VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

Modeliais paremtas testavimas: testavimo įrankių analizė

Model-based testing: analysis of testing tools

Magistro baigiamasis darbas

Atliko:	Mindaugas Daukšys	(parašas)
Darbo vadovas:	Prof. dr. Linas Laibinis	(parašas)
Recenzentas:	Asist. dr. Haroldas Giedra	(parašas)

Vilnius
2020

Santrauka

Šiame darbe yra apžvelgiamos ir palyginamos pagrindinės modeliais paremto testavimo (MPT) charakteristikos bei susijusių testavimo įrankių funkcionalumas. Literatūros apžvalgos pagrindu darbe yra pristatytos modeliais paremto testavimo proceso sudedamosios dalys, susipažinimui pateiktos įvairios testavimui naudojamos modeliavimo kalbos bei dažniausiai naudojami algoritmai testų atvejams generuoti. Atlikus literatūros apžvalgą, darbe yra supažindinama su konkrečiais modeliais paremto testavimo įrankiais. Tolesnėje darbo dalyje pateikiamas trijų pasirinktų MPT įrankių – GraphWalker, ProB ir MBTsuite – palyginimas remiantis bandymu kurti SIM kortelės failų sistemos testavimo modelį ir generuoti testų atvejus. Atlikus atvejo analizę, yra pateikiamos įrankių įverčių reikšmės pagal iš anksto nustatytus kriterijus. Modeliais paremto testavimo įrankiai yra aptariami išskiriant naudojamus technologinius sprendimus ir galimas pritaikymo problemas modeliavimo ir testų generavimo srityse. Remiantis sudarytais palyginimo kriterijais, pateikiama informacija apie įrankių privalumus ir trūkumus modelių kūrimo ir testų generavimo procesuose. Detalesnis tyrimas taip pat atskleidžia, kuriose situacijose konkretūs įrankiai yra pranašesni už kitus.

Raktiniai žodžiai: modeliais paremtas testavimas, testų generavimas, testų filtravimas, modelių kalbos, testų generavimo įrankiai.

Summary

This work describes and compares essential model-based testing characteristics and the associated testing tool techniques. In the first work part, main parts of the model-based testing process, various modeling languages, and the most commonly used algorithms for generating test cases are presented. Next, an overview of the most commonly used model-based testing tools is given. The following sections present a comparison between three selected model-based testing tools – GraphWalker, ProB, and MBTsuite – based on a common case study, the SIM card file system, used to create system models and generate test cases by each tool. This allows to present and compare the key technological solutions of the analyzed tools, as well as to describe the issues (and their possible solutions) that occurred while creating models and generating test cases with these tools. Based on the determined in advance comparison criteria, the advantages and disadvantages in the model creation and test generation processes are summarized, emphasizing where a specific tool has advantages in comparison to the others.

Keywords: model-based testing, test generation, test filtering, modelling languages, test generation tools.

Turinys

Įvadas

Testavimas yra programų sistemos kūrimo proceso dalis, kurios metu iš anksto sudarytų testų pagalba surenkama informacija apie programinės įrangos kokybę. Testavimo procesas susideda iš testų atvejų rinkinio (angl. test suite) sudarymo pagal žinomus sistemos reikalavimus ir jo įvykdymo testuojamai sistemai. Rinkinys susideda iš testų atvejų (angl. test case), kurių kiekvienas turi savo apibrėžtas duomenų įvestis, vykdymo sąlygas, testavimo procedūrą ir laukiamus rezultatus. Laukiami rezultatai (tuo pačiu ir testų rinkinys) paprastai susiejami su testavimo tikslais (angl. testing objective), pavyzdžiui, įvykdyti tam tikrą testuojamos sistemos veiksmų eigą, patikrinti suderinamumą su konkrečiu reikalavimu ar panašiai.

Dauguma įmonių, užsiimančių programinės įrangos kūrimu, naudoja vienokius ar kitokius pagalbinius įrankius testavimui. Dauguma atveju testų vykdymo procesas būna automatizuotas. Tačiau testų kūrimas vis dar dažnai vyksta rankiniu būdu. Dėl šios priežasties testų kūrimo procesas nėra pritaikytas sistemos reikalavimų pasikeitimams, nemažai laiko trunka testų perrašymas, kuriant testus rankiniu būdu dažnai pasitaiko klaidos [BJK+05]. Be to, programų sistemoms tampant vis sudėtingesnėms, jų kokybės užtikrinimo užduotis tampa vis didesniu iššūkiu. Dėl brangiai kainuojančių programinės įrangos klaidų ir testavimo brangumo kuriant sudėtingas programų sistemas atsirado poreikis ieškoti naujų automatizuotų testavimo būdų, kad būtų pasiektas kiek galima didesnis testavimo efektyvumas ir kokybė.

Tobulinant testavimo procesą, kaip vienas iš modeliais paremto projektavimo (angl. model-based development) pritaikymo būdų atsirado modeliais paremtas testavimas (MPT). Šis metodas remiasi iš anksto sukurtais programų sistemų modeliais, kurie abstrakčiai atvaizduoja norimą sistemos funkcionalumą ar kitas sistemos pageidaujamas savybes. Remiantis modeliais kaip formalizuotais ar struktūrizuotais reikalavimais, automatizuojamas testų generavimo procesas. Tačiau kartu atsiranda papildomų iššūkių dėl papildomo modeliavimo etapo bei testų generavimo ir pritaikymo pasirinktų modelių pagrindu. Norint pasinaudoti MPT privalumais, testuotojams reikalingos modeliavimo ir modelių interpretavimo žinios. Taip pat reikia pasirinkti vieną iš sukurtų MPT įrankių, su gana skirtingomis galimybėmis ir pritaikymo charakteristikomis, kuris būtų labiausiai tinkamas konkrečiai sistemai.

Darbo aktualumas

Pastarajame dešimtmetyje modeliais paremtas testavimo būdas (pirmiausia dėl teikiamų papildomų testavimo proceso automatizavimo galimybių) susilaukė nemažai dėmesio iš programinės įrangos industrijos. Nors ir tyrimais patvirtinta, kad šis metodas tam tikrais atvejais atsiperka labiau nei įprastas automatiškai vykdomų testų rašymas[UL10], tačiau dėl

modeliavimo sudėtingumo iki šiol yra nemažai atvejų, kada pasirenkama taikyti kitokias mažiau automatizuotas testavimo metodikas [BLK15] [SVG+08].

Modeliais paremtas testavimo būdas reikalauja papildomų įrankių, kurių jau sukurta nemažai, tiek mokamų, tiek atvirojo kodo, pavyzdžiui, *PragmaDev, CertifyIt, Conformiq, ProB, DIVERSITY*. Nors šie įrankiai palaiko bendrąsias MPT veiklas, tačiau jų charakteristikos, tokios kaip, palaikomų modelių tipai, testų tipai (vienetų testai, integracijos testai ir kt.) ar kuriamų testų išvesties formatai (programavimo kalbos, aprašomosios kalbos ir kt.) labai skiriasi [LLS17]. Tam, kad būtų galima pasinaudoti jų privalumais, testuotojus reikia apmokyti naudotis konkrečiu testavimo įrankiu. Testų generavimas turi būti pagrįstas suderintais (ir vienokia ar kitokia forma struktūrizuotais ar formalizuotais) kuriamos ar esamos sistemos reikalavimais, kas yra viena iš sudėtingiausių MPT pritaikymo problemų. Be to, dažnai nėra aiškiai apibrėžta, kokio tipo sistemai kuris testavimo įrankis labiau pritaikytas bei ar apskritai verta konkrečiam projektui naudoti modeliais paremtą testavimą. Testų kūrimo įrankis gali neleisti paversti testų į norimą programavimo kalbą, nepalaikyti tam tikrų platformų, leisti generuoti tik tam tikrus testų tipus, gali turėti nepakankamą modelių kūrimo funkcionalumą ar turėti kitų trūkumų [GLA+16]. Todėl svarbu vartotojams pateikti bendrą supratimą apie MPT įrankius ir jų palyginimą.

Tikslai ir siekiami rezultatai

Šiame darbe analizuojami dažniausiai naudojamų MPT įrankių struktūra, veikimo principai ir pritaikymo pavyzdžiai. Pagrindinis darbo tikslas yra sudaryti vertinimo kriterijus ir pagal juos įvertinti įvairių testavimo įrankių pritaikymą programų sistemoms.

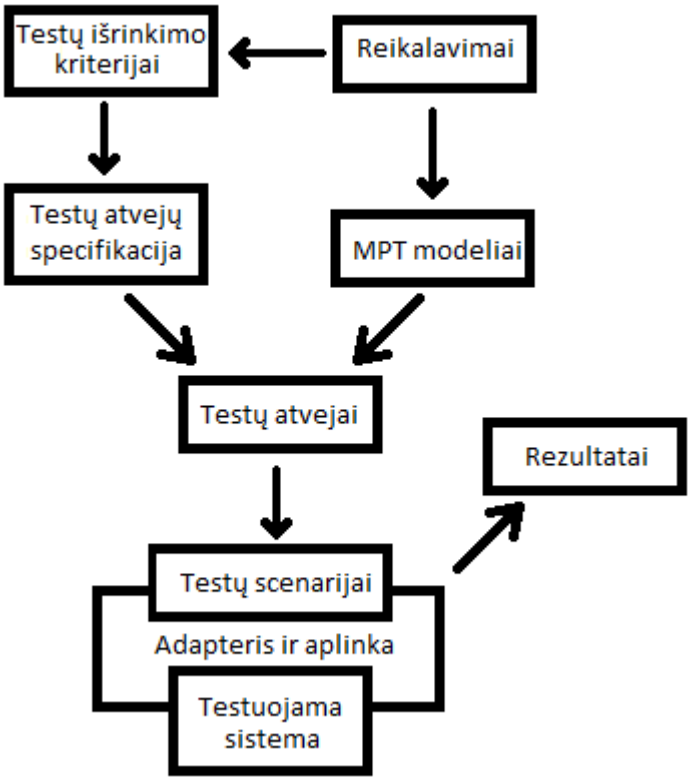
Pagrindiniai darbo uždaviniai:

1. **Identifikuoti įprasto MPT proceso sudedamąsias dalis**. Pradžioje bus MPT testavimo procesas išskaidytas į nagrinėjamas dalis. Pateikiamas kiekvienos dalies aprašymas bei savybės būdingos MPT.
2. **Palyginti konkrečių MPT įrankių komponenčių algoritmus.** Kalbant apie MPT įrankių panaudojimą reiktų žinoti apie pagrindines MPT įrankių testų kūrimo algoritmo komponentes ir papildomas MPT įrankių galimybes, todėl bus aptarti modeliais paremto testavimo įrankiams naudojami technologiniai sprendimai, pateikti įrankių pranašumai ir trūkumai.
3. **Pateikti MPT įrankių vertinimo kriterijus ir įverčių reikšmes pagal pirminę analizę**. Pasitelkus įvairius šaltinius bus išsiaiškintas įrankių funkcionalumas, sudaromi įvertinimo kriterijai, juos naudojant palyginti labiausiai paplitę MPT įrankiai.

4. **Pateikti MPT įrankių vertinimo kriterijus ir įverčių reikšmes pagal atvejo analizės tyrimus**. Gilesnė pasirinktų įrankių analizė bus atlikta konkrečios sistemos viešai prieinamų reikalavimų pagrindu, iš kurių bus sukurti potencialiai skirtingų tipų modeliai atskiriems pasirinktiems MPT įrankiams. Remiantis sukurtais modeliais atliekamas testų generavimas pasirinktiems įrankiams, vertinimo kriterijų sudarymas bei jų įverčių palyginimas, tuo pačiu įvertinant, kuris įrankis geriau pritaikytas konkrečioms testavimo užduotims.

1. MPT bendrasis procesas

Kaip teigia literatūros apžvalgos metu išnagrinėti šaltiniai, modeliais paremto testavimo procesas susideda iš penkių žingsnių: modelių kūrimo, testų generavimo, testų konkretizavimo, testų vykdymo ir rezultatų analizės. Šiame skyriuje pateikiama detalesnė informacija apie kiekvieną iš šių žingsnių. Aiškinantis MPT proceso sudedamąsias dalis, daugiausia buvo remiamasi [UPL12] ir [UL10] šaltiniais.



1 pav. MPT procesas [UPL12]

1.1. Modelių kūrimas

Pirmame MPT proceso žingsnyje yra sudaromi testuojamos sistemos modeliai. Modeliuose abstrakčiai atvaizduojama kaip pagal sistemos reikalavimus arba jos specifikaciją turėtų atrodyti sistemos struktūra ir funkcionalumas. Jie aprašomi formaliomis ar pusiau formaliomis kalbomis bei žymėjimo sistemomis. Konkretus modelių formatas taip pat priklauso nuo testuojamos sistemos charakteristikų ir MPT įrankio reikalaujamos įvesties formato [LLS17]. Modelio abstrakcijos yra susiejamos su testavimo tikslais, todėl toks modelis dažnai vadinamas testavimo modeliu (angl. testing model). Kartais testuojant sistemą būna panaudojami projektavimo modeliai (angl. design models). Tokiu atveju svarbu, kad modelis naudojamas testavimui būtų nepriklausomas nuo modelio naudojamo kitiems tikslams, nes klaidos projektavimo modelyje gali persikelti į generuojamus testus. Dėl tos priežasties įprasta sukurti vien testams naudojamą

modelį tiesiogiai iš neformalių sistemos reikalavimų arba panaudoti tik kelis trūkstamus aspektus iš kito modelio [UPL12].

Vienas svarbiausių MPT modelių privalumų – galimybė atskirai patikrinti modelio validumą (korektiškumą). Modelio validumo patikrinimas yra reikalingas įsitikinti, kad modelis yra neprieštaringas ir pilnas sistemos reikalavimų atžvilgiu. Taip gana dažnai tokio patikrinimo metu randamos klaidos ar trūkstama informacija reikalavimuose [UPL12]. Iš modelio patikrinimo būtinumo seka, kad modelis turi būti „paprastesnis“ (t. y., labiau abstraktus) nei testuojama sistema ar bent lengviau tikrinamas, keičiamas ir palaikomas. Testavimo modelis gali priklausyti įvairiems abstrakcijos lygiams. Aukščiausias abstrakcijos lygis paprastai atvaizduoja sistemą kaip "juodą dėžę", susiejant visus įmanomus sistemos įvesties duomenis su reikalaujamais išvesties rezultatais. Priklausomai nuo naudojamo abstrakcijos lygio, modelis gali neįtraukti tam tikro sistemos funkcionalumo ar kai kurių kokybės atributų, pavyzdžiui, saugumo ar vykdymo laiko [UPL12].

Iš kitos pusės, modelis turi būti pakankamai tikslus, kad būtų naudojamas kaip pagrindas „prasmingų“ testų generavimui. Tai reiškia, kad testai, generuojami modelio pagrindu, turi būti pakankamai išbaigti veiksmų, įvesties parametrų ir laukiamų rezultatų atžvilgiu tam, kad MPT procesas sukurtų didesnę pridėtinę vertę, lyginant su tradiciniais testavimo būdais [UPL12].

1.2. Testų atvejų generavimas

Antrame MPT proceso žingsnyje atliekamas testų atvejų generavimas. Pagrindinis šio žingsnio rezultatas yra abstrakčių testų atvejų rinkinys. Kadangi modelis naudoja testuojamos sistemos supaprastintą vaizdą, šie abstraktūs testai neturi tam tikrų detalių reikalingų testuojamai sistemai ir nėra vykdomi tiesiogiai [UL10].

Tam, kad pilnai patikrinti vidutinio ar didesnio sudėtingumo sistemą, reikalingas didžiulis kiekis visų įmanomų testų (kas daro pilną sistemos testavimą nepraktišku užsiėmimu), todėl būtina pasirinkti testų atrinkimo kriterijus. Kaip pabrėžiama literatūros šaltiniuose, iš principo neįmanoma apibrėžti geriausio kriterijų rinkinio [UPL12]. Pasirinkti testų atrinkimo kriterijai kontroliuoja jų generavimo procesą susitelkiant į konkrečius esminius atvejus, kuriuos norima ištestuoti. Testų išrinkimo kriterijai gali būti susiję su esamais sistemos reikalavimais, testavimo modelio struktūra (pvz., skirtingų būsenų ir perėjimų tarp jų padengimu), duomenų padengimo heuristika, stochastinėmis charakteristikomis (pvz., vartotojo veiksmų tikėtinumu), sistemos aplinkos savybėmis arba apibrėžtu galimų defektų rinkiniu [UPL12]. Testų išrinkimo kriterijai yra transformuojami į testų atvejų specifikacijas. Testo atvejo specifikacija yra aukšto abstrakcijos lygio norimo testo aprašymas. Testų atvejų specifikacijos įformina testų išrinkimo

kriterijus ir pateikia juos paruoštus naudojimui taip, kad turint modelį ir testo atvejo specifikaciją testų atvejų generatorius sugebėtų sudaryti testų rinkinį.

Dauguma modeliais paremtų testavimo įrankių taip pat automatiškai sukuria reikalavimų atsekamumo matricą (angl. traceability matrix) ar modelio padengimo ataskaitas kaip papildomus testų generavimo žingsnio rezultatus. Reikalavimų atsekamumo matrica nurodo ryšį tarp funkcinių sistemos reikalavimų ir sukurtų testų atvejų. Reikalavimas gali būti padengtas keliais testų atvejais, taip pat vienas testų atvejis gali tikrinti kelis reikalavimus. Modelio padengimo ataskaitos taip pat pateikia analizę apie tai, kaip gerai sukurti testai patikrina visus modeliuojamos sistemos veiksmus. Pavyzdžiui, padengimo ataskaita gali pateikti tam tikras padengimo statistikas sistemos operacijoms ar būsenų perėjimams (pasikeitimams), nusakyti mums, kokią dalį svarbiausių loginių sistemos sprendimų modelis pratestavo su teisingomis ir klaidingomis reikšmėmis, ar pateikti kitus padengimo matavimus. Tokias padengimo ataskaitas gali būti panaudojamos grįžtamajam ryšiui apie generuojamo testų rinkinio kokybę arba galima jas panaudoti siekiant identifikuoti tas modelio dalis, kurios galėjo nebūti gerai ištestuotos, ir išnagrinėti, kodėl taip atsitiko. Jei kuri nors leistina veiksmų seka neturi sugeneruotų testų, galima pamėginti pakeisti dalį generavimo parametrų ir pakartoti testų generavimo žingsnį [UL10].

Kai modelis ir testų atvejų specifikacija yra pilnai apibrėžti, testų atvejai yra sugeneruojami su tikslu patenkinti visas testų atvejų specifikacijas. Jei testų atvejų specifikacijai nėra sukuriamas nei vienas testo atvejis, testų atvejų specifikacija laikoma nepatenkinama. Tačiau paprastai yra bent keli testų atvejai, kurie tenkina specifikaciją. Tokiose situacijose testų atvejų generatorius pasirinks kelis iš tokių testų atvejų. Kai kurie testų generatoriai gali skirti daug pastangų siekiant sumažinti testų atvejų skaičių taip, kad mažas kiekis sukurtų testų padengtų didelį kiekį testų atvejų pagal jų specifikaciją. [UPL12]

1.3. Testų atvejų konkretizavimas

Trečiame žingsnyje abstraktūs testų atvejai, gauti iš antro žingsnio, konkretizuojami į vykdomus testų atvejus naudojant iš anksto paruoštus sąryšius ar "vertimus" tarp abstrakčių modelių ir realios sistemos elementų. Vykdomi testų atvejai sukuriami atsižvelgiant į sistemos realizacijos savybes ir gali būti tiesiogiai vykdomi testuojamoje sistemoje [LLS17]. Ši dalis gali būti atliekama transformavimo įrankio, kuris naudoja įvairius šablonus ir žinomus sąryšius, kad išverstų kiekvieną abstraktų testo atvejį į vykdomą testo scenarijų (angl. test script). Testo scenarijus yra vykdomas kodas, kuris vykdo testo atvejį, apibendrina sistemos rezultatą ir pateikia verdiktą, ar testas pavyko. Konkretizavimas taip pat gali būti atliktas naudojant adapterį.

Tai yra testavimo aplinkos sudedamoji dalis, kuri pritaiko abstrakčius testo duomenis konkrečios testuojamos sistemos interfeisui.

Dviejų abstrakcijos lygių testų kūrimo vienas iš pranašumų yra tai, kad abstraktūs testai gali būti nepriklausomi nuo testavimo aplinkos ir testams rašyti naudojamos kalbos. Keičiant vien adapterio kodą arba vertimų šablonus, galima pakartotinai panaudoti tą patį testų rinkinį skirtingose testų vykdymo aplinkose [UL10].

1.4. Testų atvejų vykdymas

Testų atvejai yra vykdomi testuojamoje sistemoje rankiniu būdu arba per automatizuotą testų vykdymo aplinką. Vykdant testus, į sistemą yra kreipiamasi parenkant įvesties duomenis, atitinkančius kiekvieną testų atvejį. Kiekvienam testų atvejui yra gaunamas rezultatas, pagal kurį galima vienareikšmiškai nustatyti, ar testas pavyko [LLS17].

Testų vykdymas susideda iš kelių žingsnių. Reikia prisiminti, kad modelis ir testuojama sistema yra skirtingų abstrakcijos lygių ir kad jie turi būti susiejami, verčiant modelio elementus į atitinkamus sistemos elementus. Testo atvejo vykdymas prasideda testo įvesčių konkretizavimu (pavyzdžiui, detalus kreipinys į web servisą) ir jų siuntimu į testuojamą sistemą. Gautas konkretus testuojamos sistemos rezultatas privalo būti palygintas su rezultatu, kurio tikimasi.

Testavimas gali būti *online* arba *offline*. Skirtumas tarp jų yra tai, kad *online* testai vykdomi iškart po jų generavimo, o *offline* testai įrašomi ir gali būti įvykdomi vėliau. Pirmuoju atveju paprastai 2-4 MPT proceso žingsniai yra sujungiami, o antruoju atveju jie yra vykdomi atskirai.

Naudojant *online* testavimą testų generavimo algoritmai gali reaguoti į faktinius testuojamos sistemos rezultatus. Jei testuojama sistema yra nedeterministinė, kartais testuoti *online* būdu būtina, nes testų generatorius gali matyti, kurią sistemos veiksmų seką testuojama sistema pasirinko, ir atsekti tą būsenų pasikeitimų grandinę modelyje.

