo kalba (*angl. Unified Modeling Language*)

OCL – objektų apribojimo kalba (*angl. Object Constraint Language*)

SUT – testuojama sistema *(angl. System under test*)

API – aplikacijų programavimo sąsaja (*angl. Application Programming Interface*)

HTML – Hiperteksto žymėjimo kalba (*angl. Hyper text Markup Language*)

AsmL – abstrakti baigtinių automatų kalba (*angl. Abstract State Machine Language*)

XML – praplėsta žymėji kalba (*angl. Extensible Markup Language*)

AML – AGEDIS modeliavimo kalba (*angl. AGEDIS Modeling language*)

JML – Java modeliavimo kalba (*angl. Java Modeling Language*)