Kauno technologijos universitetas

Informatikos fakultetas

Tekstinių defektų paieška mobiliosiose programose analizuojant programos paveikslėlius

Baigiamasis magistro studijų projektas

Arnoldas Aukštikalnis

Projekto autorius

doc. dr. Šarūnas Packevičius

Vadovas

Kaunas, 2021

Kauno technologijos universitetas

Informatikos fakultetas

Tekstinių defektų paieška mobiliosiose programose analizuojant programos paveikslėlius

Baigiamasis magistro studijų projektas

Programų sistemų inžinerija (6211BX011)

Arnoldas Aukštikalnis

Projekto autorius

doc. dr. Šarūnas Packevičius

Vadovas

lekt. Dominykas Barisas

Recenzentas

Kaunas, 2021

Kauno technologijos universitetas

Informatikos fakultetas

Arnoldas Aukštikalnis

Tekstinių defektų paieška mobiliosiose programose analizuojant programos paveikslėlius

Akademinio sąžiningumo deklaracija

Patvirtinu, kad:

1. baigiamąjį projektą parengiau savarankiškai ir sąžiningai, nepažeisdama(s) kitų asmenų autoriaus ar kitų teisių, laikydamasi(s) Lietuvos Respublikos autorių teisių ir gretutinių teisių įstatymo nuostatų, Kauno technologijos universiteto (toliau – Universitetas) intelektinės nuosavybės valdymo ir perdavimo nuostatų bei Universiteto akademinės etikos kodekse nustatytų etikos reikalavimų;

2. baigiamajame projekte visi pateikti duomenys ir tyrimų rezultatai yra teisingi ir gauti teisėtai, nei viena šio projekto dalis nėra plagijuota nuo jokių spausdintinių ar elektroninių šaltinių, visos baigiamojo projekto tekste pateiktos citatos ir nuorodos yra nurodytos literatūros sąraše;

3. įstatymų nenumatytų piniginių sumų už baigiamąjį projektą ar jo dalis niekam nesu mokėjęs (-usi);

4. suprantu, kad išaiškėjus nesąžiningumo ar kitų asmenų teisių pažeidimo faktui, man bus taikomos akademinės nuobaudos pagal Universitete galiojančią tvarką ir būsiu pašalinta(s) iš Universiteto, o baigiamasis projektas gali būti pateiktas Akademinės etikos ir procedūrų kontrolieriaus tarnybai nagrinėjant galimą akademinės etikos pažeidimą.

Arnoldas Aukštikalnis

Patvirtinta elektroniniu būdu

Kauno technologijos universitetas

Informatikos fakultetas

Baigiamojo magistro projekto užduotis

Projekto tema	Tekstinių defektų paieška mobiliosiose programose analizuojant programos paveikslėlius
Reikalavimai ir sąlygos (tikslinti pavadinimą pagal poreikį)	Sukuti testavimo įrankį, aptinkantį vartotojo sąsajos defektus mobiliosiose programose
Vadovas / Vadovė	
	(vadovo pareigos, vardas, pavardė, parašas) (data)

Aukštikalnis Arnoldas. Tekstinių defektų paieška mobiliosiose programose analizuojant programos paveikslėlius. Magistro baigiamasis projektas / vadovas doc. dr. Šarūnas Packevičius; Kauno technologijos universitetas, Informatikos fakultetas.

Studijų kryptis ir sritis (studijų krypčių grupė): Programų sistemų inžinerija.

Reikšminiai žodžiai: vaizdo analizė, teksto analizė, testavimas, tekstiniai defektai.

Kaunas, 2021. 42 p.

Santrauka

Šiame darbe pristatomas sukurtas įrankis, skirtas tekstiniams defektams aptikti mobiliosiose programėlėse naudojant programėlės paveikslėlius bei tiriamas jo veikimas. Pirmajame skyriuje analizuojama literatūra, kurioje tiriamas mobiliųjų programėlių vartotojų sąsajos automatinis testavimas, palyginamos rinkoje egzistuojančios technologijos ir trumpai aprašomas pasirinktas sprendimas. Antrajame skyriuje pateikiamas sukurto įrankio aprašymas, panaudojimo atvejai, iškelti funkciniai ir nefunkciniai reikalavimai bei statinis ir dinaminis sistemos vaizdas. Trečiajame skyriuje detaliau aprašomi sukurti defektų aptikimo metodai bei aprašomas atliktas eksperimentas, kurio metu įrankis analizavo apie 95 tūkst. mobiliųjų programėlių nuotraukų. Galiausiai, analizuojami atlikto eksperimento rezultatai, o darbo pabaigoje pateikiamos išvados.

Aukštikalnis Arnoldas. Text Defect Detection in Mobile Applications Using Mobile Application Screenshot Analysis. Master's Final Degree Project / supervisor doc. dr. Šarūnas Packevičius; Informatics Faculty, Kaunas University of Technology.

Study field and area (study field group): Software Engineering.

Keywords: image analysis, text analysis, testing, text defects.

Kaunas, 2021. 42 p.