Tuo tarpu *offline* testavimas reiškia, kad testų atvejai yra generuojami griežtai prieš jų vykdymą. *Offline* testų generavimas iš nederministinių modelių yra sudėtingesnis, nes testai sukuriami anksčiau nei žinoma, į kokią būseną juos įvykdžiusi sistema pateks. Dėl to testai turi būti vykdomi atsižvelgiant į sistemos būseną vykdymo metu. *Offline* testavimo privalumas yra tiesiogiai susietas su sugeneruotų testų išsaugojimu. Išsaugoti testų atvejai gali būti valdomi ir vykdomi naudojant testų valdymo įrankius (angl. test management tools), kurių pagalba reikės mažiau pakeitimų testavimo procesui. Testai gali būti sukurti vieną kartą ir kartojami daug kartų tai pačiai sistemai. Taip pat testų generavimas ir vykdymas gali būti atliekamas skirtingose mašinose, skirtingose aplinkose ar skirtingu laiku. Testų rinkiniai gali būti skaidomi ir pritaikomi kelioms testuojamoms sistemoms lygiagrečiai. Taip pat įmanoma atlikti testų rinkinio

minimizavimą. Be to, testuoti realaus laiko sistemas gali būti neįmanoma, jei testų generavimas trunka per ilgai. Jei testų generavimo procesas yra lėtesnis nei testų vykdymo, matomas akivaizdus pranašumas atliekant testų generavimą tik kartą [UPL12].

1.5. Rezultatų analizė

Po kiekvieno testo įvykdymo būtina nuspręsti, ar stebėtas sistemos elgesys buvo teisingas. Tai vadinama orakulo problema (angl. oracle problem). Orakulo problema dažnai sprendžiama rankiniu būdu tyrinėjant testų rezultatus, tačiau efektyviam ir pakartotinam testavimui ši testavimo procedūra būti automatizuota. MPT automatizuoja orakulų generavimą bei testų įvesčių pasirinkimą.

MPT proceso pabaigoje testų rezultatai yra pranešami vartotojams. Kiekvieno testo nesėkmės atveju reikia nuspręsti, kas nulėmė nesėkmę. Tai panašu į tradicinį testavimo analizės procesą. Paprastai, kai testas nepavyksta, surandama, kad testas nepavyko dėl klaidos testuojamoje sistemoje arba dėl klaidos pačiame teste. Kai naudojamas MPT, paprastai klaida būna dėl adapterio kodo arba dėl modelio apibrėžimo (galimai ir dėl reikalavimų apibrėžimo) [UL10]. Dėl to nesėkmingiems testų atvejams MPT procese išsaugojami susiejimo ryšiai tarp specifikacijos, modelių ir testų atvejų, kurie gali būti panaudojami ieškant klaidos vietos.

2. Modelių kalbos

Remiantis aprašytu MPT procesu, testams kurti reikalingi modeliai. Jie gali būti aprašomi įvairiomis specifikacijų ar modeliavimo kalbomis. Kiekvienas MPT įrankis palaiko konkrečias modelių kalbas. Šiame skyriuje pateikta informacija turėtų suteikti žinių apie įrankiuose naudojamas specifikacijų kalbas. Pagal tai, kaip modeliai yra kuriami ir ką semantiškai jie atvaizduoja, juos galima suskirstyti į kelias paradigmas, kurios yra parašytos [UPL12] ir [Jor17] šaltiniuose. Šiame skyriuje yra pateiktos tik dažniau naudojamos modelių paradigmos, susijusios su testų kūrimu. Surinkta ir apibendrinta informacija yra reikalinga gilesniam supratimui apie testų generavimo modelių pagrindu procesą bei modelių testavimo įrankių veikimą, kurie bus pristatyti kituose skyriuose.

2.1. Būsenomis paremtos specifikacijos kalbos

Šiomis kalbomis sistema modeliuojama kaip rinkinys kintamųjų, kurie reprezentuoja sistemos būseną konkrečiu jos veikimo momentu, bei grupė operacijų, kurios keičia šiuos kintamuosius. Kiekviena operacija apibrėžta pradine sąlyga (angl. precondition) ir galutine sąlyga (angl. postcondition) arba gali būti duota tik galutinė būsena, kuri gali būti apibrėžta konkrečiu kodu, kuris atnaujina būseną. Keletas tokių kalbų pavyzdžių: Z, B, VDM, JML, OCL. Šiuo atveju yra akivaizdu, kad operacijų apibrėžimas pradinėmis ir galutinėmis sąlygomis yra šaltinis testų generavimui, kaip pradinės sąlygos apriboja leistinus įvesties duomenis, o galutinės sąlygos apibrėžia laukiamą rezultatą.

2.2. Perėjimais paremtos specifikacijos kalbos

Šios kalbos didžiausią dėmesį atkreipia į perėjimus tarp skirtingų sistemos būsenų. Paprastai jie yra grafiškai žymimi mazgais ir jungtimis tarp jų. Mazgai atstoja pagrindines sistemos būsenas, o lankai – operacijas. Tekstinis ar lentelės pavidalo žymėjimas taip pat gali būti naudojamas norint apibrėžti būsenų perėjimus. Dažnai perėjimais paremtos kalbos yra išraiškingesnės, pridedant duomenų kintamuosius, sistemos komponenčių hierarchijas ar paralelizmą tarp tokių komponenčių. Pagal visų galimų ar leistinų perėjimų modelį generuojami testai. Kaip pavyzdžius tokių perėjimais paremtų kalbų, naudojamų MPT procese, galima paminėti: baigtinių būsenų mašinos (angl. finite state machines), būsenų diagramos (angl. statecharts), suženklintos perėjimų sistemos (angl. labelled transition systems) ir įvesties-išvesties automatai.

2.3. Istorija paremtos specifikacijos kalbos

Šiose kalbose modeliuojama sistema yra apibrėžiama, susiejant sistemos būsenas ir jų perėjimus su tam tikrais laiko momentais ar leistina veiksmų tvarka laike. Įvairūs laiko žymėjimai gali būti naudojami (diskretūs ar tiesiniai, tiesiniai ar išsišakojantys, taškai ar intervalai ir pan.). Jie remiasi skirtingomis laiko logikomis. Pranešimų sekų diagramos (angl. message-sequence charts) ir susiję formalizmai taip pat įtraukiami į šią grupę. Yra taip pat grafinių ir tekstinių specifikacijų kalbų, papildomai nurodančių sąveikas ir leistinas veiksmų sekas tarp skirtingų komponenčių. Testų generavimui, be galimų sistemos perėjimų, panaudojami vykdymo laiko momento apribojimai (angl. time constraints) ir apribojimai pagal vykdymo istoriją.

2.4. Operacinės specifikacijos kalbos

Šie kalbos aprašo sistemą kaip rinkinį lygiagrečiai vykdomų procesų. Jie yra ypatingai tinkami aprašyti paskirstytas sistemas ir komunikacijų protokolus. Tokių kalbų pavyzdžiai: Petri tinklai ir įvairios procesų algebros (CSP, CCS ir pan.). Čia taip pat testų generavimui panaudojamos pradinės ir galutinės sąlygos bei apibrėžtais protokolais apribotos leistinų komunikacijų tarp skirtingų komponenčių ar procesų sekos.

2.5. Stochastinės specifikacijos kalbos

Šios kalbos aprašo sistemą kaip tikimybinį įvykių ir įvesties reikšmių modelį ir dažniau panaudojami ne testuojamos sistemos, o aplinkos modeliavimui. Aplinkos savybės panaudojamos ir testų generavimui. Pavyzdžiui, Markovo grandinės panaudojamos su veiklos profiliais (angl. operational profiles), kad testai būtų generuojami atsižvelgiant į atliekamų veiksmų sekų dažnumą.

2.6. Duomenų srauto specifikacijos kalbos

Šios kalbos koncentruojasi labiau į duomenis ir jų transformacijas nei į valdymo srautą. Žinomi pavyzdžiai yra Lustre ir Matlab Simulink blokinės diagramos, kurios dažnai naudojamos modeliuoti tęstinėms sistemoms (angl. continuous systems). Generuojant testus atsižvelgiama į leistinas duomenų transformacijas (taip pat naudojant pradines ir galutines sąlygas).

2.7. Keleto specifikacijos kalbų tipų panaudojimas

Kelios paradigmos gali būti atvaizduojamos viena kalba. Pavyzdžiui, UML turi perėjimais paremtą paradigmą, panaudojant mašinos būsenų diagramą, ir būsenoms paremtą paradigmą, panaudojant OCL kalbą. Dvi paradigmos gali būti panaudotos vienu metu tame pačiame

testavimo modelyje. Tai gali padėti išreikšti kintančius sistemos perėjimus ir tam tikras taisykles diskretiems duomenų tipams. Kitas pavyzdys – Matlab, kuriuo galima modeliuoti įterptines realaus laiko sistemas, naudojant Simulink blokinių diagramų (duomenų srauto kalba) ir Stateflow būsenų schemų (perėjimais paremta kalba) kombinaciją.

3. Testų generavimo algoritmai

Viena patraukliausių MPT savybių – galimybė automatizuoti testų generavimo procesą. Šis žingsnis atliekamas jau turint testavimo modelį. Testuojant sistemą, „vaikščiojimas“ tarp sistemos būsenų nusako sistemos vykdymo "kelią" (angl. execution path). Dauguma minimų MPT algoritmų, naudojamų testų generavimui, remiasi tokiais iš anksto modelių pagrindu paruoštais ar iš jų sugeneruotais keliais, kurie matematiškai atvaizduojami kaip grafai. Panaudojant esamus modelius ir kai kurias testų atvejų specifikacijas, testų atvejai gali būti išvedami stochastiškai arba panaudojant paieškos grafe algoritmus ar kitas paieška paremtas technikas, modelio patikrinimą (angl. model checking), simbolinį vykdymą dedukcinį teoremų įrodymą (angl. theorem proving) ar apribojimų sprendimą (angl. constraint solving) [UPL12]. Toliau skyriuje pateikiama informacija apie testų generavimą šiais būdais.

Atsitiktinis testų generavimas gali būti atliktas, imant atsitiktines įvesties reikšmes iš sistemos įvesties erdvės (t. y., visų galimų įvesties reikšmių aibės). Reaktyvių sistemų atveju sistemos vykdymo keliai gali būti parenkami, atsitiktinai imant duomenis iš įvesties erdvės ir pritaikant testavimo modelį rezultatų patikrinimui. Atsitiktinio klaidžiojimo algoritmą taip pat galima naudoti „vaikščiojimui“ per būsenas, siekiant sudaryti skirtingus testų rinkinius. Atsitiktiniai klaidžiojimai taip pat gali būti atliekami su aplinkos modeliais, pateikiant stochastinius formalizmus.

Paieška paremti algoritmai panaudojami modelio perėjimams ištirti. Jiems priklauso grafų paieškos algoritmai, tokie kaip, viršūnių ir briaunų padengimo algoritmai (pavyzdžiui, „kiniečių paštininko algoritmas“, kuris padengia visas briaunas bent kartą) bei kiti paieškos algoritmai, tokie kaip, metaheuristinė paieška, genetiniai algoritmai ir imituoto grūdinimo (angl. simulated annealing) algoritmai.

Apribotas modelio tikrinimas (angl. bounded model checking) yra technologija modelio verifikavimui arba jo savybių simuliavimui. Jei savybė yra netenkinama, modelių tikrintojai dažnai gali sugeneruoti konkrečias būsenas ir perėjimų sekas, tą demonstruojančias (angl. counterexample). Susiejant šią technologiją su MPT, pagrindinis principas yra pirma suformuoti testų specifikacijas kaip pasiekiamumo (angl. reachability) savybes, nusakančias, kad galiausiai tam tikra būsena yra pasiekiama arba tam tikras perėjimas įvyksta. Modelio tikrintojas ieškodamas pavyzdžių, kuriuose savybė yra netenkinama, sugeneruoja kelius, kuriais pasiekiama tokia neigiama būsena arba verifikuoja, kad galiausiai reikiamas perėjimas įvyks. Kiti modelių tikrintojų panaudojimo variantai testų atvejų generavimui naudoja modelių arba savybių mutacijas.

Simbolinis vykdymas (angl. symbolic execution) paleidžia modelį ne su vienos įvesties reikšmėmis, bet su įvesčių rinkiniais. Jie atvaizduojami kaip apribojimai (angl. constraints). Šiuo

būdu simboliniai sistemos vykdymo keliai yra sugeneruojami: vienas simbolinis kelias atstoja daug pilnai išreikštų kelių. Demonstravimas su konkrečiomis reikšmėmis turi būti atliktas tam, kad būtų gauti testų atvejai testuojamai sistemai. Simbolinis vykdymas atliekamas pagal testų atvejų specifikacijas. Pakankamai dažnai tokios specifikacijos remiasi pasiekiamumo savybių apibrėžimais, kaip ir modelio tikrinimo atveju. Kitais atvejais testų atvejų specifikacijos yra pateikiamos kaip išreikštiniai apribojimai (angl. explicit constraints) ir simbolinis vykdymas atliekamas atsižvelgiant į šiuos apribojimus.

Dedukcinis teoremos įrodymas (angl. deductive theorem proving) taip pat naudojamas testų generavimui, ypač pasitelkiant įrodytojus (angl. theorem prover), kurie palaiko įrodymo kelio ar priešingų pavyzdžių generavimą. Vienas tokio pritaikymo variantas yra panašus į modelių tikrintojų panaudojimą, kur modelio tikrintojas pakeičiamas teoremos įrodytoju. Tačiau dažniausiai teoremos įrodytojas yra naudojamas patikrinti formulių patenkinamumą (angl. satisfiability), kurie dažniausiai yra perėjimų pradinės sąlygos būsenomis paremtuose modeliuose. Teoremos įrodytojas gali paeiliui įvykdyti kintamųjų priskyrimus, kai tenkinamos pradinės sąlygos veiksmams atlikti, taip atliekant perėjimus tarp būsenų. Pagal perėjimų sekas sukuriami testų atvejai.

Apribojimų sprendimas (angl. constraint solving) yra naudingas išrenkant reikšmes iš sudėtingų duomenų tipų, pavyzdžiui, kombinatoriniame n-tosios eilės testavime. Taip pat dažnai naudojamas kartu su kitais metodais, tokiais kaip, simboliniu vykdymu, modelio tikrinimu, kur specifiniai ryšiai tarp kintamųjų pradinėse sąlygose yra išreiškiami kaip apribojimai ir efektyviai išsprendžiami atitinkamais apribojimų sprendėjais (angl. constraint solver).

4. Dažniausiai naudojami įrankiai

Sudarant šį skyrių buvo atsižvelgta į naujausius atliktus įrankių tyrimus mokslinėje literatūroje. Taip pat renkantis aprašomus įrankius atsižvelgta į jų palaikymą šiuo metu. Įrankiai atrinkti pagal [Jor17] [LLS17] [SSB18] [Mic17] šaltinius, jų aprašymams panaudota ir informacija esanti kūrėjų puslapiuose.

4.1. FMBT

FMBT (free model-based testing) yra atvirojo kodo įrankis, sukurtas Intel [Int]. Įrankis yra tinkamas testuoti plačiam ratui programinių sistemų, imant nuo pavienių C++ klasių iki grafinio interfeiso aplikacijų, mobilių įrenginių bei paskirstytų sistemų. FMBT turi modelių redaktorių, testų generatorių, adapterių, naudojamų įvairiems tikslams, rinkinį bei įrankius atsekamumui analizuoti. Dabar FMBT palaiko visus MPT žingsnius be grafinio interfeiso [Jor17].

FMBT palaikomas *online* ir *offline* testavimas. FMBT yra naudojama Linux platformoje. „Ubuntu“ operacinė sistema yra populiarus ir rekomenduojamas pasirinkimas testavimui naudojant FMBT. Testų žingsniai gali būti parašyti Python, C++, JavaScript ir Shell kalbomis. Vieno paleidimo metu gali būti vykdomi testai parašyti skirtingomis kalbomis ir vykdomi skirtinguose įrenginiuose [Int].

Jorganesen knygoje [Jor17] yra tvirtinama, kad, nepaisant FMBT stiprių pusių esama dokumentacija gana silpna, kas apsunkina apsimokymą ir jos naudojimą. Intel pateikia labai naudingus komandinės eilutės kodo fragmentus visų reikalingų paketų diegimui FMBT eksploatavimui, tačiau fragmentai pateikiami skirtingose vietos ir juos nėra lengva rasti [Jor17]. Diegimas yra gana greitas ir FMBT programa turi būti paleidžiama naudojant terminalą. Paleidus FMBT programą, grafinės vartotojo sąsajos pagalba galima kurti naujus testus. Iš čia vartotojas gali sukurti naują AAL/Python modelį. AAL (Adapter Action Language) kalba yra naudojama testų kūrimui. AAL modeliavimo kalba apibrėžiami modeliai, naudojant pradines ir galutines sąlygas.

Deja, buvo pastebėta, kad šį įrankį yra labai sudėtinga naudoti, ypač vartotojams neturintiems programavimo patirties Python kalba. Tačiau pakartotinai naudojant įrankį skirtingoms sistemoms testuoti, pastebėta, kad FMBT naudojimo privalumas yra ne vien dėl to, kad programa yra atvirojo kodo, bet ir taip pat įrankis padeda labai greitai sukurti testų atvejus.

FMBT gali būti labai veiksmingas įrankis ir puikus pasirinkimas vartotojams, kurie yra susipažinę su OS aplinkomis ir kalbomis. Įrankis pateikia skirtingas labai naudingas funkcijas, tokias kaip, programos grafų generavimas su kintamųjų sekimu ir grafikais su žingsnių sekimu kiekviename bloke. Funkcijų, kurios būtų lengvai naudojamos, palyginus su komerciniais yra mažai, tačiau yra IDE įskiepių integruotis su „Visual Studio“ ir „Eclipse“ [Jor17].

4.2. MISTA

MISTA taip pat yra atvirojo kodo įrankis, kuris generuoja testus iš modelių į baigtinių būsenų mašiną arbą funkcijų tinklą [Xu15]. Įrankis naudoja aukšto lygio Petri tinklus, kaip vizualią modeliavimo kalbą. Testų modeliai gali būti animuojami ir patikrinami. Testų atvejų formatai padengia daug kalbų (Java, C, C++, C#, PHP, Python, HTML ir VB) bei keletą testavimo karkasų (xUnit, Selenium IDE, Robot framework). MISTA palaiko ir *online* ir *offline* testavimą [LLS17] [Jor17].

Įsigyti MISTA paprasta: įėjus į kūrėjų puslapį [Xu15], galima parsisiųsti zip failą, jokia licencija yra nereikalinga. MISTA paleidžiamas kaip Java programa. Įrankis pateikia grafinę sąsają, kuri leidžia vartotojams kurti modelius, skaičiuoklės lapo tipo (angl. spreadsheet) redaktorius leidžia tai daryti tekstiniu pavidalu, galima įvesti pagalbinį kodą testų kodo generavimui bei testų medžio generavimui.

Įrankyje yra galimybė verifikuoti modelius. Tai leidžia įsitikinti, kad visos būsenos ir perėjimai modelyje gali būti pasiekiami, pateikiant pradinę ir tikslo būseną. Įrankis taip pat leidžia imituoti Petri tinklus tam, kad būtų įsitikinta, kad modelis veikia taip, kaip tikimąsi. Yra platus testų padengimų ir kriterijų pasirinkimas.

4.3. Yest

Yest yra „Smartesting“ įmonės produktas [Sma], skirtas verslo aplikacijoms testuoti. Yest palaiko testų analizę, projektavimą ir įgyvendinimą remiantis grafiniais verslo procesų modeliais. Vartotojas gali apibrėžti verslo procesus ir testų atvejus. Verslo procesas, išreiškiamas užduotimis ir alternatyvomis, apibrėžia programos veikimą. Kai kurios taisyklės, atvaizduojamos sprendimų lentelėmis arba sprendimų medžiais, yra sujungiamos į proceso užduotis. Testų atvejai gali būti generuojami pačiu įrankiu, sekant procesą ir pritaikant susijusias taisykles, arba parašyti rankiniu būdu ir palyginami su procesu ir taisyklėmis. Sukurti testų atvejai gali būti saugomi testų talpykloje. Keičiant modelį, galima pakeisti egzistuojančius testų atvejus arba juos panaikinti. Išsaugant juos pakartotinai, testų atvejai saugykloje keičiasi atitinkamai, taip padidinant vartotojo produktyvumą. Naudojant Yest keli vartotojai gali dirbti vienu metu prie to paties projekto. Daugiau informacijos galima rasti kūrėjų puslapyje [Sma].

4.4. TestCast

TestCast yra vienas iš „Elvior“ įmonės produktų. Įrankis naudojamas Windows, Mac ir Linux platformose. Testai generuojami į TTCN-3 kalbą, todėl nėra programinės kalbos apribojimų, kuria turėtų būti parašyta testuojama sistema [Elv]. TestCast susideda iš:

- TestCast MPT kliento. Kliento pagalba galima redaguoti būsenų mašiną, testų tikslus, testų rinkinio įgyvendinimą skirtingose platformose;
- TestCast MPT serverio (pasiekiamo nuotoliniu būdu). Jis yra paleidžiamas Elvior serveryje ir palaiko daugiau nei vieną TestCast klientą. Čia yra projektuojami ir generuojami testų atvejai atsižvelgiant į kliento pateiktus testų padengimo kriterijus;
- TestCast T3 testų kūrimo ir vykdymo platformos. Ji naudojama automatinių testų vykdymui pagal TTCN-3 testų scenarijus. Tai yra pilnai funkcionalus įrankis testų duomenų kūrimui ir testų vykdymui.

Tam, kad būtų galima pasinaudoti įrankiu, pirma vartotojas turi sukurti testuojamos sistemos modelį naudodamas UML būsenų diagramas, TTCN-3 (testing and test control notation) duomenų apibrėžimus ir testų konfigūracijos apibrėžimą. Vėliau apibrėžiamas norimas testų padengimas ir generuojami testų atvejai. Abstrakčios testų sekos gali būti nagrinėjamos vizualiai. Jos paverčiamos į TTCN-3 testų scenarijus, kurie yra vykdomi TTNC-3 įrankiu. Testų vykdymo rezultatai pateikiami tolimesnei analizei [Jor17].

4.5. Conformiq 360° Test Automation

Conformiq produktų linija „Conformiq 360° Test Automation“ [Con] yra labai įvairiapusiška ir gali daug daugiau nei generuoti testus. Siūlomi įrankiai leidžia integraciją su jau egzistuojančios programos kūrimo ciklu (angl. existing software development life), pradedant testavimo procesą nuo reikalavimų valdymo ir programos gyvavimo ciklo valdymo bei programos gyvavimo ciklo valdymo per testų valdymą, dokumentavimą ir automatinio testų vykdymo įrankius. Jų įmonės svetainėje [Con] galima rasti daugiau informacijos apie produktus.

„Conformiq 360° Test Automation“ produktų linija yra įvairiapusiška, įtraukiant galimybes testų generavimui, dvi modeliavimo kalbas, vartotojo sąsają ir lankstų interfeisą su kitais įrankiais. Ji susideda iš penkių įrankių:

- **Conformiq Creator** suteikia galimybę automatiniu būdu testuoti įmonės IT programas, web aplikacijas ir web servisus. Creator sukurtas tam, kad automatizuoti funkcinį, aplikacijos, sistemos testavimą. Šis produktas nereikalauja programavimo patirties ir palaiko pilnai grafinį modeliavimą testavimui;
- **Conformiq Designer** naudoja Java ir UML2 kombinaciją, kuri leidžia supaprastinti sudėtingų sistemų (pavyzdžiui, įterptinių programų, tinklo įrankių) testavimą. Algoritmai automatiškai generuoja būtinus testus, kurie padeda tobulinti programinės įrangos kokybę;

- **Conformiq Transformer** automatizuoja vartotojo nurodytų ir automatiškai sugeneruotų testų vykdymą, palaikant platų pasirinkimą programos kūrimo gyvavimo ciklo įrankių ir tobulinant produktyvumą kokybės užtikrinimo procesą;
- **Conformiq 360° Integrations** pateikia svarbius palengvinimus integracijai su programos gyvavimo ciklo valdymu ir kitoms sistemoms programų kūrimo gyvavimo cikle, naudojant su Conformiq Designer ir Conformiq Creator;
- **Conformiq Grid** yra suderinamas produktas su „Conformiq Designer“ ir „Conformiq Creator“. Tai yra debesų kompiuterijos pritaikymas testų generavimui, kuris leidžia organizacijoms efektyviai panaudoti Conformiq produktus ir pridėti daugiau skaičiavimo resursų kai to reikia.

4.6. MBTsuite

Tai yra įrankis, sukurtas kompanijos „Sepp.med“ [Sep]. Jis automatiškai generuoja vykdomus testų atvejus ir testų duomenis iš grafinių modelių. Įrankis pritaikytas integracijai su įvairiais egzistuojančiais testavimo procesais. MBTsuite palaiko UML modelius, kurie kuriami išorinių įrankių pagalba. Modelio tvarkymo palengvinimui ir MPT specifiniam modeliavimui „Sepp.med“ sukūrė įrankį „MBTassist“, kuris pateikia vartotojo sąsają bet kurio įrankiui reikalingo parametro įvedimui.

MBTsuite naudoja grafinį modelį kaip įvestį ir sukuria testų atvejus remiantis pasirinktais testų atrinkimo kriterijais. Sukurti testų atvejai gali paeiliui būti perkeliami į esamą testų valdymo įrankį. Taigi, MBTsuite sujungia modeliavimą, testų valdymą, testų automatizavimo įrankius, griežtai susitelkiant ties esmine MPT dalimi – testų generavimu iš modelio [Sep].

Testų generavimui MBTsuite įgyvendina labai įvairius testų atrinkimo kriterijus. Keičiant testų generavimo parametrus ir filtravimo mechanizmus, galima gauti minimalų testų atvejų rinkinį, kuris atitinka testų tikslus geriausiai. Įmanoma ir atskirai apibrėžti išrinkimo strategijas diagramos dalims. Testams filtruoti naudojamos kelios filtrų kategorijos.

MBTsuite pateikia įvairų papildomą funkcionalumą testų valdymo palaikymui, pavyzdžiui, reikalavimų atsekamumą, prioritetų nustatymą ir vizualizavimą specifinių testų atvejų modelyje. Įrankis yra pateikiamas kartu su plačia dokumentacija, įtraukiant vartotojo vadovą, specifines modeliavimo gaires, profilius kai kuriems modeliavimo įrankiams.

Šis įrankis taip pat naudoja išorinius įrankius modelių kūrimui. Modelių kūrimui buvo pasirinktas įrankis „Enterprise Architect“, kurio pagalba galima sukurti UML diagramas, tačiau testavimo modeliams naudoti galima UML veiklos (angl. activity) arba būsenų (angl. state) diagramas. Naudojantis šiuo įrankiu sudaryti modeliai turi pradines ir galutines vykdymo sąlygas bei susijusį programinį kodą, kuris bus įtrauktas generuojamą į testo atvejį. Šiuo atveju ne kaip

GraphWalker sugeneruotas testų atvejis yra ne per daug funkcijų, o per vieną. Beje, tiesiogiai jie nėra rašomi – maži kodo gabalai sujungiami, kad būtų gautas pilnas testo atvejis. Testo kodą, kurį atstoja modelio dalis, galima rašyti pasirinkta kalba, tačiau testų šablonai generuojami tik Python, Java, C ir C++ kalboms.

4.7. GraphWalker

GraphWalker yra atvirojo kodo įrankis [Kar20]. Jis sukuria *offline* ir *online* testų rinkinį iš būsenų mašinos. Vartotojai gali pasirinkti bet kurį iš esamų septynių padengimo kriterijų testų generavimui. Įrankis gali būti integruotas naudojant Java arba iškviečiamas kaip web servisas.

Testai generuojami iš modelių, laikomų GraphML ar JSON formoje. Tai yra grafai, sukurti naudojant „yEd“ ar „GraphWalker Studio“. Abu modeliavimo įrankiai pritaikyti integracijai su Java ir „Maven“. Iš modelių generuojami testai, kuriuos būtų galima paleisti naudojant testavimo įrankius, tokius kaip, „JUnit“ ar „Selenium“. Testų generavimas atliekamas naudojant komandinės eilutės instrukcijas [Kar20].

Dažnai atvirojo kodo MPT įrankiai turi nepakankamą dokumentaciją ir sudėtingą diegimą, reikalaujantį gilių žinių apie kiekvieną neaiškų naudojamą įrankį. GraphWalker yra šiuo atžvilgiu taip pat ne išimtis. GraphWalker naudojimo instrukcijos nėra aiškios, be to, įrankis reikalauja gilių žinių norint jį panaudoti prasmingai. Pagalbai skirtas puslapis turi minimalų kiekį instrukcijų, o įrankio palaikymo bendruomenė yra gana nedidelė.

4.8. ProB

ProB įrankis plėtojamas Diuseldorfo Heinricho Heine universitete [HHU]. Įrankis suteikia galimybe naudotis B modeliavimo kalba. B kalba remiasi predikatų logika, aritmetika ir aibių teorija bei palaiko duomenų struktūras, tokias kaip, ryšiai, funkcijos ir sekos. Pakankamai detaliai apibrėžus modelius, ši modeliavimo kalba leidžia sukurti įrodomai teisingą programą. B modeliavimo kalba buvo sėkmingai panaudota reikšmingų, saugumą užtikrinančių sistemų kūrimui, pavyzdžiui, automatinėms (bepilotėms) Paryžiaus ir kitų miestų metro linijoms. Įrankis turi modelio animavimo, modelio tikrinimo ir apribojimų sprendimo funkcionalumą. ProB gali būti naudojamas plataus spektro klaidų paieškai apibrėžtuose modeliuose, aklaviečių paieškai ir testų atvejų generavimui. Testavimui skirti modeliai yra žymiai paprastesni, jiems apibrėžti reikia daug mažiau pastangų ir nereikalingi gilūs įrodymai [UL10].

5. Pasirinktos sistemos specifikacija

Modelių kūrimui buvo pasirinkta GSM (Global Standart for Mobile Communications) 11.11 protokolo specifikacija. Tai yra labiausiai paplitęs mobiliųjų telefonų ryšio standartas pasaulyje, kuriam antrojo dešimtmečio viduryje priklausė daugiau nei 90 procentų pasaulinio mobiliojo ryšio rinkos dalies [Ame14]. Standartas yra palaikomas Europos telekomunikacijų standartizacijos instituto [ETS96]. Jis apibrėžia numatytą SIM kortelės elgesį. Reikalavimų specifikacijai yra pasitelktas [UL10] šaltinis.

5.1. Specifikacijos savybės

Saugumas ir mobilumas yra svarbios mobilaus ryšio tinklo savybės. Kad būtų užtikrintas saugumas, GSM turi optimizuotą radijo interfeisą su skaitmeninėmis komunikacijomis, kurios apsaugotos kriptografiniais algoritmais. SIM kortelė turi šifravimo raktus, autentifikavimo algoritmus ir informaciją apie abonento tapatybę ir galimas paslaugas bei atmintį teksto žinutėms, telefono knygos informacijai ir pan. Kadangi kortelė turi visą vartotojo informaciją, galima SIM kortelę perkelti iš vieno mobilaus įrenginio į kitą ir išlaikyti visus vartotojo duomenis ir saugumo raktus. GSM 11.11 standartas apibrėžia fizines ir elektrines SIM kortelės savybes bei loginę duomenų struktūrą ir saugumo savybes, kurios apsaugo vartotojo duomenis.

GSM 11.11 standarte yra apibrėžiamas ir programinis interfeisas tarp SIM ir mobilaus įrenginio. Aplikacijos mobiliajame telefone gali pasiekti ir redaguoti failus naudodamos SIM kortelės komandas. Testuojamo branduolio abstrakcijos lygis apima tik standarte apibrėžtas komandas, t. y. nepadengia aukštesnio lygio (aplikacijos) ir žemesnio lygio (komunikacijos protokolo tarp SIM ir mobiliojo įrenginio).

Testavime labiausiai atkreipta dėmesį į SIM komandas ir pasiekiamus failus. Šios komandos suteikia galimybę pasirinkti failą ir įvesti PIN kodą. Priklausomai nuo PIN kodo validumo ir prieigos teisių tipo nustatoma, ar vartotojas gali skaityti pasirinktą failą. Vartotojo prieigos teisės gali būti užblokuotos, jei autentifikavimo procesas nepavyksta tris kartus paeiliui, ir gali būti atblokuotos, jei įvestas atitinkamas PUK kodas.

5.2. Failų sistema

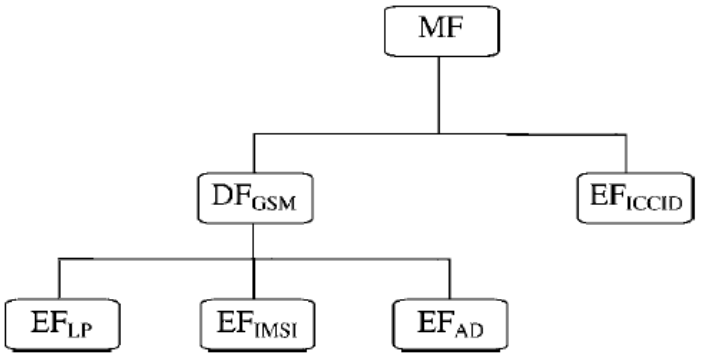
Pagal GSM 11.11 standartą sistemoje yra laikomi dviejų tipų failai:

- **Elementarusis failas** (angl. Elementary file arba EF). Tai yra failai, kurie turi tik duomenis. Pagal specifikaciją yra keli EF potipiai, tačiau šiame darbe bus nagrinėjami EF, kurie susideda iš baitų sekos;
- **Dedikuotasis failas** (angl. Dedicated file arba DF). Tai yra aplankas, kurio viduje gali būti laikomi kiti DF ir EF. Jie apibrėžia failų medžio struktūrą. Specialus failas,

vadinamas vyriausiuoju failu (angl. Master file arba MF), apibrėžia medžio „šaknį“. Tai yra aplankas, kuriame laikoma visa atminties struktūra.

Testavimo tikslams nuspręsta apriboti saugomų failų struktūrą iki dviejų DF (iš kurių vienas yra MF) ir keturių EF (2 pav.). Pasirinktas sistemos aplankas visada yra kuris nors iš DF. Pasirinktas failas visada yra EF iš tuo metu naudojamo aplanko, išskyrus atvejį, kai failas iš viso nėra pasirinktas, pavyzdžiui, po to, kai pasirinktas naujas aplankas. Pradžioje pasirinktas aplankas yra MF ir nepasirinktas joks failas. Failo pasirinkimas (DF ar EF) atliekamas pagal šias taisykles:

- Bet kuris failas, kuris yra esamoje direktorijoje gali būti pasirinktas;
- Bet kuris DF failas, kuris yra pasirinkto aplanko „brolis“ (turi bendrą tėvinį aplanką);
- Tėvinė direktorija gali būti pasirinkta;
- Esama direktorija gali būti pasirinkta;
- MF galima pasirinkti bet kada.



2 pav. Testavimui panaudota failų hierarchija.

5.3. Saugumo aspektai

Kiekvienas EF turi specifines prieigos sąlygas (DF neturi jokių sąlygų pagal GSM 11.11). Būtina prieigos sąlyga privalo būti išpildyta prieš prašomo veiksmo atlikimą. Šios sąlygos aprašytos lentelėje. Standarte galima rasti apibrėžtus šiuos keturis prieigos tipus:

- **ALW**ays. Veiksmas atliekamas be apribojimų;
- **NEV**er. Veiksmo atlikti negalima atlikti per mobiliojo įrenginio SIM kortelės interfeisą (kortelė gali turėti galimybę atlikti veiksmą viduje);
- **CHV** (kortelės turėtojo verifikavimas, angl. card holder verification). Veiksmas atliekamas tik tada, jei viena iš šių sąlygų yra išpildoma:
    - Įvestas teisingas PIN kodas per einamąją sesiją;
    - Įvestas teisingas PUK kodas per einamąją sesiją.

Jei patenkinama CHV prieigos sąlyga, ji išlieka validi iki GSM sesijos pabaigos, kaip ir atitinkamas PIN kodas išlieka neužblokuotas. Tai reiškia, kad po trijų bandymų paeiliui, nebūtinai toje pačioje sesijoje, prieigos teisės, suteiktos pagal PIN ar PUK kodą, yra iškart prarandamos;

- **ADM**inistrator. Už šių teisių skyrimą ir įvykdymą yra atsakinga atitinkama administracinė institucija. Atvejo analizėje ši prieiga visada bus uždrausta.

5.4. Pasirinktos komandos

Tyrimui buvo pasirinktos penkios komandos iš komandų rinkinio apibrėžto GSM 11.11 standarte. Šis pasirinkimas pakankamas failų minimaliam skaitymo funkcionalumui testuoti.

- **SELECT_FILE**. Ši funkcija pasirenka naują failą pagal sąlygas apibrėžtas anksčiau.
- **READ_BINARY**. Ši funkcija skaito duomenis iš esamo EF. Funkcija gali būti įvykdyta tik jei prieigos sąlygos šiam EF tenkinamos.
- **VERIFY_CHV**. Ši funkcija verifikuoja PIN kodą pateiktą mobilaus įrenginio, palygindama jį su reikalaujamu, kuris saugomas SIM kortelėje. Verifikavimo procesas leidžiamas tik jei CHV nėra užblokuota. Jei pateikiamas teisingas PIN kodas, galimų CHV bandymų skaičius atstatomas iki 3 ir CHV prieigos sąlygos tampa patenkinamomis. Jei PIN kodas nėra teisingas, galimų CHV bandymų skaičius sumažėja. Po trijų kartų paeiliui neteisingo PIN kodo įvedimo, nebūtinai toje pačioje sesijoje, CHV užblokuojamas ir CHV prieiga blokuojama tol, kol UNBLOCK_CHV funkcija sėkmingai atliekama.
- **UNBLOCK_CHV**. Verifikuojant pateiktą PUK kodą, ši funkcija atblokuoja CHV, kuri buvo užblokuota po trijų kartų paeiliui neteisingai įvedus PIN kodą. Ši funkcija gali būti sėkmingai įvykdyta net jei CHV nėra užblokuota. Jei pateiktas PUK kodas yra teisingas, PIN kodo reikšmė, pateikta kartu su PUK kodu, pakeičia esamą CHV PIN kodo reikšmę. Likusių UNBLOCK_CHV bandymų skaičius yra atstatomas iki pradinio skaičiaus 10 ir skaičius likusių CHV bandymų yra atstatomas iki 3. Po sėkmingo atblokavimo CHV prieigos sąlygos yra patenkinamos. Jei pateiktas PUK kodas yra neteisingas, galimų UNBLOCK_CHV bandymų skaičius sumažėja. Neteisingas PUK kodas neturi jokios įtakos CHV statusui. Jei 10 kartų paeiliui neteisingai suvestas PUK kodas, nebūtinai toje pačioje kortelės sesijoje, UNBLOCK_CHV komanda bus užblokuota. Tai padaro kortelę nenaudojamą visam laikui.
- **STATUS**. Ši funkcija suteikia galimybe matyti SIM būseną, į kurią įtraukta pasirinkta direktorija, pasirinktas failas (jei toks yra) ir CHV bei UNBLOCK CHV skaitliukų reikšmės.

Kiekviena funkcija, išskyrus STATUS operaciją, visada grąžina rezultato reikšmę. Šis atsakymas yra šešioliktainio skaičiaus kodas iš 1 lentelės.

Atsakymo kodas	Aprašymas
9000	- Normalus komandos grįžimas
9400	- Nepasirinktas EF
9404	- Nerastas failas
9804	- Prieigos sąlygos neišpildytos - Nesėkmingas CHV verifikavimas, liko dar bent vienas bandymas - Nesėkmingas UNBLOCK CHV verifikavimas, liko dar bent vienas bandymas
9840	- Nesėkmingas CHV verifikavimas, nebeliko bandymų - Nesėkmingas UNBLOCK CHV verifikavimas, nebeliko bandymų - CHV užblokuotas - UNBLOCK CHV užblokuotas

1 lentelė. Komandų atsakymų kodai

5.5. Sistemos modelio supaprastinimas

Kaip jau minėta, vienas iš modelio supaprastinimų buvo atliktas nusprendus susitelkti ties paprasčiausio EF potipio testavimu. Tai leidžia mums sumodeliuoti tik penkias komandas ir ignoruoti visas likusias GSM standarte. Rezultate atvejo analizės tyrimas turėtų išlikti pakankamai mažas. Kitas supaprastinimo sprendimas yra testuoti tik vieną CHV rūšį (GSM standarte yra dvi rūšys: CHV1 ir CHV2, kur CHV1 yra paprastas PIN kodas telefono naudojimui, CHV2 atskiras PIN, kuris gali būti naudojamas kitam funkcionalumui).

GSM 11.11 standarte kiekviena komanda yra koduojama šešioliktaine žyme su penkiais vieno bito parametrais (CLASS, INS, P1, P2 ir P3) ir kintamo ilgio parametru DATA, kuriame gali būti įvairūs duomenys. CLASS parametras visada yra konstanta „A0“ GSM aplikacijoms. INS yra baitas, nulemiantis, kuri instrukcija bus kviečiama. P1, P2, P3 yra baitai, kuriuose yra kviečiamos komandos parametrai. Tokios komandos pavyzdys yra pateiktas 2 lentelėje. Vienintelis VERIFY_CHV komandos parametras DATA yra svarbus iš funkcinės pusės, nes jis turi PIN kodą, kuris naudojamas vartotojo autentifikavimui. Parametrai P1 ir P3 yra fiksuoti šiai komandai, ir parametras P2 turi reikšmę „1“ (CHV1). Testų adapteris šias parametrų reikšmes gali panaudoti VERIFY_CHV komandai iškviesti.

CLASS	INS	P1	P2	P3	DATA
‘A0’	‘20’	‘20’	’01’	‘08’	‘1234’

2 lentelė. Užkoduota VERIFY_CHV komanda su PIN kodu

Kuriamo modelio tikslas buvo atvaizduoti normalų SIM kortelės elgesį, įtraukiant ir klaidų atvejus. Pavyzdžiui, sumodeliuotoje sistemoje testo atvejis galėtų būti iškviesti VERIFY_CHV komandą su teisingu PIN kodu, kai PIN kodas jau užblokuotas ir komanda grąžina statuso žodį 9840. Šioje atvejo analizėje nėra tikrinamas sintaksės patikimumas, kaip pavyzdžiui, komandos iškvietimas su klaidos ir parametrų formatais. Kitais žodžiais, yra laikoma, kad visos komandos yra teisingai suformuotos ir dėmesys kreipiamas į aplikacijos funkcionalumą.

5.6. Tikrinami reikalavimai

Šioje atvejo analizėje didžioji dalis dėmesio skirta pilnam pasirinktų komandų funkcionalumo padengimui. Reikalavimai nėra tiesiogiai pateikti specifikacijoje, tačiau pagal aprašytą operacijų veikimą galima juos identifikuoti. 3 lentelėje yra pateikti keli tokių reikalavimų pavyzdžiai.

Identifikuoti sistemos reikalavimai yra naudojami užtikrinti pilnu sistemos operacijų šakų padengimą testavimo metu. Ne visas operacijų vykdymo šakas galima pasiekti iš karto, pavyzdžiui, norint gauti sėkmingą failo perskaitymą, reikia pasirinkti failą suteikti atitinkamas prieigos teises arba turėti užblokuotą atitinkamą prieigą reikia neteisingai suvesti tam tikrą kartų skaičių PIN arba PUK kodą.

Reikalavimo identifikatorius	Reikalavimo aprašymas
REQ1	Jei PIN užblokuotas, tai atitinkamos teisės yra uždraustos.
REQ2	Jei PIN skaitliukas yra lygus 0, PIN užblokuotas.
REQ3	Jei atblokavimo skaitliukas lygus 0, kortelė visiškai užblokuota.
REQ4	Pasirinktas failas yra esamos direktorijos vaikinis arba išvis nepasirinktas.
...	...

3 lentelė. Reikalavimų pavyzdžiai

6. Pasirinktos sistemos testavimas

Šiame darbo etape buvo pasirinkti trys įrankiai: GraphWalker ir ProB ir MBTsuite. Siekiant savo tyrime kuo plačiau padengti skirtingų įrankių funkcionalumą ir galimybes, įrankiai tikslingai buvo pasirinkti kaip labai besiskiriantys pagal savo charakteristikas. Šiai analizei atlikti nebuvo naudojamas realios sistemos įgyvendinimas, vietoje to buvo pasirinkta dalis realios sistemos reikalavimų specifikacijos, pagal juos sukurti modeliai ir sugeneruoti testai.

6.1. Sistemos modeliavimas

Naudojantis [UL10] šaltiniu nepavyko atkartoti bandymų su B modeliavimo kalba pasitelkiant LEIRIOS įrankį, nes įrankio palaikymas yra seniai nutrauktas ir rasti jo nepavyko. Dėl to modeliai buvo dalinai perdaryti ir perkelti į ProB įrankį. Šiame skyriuje yra paaiškinamos tik esminės ProB modelio savybės, pilnas modelio kodas B kalba yra pateiktas 1 priede.

Antrasis pasirinktas MPT įrankis GraphWalker palaiko dviejų modeliavimo įrankių pateikiamus modelius. Vienas iš jų gali būti sukurtas „Yed“ įrankio pagalba ir importuojamas kaip „GraphML“ failas. Kitas palaikomas modelio formatas yra aprašytas JSON faile, kuris yra eksportuojamas įrankiu „GraphWalker Studio“. Kadangi jų veikimas nedaug kuo skiriasi, sistemos modeliams kurti buvo pasirinktas „GraphWalker Studio“.

Trečiajam pasirinktam įrankiui MBTsuite modeliai kuriami išorinių įrankių pagalba. Galima rinktis iš dviejų įrankių: „Enterprise Architect“ ir „IBM Rhapsody“. Abiem įrankiais galima kurti grafinius modelius. „Enterprise Architect“ modeliai kuriami UML veiklos (angl. activity) arba UML būsenų (angl. state) diagramomis. „IBM Rhapsody“ galima kurti modelius naudojant UML veiklos diagramomis. Modeliavimui pasirinktas „Enterprise Architect“ dėl platesnių įrankio galimybių.

6.1.1. ProB sistemos modelis

B kalbos duomenų modelis susideda iš statinės dalies, kuri apibrėžia duomenų struktūras (aibes ir konstantas) naudojamas modelyje, ir dinaminės dalies, kuri apibrėžia modelio būsenos kintamuosius, nekintamas savybes (taip vadinamus invariantus) bei sistemos operacijas. Toliau yra aprašomas sistemos modelis pagal minėtąsias modelio dalis.

Naudojamos aibės (SETS):

- **FILES** apibrėžia SIM failų rinkinį ir direktorijas.
- **PERMISSION** apibrėžia įvairias skaitymo teises.
- **VALUE** apibrėžia dvi galimas logines reikšmes tiesa (true) ir netiesa (false).

- **BLOCKED_STATUS** apibrėžia blokavimo būsenas.
- **CODE** atvaizduoja galimus PIN kodus, kurie gali būti naudojami verifikuoti prieigos sąlygas. Kodai gali būti apriboti iki keturių įmanomų reikšmių, kad būtų paprasčiau perskaityti sugeneruotas testų sekas.
- **DATA** yra abstraktus rinkinys visų duomenų, kurie gali būti įrašyti į failus. Testavimo tikslams, nebūtina sumodeliuoti failų turinio, kaip baitų sekos. Todėl norint abstraktesnio vaizdo, keturių naudojamų EF turiniai apibrėžiami kaip keturios skirtingos konstantos.
- **STATUS_WORDS** yra visų operacijų rezultatų kodų aibė, kurie gali būti grąžinami iš skirtingų operacijų.

Be to, yra apibrėžtos penkios konstantos (CONSTANTS):

- **FILES_CHILDREN** yra sąryšis naudojamas apibrėžti failo hierarchijos medžio struktūrą. Tai apima kelias (tėvas, vaikas) failų poras.
- **PERMISSION_READ** yra funkcija, kuri apibrėžia skaitymo prieigos sąlygas kiekvienam failui.
- **MAX_CHV** yra maksimalus nesėkmingų bandymų paeiliui suvesti PIN kodą skaičius iki CHV užblokavimo.
- **MAX_UNBLOCK_CHV** yra nesėkmingų bandymų paeiliui suvesti PUK kodą skaičius iki kortelės užblokavimo.
- **PUK** yra SIM kortelės PUK kodo konstanta.

Trys papildomi apibrėžimai (DEFINITIONS), kurie naudojami kaip sutrumpinimai:

- **MF** yra aibė, kuri turi tik pagrindinį failą;
- **DF** yra aibė, kuri turi tik direktorijas, išskyrus pagrindinį failą;
- **EF** yra aibė, kuri turi visus paprastus failus.

Dinaminėje modelio dalyje yra apibrėžti devyni sistemos kintamieji (VARIABLES):

- **current_file** nurodo pasirinktą failą.
- **current_directory** nurodo pasirinktą direktoriją.
- **counter_chv** apibrėžia likusį PIN kodo įvedimo bandymų skaičių, kol kortelė bus užblokuota.
- **counter_unblock_chv** nurodo likusių neteisingų PUK kodo įvedimo bandymų skaičių, kol kortelė bus užblokuota visam laikui.
- **blocked_chv_status** nurodo, ar CHV yra užblokuotas.

- **permission_session** atvaizduoja kiekvieną PERMISSION lygį į loginę reikšmę, ar vartotojui galima prieiti prie visų failų, kurie turi tokį prieigos lygį.
- **pin** einamąjį teisingą CHV PIN kodą.
- **data** apibrėžia kiekvieno EF turinį.

Modelio INVARIANT ir INITIALISATION skyreliai įtraukia atitinkamai invariantus ir būsenos kintamųjų inicijavimą. Invariantai skirti aprašyti taisyklėms, kurios galioja modelyje nepriklausomai nuo modelio būsenos. Kai padaroma klaida kurioje nors operacijoje, invariantai padėjo pastebėti klaidą. Sistemos modelio atveju, didžioji dalis invariantų yra kintamųjų tipų apribojimai, išskyrus paskutines keturias apibrėžtas sistemos nuoseklumo sąlygas, kurios turėtų visada būti išpildomos. Šių keturių invariantų intuityvus paaiškinimas yra toks:

- Jei PIN užblokuotas, tai atitinkamos teisės yra uždraustos.
- Jei PIN skaitliukas yra lygus 0, PIN užblokuotas.
- Jei atblokavimo skaitliukas lygus 0, kortelė visiškai užblokuota.
- Pasirinktas failas yra esamos direktorijos vaikinis arba išvis nepasirinktas.

Modelio operacija STATUS formalizuoja pateiktą komandą pagal duotą specifikacijos aprašą. Siekiant padengti visas galimas operacijų šakas testų generavime, kitos komandos padalintos į smulkesnes. Jos pateiktos 4 lentelėje.

Komanda	Išskaidytos operacijos
SELECT_FILE	SELECT_DF, SELECT_EF, SELECT_DF_FAIL, SELECT_EF_FAIL
READ_BINARY	READ_EMPTY_BINARY, READ_BINARY, READ_BINARY_NOT_ALLOWED
VERIFY_CHV	VERIFY_CHV_BLOCKED, VERIFY_CHV_GOOD_PIN, VERIFY_CHV_LAST_FAIL, VERIFY_CHV_FAIL
UNBLOCK_CHV	UNBLOCK_CHV_BLOCKED, UNBLOCK_CHV_SUCCESS, UNBLOCK_CHV_LAST_FAIL, UNBLOCK_CHV_FAIL

4 lentelė. Išskaidytos operacijos pagal SIM kortelės komandas

Žemiau yra pateikti jų aprašymai:

- **SELECT_DF**. Ši funkcija priskiria nurodytą DF ar MF kaip esamą direktoriją. Funkcija gali būti įvykdyta, jei tenkinamos 5.2 skyrelyje apibrėžtos sąlygos.
- **SELECT_EF**. Ši funkcija priskiria nurodytą EF kaip pasirinktą failą. Funkcija gali būti įvykdyta, jei tenkinamos 5.2 skyrelyje apibrėžtos sąlygos.

- **SELECT_DF_FAIL**. Ši funkcija grąžina klaidą apie DF failo pasirinkimą netenkinant apibrėžtų failo pasirinkimo taisyklių. Funkcija gali būti įvykdyta, jei netenkinamos 5.2 skyrelyje apibrėžtos sąlygos.
- **SELECT_EF_FAIL**. Ši funkcija grąžina klaidą apie EF failo pasirinkimą netenkinant apibrėžtų failo pasirinkimo taisyklių. Funkcija gali būti įvykdyta, jei netenkinamos 5.2 skyrelyje apibrėžtos sąlygos.
- **READ_EMPTY_BINARY**. Ši funkcija grąžina klaidą apie nepasirinktą failą. Funkcija gali būti įvykdyta jei joks EF nėra pasirinktas.
- **READ_BINARY**. Ši funkcija grąžina duomenis iš tuo metu pasirinkto EF. Funkcija įvykdoma tik jei prieigos sąlygos šiam EF tenkinamos.
- **READ_BINARY_NOT_ALLOWED**. Ši funkcija grąžina klaidą apie trūkstamas failo skaitymo teises. Funkcija gali būti įvykdoma, kai nėra reikiamų teisių skaitant duomenis iš EF.
- **VERIFY_CHV_BLOCKED**. Ši funkcija gali būti vykdoma tik tada, kai PIN kodas yra užblokuotas. Jos paskirtis grąžinti klaidos kodą, kad verifikavimo funkcija yra užblokuota.
- **VERIFY_CHV_GOOD_PIN**. Ši funkcija gali būti vykdoma tik tuo atveju, jei PIN kodas yra teisingas. Kai pateikiamas teisingas PIN kodas, galimų CHV bandymų skaičius atstatomas iki maksimalios nustatytos reikšmės ir CHV prieigos sąlygos tampa patenkinamomis.
- **VERIFY_CHV_LAST_FAIL**. Ši funkcija gali būti vykdoma tik tada, vedamas paskutinysis PIN kodo bandymas ir pateiktas kodas yra neteisingas. Po paskutiniojo neteisingo PIN kodo įvedimo, CHV prieigos teisės užblokuojamos iki operacijos UNBLOCK_CHV sėkmingo įvykdymo (UNBLOCK_CHV_SUCCESS).
- **VERIFY_CHV_FAIL**. Ši funkcija gali būti vykdoma tik tada, kai pateiktas PIN kodas nėra teisingas ir yra daugiau nei vienas likęs bandymas jį įvesti. Tokiu atveju likusių PIN kodo įvedimo bandymų skaičius bus sumažintas.
- **UNBLOCK_CHV_BLOCKED**. Ši funkcija gali būti vykdoma tik tuo atveju, jei kortelė yra visiškai užblokuota. Jos paskirtis grąžinti užblokuotos kortelės būsenos kodą.
- **UNBLOCK_CHV_SUCCESS**. Ši funkcija gali būti įvykdyta tik tada, kai pateiktas PUK kodas yra teisingas. Ši funkcija atblokuoja CHV prieigos teises, jei jos buvo užblokuotos. Pateikus teisingą PUK kodą, PIN kodo reikšmė, pateikta kartu su PUK kodu, pakeičia esamą PIN kodo reikšmę. Likusių UNBLOCK_CHV bandymų skaičius yra atstatomas iki nustatyto maksimalaus skaičiaus, taip pat likusių CHV bandymų yra

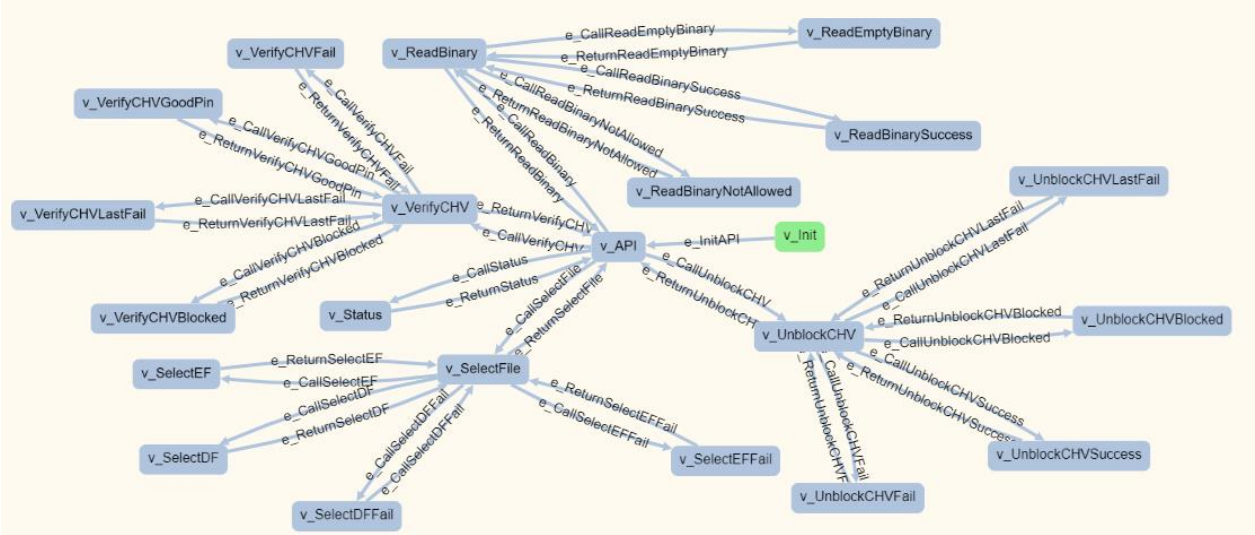
atstatomas iki nustatytos maksimalios reikšmės. Po sėkmingo atblokavimo failai pagal CHV prieigos teises tampa pasiekiami.

- **UNBLOCK_CHV_LAST_FAIL**. Ši funkcija gali būti įvykdyta tik tada, kai per paskutinį leistiną bandymą pateikiamas neteisingas PUK kodas. Funkcija padaro kortelę nenaudojamą visam laikui.
- **UNBLOCK_CHV_FAIL**. Ši funkcija vykdoma, kai pateiktas PUK kodas yra klaidingas, bet tai nėra paskutinis likęs bandymas iki kortelės užblokavimo. Tokiu atveju bus sumažintas PUK kodo įvedimo likusių bandymų skaičius. Neteisingas PUK kodas neturi jokios įtakos CHV prieigos statusui.

6.1.2. GraphWalker sistemos modelis

Modeliavimo įrankių pagalba galima sukurti grafinius modelius. Dažniausiai naudojami modeliavimo įrankiai yra „GraphWalker Studio“ ir „yEd“. Kadangi pagal dokumentaciją įrankių galimybės nesiskiria, modeliavimui pasirinktas „GraphWalker Studio“ įrankis.

Kuriant GSM sistemą buvo apibrėžtas paprastas modelis, kuris skirtas visiems perėjimams sumodeliuoti. Siekiant galimybės apibrėžti analogiškus modelio generavimo tikslus ProB, sukurtos tos pačios operacijos. Papildomai pridėtos pirminės operacijos (VERIFY_CHV, READ_BINARY ir t. t.) modelio atvaizdavimo supaprastinimui ir siekiant sumažinti reikalingų briaunų skaičių – operacija „v_API“, iš kurios galima patekti į bet kurią operaciją. Toliau (3 pav.) pateiktas sistemos modelis, sukurtas „GraphWalker Studio“ įrankiu.



3 pav. GraphWalker operacinis modelis. Žalia spalva pažymėta pradinė viršūnė.

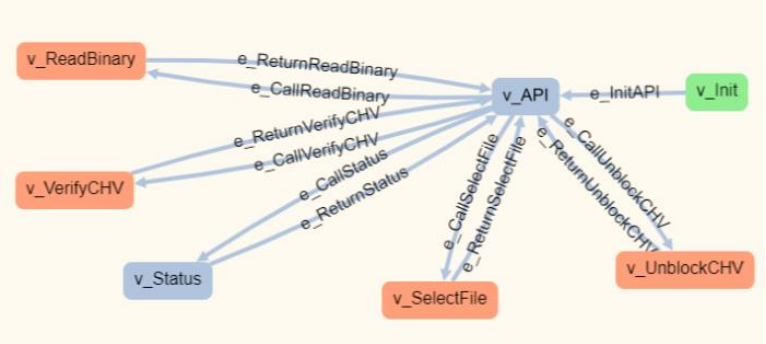
Šio tipo modelis turi panašų vaidmenį MPT kaip ir būsenų diagrama. Rodyklės reiškia veiksmus arba perėjimus iš vienos būsenos į kitą, o viršūnėse atliekami patikrinimai, ar sistemos

būsena yra tokia, kokia turėtų būti. Modelyje galima apibrėžti atliekamus veiksmus ir pradines vykdymo sąlygas. Aprašant modelio veiksmus ir tikrinimo sąlygas naudojama JavaScript programavimo kalba. Kintamųjų reikšmės gali būti panaudotos testų atvejuose arba nustatant, ar konkreti operacija gali būti vykdoma. Pasinaudojant šia savybe buvo dalinai atkartotas ProB duomenų modelis.

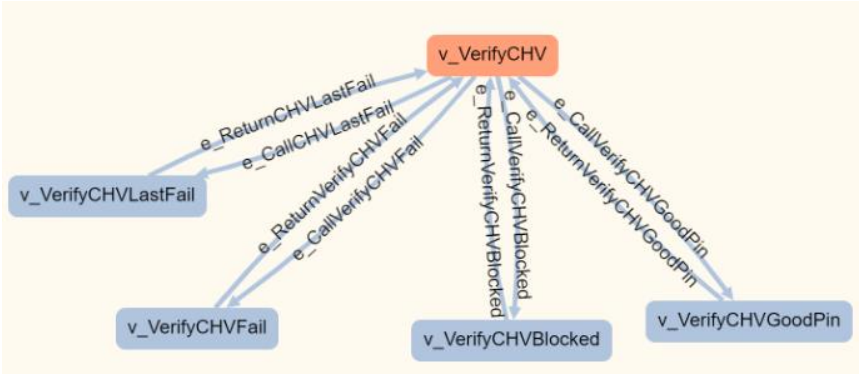
Aprašant duomenų modelį kilo nemažai problemų. Visų pirma, įrankis labai silpnai pritaikytas dideliems duomenų modeliams. Atliekamų veiksmų įvedimui pateikiama tik viena eilutė, todėl norint pateikti didesnį kiekį atliekamų veiksmų būtina naudoti išorinį teksto redaktorių ir iš jo nukopijuoti kodą į įrankio įvesties lauką. Klaidų ieškojimas modelyje labai sudėtingas. Sudėtingesnės modelio duomenų struktūros atvaizduojamos neinformatyviu tekstu. Prireikė kitos aplinkos skriptų kodui įvykdyti, norint išsiaiškinti, kur klaida. Kitos problemos susijusios su bandymais atkartoti ProB modelį. Pavyzdžiui, norint aprašyti invariantus, reikia modelio operacijose kartu su reikšmių priskyrimu tikrinti, ar sistemos būsena validi. Nemaža tikimybė būsenos tikrinimą kur nors praleisti, todėl invariantų aprašymas šiame įrankyje buvo praleistas. Parametrų reikšmių parinkimas operacijoms kitaip nei ProB, atliekamas rankiniu būdu, todėl į modelį taip pat reikia įtraukti JavaScript kodą, kuris generuotų atsitiktines reikšmes parametrams.

„GraphWalker Studio“ pateikia modelį JSON formatu. GraphWalker palaiko keturių failo formatų modelius: JSON, GraphML, Dot ir Java. Yra galimybė sudaryti ir kitokius šablonus, į kurius modelis būtų eksportuojamas. Komandinės eilutės pagalba įrankio numatytieji formatai gali būti konvertuojami iš vieno į kitą. Buvo laikytąsi prielaidos, kad nesvarbu, kuris įrankis bus pasirinktas, analogiškus rezultatus gauti kitu įrankiu bus galima gauti konvertavus modelio formatą. Tačiau atlikus bandymus su konvertavimo galimybe, teko nusivilti dėl įrankio programinių klaidų.

GraphWalker taip pat palaiko submodeliavimą. Tai yra funkcionalumas leidžiantis išskaidyti modelį į kelis smulkesnius, taip palengvinant jo skaitymą. 4 ir 5 pav. pateiktas submodeliavimo pavyzdys.



4 pav. Pagrindinis modelis. Raudona spalva pažymėtos viršūnės turi submodelius.



5 pav. „VerifyCHV“ submodelis. Raudona spalva pažymėta viršūnė yra pradžios ir pabaigos taškas.

GraphWalker komandinės eilutės įrankis iš pateikto JSON failo gali sudaryti pagal turimą šabloną kitokio formato modelio aprašymą. Šablonas gali būti sudarytas ir norima programavimo kalba, pavyzdžiui, Python. Be to, prie įrankio yra pridėtas šablonas, pagal kurį gali konvertuoti JSON faile esantį modelio aprašymą į Java kalbą (6 pav.). Naudojant submodeliavimą nebuvo išvengta problemų dėl konvertavimo į kitus formatus, buvo gautos neinformatyvios klaidos. Pagal internete rastą informaciją, vietoj „GraphWalker Studio“ naudojant „yEd“ modeliavimo įrankį šios problemos nebūtų. Tačiau „GraphWalker Studio“ įrankis dažnai atnaujinamas, todėl tikėtina, kad ateityje problema bus išspręsta.

```
import org.graphwalker.core.condition.*;
import org.graphwalker.core.generator.*;
import org.graphwalker.core.machine.*;
import org.graphwalker.core.model.*;

public class GSM {

  public final class ModelTestContext extends ExecutionContext {
  }

  public static void main(String... aArgs) {
    GSM modeltest = new GSM();
    modeltest.run();
  }

  private void run() {
    Vertex v_Init = new Vertex().setName("v_Init").setId("fba1e7e9-7c5b-467f-b462-484eb84d39c0");
        Vertex v_Status = new Vertex().setName("v_Status").setId("d7bdec6d-f764-48c4-9453-2cdce54808db");
        Vertex v_SelectFile = new Vertex().setName("v_SelectFile").setId("a42ed560-974d-4b72-af75-5b2ce4b78cdb");
        Vertex v_ReadBinary = new Vertex().setName("v_ReadBinary").setId("03b4a545-5d3f-4745-8700-e02178313dce");
        Vertex v_VerifyCHV = new Vertex().setName("v_VerifyCHV").setId("fe98efb0-ba9f-47e7-9a85-4079537de390");
        Vertex v_UnblockCHV = new Vertex().setName("v_UnblockCHV").setId("8826ea5d-7a9b-43c6-a64b-178bfe504f35");
        Vertex v_API = new Vertex().setName("v_API").setId("583aac1d-5175-4938-a23d-f849628a5ec4");
        Vertex v_VerifyCHVFail = new Vertex().setName("v_VerifyCHVFail").setId("f1a36c4c-169f-4655-b6be-13dd69017223");
        Vertex v_VerifyCHVGoodPin = new Vertex().setName("v_VerifyCHVGoodPin").setId("1389c24b-9c07-4bea-a7a6-1525de6a8349");
        ...


    Model model = new Model();
    model.addEdge( new Edge().setSourceVertex(v_Status).setTargetVertex(v_API).setName("e_ReturnStatus").setId("5afd8966-4d41-4601-837e-6d750f64820d"));
        model.addEdge( new Edge().setSourceVertex(v_API).setTargetVertex(v_VerifyCHV).setName("e_CallVerifyCHV").setId("82cea5f6-af42-4227-b3ec-4df2b5209655"));
        model.addEdge( new Edge().setSourceVertex(v_VerifyCHV).setTargetVertex(v_API).setName("e_ReturnVerifyCHV").setId("24be4652-1acc-42bf-a330-c9a79928ffae"));
        model.addEdge( new Edge().setSourceVertex(v_Init).setTargetVertex(v_API).setName("e_InitAPI").setId("87696ab1-d572-4171-bbbf-2b3ebaecdc07"));
        model.addEdge( new Edge().setSourceVertex(v_SelectFile).setTargetVertex(v_API).setName("e_ReturnSelectFile").setId("ddafa128-775c-4c3d-b214-624f329c4c58"));
        model.addEdge( new Edge().setSourceVertex(v_API).setTargetVertex(v_SelectFile).setName("e_CallSelectFile").setId("f2b7b042-aa7e-4db6-9bad-dcaaa5207c37"));
        model.addEdge( new Edge().setSourceVertex(v_API).setTargetVertex(v_UnblockCHV).setName("e_CallUnblockCHV").setId("97d4f37a-785c-436a-9703-67a82879f9ef"));
        ...


    Context context = new ModelTestContext();
    context.setModel(model.build()).setPathGenerator(new RandomPath(new EdgeCoverage(100)));
    context.setNextElement(context.getModel().findElements("v_Init").get(0));

    Machine machine = new SimpleMachine(context);
    while (machine.hasNextStep()) {
      machine.getNextStep();
      System.out.println(context.getCurrentElement().getName());
    }
  }
}
```

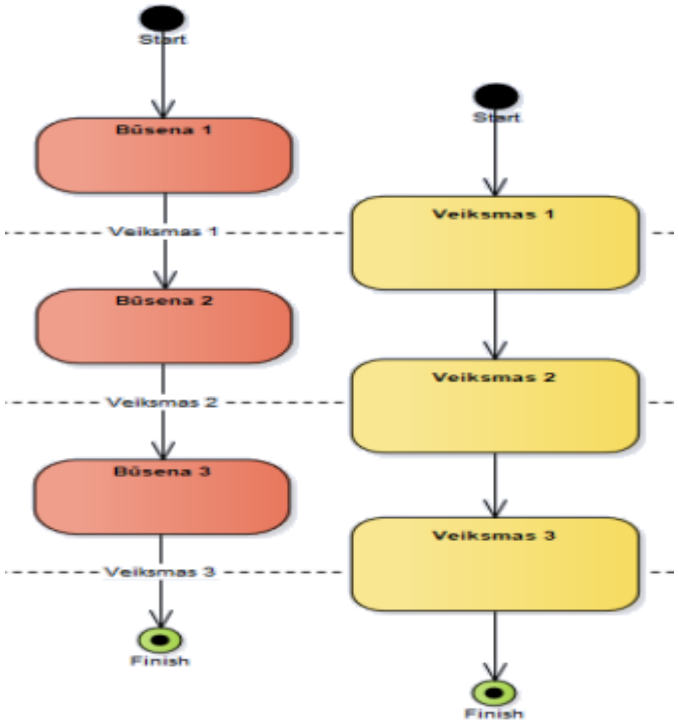
6 pav. Sugeneruoto GraphWalker modelio aprašymo Java kalba pavyzdys

6.1.3. MBTsuite sistemos modelis

MBTsuite neturi savo vidinio modeliavimo įrankio, bet palaiko integraciją su išoriniais modeliavimo įrankiais. MBTsuite dokumentacijoje pateikiama informacija, kaip naudotis IBM

„Rhapsody“ ir „Enterprise Architect“ modeliavimo įrankiais. Sprendžiant pagal pateiktą dokumentaciją „Enterprise Architect“ yra labiau pritaikytas MBTsuite įrankiui („Rhapsody“ nepalaiko būsenų diagramos tipo), todėl šiame skyrelyje aprašomos „Enterprise Architect“ modeliavimo įrankio savybės.

Modeliai atvaizduojami UML veiklos arba būsenų diagramos tipu. Tai yra du panašūs UML elgesio diagramos tipai. Kuriant modelius su bet kuriuo iš minėtų dviejų diagramos tipų dažniausiai naudojamos komponentės yra viršūnės, briaunos ir loginių šakų žymės. Tačiau viršūnės ir briaunos diagramose atlieka skirtingas funkcijas (7 pav.). Būsenų diagramoje briaunose nurodyti veiksmai atliekami veiksmai kaip reakcija į konkrečius įvykius. Veiklos diagramai nereikalingi konkretūs įvykiai, o tik perėjimai iš vienos viršūnės į kitą automatiškai atliekant veiksmus.



7 pav. Būsenų diagramos (kairėje) ir veiklos diagramos (dešinėje) palyginimas.

Būsenų diagramos apibrėžia visas įmanomas būsenas, į kurias tam tikras objektas (ar net sistema) gali patekti. Būsenų diagramoje viršūnės atvaizduoja objekto būsenas, briaunos atvaizduoja įvykius. Šio tipo diagramos taip pat suteikia galimybę kontroliuoti sprendimus – priskirti briaunoms pradines vykdymo sąlygas. Būsenų diagramos naudojamos modeliuojant objekto ar sistemos gyvavimo ciklą, detalizuojant naudojimo atvejus, protokolus arba veiksmų sekas [Pea09].

Veiklos diagrama yra būsenų diagramos tipas, kuris naudojamas veiksmų sekos modeliavimui [Pea09]. Veiklos diagramos fiksuoja aukšto lygio veiklos aspektus. Šio tipo

diagramose atvaizduojamas elgesys gali būti iš daugiau nei vieno objekto. Be minėtų savybių diagramos skiriasi ir tuo, kad veiklos diagrama apibrėžia veiklas, kurios gali įtraukti lygiagretumą ir sinchronizavimą, tačiau naudojant MBTsuite tai nėra palaikoma. Pagal UML semantiką veiklos diagramos yra redukuojamos į būsenų mašinas, pridedant papildomas žymes, kur viršūnės atvaizduoja perėjimą iš vieno baigiamo veiklos rinkinio į naujo pradedamo veiklos rinkinio pradžią. Kadangi veiklos diagrama gali vienodai atvaizduoti tos pačios sistemos elgesį, kaip ir būsenų diagrama, nuspręsta ja naudotis atliekant testų generavimo bandymus.

Būsenų diagramoje viršūnės aprašo esamą būseną, o veiklos – esamą veiksmą. Abiejuose diagramos tipuose viršūnėse galima aprašyti generuojamą testo atvejo kodą bei pridėti papildomus duomenis testų atvejų generavimui. Būsenų diagramos testavimo modelyje atvaizduojamos viršūnės, panaudotos kaip būsenų patikrinimo taškai, o veiklos diagramoje kaip atliekamos operacijos ir jų rezultatų patikrinimo taškai.

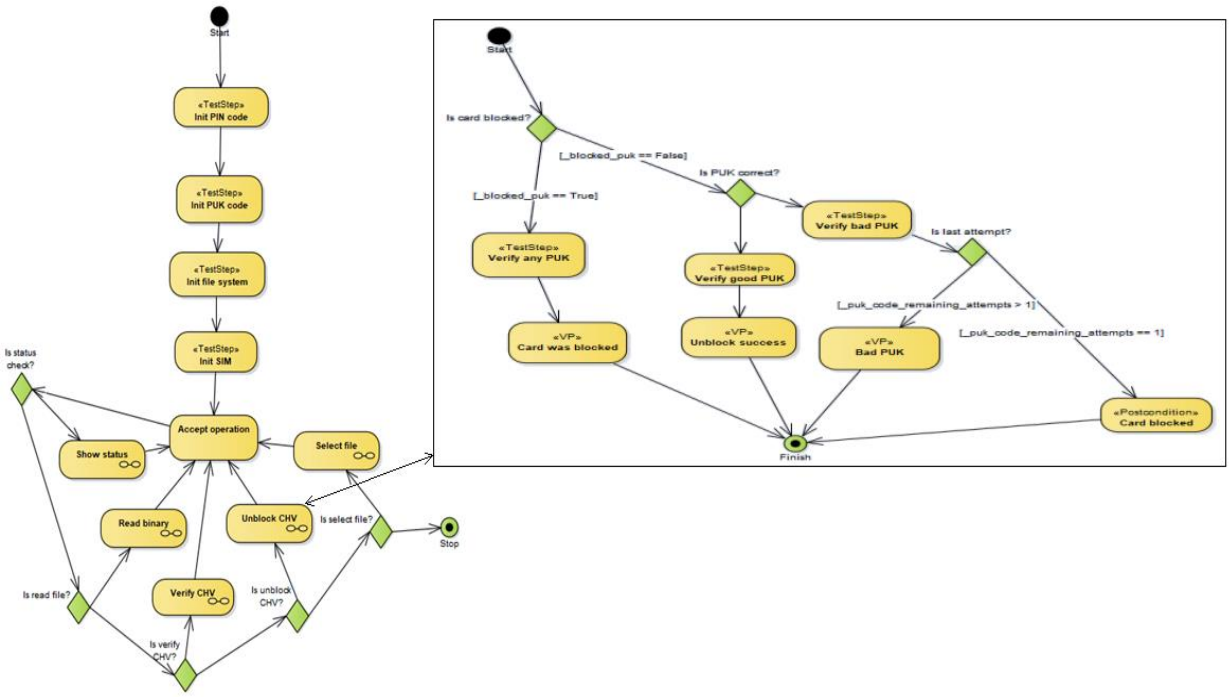
Veiklos ir būsenų diagramose perėjimai turi skirtingas paskirtis ir galimybes. Veiklos diagramos perėjimuose gali būti aprašytos pradinės sąlygos, kurias esama modelio būsena turi tenkinti tam, kad būtų galima ją pereiti, tačiau perėjimuose negali būti aprašomas kodas testo atvejo kodas. Būsenų diagramoje perėjimai skirti aprašyti sąlygų ar atliekamų veiksmų arba įvykių, kuriems įvykus pereinama į nurodytą būseną.

Loginių šakų žymės neturi jokio funkcionalumo testų generavime ir modeliuojant gali būti visiškai nenaudojamos, jei vietoj jų pagal atitinkamas sąlygas atliekami perėjimai iš vienos viršūnės į kitas. Tačiau šios žymės palengvina skaitomumą, todėl jos buvo panaudotos modeliuojant sistemą.

Be loginių šakų, modeliui supaprastinti buvo pasinaudota submodeliavimo galimybe. Tai yra modelio supaprastinimas paslepiant dalį modelio po viena komponente – submodeliu. Norint išlaikyti modelį lengviau skaitomą pagrindinėje diagramoje kiekviena SIM kortelės operacija buvo pateikiama subdiagrama. 8 pav. pateiktas submodelio pavyzdys. Diagramoje panaudota subdiagrama yra tarsi viršūnė, turinti įeinantį ir išeinantį perėjimus, tačiau į ją patekus pradedama vykdyti nurodytą diagrama nuo jos pradžios iki pabaigos taško, o išėjus grįžtama atgal į diagramos viršūnę ir vykdomas perėjimas iš jos.

Norint atvaizduoti tikslesnį veikimą generavimo keliui sudaryti buvo pasinaudota skriptų rašymo galimybe. Skriptai rašomi Python programavimo kalba. Aprašant konkretų testo vykdymo kodą galima manipuliuoti skriptų fragmentais, specialiu interpoliacijos žymėjimu įtraukiant modelio būsenos kintamuosius. Tačiau negalima tiesiogiai įtraukti sudėtingesnių išraiškų. Norint pridėti sudėtingesnę išraišką, reikia sukurti kintamąjį, jam priskirti išraišką modelio skriptuose bei pasinaudoti kintamuoju vietoj išraiškos. Dar vienas išryškėjęs trūkumas –

skripto ilgis vienam komponente yra riboto ilgio, dėl to reikėjo inicializavimo procesą skaidyti į keletą komponentų.



8 pav. Veiklos diagramos submodelio pavyzdys

Generuojamas testo atvejo kodas minėtų tipų diagramose aprašomas skirtingai: veiklos diagramoje vykdomas kodas aprašomas modelio atribute, o būsenų diagramoje operacijos žymėje. Sukūrus operacijos žymę, joje esančias funkcijas galima panaudoti pakartotinai. Operacijos remiamasi savybe, kad perėjimams yra sukuriama nuoroda į operaciją, o būsenoms (viršūnėms) nukopijuojama operacija. Tai reiškia, kad padarius pakeitimus operacijoje, perėjimuose ji atsinaujins automatiškai, tačiau būsenose liks prieš tai buvusi. Norint atnaujinti operaciją būsenoje, reikia išimti iš jos senąją operaciją ir pridėti iš naujo.

Įrankis leidžia importuoti ir panaudoti specifinius MBTsuite įrankiui sukurtus stereotipus (9 pav.), kurių pagalba testų generavimo įrankis gali atitinkamai interpretuoti diagramos elementus. Vieni svarbiausių stereotipų „TestStep“, „VP“, „Precondition“ ir „Postcondition“ yra naudojami aprašyti testo atvejo dalims. „TestStep“ – testo vykdymo žingsnis, kuriame aprašytas programos funkcijų kvietimai pagal parametrus. Stereotipas „VP“ (verification point) skirtas apibrėžti testo dalį, kuri patikrina sistemos būseną. Pažymėto elemento „TestStep“ arba „VP“ stereotipu kodo fragmentas bei kita informacija testų generavimo metu bus įtraukta į testo atvejį, kai elementas bus pasiektas. Testų generavime pasiekus elementą, kuris pažymėtas „Precondition“ arba „Postcondition“ stereotipu pridedamas kodo fragmentas atitinkamai į testo pradžią arba pabaigą.

9 pav. Diagramos elementai su „TestStep“ ir „VP“ stereotipais

Stereotipai naudojami ir kitiems tikslams. Iš pradžių norint ištestuoti tik tam tikrus kelius buvo pasinaudota prioritetų stereotipais. Prioritetai skaičiuojami nuo nulio iki dešimties. Nulinis laikomas aukščiausiu ir jis nustatytas visiems perėjimams ir viršūnėms pagal nutylėjimą. Pažymėjus nereikalingus keliuose bent vieną elementą žemesniu prioritetu už nustatytą slenkstį, į testų generavimą jie palieka neįtraukti. Norint pakeisti, kad būtų testuojami kiti keliai, nes prioritetus reikėjo perskirstyti, todėl ši savybė nebuvo teikianti daugiau naudos kaip perėjimų pradinės sąlygos. Taip pat stereotipų pagalba submodeliams galima nustatyti testų generavimo strategijas. Apie testų generavimo strategijas daugiau informacijos pateikiama testų generavimo skyrelyje (6.4.3).

6.2. Reikalavimų atsekamumas

6.2.1. ProB reikalavimų atsekamumas

ProB neturi galimybės reikalavimų atsekamumui įgyvendinti, tačiau yra galimybė reikalavimo atsekamumą įgyvendinti patiems. Testai yra generuojami pagal vieną apibrėžtą tikslą, kurį pasiekus vykdymas sustoja. Turint daugiau nei tikslą, juos galima apibrėžti atskiruose failuose. Skriptų pagalba galima įvykdyti testų generavimą, nuskaityti iš parametrų failo tikslą ir jį susieti su gautais testų atvejais.

6.2.2. GraphWalker reikalavimų atsekamumas

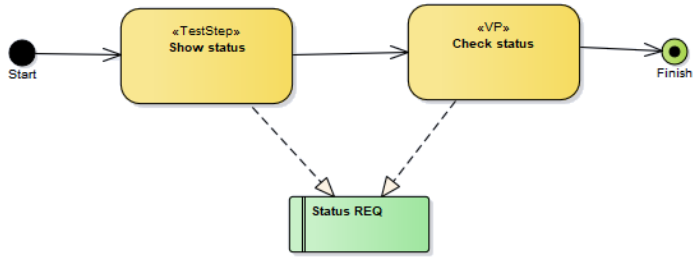
Modeliavimo įrankių pagalba galima į modelio elementus arbą į patį modelį įterpti atitinkamų reikalavimų žymes. Jas bus galima išspausdinti. Generuojant testų atvejus su Maven rodomi padengti ir nepadengti reikalavimai bei visų reikalavimų padengimo dalis procentais (10 pav). Naudojant komandinės eilutės įrankį susiejimą reikia susidaryti rankiniu būdu. Generuojant testus su kiekviena pereinama briauna ar viršūne yra pateikiamas ir jos identifikatorius, todėl galima parašyti skriptą, kuris pagal modelio elementų identifikatorius susietų reikalavimus iš modelio.

```
    },
    {
      "modelName": "GSM",
      "edgeId": "f321fc9b-4814-4829-ad32-7595a599358c",
      "edgeName": "e_ReturnReadBinarySuccess"
    }
  ],
  "requirementCoverage": 100,
  "requirementsPassed": [{
    "modelName": "GSM",
    "requirementKey": "REQ1"
  }],
  "requirementsFailed": [{
    "modelName": "GSM",
    "RequirementKey": "REQ2"
  }],
  "vertexCoverage": 47,
  "totalNumberOfEdges": 43,
  "totalNumberOfVisitedVertices": 11,
  "totalNumberOfRequirement": 2,
  "totalNumberOfUncoveredRequirement": 0,
  "edgeCoverage": 41,
  "totalNumberOfVertices": 23,
  "totalNumberOfPassedRequirement": 1,
  "totalNumberOfUnvisitedEdges": 25
}
```


10 pav. Maven įrankio pateikiama informacija apie reikalavimų padengimą

6.2.3. MBTsuite reikalavimų atsekamumas

Reikalavimų aprašymui modelyje pridedamos specialios komponentės, kurios nenaudojamos testų generavimo perėjimams. Iš jų išeina nuorodos į testų generavimo metu pereinamas modelio komponentes. Sugeneruotų testų atvejų lentelėse pavaizduojama, kokius reikalavimus perėjimas padengia. „Show status“ subdiagramoje (11 pav.) pateiktas reikalavimas, susijęs su pereinamomis modelio dalimis generuojant testus. Sugeneruoto testo atvejo lentelėje (12 pav.) matomas jo testo atvejyje.



11 pav. MBTsuite „Show status“ subdiagrama

Step	Type	Step Name	Step Description	Expected Result	Requirements	Code
1	TESTSTEP	Init PIN code	Creating PIN code & initializing maximum available attempts in model			String pin = "3663";
2	TESTSTEP	Init PUK code	Creating PUK code & initializing maximum available attempts in model			String puk = "97375782";
3	TESTSTEP	Init file system	Initializing file system for SIM			FileSystem fileSystem = new FileSystem();
4	TESTSTEP	Init SIM	Initiializing SIM			SIM sim = new SIM(pin, puk, fileSystem);
5	TESTSTEP	Show status	Getting SIM status		Status REQ	SimStatus simStatus = sim.getStatus ();
6	VP	Check status	Check current directory, file and pin, puk remaining attempts	Status matches model status	Status REQ	assert Objects.equals (simStatus.getCurrentDirectory(), "MF"); assert Objects.equals (simStatus.getCurrentFile(), null); assert simStatus.getRemainingPinAttempts() == 3; assert simStatus.getRemainingPukAttempts () == 10;

12 pav. Sugeneruoto testo atvejo lentelė. Requirements stulpelyje pateikiami padengiami reikalavimai.

6.3. Validavimas ir verifikavimas

Prieš pradedant generuoti testų atvejus naudinga patikrinti modelio kokybę pritaikant tam tikras validavimo ir verifikavimo procedūras:

- Modelio validavimas reiškia patikrinimą, kad jis atitinka neformalius reikalavimus, tai atliekama modelio peržiūrų metu ir tam tikrais atvejais animuojant modelį.
- Modelio verifikavimas susideda iš vidinių patikrinimų, pavyzdžiui, patikrinimas, kad visos operacijos laikosi modelio invariantų, nėra aklaviečių ir panašiai.

6.3.1. ProB validavimas ir verifikavimas

Panaudojus modelio kopiją iš [UL10] šaltinio, ProB padėjo rasti paliktas knygoje modelio sintaksės klaidas. Modelio tikrinimo funkcionalumas įrankyje leidžia ieškoti aklaviečių ir rasti invariantų pažeidimus. Įrankis turi septynias paieškos strategijas, kurios pritaikytos skirtingoms situacijoms (paieška į plotį, paieška į gylį, maišyta paieška į plotį ir į gylį, išeinančių kelių skaičiaus paieška, skirta aptikti aklavietėms, bei įvairūs atsitiktinio klaidžiojimo algoritmai).

Specialiai įvėlus invariante klaidą, kad PIN kodas turėtų būti užblokuotas po pirmojo nesėkmingo bandymo jį įvesti, modelio tikrintojas parodė, kad yra klaida. Kartu su klaida pateikiami įvykdyti žingsniai jai atkartoti ir būsenos savybės, kurias galima analizuoti, norint rasti klaidą. Aklavietė yra pasiekiama būsena, kai modelyje nebegalima vykdyti jokios operacijos. Operacijų vykdymą apriboja jos pradinės sąlygos. Visos modelio operacijos gali būti vykdomos besąlygiškai, todėl aptikti aklavietę neįmanoma. Norint sudaryti aklavietės situaciją, būtų galima panaikinti visas operacijas išskyrus VERIFY_CHV_FAIL ir VERIFY_CHV_LAST_FAIL. Tokiu atveju pasiekus situaciją, kai PIN kodas yra užblokuotas, nebebus galima vykdyti jokios operacijos. Norint panaikinti aklavietę, galima pridėti operacija VERIFY_CHV_BLOCKED, kurią galima vykdyti, kai PIN užblokuotas. Tiesa, tikrinant modelį įrankis gali nerasti klaidų, net jei jų yra. Paprastai taip atsitinka, kai atidaromas failas su ProB ir

pakeičiamas jo turinys. Norint išvengti šios problemos reikia pakeistą modelį išsaugoti, iš naujo jį atidaryti su ProB ir paleisti modelio tikrinimą.

6.3.2. GraphWalker validavimas ir verifikavimas

GraphWalker modelio verifikavimas atliekamas tik modelio sintaksės tikrinimui. Pavyzdžiui, pakeitus pradinio elemento identifikatorių, elementas nerandamas, todėl įrankis grąžina klaidą. Klaidos skriptuose nėra aptinkamos, apie jas gaunami pranešimai tik generuojant testus. Kaip ir testų kūrimas, modelio validavimas atliekamas „Apache Maven“ karkaso pagalba, GraphWalker komandinės eilutės sąsaja arba naudojant „GraphWalker Studio“.

6.3.3. MBTsuite validavimas ir verifikavimas

„Enterprise Architect“ modeliavimo įrankis leidžia eksportuoti XMI failo formatu modelius, kurie nėra validūs, tačiau MBTsuite patikrina modelio validumą prieš jį importuojant. Įrankis neleis importuoti modelio ir pateiks atitinkamą pranešimą, jei nėra išeinančių ar įeinančių briaunų į bent vieną viršūnę. Kiekviena diagrama turi turėti pradžios tašką ir pabaigos tašką. Galutinis taškas negali turėti išeinančių briaunų, pradinis – įeinančių. Jei yra MBTsuite įrankiui nežinomų diagramos elementų pateikiami įspėjimai, tačiau modelis importuojamas. Įspėjimai matomi, jei ir aprašant modelio atributus panaudojami ne tik raktiniai žodžiai („Code“, „Script“). Kai kurių klaidų nėra ieškoma modelio nuoseklumo patikrinime, tačiau apie jas pranešama generuojant testų atvejus. Dažniausiai tai yra klaidos skriptuose. Kaip jau minėta, negalima į konkretaus sugeneruoto kodo aprašymą įterpti sudėtingesnių išraiškų kaip kintamieji, apie tai parašoma, kai bandoma generuoti testus. Tai pat pateikiami pranešimai, jei modelio perėjimo skriptuose sintaksės, vykdymo klaidų arba yra vykdymo kelių, kuriuos nebuvo galima pereiti arba nepavyko sukurti testų atvejų. Kuriant modelius teko pastebėti, kad skriptuose kintamieji nėra globalūs, nebent prieš juos pridedamas apatinio brūkšnio simbolis („_“) arba kintamieji nenurodomi kaip globalūs pagal Python kalbos sintaksę. Tačiau yra loginių klaidų, kurių įrankis neaptinka. Pavyzdžiui, kai lokalaus kintamo reikšmė yra neapibrėžta, bet jis panaudojamas, apie tai nėra pateikiama klaidų.

6.4. Testų generavimas

6.4.1. ProB testų generavimas

ProB gali generuoti testus iš formalių modelių, kurie aprašyti B, Event-B, TLA ar Z kalba. Naudojant šį įrankį galimas tik operacijų padengimo kriterijus. Tai reiškia, kad bandoma užtikrinti, kad kiekviena modelio operacija yra įvykdoma bent kartą. Testo atvejus sudaro operacijų seka kartu su operacijų parametrų reikšmėmis, konkrečiomis konstantų reikšmėmis ir

formalaus modelio pradinėmis reikšmėmis. Galima nustatyti, kad testo atvejo generavimas baigtųsi būsenoje pagal generavimo parametruose pateiktą predikatą.

ProB turi du pagrindinius algoritmus, kurie randa testų atvejus:

- **Paremtas modelio tikrinimu**. Naudojantis šiuo būdu modelio tikrintojas generuos modelio būsenų erdvę, kol bus patenkintas padengimo kriterijus. Pilna būsenų erdvė susideda iš visų įmanomų sistemos inicializavimo kombinacijų, įskaitant ir visas galimas konstantų reikšmes.
- **Paremtas suvaržymais** (angl. constraint-based). Šiame būde ProB apribojimų sprendėjas panaudojamas įmanomų įvykių sekų generavimui pagal paieškos į plotį algoritmą. Generavimas vykdomas, kol patenkinamas padengimo kriterijus.

Abu algoritmai panašūs: atliekamas perrinkimas, kol padengimo kriterijai yra patenkinami. Pagrindiniai skirtumai tarp algoritmų yra tokie:

- Suvaržymais paremtas būdas sugeneruos konstantas ir operacijų parametrus tokius, kokie reikalaujami pagal tikslą. Naudojant šį būdą nebus tikrinama kiekviena įmanoma konstantų, pradinių reikšmių ar operacijų parametrų reikšmių kombinacija.
- Suvaržymais paremtas būdas konstruoja ne pilną būsenų erdvę, o sudaro įmanomų vykdymo kelių medį. Šis būdas negali aptikti ciklų. Net jei būsenų erdvė baigtinė, vykdymas niekada nesustos, jei visos būsenos negalės būti padengtos.

Suvaržymais paremtas būdas naudingas, kai yra didelė galimų reikšmių aibė konstantoms, pradinėms reikšmėms, operacijų parametrams, taip pat, kai generuojamuose testuose vykdomų operacijų skaičius yra pakankamai mažas. Apribojimų sprendimo sudėtingumas išauga kartu su atliekamų operacijų skaičiumi viename testo atvejyje, be to, didėjant operacijų skaičiui testų atvejų kandidatų skaičius paprastai auga eksponentiškai su galimo trumpiausio testo atvejo gyliu.

Testų generavimo paleidimui naudojama komandinė eilutė. Testų generavimo parametrai gali būti apibrėžti joje arba naudojant XML parametrų failą. Pagrindiniai naudojami parametrai (13 pav.):

- Failo pavadinimas, į kurį bus surašomi rezultatai (output-file).
- Padengiamos operacijos (event-coverage).
- Vykdymo sustojimo sąlyga (target).
- Maksimalus paieškos į plotį gylis (maximum-depth).

```
<test-generation-description>
  <output-file>test_results_refactored_req1.xml</output-file>
  <event-coverage>
    <event>VERIFY_CHV_BLOCKED</event>
    <event>VERIFY_CHV_GOOD_PIN</event>
    <event>VERIFY_CHV_LAST_FAIL</event>
    <event>VERIFY_CHV_FAIL</event>
    <event>UNBLOCK_CHV_BLOCKED</event>
    <event>UNBLOCK_CHV_SUCCESS</event>
    <event>UNBLOCK_CHV_LAST_FAIL</event>
    <event>UNBLOCK_CHV_FAIL</event>
    <event>STATUS</event>
    <event>SELECT_DF</event>
    <event>SELECT_DF_FAIL</event>
    <event>SELECT_EF</event>
    <event>SELECT_EF_FAIL</event>
    <event>READ_EMPTY_BINARY</event>
    <event>READ_BINARY</event>
    <event>READ_BINARY_NOT_ALLOWED</event>
  </event-coverage>
  <target>blocked_chv_status = blocked</target>
  <parameters>
    <maximum-depth>10</maximum-depth>
    <preference name="CLPFD" value="true"/>
    <preference name="TIME_OUT" value="2000"/>
  </parameters>
</test-generation-description>
```

13 pav. Parametrų failo pavyzdys

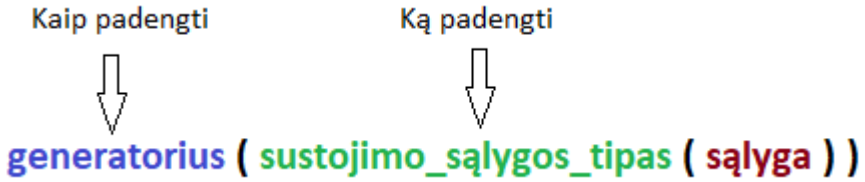
6.4.2. GraphWalker testų generavimas

Online testus generuoti naudojant GraphWalker yra du būdai. Pirmuoju GraphWalker paleidžiamas kaip serveris, su kuriuo galima bendrauti WebSocket protokolu arba HTTP REST interfeisu. GraphWalker serveris vykdo komandas pagal modelį ir atlikti perėjimai perduodami į serverį besikreipiančiam klientui, kuris žino, kokie veiksmai turėtų būti atliekami įvykus konkrečiam perėjimui. Antrasis būdas *online* testams generuoti – Maven karkaso pagalba. Susikūrus Java programą pagal dokumentacijoje pateiktą šabloną reikia pridėti į programos atitinkamą katalogą modelį ir įvykdyti komandą, kuri aprašyta į Maven, kad būtų sukuriamas modelio interfeisas Java programavimo kalba. Interfeise aprašyta kiekviena modelio briauna ir viršūnė. Įgyvendinant Java interfeisą galima aprašyti koks programinis kodas bus vykdomas atitinkamoje briaunoje ar viršūnėje. Kaip minėta, briaunose turėtų būti atliekami testų žingsniai, o viršūnėse – atliekami sistemos būsenos patikrinimai. Interfeiso realizacijoje taip pat reikia aprašyti Java kalbos GraphWalker anotaciją, kurioje būtų pateikta pradinė briauna ar viršūnė ir generavimo parametrai. Paleidus programą nurodytu būdu pereinamas modelis ir vykdomas programinis kodas pagal modelio interfeiso realizaciją. Baigus vykdymą konsolėje pateikiamas pereitas kelias bei kita su generavimu susijusi informacija.

Offline testus generuoti taip pat yra du būdai. Testų generavimo simuliacija pateikiama „GraphWalker Studio“. Paleidus testų vykdymą grafiškai atvaizduojamas pereitas kelias, tačiau jo eksportuoti ar panaudoti kitur nėra jokios galimybės, tam reikia kito generavimo įrankio. Paprasčiausia panaudoti komandinės eilutės įrankį. Komandoje turi būti pateiktas kelias iki

modelio bei generavimo parametrai. Nurodant modelį ir testų generavimo sustojimo sąlygą, komandinės eilutės sąsajoje JSON formatu išspausdinamas kelias ir, jei norima, papildoma informacija apie modelio būseną.

GraphWalker testų generavimui naudojami parametrai, kurie nurodo, kokiu būdu padengti, kada ir kaip nustatyti sustojimą. Parametrų aprašymo šablonas yra pateiktas 14 pav. Vienas iš galimų naudoti pavyzdžių – „random(edge_coverage(100))“. Šiuo atveju būtų testai generuojami atsitiktinio klaidžiojimo algoritmu, kol būtų padengtos visos kraštinės. Galima aprašyti ir kelias sustojimo sąlygas, atskirtas loginiais operatoriais „AND“ ir „OR“. Taip pat galima nurodyti kelis generavimo būdus, kurie bus vykdomi paeiliui. Tiek generatoriaus, tiek sustojimo sąlygos tipo funkcijas galima sukurti pačiam testuotojui.



14 pav. Testų išrinkimo kriterijų aprašymo šablonas

Testo kelio sudarymui dokumentacijoje pateikiami tokie generatoriai:

- **random**. Keliaus per modelį visiškai atsitiktiniu būdu. Algoritmas atsitiktiniu būdu pasirenka išeinančią briauną iš viršūnės ir kartoja procesą sekančioje viršūnėje.
- **weighted_random**. Veikia panašiai kaip ir atsitiktinio kelio generatorius, tačiau yra naudojamos svorių reikšmės generuojant kelią. Svoris yra priskiriamas tik briaunoms. Jis išreiškia tikimybę, kad briauna bus pasirinkta.
- **quick_random**. Bandoma įvykdyti trumpiausią kelią per modelį, bet greitu būdu. Algoritmas veikia taip:
  - Atsitiktinai pasirenkama viršūnė, kuri dar nėra aplankyta.
  - Pasirenkamas trumpiausias kelias iki briaunos pagal Dijkstros algoritmą [Dij59].
  - Einama tuo keliu ir žiūrimos visos įvykdytos briaunos kaip aplankytos.
  - Kai pasiekiama pasirinkta briauna, grįžtama atgal į pirmąjį žingsnį.

Sustojimo sąlygų tipai:

- **a_star** sugeneruos trumpiausią kelią iki nurodytos viršūnės ar briaunos.
- **edge_coverage.** Sustojimo sąlyga yra sveikais skaičius, nurodantis norimą padengimo procentą. Vykdymas bus nutrauktas, kai bus pereitas atitinkamas procentų kiekis briaunų. Jei briauna yra pereinama antrą kartą, laikoma, kad ji aplankyta buvo tik kartą.

- **vertex_coverage**. Sustojimo sąlyga, taip pat yra sveikasis skaičius, kuris nurodo norimą padengimo procentą. Veikimas panašus, kaip ir edge_coverage, tačiau skaičiuojami aplankyti procentai viršūnių.
- **requirement_coverage**. Sustojimo sąlyga, yra norimas reikalavimų padengimo procentas, veikia analogiškai ir edge_coverage ir vertex_coverage, tačiau skaičiuojami padengtų reikalavimų procentai.
- **dependency_edge_coverage**. Sustojimo sąlyga yra sveikasis skaičius, nurodantis priklausomybių slenkstį. Vykdymas sustabdomas, jei visos briaunos, turinčios lygų nurodytam priklausomybės slenkstį arba už jį didesnį, yra aplankytos.
- **reached_vertex**. Sustojimo sąlyga yra viršūnės pavadinimas. Kai vykdymo viršūnė yra pasiekta, testas baigiamas.
- **reached_edge**. Sustojimo sąlyga yra briaunos pavadinimas. Kai vykdant briauna yra pasiekta, testas baigiamas.
- **time_duration**. Sustojimo sąlyga yra laikas sekundėmis, kuris reiškia, kiek laiko bus leista generatoriui kurti kelią. Reikia atkreipti dėmesį, kad laikas yra pritaikomas visam testui. Pavyzdžiui, jei yra du modeliai su bendromis būsenomis ir abu turi „time_duration“ nustatytą 60 sekundžių, generavimas iš abiejų modelių bus sustabdytas po 60 sekundžių, net jei vienas iš modelių nebuvo aplankytas.
- **length**. Sustojimo sąlyga yra skaičius atstojantis viršūnių-briaunų poros, sukurtos generatoriaus. Pavyzdžiui, jei tai būtų 110, būtų pereita 110 briaunų ir 110 viršūnių.
- **never**. Tai yra specialus sustojimo sąlygos tipas, neturintis parametro. Naudojant jį generatoriaus darbas niekada nebus nutraukiamas.

6.4.3. MBTsuite testų generavimas

Testai generuojami pasirinkus vieną iš šešių strategijų. Toliau pateikiamas jų aprašymas.

- **Pilnas kelių padengimas**. Įvykdomos visos įmanomos kelių kombinacijos pagal duotą maksimalų vykdymo ciklų kiekį. Papildomai prieš generavimą galima apriboti paties testo atvejo dydį pagal atliekamų žingsnių skaičių testo atvejyje ir nustatyti maksimalų leistiną ciklų skaičių. Iš visų strategijų gaunamas didžiausias testų rinkinys.
- **Nurodytas kelias**. Galima sukurti specialią modelio komponentę, kurioje aprašomi visi keliai, pagal kuriuos bus generuojamas testų rinkinys. Briaunos identifikuojamos pagal joms priskiriamus pavadinimus.

- **Kelias pagal pavadinimą**. Įrankis leidžia briaunoms priskirti pavadinimus ir pagal pasirinktą pavadinimą turinčias briaunas generuoti testus. Galima pasirinkti tik vieną pavadinimą vienam testų rinkiniui sudaryti.
- **Atsitiktinis kelias**. Testai generuojami pagal atsitiktinio klaidžiojimo algoritmu. Kaip parametrai gali būti gali būti maksimalus nueito kelio ilgis, norimų testų atvejų skaičius. Maksimalus bandymų skaičius sėkmingai sugeneruoti testus. Testo generavimas laikomas sėkmingu, jei per nustatytą žingsnių skaičių pavyksta pasiekti galutinę būseną. Prieš generuojant testus nustatoma viršutinė ribos, kiek bus bandymų sudaryti testų atvejus bei kiek tikimasi gauti testų atvejų. Bandymas laikomas nepavykusiu, jei nepasiekus pabaigos taško negalima pereiti iš į sekančią viršūnę. Testų gali būti mažiau nei tikimasi dėl keleto priežasčių:
    - Galimų modelio perėjimų kelių yra mažiau nei nurodytas skaičius;
    - Bandymų skaičius mažesnis nei tikimasi gauti testų;
    - Testai, viršijantys nurodytą maksimalų žingsnių skaičių, nepridedami į testų rinkinį;
    - Sugeneruojamas besidubliuojantys testo atvejai;
    - Perėjimų pradinės sąlygos nepatenkinamos.
- **Trumpiausias galimas kelias**. Generavimo metu ieškoma trumpiausio kelio nuo pradinės iki galutinės būsenos. Nustatyti trumpiausią kelią galima pagal atliekamų žingsnių skaičių, kelio kainą arba trukmę. Jei pagal pasirinktą kriterijų yra daugiau nei vienas trumpiausias kelias, testų rinkinyje pateikiami visi trumpiausi keliai.
- **Panaudojimo padengimo**. Šiuo metu įrankyje yra šis pasirinkimas, tačiau dokumentacijoje jis neaprašytas. Išbandžius šią strategiją buvo gautos vartotojui neinformatyvios klaidos. Tikėtina, kad tai nėra pilnai įgyvendinta sistemos dalis.

Norint sumažinti testų atvejų rinkinį galima pasinaudoti filtrais. Filtrai skirstomi pagal šias kategorijas:

- **Padengimo**. Padengimo filtrai išrenka minimalų testų atvejų skaičių padengiant visus susijusius objektus bent karta. Galimi šie padengimo filtrai:
    - **Viršūnių padengimo**. Sukuria minimalų testų rinkinį, kuris padengia visas viršūnės bent kartą.
    - **Briaunų padengimo**. Sukuria minimalų testų rinkinį, kuris padengia visas briaunas bent kartą.

    - **Testų žingsnių ir patikrinimo taškų padengimo**. Sukuria minimalų testų rinkinį, kuris padengia visas viršūnes ir briaunas, kurios turi „TestStep“ ar „VP“ stereotipus.
    - **Reikalavimų padengimo**. Sukuria minimalų testų rinkinį, kuris padengia pasirinktus reikalavimus bent kartą.
- **Intervalo filtrai**. Srities filtrai sukuria minimalų testų rinkinio poaibį, kuriame filtro kriterijus, susumavus kiekvieną testo atvejį, atitinka nurodytą intervalą. Galimi šie intervalo filtrai:
    - **Kainos filtras**. Sukuria minimalų testų rinkinį, kurio kainos reikšmių suma testų atvejuose atitinka nustatytą intervalą.
    - **Trukmės filtras**. Sukuria minimalų testų rinkinį, kurio trukmės reikšmių suma testų atvejuose atitinka nustatytą intervalą.
    - **Ilgio filtras**. Sukuria minimalų testų rinkinį, kuriame testų atvejų ilgis atitinka nustatytą intervalą.
- **Dinaminių žymių filtras**. Sukuria testų atvejų rinkinį remiantis pasirinktomis žymėmis ir loginiais operatoriais.

6.5. Gauti testų kūrimo rezultatai

6.5.1. ProB rezultatai

Sugeneruoti testų atvejai yra pateikiami XML formatu (15 pav.). Kiekvienas testų atvejis atskiriamas „test-case“ žyme. Viduje paprastai yra pradinės būsenos inicijavimo parametrai („initialisation“ žymė), kurie nagrinėjamoje sistemoje visada yra vienodi, ir teste atliekami žingsniai („step“ žymės). Žingsnių žymėse aprašoma, kokia operacija atliekama, o jų vaikiniuose elementuose yra pateikti kvietimo parametrai ir kokie būsenos pakeitimai gauti. Pagal tai galima atlikti tikrinimus, ar sistemoje būsena pasikeitė atitinkamai.

```
<?xml version="1.0" encoding="UTF-8"?>
<extended_test_suite>
  <test_case>
    <initialisation>
      <value type="constant" name="FILES_CHILDREN">{(mf|->df_gsm),(mf|->ef_iccid),(df_gsm|->ef_lp),(df_gsm|->ef_imsi),(df_gsm|->ef_ad)}</value>
      <value type="constant" name="PERMISSION_READ">{(ef_iccid|->never),(ef_lp|->always),(ef_imsi|->chv),(ef_ad|->adm)}</value>
      <value type="constant" name="MAX_CHV">2</value>
      <value type="constant" name="MAX_UNBLOCK_CHV">2</value>
      <value type="constant" name="PUK">c3</value>
      <value type="variable" name="current_file">{}</value>
      <value type="variable" name="current_directory">mf</value>
      <value type="variable" name="counter_chv">2</value>
      <value type="variable" name="counter_unblock_chv">2</value>
      <value type="variable" name="blocked_chv_status">unblocked</value>
      <value type="variable" name="blocked_status">unblocked</value>
      <value type="variable" name="permission_session">{(always|->true),(chv|->false),(never|->false),(adm|->false)}</value>
      <value type="variable" name="pin">c1</value>
      <value type="variable" name="data">{(ef_iccid|->data1),(ef_lp|->data2),(ef_imsi|->data3),(ef_ad|->data4)}</value>
    </initialisation>
    <step name="VERIFY_CHV_FAIL">
      <value name="code">c2</value>
      <modified name="counter_chv">1</modified>
    </step>
    <step name="VERIFY_CHV_LAST_FAIL">
      <value name="code">c2</value>
      <modified name="counter_chv">0</modified>
      <modified name="blocked_chv_status">blocked</modified>
    </step>
  </test_case>
</extended_test_suite>
```

15 pav. ProB gautas testo atvejis pagal vieną iš testavimo tikslų

Pirmieji bandymai buvo atlikti pagal [UL10] pateiktą operacinį modelį su pateiktais testavimo tikslais, padengimo kriterijuose nurodžius vien tik VERIFY_CHV komandą, kuri turėjo kelias vykdymo šakas. Nenurodžius testavimo tikslo, apribojimų sprendimo algoritmu buvo gautas tik vienas testo atvejis. Pasidarė aišku, kad keleto vykdymo šakų aprašymo galimybė vienoje modelio operacijoje testų generavimui yra nenaudinga, ir buvo priimtas sprendimas perdaryti operacinį modelį, kad būtų sukurta po vieną operaciją kiekvienai operacijos šakai. Perdarius operacijas gautos keturios VERIFY_CHV komandos (joms priskyrus naujus operacijų pavadinimus), su skirtingomis pradinėmis vykdymo sąlygomis. Po operacinio modelio perdarymo pasidarė įmanoma įrankiu pasirinkti padengti ne komandos vykdymą, bet visų jos šakų.

Svarbu paminėti, kad testavimo tikslų aprašyme negali būti AND ir OR loginių operatorių, todėl apibrėžtus testavimo tikslus reikėjo performuluoti, jie pateikti 5 lentelėje. Perrašant tikslus buvo atsižvelgta į veikimo algoritmą ir perdarytas operacijas. Pavyzdžiui, atkartojant tikslą, kai PIN kodas neužblokuotas ir įvesta teisinga reikšmė, pakako nurodyti padengti funkciją VERIFY_CHV_GOOD_PIN ir nurodyti sustoti, kai PIN kodas nėra užblokuotas (PIN kodas vos pradėjus vykdyti jau nėra užblokuotas, bet būtina padengti VERIFY_CHV_GOOD_PIN). Priklausomai nuo testavimo tikslo, buvo sugeneruojami 3 arba 4 testų atvejai.

Pradinis tikslas (pagal šaltinį)	Testo tikslas
blocked_chv_status = blocked	blocked_chv_status = blocked
blocked_chv_status /= blocked & pin = code	Padengti funkciją VERIFY_CHV_GOOD_PIN ir sustoti, kai blocked_chv_status /= blocked
blocked_chv_status /= blocked & pin /= code & counter_chv = 1	Padengti funkciją VERIFY_CHV_LAST_FAIL ir sustoti, kai counter_chv = 0

blocked_chv_status /= blocked & pin /= code & counter_chv /= 1	Padengti funkciją VERIFY_CHV_FAIL ir sustoti, kai counter_chv = MAX_CHV – 1

5 lentelė. Performuluoti testų tikslai dėl galimybės nebuvimo testų tiksluose panaudoti AND ir OR loginių operacijų.

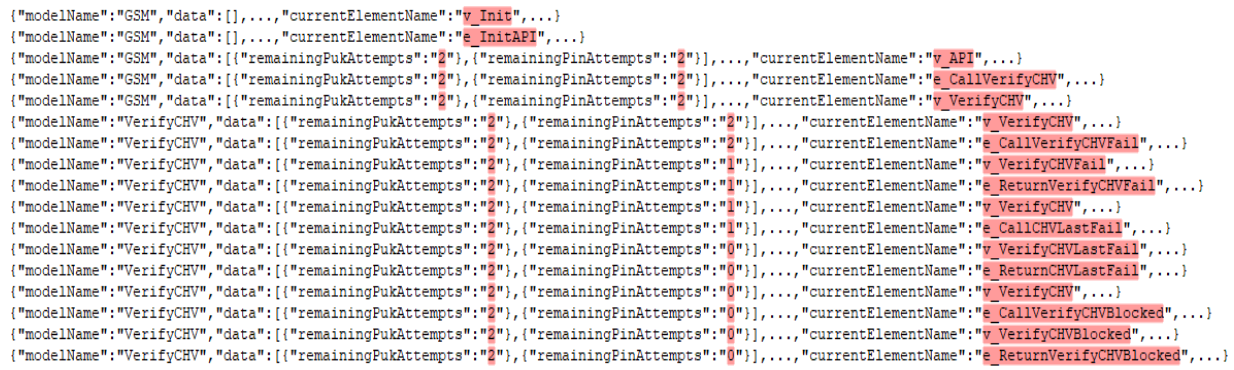
Buvo atliktas dar vienas bandymas norint gauti didesnį atliktų operacijų skaičių viename testo atvejyje – testavimo tiksle nurodyti, kad kortelė būtų visiškai užblokuota. Norint pasiekti šią būseną reikėjo bent 10 iteracijų, nes PUK kodui užblokuoti reikia paeiliui 10 kartų jį suvesti neteisingai. Einant į didesnį gylį kiekvienos iteracijos vykdymo trukmė didėjo eksponentiškai, todėl nepavyko gauti rezultatų. Tam, kad būtų pasiekta būsena užblokuotos kortelės būsena, užblokavimui reikalingų paeiliui bandymų skaičius buvo sumažintas iki dviejų.

Po minėtų pakeitimų, padengiant visas aprašytas operacijas modelio tikrinimo būdu buvo sugeneruota 12 testų atvejų, o naudojant apribojimų sprendimą gauta 16 testų atvejų. Apribojimų sprendime buvo galima tikėtis tokio skaičiaus dėl to, kad operacijų padengimo kriterijuose aprašyta būtent 16 operacijų. Modelio tikrinime gauta mažiau testų atvejų, nes modelio tikrintojas gali aptikti pasikartojančias būsenas bei yra operacijų, kurias norint įvykdyti būtina prieš tai įvykdyti kitas operacijas. Pavyzdžiui, norint įvykdyti VERIFY_CHV_BLOCKED operaciją, būtina prieš tai įvykdyti UNBLOCK_CHV_FAIL ir UNBLOCK_CHV_LAST_FAIL. Tačiau testų atvejai, kuriuose dalyvauja failų direktorijų pasirinkimas nebuvo sutraukiami į vieną testo atvejį, nes testuose gaunama skirtinga būsena – vienuose testuose pakako pasirinkti MF direktoriją, kituose buvo būtina pasirinkti sėkmingai kitą direktoriją.

6.5.2. GraphWalker rezultatai

GraphWalker įprastai sugeneruoja vieną testo kelią, todėl būtų gaunamas tik vienas testo atvejis. Dėl to kilo dvi problemos: kortelė gali būti užblokuota, todėl modelis gali būti pilnai nepereitas bei nėra testų atvejų rinkinio. Norint išspręsti šias problemas įvesta papildoma viršūnė, kurioje modelis nustatomas į pradinę būseną. Atėjus į šią viršūnę laikoma, kad testo atvejis baigtas ir pradedamas naujas testas, nukreipiant į pirmąją viršūnę. Kadangi norint sudaryti testų rinkinį iš *online* būdu sugeneruoto testo kilo papildomų iššūkių, bandymai atlikti *offline* testavimo būdu.

Kita problema buvo susijusi su Maven įrankiu. Generuojant testus negalima pasinaudoti duomenų modeliu. Ši savybė labai naudinga atliekant sistemos būsenos verifikavimą. Duomenų modeliu buvo galima pasinaudoti tik generuojant testus komandinės eilutes pagalba, komandos parametruose nurodžius spausdinti visą modelio būseną. Taip gaunamas pilnas duomenų modelis, iš kurio reikia papildomų įrankių pagalba išsiskirti reikalingus duomenis (16 pav.).



16 pav. GraphWalker *offline* būdu sukurtas testas. Raudonai pažymėti modelio būsenos atributai bei operacijos pavadinimai, kuriuos galima panaudoti testų atvejuose būsenų tikrinimui.

Testų generavimo metu vienintelis parametras, kuriuo galima manipuliuoti buvo generatoriaus funkcija. Norint apeiti visą modelį kuo greičiau, buvo pasirinka funkcija „quick_random“, tačiau įrankis vykdydamas perėjimus 6.4.2 skyrelyje aprašytą algoritmą neatsižvelgia į perėjimų pradines sąlygas, todėl vykdymas užstringa. Nepavykus pasinaudoti šia funkcija, pereita prie kitų testų generavimo funkcijų.

Norint panaudoti „weighted_random“ funkciją, modelyje reikia papildomai uždėti ant perėjimų vykdymo tikimybes. Sukurtame modelyje svoriais galima įtakoti operacijų vykdymo dažnį, tačiau negalima įtakoti operacijų šakų vykdymo, nes jų vykdymas priklauso nuo skriptų. Skriptai gali suteikti platesnes galimybes, nei vien tik paveikti operacijų šakų vykdymo dažnius. Be to, atsitiktinių reikšmių parinkimas pridėjus naujus perėjimus kiekvienai galimai atsitiktinei reikšmei smarkiai padidintų grafinį modelį. Dėl šių priežasčių buvo nuspręsta vietoj galimybės naudotis svoriais, naudoti „random“ generavimo strategiją bei skriptų pagalba arba kitais būdais įtakoti atsitiktinių reikšmių parinkimo dažnį.

Naudojant „random“ buvo bandoma padengti visas modelio briaunas. Sudėtingiausia pagal SIM kortelės reikalavimus būtų pasiekti po 10 kartų paeiliui nesėkmingų bandymų suvesti PUK kodą gauti atsakymą, kad kortelė užblokuota. Todėl, kaip ir ProB atveju, padaryta, kad PIN ir PUK kodas užsiblokuotų po 2 nesėkmingų bandymų. Kita sudėtingai pasiekiama būsena – sėkmingas perskaitymas iš failo. Norint greičiau atsitiktinio vaikščiojimo algoritmu pasiekti šią būseną, buvo papildomai įvestas apribojimas, kad turint CHV failo skaitymo teises nebūtų galima kviesti funkcijų VERIFY_CHV ir UNBLOCK_CHV. Su šiais apribojimais, iš dvidešimt bandymų generuoti testus, vidutiniškai prireikė 929 testų atvejų padengti visoms kraštinėms. Iš atliktų bandymų mažiausiai reikėjo 164 testų atvejų, kol visos briaunos buvo padengtos, daugiausia – 2965. Testus filtruoti galima tik rankiniu būdu.

6.5.3. MBTsuite rezultatai

Sugeneruoti testų atvejai prasideda nuo SIM kortelės inicializavimo. Tai atliekama kiekviename sugeneruotame teste. Skripto pagalba sugeneruotas atsitiktinis PIN ir PUK kodas turėtų būti parinktas skirtingas kiekvienam testo atvejui, tačiau pavyko tik sugeneruoti atsitiktinius PIN ir PUK, kurie vienam testų rinkiniui yra vienodi. Tikėtina, kad taip įvyksta dėl testų generavimo algoritmo optimizavimo. Įvykdžius generavimą iš naujo, PIN ir PUK reikšmės pasikeičia į naujas atsitiktines reikšmes.

Po inicializavimo einama per logines šakas ir atrenkama vykdoma komanda. Testuojant komandų vykdymą iš pradžių įvykdoma komandą ir pagal esamą sistemos modelio būseną patikrinama, ar SIM kortelės būsena atitinka. Po to grįžtama atgal ir atrenkama komanda vykdymui iš naujo. Maksimalus įvykdytų komandų skaičius priklauso nuo testų generavimo parametro, apibrėžiančio leistiną vykdymo ciklų kiekį. Nebeleidžiant daugiau pasikartojimų, pasinaudojama likusia galima vykdymo šaka, nukreipiančia į vykdymo pabaigos tašką. Gauto testo pavyzdys pateiktas 17 pav.

Main_0002

Test Steps:

Step	Type	Step Name	Step Description	Expected Result	Requirements	Code
1	TESTSTEP	Init PIN code	Creating PIN code & initializing maximum available attempts in model			String pin = "8930";
2	TESTSTEP	Init PUK code	Creating PUK code & initializing maximum available attempts in model			String puk = "52285825";
3	TESTSTEP	Init file system	Initializing file system for SIM			FileSystem fileSystem = new FileSystem();
4	TESTSTEP	Init SIM	Initiializing SIM			SIM sim = new SIM(pin, puk, fileSystem);
5	TESTSTEP	Verify good PIN	Entering valid PIN code		VERIFY_CHV REQ1	String result = sim.verifyChv("8930");
6	VP	Access granted	Checking function status code and remaining PIN code attempts	Function returns success status code, remaining PIN code attempts is set to maximum number of attempts	VERIFY_CHV REQ1	assert(result.equals ("SW_9000")); assert (sim.status().contains ("remaining_pin_attempts: 2"));

17 pav. MBTsuite sugeneruoto testo atvejo pavyzdys

Bandant generuoti testų atvejus pagal trumpiausią galimą kelią gautas tik vienas testas, kuriuo iš inicializavimo pereinama į galutinę būseną nevykdžius nė vienos komandos. Kuriant modelius galima nustatyti būsimų perėjimų kainą ir trukmę. Jie galėjo būti panaudoti testų generavime nustatant trumpiausią kelią, tačiau testuojamai sistemai naudoti šiuos atributus nebūtų prasminga.

Vykdant testų generavimą keletą kartų pagal atsitiktinius vykdymo kelius, gaunami skirtingų dydžių testų rinkiniai. Patys testai sugeneruojami įvairaus ilgio. Paleidus testų generavimą keletą kartų su ta pačia atsitiktinių skaičių generatoriaus pradine parametro reikšme (angl. seed), gaunami vienodi testų rezultatai. Norint gauti kitokius rezultatus reikia pakeisti atsitiktinių skaičių generatoriaus parametro reikšmę arba iš naujo importuoti modelį. Šioje strategijoje nebuvo galimybės pilnai padengti modelio funkcionalumo.

Naudojant kelio pagal pavadinimą strategiją, reikėjo loginėse šakose vienai briaunai priskirti pavadinimą, kuriuo remiantis bus generuojamas testo atvejis. Šios strategijos esamam modeliui nebuvo galima prasmingai pritaikyti dėl ciklų vykdyme. Įvykdžius komandą norint išeiti iš ciklo grįžtus atgal į tą pačią loginę šaką, kurioje nuspręsta įvykdyti komandą, reikia įvykdyti kitokį perėjimą nei buvo vykdoma prieš tai. Tai reiškia, kad abiejų perėjimų pavadinimai turi būti vienodi, bet įrankis to neleidžia, nes tokiu atveju negali nuspręsti, kuri šaka turėtų būti vykdoma. Dėl to šios strategijos pagal turimą modelį neįmanoma prasmingai panaudoti.

Prasmingai panaudoti nurodytų kelių strategiją buvo daug pastangų reikalaujanti užduotis. Testų atvejų gaunama būtent tiek, kiek kelių aprašoma ir įtraukiama į testų generavimą. Kiekvieno testo atvejo kelias aprašomas su jo pavadinimu. Į aprašymą reikėjo įtraukti ir subdiagramos kelią, todėl aprašyti keliai buvo labai ilgi. Pavyzdžiui, norint ištestuoti vieno neteisingo PIN kodo įvedimą reikėjo aprašyti 22 žingsnius. Generavimo procese patekus į subdiagramą, tolimesnis kelias skaitomas iš ten pagal atitinkamo testo atvejo pavadinimą.

Efektyviausi rezultatai gauti naudojant pilną kelių padengimą. Maksimalus įvykdomų komandų skaičius nustatomas leistinų ciklų parametru. 6 lentelėje matyti testų atvejų kiekio ir generavimo laiko priklausomybė nuo maksimalaus leistino vykdomų operacijų kiekio viename testo atvejyje. Deja, leidžiant generuoti testų atvejus iš daugiau kaip 3 komandų, sugeneruotų testų atvejų užkrovimas aplikacijoje truko labai ilgai. Norint to išvengti reikėjo perkrauti MBTsuite aplikaciją. Perkrovus aplikaciją buvo galima sugeneruotiems testams pritaikyti filtrus arba juos konvertuoti į kitus formatus, tačiau pačioje MBTsuite aplikacijoje bandant atidaryti sugeneruotų testų atvejų rinkinį, sistema užstrigdavo.

Maksimalus leistinas testo kelio ilgis	Sugeneruotų testų atvejų kiekis	Generavimo trukmė, s
1	10	0,5
2	91	7,6
3	820	18,3
4	7380	150,9
5	66388	1204,8

6 lentelė. MBTsuite sugeneruotų testų atvejų kiekio ir generavimo trukmės priklausomybė nuo maksimalaus leistino testo kelio ilgio. Kelio ilgis nurodo vykdomų operacijų skaičių viename teste.

Norint pasiekti būseną, kai kortelė yra užblokuota, reikia neteisingai suvesti PUK kodą 10 kartų paeiliui neteisingai. Kadangi vykdymo trukmė didinant vykdomų komandų skaičių viename testo atvejyje augo eksponentiškai, reikėjo sumažinti reikalingų komandų skaičių kortelės užblokavimui. Buvo nustatyta, kad PIN ir PUK užblokavimui pakaktų juos suvesti neteisingai po 2 kartus. Mažinti skaičiaus labiau nebuvo galima, nes būtų prarandama dalis sistemos būsenų. Todėl norint gauti pranešimą iš sistemos, kad PUK užblokuotas mažiausiai reikėjo 2 komandų.

Sukūrus apribojimus, kad kortelė būtų užblokuojama po dviejų nesėkmingų PUK kodo įvedimų bandymų, kai galimi 5 ciklai, sugeneruotų testų atvejų liko tik 3574. Pritaikius viršūnių arba kraštinių padengimo filtrą, liko tik 5 testai. Visuose iš jų bandoma vykdyti po keletą komandų.

Sugeneruoti testų atvejai galėjo būti eksportuojami įvairiais formatais. Kadangi modelio susiejimas su konkrečiais testų atvejais Java kalba pasirinktas Java formato šablonas. Žemiau pateiktas MBTsuite sugeneruotas testo atvejis eksportuotas į Java šabloną (18 pav.). Matoma, kad testai atkuriami pagal priskirto kodo reikšmes, virš kodo pridėjus komentarą apie testo žingsnio aprašymą, po kodo pridėjus komentarą apie laukiamus rezultatus. Virš sugeneruoto testo atvejo kodo pateikiami duomenys apie testų generavimo procesą bei kokius reikalavimus testo atvejis padengia.

```
// do your imports here

/**     Test Class Main_0002
        Source Model: Main
        Source Tree: FPC
        Model was imported on 4/14/20 8:24 PMby user 'abc' from 'Enterprise Architect' with start diagram 'Main'
        Test case was exported on 4/14/20 8:25 PM by user 'abc'
*/
public class Main_0002{
    public static void main(String[] args){
                Main_0002 ts = new Main_0002();
                ts.Main_0002();
        }


        /** TestCase: Main_0002
        Requirements:- VERIFY_CHV REQ1
        - VERIFY_CHV REQ1


        */
        public void Main_0002(){

                // Creating PIN code & initializing maximum available attempts in model
                String pin = "8930";

                // Creating PUK code & initializing maximum available attempts in model
                String puk = "52285825";

                // Initializing file system for SIM
                FileSystem fileSystem = new FileSystem();

                // Initiializing SIM
                SIM sim = new SIM(pin, puk, fileSystem);

                // Entering valid PIN code
                String result = sim.verifyChv("8930");

                // Checking function status code and remaining PIN code attempts
                assert(result.equals("SW_9000"));
                assert(sim.status().contains("remaining_pin_attempts: 2"));
                // Expected Result: Function returns success status code, remaining PIN code attempts is set to maximum number of attempts

        }
}
```

18 pav. Eksportuotas testo atvejis su Java kalbos testo šablonu

6.6. Atvejo analizės rezultatų apibendrinimas

Visuose modeliuose vienodai apibrėžtas sistemos veikimas: vykdomos operacijos ir sekama SIM kortelės būsena. Skirtumai tarp įrankių išryškėjo palaikomuose testų generavimo ir išrinkimo algoritmuose. Daugiausia generavimo algoritmų turėjo MBTsuite, kiek mažiau GraphWalker ir tik du ProB. Įrankiai turintys mažiau testų generavimo būdų turi savybių, kurių neturi kiti įrankiai. Įrankyje, kuris turi daugiausia generavimo algoritmų, dalis algoritmų buvo netenkinantys testavimo tikslų, dalis net negali būti pritaikyta.

Norint palyginti sugeneruotų testų atvejų kiekį, GraphWalker įrankyje reikėjo papildyti modelį nauja viršūne, kurios paskirtis – pradėti naują testo atvejį, tuo tarpu ProB ir GraphWalker sudarė testų rinkinį be papildomų veiksmų. Sudarius testų rinkinius visoms modelio operacijoms padengti, daugiausia testų atvejų gauta naudojant GraphWalker ir MBTsuite. MBTsuite buvo galima generuoti testus iki norimo operacijų padengimo kiekio viename teste. MBTsuite turi galimybę sugeneruotus testų atvejus filtruoti automatiniu būdu. Kadangi MBTsuite taip pat palaiko testų iki pasirinkto ilgio generavimą, taip juose padengiant daugiau operacijų, po tokio filtravimo buvo gautas mažesnis testų rinkinys nei pateikė ProB. GraphWalker neturi galimybės automatiniu būdu filtruoti testų atvejų. Dėl nesuderinamumo su aprašytais skriptais nebuvo galimybės panaudoti GraphWalker operacijų padengimui efektyviausią strategiją

„quick_random“. Dėl to testų atvejų skaičiaus sumažinimui bet kuriuo atveju reikia papildomų pastangų.

GraphWalker atsitiktinio testavimo algoritmas yra efektyvesnis nei MBTsuite dėl naudingesnių sustojimo sąlygų (šiuo atveju norėta padengti visus perėjimus). Pagal tai galima spręsti, kad GraphWalker labiau pritaikytas atsitiktiniam testavimui, o MBTsuite – išsamiam testavimui.

Norint mažo ir efektyvaus rinkinio, konkuruoja ProB ir MBTsuite. ProB generavimo metu vykdomas filtravimas automatiškai sudarant mažą, efektyvų testų rinkinį, o su MBTsuite sukūrus testų rinkinį filtrai parenkami ir pritaikomi rankiniu būdu, todėl testų rinkinio efektyvumas priklauso nuo testuotojo.

Visais įrankiais pagal testavimo tikslus reikalaujant didesnio ilgio testų generavimo trukmė auga eksponentiškai, todėl modeliuose buvo būtina vietoj realaus modelio įvesti supaprastinimus. Testus generuojant atsitiktiniu būdu (GraphWalker), supaprastinimų reikėjo daugiau.

Modeliuojant buvo atsižvelgta ir į klaidų paiešką modeliuose. GraphWalker surasti sintaksės klaidą be papildomų įrankių gali būti daug laiko reikalaujanti užduotis, tačiau ir aprašius modelį pagal taisykles reikia papildomos JavaScript palaikančios aplinkos patikrinimui, ar visi keliai gali būti pasiekti, nes dideliame grafe viršūnė gali būti ilgai nepasiekiama ir dėl atsitiktinumo. Su MBTsuite dauguma klaidų buvo tiksliai nurodomos po testų generavimo (išspausdinamos nepasiekiamos viršūnės arba nurodoma tiksli sintaksės klaidos vieta). Tačiau papildoma problema rašant skriptus yra ribotas įvesties ilgis, padalinti skriptą į mažesnes dalis reikia papildomų pastangų. Naudojant ProB pasinaudojus invariantais sistema prieš testų generavimą parodo, kuri operacija pažeidžia nekintamas sistemos savybes, šiuo įrankiu reikėjo pastebimai mažiau laiko ieškant klaidų modelyje.

Duomenų modelyje buvo mažo dydžio aibės PIN kodo, PUK kodo ar failo pasirinkimui, iš kurių MBTsuite ir GraphWalker reikšmes išrinkdavo atsitiktiniu būdu, o ProB išrenka reikšmes perrinkimo būdu, kol randama tinkama. Kol modelis yra nedidelis, jei atsitiktinių reikšmių parinkimas netenkina ir norima pilno galimų reikšmių padengimo, galima pridėti papildomus perėjimus modelyje kiekvienai reikšmei įgyti. Tačiau jei būtų neribota galimų reikšmių aibė norint patikrinti, tinkamai veikia sistema modeliuojant su GraphWalker ir MBTsuite reikalautų papildomo darbo apibrėžiant skriptuose, kaip reikšmės turėtų būti parenkamos, o ProB turi algoritmą, kuris šią problemą sprendžia automatiškai.

Testų atvejų sukonkretinimas be papildomų įrankių galimas tik naudojant MBTsuite. Kituose įrankiuose abstrakčiuose testuose su atliekamais žingsniais pateikiami modelio

duomenys, kurių dalis gali būti panaudojami, kaip parametrai atliekamiems veiksmams ar kaip testavimo orakulas.

Kiekviename įrankyje buvo rasta nedidelių problemų dėl neveikiančio funkcionalumo arba kliūčių kuriant modelius. Problemos nebuvo aprašytos dokumentacijose, taip pat jose trūko tikslaus veikimo paaiškinimo, kai kur tik buvo nuorodos į pavyzdžius. Taip pat įvedimo laukeliai skriptams labai maži, su šia problema nebuvo susidurta tik naudojant ProB, tačiau šiam įrankiui visas modelis pateikiamas tekstiniu pavidalu.

7. Testavimo įrankių palyginimas

Šiame skyrelyje pateikiami bendri įrankių palyginimai, neatsižvelgiant į atvejo analizę. Po kiekvienos lentelės pateikiamas rezultatų apibendrinimas. Į lenteles neįtraukiami kriterijai, kuriuose visi įrankiai įvertinti vienodai.

7.1. Testavimo modeliai ir jų galimybės

Kriterijai	GraphWalker	ProB	MBTsuite
Modelio atvaizdavimo pavidalas	Grafinis	Tekstinis	Grafinis
Modelio kalbos tipas	Perėjimų	Būsenų	Perėjimų
Submodeliavimas	+	-	+
Reikalavimų žymių aprašymas	+	-	+
Invariantų aprašymas	-	+	-
Pritaikymas sudėtingiems duomenų modeliams	-	+	-

7 lentelė. Modeliavimo galimybių palyginimas

Pagal modelių galimybių palyginimą matyti, kad aprašomas išsamiausiai modelis ProB modeliavimo kalba. Tačiau jo atvaizdavimas nėra paprasčiausias, neturi supaprastinimo savybių – neleidžia modelio skaidyti į smulkesnius, nėra modelio grafinio atvaizdavimo. ProB modelyje nereikalingi skriptai, kuriuos reikia slėpti po grafiniais objektais – visas modelis aprašytas kodu. Kita vertus, grafiniuose GraphWalker ir MBTsuite modeliuose atvaizduojamas funkcionalumas lengviau suprantamas, juose leidžiama į modelio komponentes priskirti įgyvendinamus reikalavimus ir kitus papildomus duomenis. Be to, submodeliavimo galimybė daro šiuos įrankius žymiai tinkamesnius plėtimui (angl. scalability).

7.2. Testų generavimas

Testų generavimo bendrosios charakteristikos	GraphWalker	ProB	MBTsuite
Testų eksportavimas pagal aprašytą programavimo kalbos kodą modeliuose	Iš dalies	-	+
Testų filtravimas	-	Automatinis	+
Sustojimo sąlygų keitimas	+	-	+
Online ir *offline* testavimas	Abu	*Offline*	*Offline*
Automatizuotas testų atvejų perrinkimas testų rinkiniui sudaryti	-	+	+
Ciklišku mo tikrinimas	+	Priklausomai nuo algoritmo	+
Operacijų parametrų automatinis perrinkimas	-	+	-
Testavimo orakulai	Tik naudojant komandinę eilutę	+	+
Reikalavimų atsekamumas	Tik naudojant Maven	-	+

8 lentelė. Testų generavimo galimybių palyginimas

Visi įrankiai palaiko skriptų rašymo galimybę, todėl gali pateikti konkrečius duomenis testų atvejams, kurie gali būti pateikiami kaip testavimo orakulas, tačiau pilnai konvertavimą į konkrečius testų atvejus palaiko tik MBTsuite. Aprašytus atliekamus veiksmus galima pakartotinai panaudoti ir su GraphWalker pasitelkiant įgyvendintus modelio interfeisus. ProB turi mažiausiai pasirinkimo variantų, kaip atrinkti testus – kartojant testų vykdymą su tais pačiais duomenimis visada bus gaunamas tas pats testų rinkinys. Tačiau šis įrankis sugeba padengti duomenų modelį, kitais įrankiais tai galima atlikti tik rankiniu būdu. MBTsuite palaiko didžiausią aibę pasirinkimų, kaip bus generuojami testai, gali nustatyti testų atvejų ilgį, tačiau negali išspręsti problemų, kurias sugeba kiti įrankiai. GraphWalker turi nevienodas generavimo galimybes naudojant komandine eilute ir Maven. Reikalavimų atsekamumą palaiko tik Maven, naudojant komandinę eilutę reikalavimų nėra automatinio susiejimo. Tačiau tik komandine eilute galima pasinaudoti modelio kintamaisiais sukonkretinant testus.

7.3. Testų generavimo strategijos

Testų generavimo strategijos	GraphWalker	ProB	MBTsuite
Pilnas kelių padengimas	-	+	+
Pasirinktas kelias	-	-	+
Kelias pagal pavadinimą	-	-	+
Atsitiktinis kelias	+	-	+
Atsitiktinis kelias be pasikartojimų	+	-	+
Trumpiausias kelias	-	-	+
Atsitiktinis kelias iki pasirinktos viršūnės	+	-	-
Rankiniu būdu sukurta generavimo strategija	+	-	-

9 lentelė. Testų generavimo strategijų palyginimas

Generavimo strategijose iš kitų įrankių išsiskiria ProB: strategijų pasirinkti negalima – jos visada panaudojamos automatiškai. GraphWalker visos palaikomos strategijos susijusios su atsitiktinio kelio generavimu. ProB palaiko tik keleto strategijų, kombinacija, kuri nesuteikia atsitiktinumo. MBTsuite palaiko tiek atsitiktinio kelio generavimą, tiek nurodomo. Atsitiktinių kelių strategija gali būti naudojama ieškant klaidų, kai jų nėra tikimąsi (pateikiamos visiškai atsitiktinės įvestys, nebūtinai tokios, kokias turi įvesti vartotojas). Taip pat testavimas įvyksta daug greičiau, nei tikrinant visą įvesčių aibę.

7.4. Padengimo kriterijai ir sustojimo sąlygos

Padengimo kriterijai	GraphWalker	ProB	MBTsuite
Viršūnių padengimas	+	-	+
Briaunų padengimas	+	-	+
Reikalavimų padengimas	+	-	+
Priklausomybių padengimas	+	-	-
Žingsnių su vykdymu kodu padengimas	-	-	+
Pasirinktų operacijų padengimas	-	+	-
Begalinis generavimas	+	-	-

10 lentelė. Padengimo kriterijų palyginimas

GraphWalker iš kitų įrankių labai išsiskiria savo savybėmis: palaiko *online* testavimą, generavimo metu sukuria tik vieną testo kelią, kuris gali būti generuojamas be sustojimo. Todėl šis įrankis labiausiai tinka nuolatiniam sistemos testavimui. Kiti įrankiai padengimo kriterijus naudoja ne kaip sustojimo sąlygas, o kaip testų filtrus. Tai suteikia galimybę nurodyti, kas turi būti ištestuota neįtraukiant nepageidaujamų operacijų vykdymo.

Sustojimo kriterijai	GraphWalker	ProB	MBTsuite
Kainos intervalas	-	-	+
Trukmės intervalas	-	-	+
Ilgio intervalas	-	-	+
Reali generavimo trukmė	+	+	-
Pasiekta viršūnė	+	+	-
Pasiekta briauna	+	+	-
Trumpiausias kelias iki briaunos ar viršūnės	+	+	-
Pagal apibrėžtas logines sąlygas	-	-	+
Kelio ilgis	+	+	+
Panaudojant svorius	+	-	+
Rankiniu būdu sukurtas sustojimo kriterijus	+	-	+

11 lentelė. Sustojimo sąlygų palyginimas

Kai kuriuose įrankiuose nėra galimybės tiesiogiai pasirinkti sustojimo sąlygų. Tačiau tokia galimybė suteikiama netiesiogiai. Pavyzdžiui, ProB įrankis naudodamas paiešką į plotį visada ras trumpiausią kelią iki tam tikros operacijos. GraphWalker ir MBTsuite kriterijų pasirinkimas skiriasi nedaug. MBTsuite leidžia pasirinkti tuos pačius kriterijus testų, kurie yra sugeneruoti, filtravimui, o GraphWalker suteikia šias galimybes pačiame testų atvejų generavime. Ši savybė tiek ProB, tiek GraphWalker suteikia vykdymo laiko pranašumą prieš MBTsuite, kai norime pilnai ištestuoti sistemą [Ham02].

Apžvelgus įrankių sustojimo bei testų generavimo sąlygas pasidarė aišku, kad GraphWalker turi pranašumą, kai nėra žinoma, kur reiktų tikėtis klaidų arba kai įvairiais, nebūtinai iš anksto nustatytais scenarijais sistema turėtų būti testuojama nuolatos. ProB galima remtis, kai modelio duomenų struktūros yra sudėtingesnės arba vengiant klaidų reikalingas matematiškai įrodomas modelio korektiškumas. MBTsuite tiriamu atveju pateikė lengviausiai suprantamą modelį, plačiausias galimybes generuojant testų atvejus, tačiau nepadengia kitų įrankių privalumų, tokių kaip, *online* testavimo ar duomenų modelio verifikavimo.

8. Išvados ir rezultatai

Šiame darbe yra apžvelgiamos ir palyginamos pagrindinės modeliais paremto testavimo charakteristikos bei susijusių testavimo įrankių funkcionalumas. Siekiant kuo plačiau padengti skirtingų įrankių funkcionalumą ir galimybes, įrankiai tikslingai buvo pasirinkti kaip labai besiskiriantys pagal savo charakteristikas. Atlikus įrankių apžvalgą pasirinkta nagrinėti GraphWalker, ProB ir MBTsuite. Norint įvertinti pasirinktų testavimo įrankių funkcionalumą, buvo atlikta atvejo analizė pasinaudojant viešai prieinama GSM protokolo specifikacija.

Kaip minėta, testavimo procesas susideda iš 5 dalių: modelių kūrimo, testų generavimo, testų konkretizavimo, testų vykdymo ir rezultatų analizės. Nagrinėjant įrankius pastebėta, kad ne visi pasirinkti įrankiai palaiko testų konkretizavimą, kai kuriems įrankiams reikalingas adapteris. Kadangi testų konkretizavimas gali būti tiek problema, kurią sprendžia testavimo įrankis, tiek problema, kurią sprendžia testuotojas, šiai daliai buvo atlikta tik pirminė analizė. Nagrinėjant pasirinktus įrankius didžiausias dėmesys buvo skirtas modeliavimui ir testų generavimui.

Darbe yra aprašytos kilusios problemos modeliavimo ir testų generavimo procesuose kuriant GSM sistemos dalį. Daliai problemų buvo išanalizuotos jas sukėlusios priežastys ir pasiūlyti sprendimo būdai. Visuose įrankiuose reikėjo bent dalinai supaprastinti modelį, kad iš jo būtų galima per protingą laiką gauti testavimo tikslus tenkinantį testų rinkinį. Kuriant sistemą, kai kurių įrankių savybių nebuvo įmanoma prasmingai panaudoti arba jos buvo perteklinės.

Visiems įrankiams buvo surasti algoritmai, kurie su mažiausiu testų rinkiniu padengė visus modelio būsenų perėjimus. ProB su abiem generavimo algoritmais padengė visas operacijas, bet modelio tikrinimo būdu sugeneruotas mažesnis testų rinkinys, nes algoritmas tikrino, ar sistemos būsena testuose nepasikartoja. MBTsuite efektyviausias algoritmas buvo pilnas kelių padengimas. Jis veikia panašiu principu, kaip ir ProB modelio tikrinimas, tik yra keletas skirtumų. Jis, kaip ir visi likę MBTsuite algoritmai, netikrina galimos parametrų aibės operacijoms bei ProB atlieka filtravimą automatiškai, o MBTsuite filtrai pritaikomi tik po generavimo. Su MBTsuite galima pasirinkti didesnio ilgio testus, ko pasėkoje jų liko mažiau nei ProB. Dėl skriptų nesuderinamumo su įrankio efektyviausiu algoritmu GraphWalker įrankis modelio operacijas galėjo padengti tik atsitiktiniu klaidžiojimu. Atsitiktiniu klaidžiojimu gautas testų rinkinių atliekant bandymus dydžiai skyrėsi daugiau nei 10 kartų, todėl efektyviam testų rinkiniui sudaryti, padengiant visas modelyje apibrėžtas operacijas ir jų šakas, įrankis nėra tinkamas.

Testuojama sistema atvejo analizei buvo pasirinkta santykinai paprasta. Buvo pastebėta, kad pasirinkti įrankiai silpnai palaiko vieną svarbią testavimui savybę – duomenų padengimą. Tačiau šiai mažai sistemai įrankiai buvo pakankamai gerai pritaikyti, todėl ši savybė nebuvo

esminė. Tikėtina, kad atliekant bandymus su didesne sistema problemų ir skirtumų tarp įrankių funkcionalumų būtų surasta daugiau.

Literatūra

[Ame14] G. Americas. GSM Global system for Mobile Communications. Archived from the original on 8 February 2014. [žiūrėta 2020-05-15]. Prieiga per internetą: <https://web.archive.org/web/20140208025938/http://www.4gamericas.org/index.cfm?fuseaction=page&sectionid=242>

[BJK+05] M. Broy, B. Jonsson, J. P. Katoen, M. Leucker, and A. Pretschner (Eds.). Model-based testing of reactive systems. In Lecture Notes in Computer Science, vol. 3472, Springer-Verlag, 2005.

[BLK15] R. V. Binder, B. Legeard, and A. Kramer. Model-based testing: where does it stand? Communications of the ACM. ACM New York, **58**(2), pp. 52-56, 2015.

[Con] Conformiq. Conformiq products. [žiūrėta 2020-05-15]. Prieiga per internetą: <https://www.conformiq.com/products/>

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. Numerische mathematik 1.1, pp. 269-271, 1959.

[Elv] Elvior. TestCast MBT – features. [žiūrėta 2020-05-15]. Prieiga per internetą: <https://elvior.com/model-based-testing/#section-features/>

[ETS96] European Telecommunications Standards Institute (ETSI). Specification of the Subscriber Identity Module - Mobile Equipment (SIM-ME) Interface; (GSM 11.11), 1996. [žiūrėta 2020-05-15]. Prieiga per internetą: <https://www.etsi.org/deliver/etsi_gts/11/1111/05.03.00_60/gsmts_1111v050300p.pdf>

[GLA+16] V. Gudmundsson, M. Lindvall, L. Aceto, J. Bergthorsson, and D. Ganesan. Model-based Testing of Mobile Systems - An Empirical Study on QuizUp Android App. In Proceedings First Workshop on Preand Post-Deployment Verification Techniques, PrePost@IFM, 2016.

[Ham02] R. Hamlet. Random testing. Encyclopedia of software Engineering. Wiley Online Library, 2002.

[HHU] Heinrich Heine University. The ProB Animator and Model Checker. [žiūrėta 2020-05-18]. Prieiga per internetą: <https://www3.hhu.de/stups/prob/index.php/Main_Page>

[Int] Intel's Open Source Technology Center. FMBT overview. [žiūrėta 2020-05-15]. Prieiga per internetą: <https://01.org/fmbt/overview/>

[JJL] J. Bendisposto, J. Clark, and M. Leuschel. ProB Handbook. [žiūrėta 2020-05-18]. Prieiga per internetą: <https://www3.hhu.de/stups/handbook/prob2/prob_handbook.html>

[Jor17] P. C. Jorgensen. The Craft of Model-Based Testing. Auerbach Publications, 2017.

[Kar20] K. Karl. Features of GraphWalker. [žiūrėta 2020-05-15]. Prieiga per internetą: <https://github.com/GraphWalker/graphwalker-project/wiki/Features-of-GraphWalker>

[LLS17] W. Li, F. Le Gall and N. Spaseski. A Survey on Model-Based Testing Tools for Test Case Generation. In Proceedings of 4th International Conference on Tools and Methods for Program Analysis (TMPA 2017), Springer International Publishing, pp. 77-89, 2017.

[Mic17] Z. Micskei. Model-based testing (MBT). Course description. Department of Measurement and Information Systems Budapest University of Technology and Economics. 2017. [žiūrėta 2020-05-15]. Prieiga per internetą: <http://mit.bme.hu/~micskeiz/pages/modelbased_testing.html>

[Pea09] J. Pearce. Modeling behavior. Lecture notes. [žiūrėta 2020-05-15]. Prieiga per internetą: <http://www.cs.sjsu.edu/~pearce/modules/lectures/uml/behavior/>

[Sep] Sepp Med GmbH. MBTsuite - The Model Based Testing Tool. [žiūrėta 2020-05-15]. Prieiga per internetą: <https://mbtsuite.com/>

[Sma] Smartesting. YEST®, Visual test design solution. [žiūrėta 2020-05-15]. Prieiga per internetą: <https://www.smartesting.com/conception-collaborative-tests-yest?lang=en>

[SVG+08] V. Santiago, N. L. Vijaykumar, D. Guimarães, A. S. Amaral, and É. Ferreira. An environment for automated test case generation from statechart-based and finite state machine-based behavioral models. In 2008 IEEE International Conference on Software Testing Verification and Validation Workshop. pp. 63-72, 2008.

[SSB18] R. Singh, A. Singhrova, and R. Bhatia. Test Case Generation Tools - A Review. International Journal of Electronics Engineering (ISSN: 0973-7383), **10**(2), pp. 586-596, 2018.

[UL10] M. Utting, B. Legeard. Practical model-based testing: a tools approach. Elsevier, 2010.

[UPL12] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approches. Software Testing, Verification and Reliability. Wiley Online Library, **22**(5), pp. 297-312, 2012.

[Xu15] D. Xu. MISTA: Model-based Integration and System Test Automation. Archived from the original on 31 October 2019. 2015. [žiūrėta 2020-05-15]. Prieiga per internetą: <https://web.archive.org/web/20191031003452/http://cs.boisestate.edu:80/~dxu/research/MBT.html>

Priedai

1 priedas. ProB modelio kodas.

```
MACHINE GSM
SETS
   FILES = {mf,df_gsm,ef_iccid,ef_lp,ef_imsi,ef_ad};
   PERMISSION = {always,chv,never,adm};
   VALUE = {true, false};
   BLOCKED_STATUS = {blocked, unblocked};
   CODE = {c1,c2,c3,c4};
   DATA = {data1,data2,data3,data4};
   STATUS_WORDS = {sw_9000,sw_9400,sw_9404,sw_9804,sw_9840}
CONSTANTS
   FILES_CHILDREN,
   PERMISSION_READ,
   MAX_CHV,
   MAX_UNBLOCK_CHV,
   PUK
DEFINITIONS
   MF == {mf};
   DF == {df_gsm};
   EF == {ef_iccid,ef_lp,ef_imsi,ef_ad}
PROPERTIES
   FILES_CHILDREN : (MF \/ DF) <-> FILES &
   FILES_CHILDREN = {(mf,df_gsm), (mf,ef_iccid),
   (df_gsm,ef_lp), (df_gsm,ef_imsi), (df_gsm,ef_ad)} &
   PERMISSION_READ : EF --> PERMISSION &
   PERMISSION_READ = {(ef_iccid,never),(ef_lp,always),
   (ef_imsi,chv),(ef_ad,adm)} &
   MAX_CHV = 3 &
   MAX_UNBLOCK_CHV = 10 &
   PUK : CODE &
   PUK = c3
VARIABLES
   current_file,
   current_directory,
   counter_chv,
   counter_unblock_chv,
   blocked_chv_status,
   blocked_status,
   permission_session,
   pin,
   data
INVARIANT
   current_file <: EF &
   card(current_file) <= 1 &
   current_directory : DF \/ MF &
   counter_chv : 0..MAX_CHV &
   counter_unblock_chv : 0..MAX_UNBLOCK_CHV &
   pin : CODE &
   permission_session : PERMISSION --> VALUE &
   (always,true) : permission_session &
   (adm,false) : permission_session &
   (never,false) : permission_session &
   blocked_chv_status : BLOCKED_STATUS &
   blocked_status : BLOCKED_STATUS &
   data : EF --> DATA &
   (blocked_chv_status=blocked => (chv,false):permission_session) &
   (counter_chv=0 => blocked_chv_status=blocked) &
   (counter_unblock_chv=0 => blocked_status=blocked) &
   (current_file = {} or
      (dom(FILES_CHILDREN |> current_file) = {current_directory}))
```

```
INITIALISATION
   current_file := {} ||
   current_directory := mf ||
   counter_chv := MAX_CHV ||
   counter_unblock_chv := MAX_UNBLOCK_CHV ||
   blocked_chv_status := unblocked ||
   blocked_status := unblocked ||
   permission_session := {(always,true),(chv,false),
                          (adm,false),(never,false)} ||
   pin := c1 ||
   data := {(ef_iccid,data1),(ef_lp,data2),(ef_imsi,data3),(ef_ad,data4)}
OPERATIONS
   sw <-- VERIFY_CHV_BLOCKED(code) =
      PRE
         code : CODE &
         blocked_chv_status = blocked
      THEN
         sw := sw_9840
      END;
   sw <-- VERIFY_CHV_GOOD_PIN(code) =
      PRE
         code : CODE &
         blocked_chv_status /= blocked &
         pin = code
      THEN
         counter_chv := MAX_CHV ||
         permission_session(chv) := true ||
         sw := sw_9000
      END;
   sw <-- VERIFY_CHV_LAST_FAIL(code) =
      PRE
         code : CODE &
         blocked_chv_status /= blocked &
         pin /= code &
         counter_chv = 1
      THEN
         counter_chv := 0 ||
         blocked_chv_status := blocked ||
         permission_session(chv) := false ||
         sw := sw_9840
      END;
   sw <-- VERIFY_CHV_FAIL(code) =
      PRE
         code : CODE &
         blocked_chv_status /= blocked &
         pin /= code &
         counter_chv /= 1
      THEN
         counter_chv := counter_chv - 1 ||
         sw := sw_9804
      END;
   sw <-- UNBLOCK_CHV_BLOCKED(code_unblock, new_code) =
      PRE
         code_unblock : CODE &
         new_code : CODE &
         sw : STATUS_WORDS &
         blocked_status = blocked
      THEN
         sw := sw_9840
      END;
   sw <-- UNBLOCK_CHV_SUCCESS(code_unblock, new_code) =
      PRE
         code_unblock : CODE &
```

```
      new_code : CODE &
      sw : STATUS_WORDS &
      blocked_status /= blocked &
      PUK = code_unblock
   THEN
      pin := new_code ||
      blocked_chv_status := unblocked ||
      counter_chv := MAX_CHV ||
      counter_unblock_chv := MAX_UNBLOCK_CHV ||
      permission_session(chv) := true ||
      sw := sw_9000
   END;
sw <-- UNBLOCK_CHV_LAST_FAIL(code_unblock, new_code) =
   PRE
      code_unblock : CODE &
      new_code : CODE &
      sw : STATUS_WORDS &
      blocked_status /= blocked &
      PUK /= code_unblock &
      counter_unblock_chv = 1
   THEN
      counter_unblock_chv := 0 ||
      blocked_status := blocked ||
      sw := sw_9840
   END;
sw <-- UNBLOCK_CHV_FAIL(code_unblock, new_code) =
   PRE
      code_unblock : CODE &
      new_code : CODE &
      sw : STATUS_WORDS &
      blocked_status /= blocked &
      PUK /= code_unblock &
      counter_unblock_chv /= 1
   THEN
      counter_unblock_chv := counter_unblock_chv-1 ||
      sw := sw_9804
   END;
cd,cf,cc,cuc <-- STATUS =
   BEGIN
      cd := current_directory ||
      cf := current_file ||
      cc := counter_chv ||
      cuc := counter_unblock_chv
   END;
sw <-- SELECT_DF(ff) =
   PRE
      ff : FILES &
      sw : STATUS_WORDS &
      ff : (DF \/ MF) &
      (((ff,current_directory) : FILES_CHILDREN)
            or ((current_directory,ff) : FILES_CHILDREN)
            or (ff = mf))
   THEN
      sw := sw_9000 ||
      current_directory := ff ||
      current_file := {}
   END;
sw <-- SELECT_DF_FAIL(ff) =
   PRE
      ff : FILES &
      sw : STATUS_WORDS &
      ff : (DF \/ MF) &
      not (((ff,current_directory) : FILES_CHILDREN)
```

```
                or ((current_directory,ff) : FILES_CHILDREN)
                or (ff = mf))
        THEN
          sw := sw_9404
        END;
    sw <-- SELECT_EF(ff) =
        PRE
          ff : FILES &
          sw : STATUS_WORDS &
          not (ff : (DF \/ MF)) &
          (current_directory,ff) : FILES_CHILDREN
        THEN
          sw := sw_9000 ||
          current_file := {ff}
        END;
    sw <-- SELECT_EF_FAIL(ff) =
        PRE
          ff : FILES &
          sw : STATUS_WORDS &
          not (ff : (DF \/ MF)) &
          not ((current_directory,ff) : FILES_CHILDREN)
        THEN
          sw := sw_9404
        END;
    sw <-- READ_EMPTY_BINARY =
        PRE
          current_file = {}
        THEN
          sw := sw_9400
        END;
    sw,dd <-- READ_BINARY =
        PRE
          current_file /= {} &
          (#(file).(file:current_file & permission_session(PERMISSION_READ(file)) = true))
        THEN
          sw := sw_9000 ||
          ANY ff WHERE ff : current_file
          THEN
            dd := data(ff)
          END
        END;
    sw <-- READ_BINARY_NOT_ALLOWED =
        PRE
          current_file /= {} &
          not (#(file).(file:current_file & permission_session(PERMISSION_READ(file)) = true))
        THEN
          sw := sw_9804
        END
END
```