Summary

This paper presents a tool, that can be used for detecting text defects in mobile applications using application screenshots and studies its operation. The first chapter analyzes the literature that examines automated testing of mobile application user interface, compares the technologies available in the market, and briefly describes the chosen solution. The second chapter provides a description of the developed tool, use cases, functional and non-functional requirements, and a static and dynamic view of the system. The third chapter describes in detail the developed defect detection methods and describes the performed experiment, during which the tool analyzed about 95 thousand screenshots of mobile applications. Finally, the results of the performed experiment are analyzed, and the conclusions are presented at the end of the work.

Turinys

Lentelių sąrašas

Paveikslų sąrašas

Santrumpų ir terminų sąrašas

Santrumpos:

API (angl. *Application Programming Interface*) – susitarimų ir procedūrų rinkinys ryšiams tarp atskirų programų ir operacinės sistemos realizuoti.

JSON (angl. *JavaScript Object Notation*) – atviro standarto formatas, perduodantis duomenų objektus, sudarytus iš atributo ir reikšmės porų, lengvai skaitomame tekste.

OCR (angl. *Optical Character Recognition*) – spausdintų arba ranka rašytų ženklų vaizdų skaitymas ir atpažinimas.

REST (angl. *Representational state transfer*) – programinės įrangos architektūros stilius, naudojantis HTTP protokolo poaibį.

RPC (angl. *Remote procedure call*) – nuotolinio procedūros iškvietimo protokolas.

Terminai:

Debesija – interneto paslaugų visuma, jungianti įvairiuose serveriuose esančius informacijos išteklius ir programinę įrangą, sudaranti sąlygas jais naudotis.

Emuliatorius – įtaisas arba programa, imituojanti (emuliuojanti) kito įtaiso arba programos darbą.

Žurnalas – dokumentas, kuriame fiksuojami darbo su kompiuterio programa faktai: pateiktos komandos, informacija apie jų vykdymą, klaidas, rezultatus ir pan.

Įvadas

Darbo problematika ir aktualumas

Mobiliųjų programėlių testavimas užima daug laiko, ypač jei testuojama rankiniu būdu. Be to, mobiliąsias programėles reikia testuoti su daugeliu skirtingų įrenginių, su skirtinga ekrano rezoliucija. Jei programėlė turi kelias skirtingas kalbas, jas visas taip pat reikėtų ištestuoti. Sukurtas įrankis leidžia sutaupyti daug testavimui skiriamo laiko bei sumažinti programėlėse esančių klaidų kiekį. Kadangi egzistuoja įrankiai, galintys išvaikščioti programėlę ir padaryti kiekvieno lango nuotrauką, galima jais pasinaudoti ir perduoti padarytas nuotraukas įrankiui, kuris skirtas defektams aptikti. Darbas aktualus ir dėl to, kad rinkoje nėra panašaus įrankio, todėl tenka testuoti rankomis, o tai užtrunka daug laiko ir nėra efektyvu.

Darbo tikslas ir uždaviniai

Darbo tikslas – sumažinti mobiliųjų programėlių testavimo trukmę ir jose esančių tekstinių defektų kiekį.

Šio tikslo įgyvendinimui iškelti šie uždaviniai:

- išanalizuoti literatūrą, kurioje tiriamas mobiliųjų programėlių vartotojų sąsajos automatinis testavimas;
- sukurti įrankį, kuris aptinka tekstinius defektus mobiliosios programėlėse analizuodamas programėlės paveikslėlius;
- ištirti sukurtų, defektams aptikti skirtų metodų veikimą, ištestuojant 95 tūkst. nuotraukų ir palyginant rezultatus su rankiniu būdu aptiktų defektų kiekiu.

Dokumento struktūra

Pirmajame skyriuje pateikiama mobiliųjų programėlių vartotojo sąsajos testavimo literatūros analizė, palyginamos rinkoje egzistuojančios technologijos ir trumpai aprašomas pasirinktas sprendimas. Antrajame skyriuje pateikiamas sukurtos programinės įrangos aprašymas, panaudojimo atvejai, iškelti funkciniai ir nefunkciniai reikalavimai bei statinis ir dinaminis sistemos vaizdas. Trečiajame skyriuje detaliau aprašomi sukurti defektams aptikti skirti metodai bei aprašomas atliktas eksperimentas ir analizuojami jo rezultatai.

1. Analitinė dalis

1.1. Testavimo programų inžinerijoje apžvalga

1.1.1. Programinės įrangos testavimas

Programinės įrangos testavimas yra plati sąvoka, apimanti įvairias veiklas programinės įrangos kūrimo cikle ir už jo ribų (1). Bendrąja prasme, testavimas yra programų sistemos vykdymo stebėjimas, siekiant patikrinti ar ji elgiasi taip, kaip yra numatyta ir siekiant nustatyti potencialias klaidas. Didelis dėmesys testavimui skiriamas kokybei užtikrinti, nes programinės įrangos atitikimas specifikacijoms ir klaidų nebuvimas yra tiesiogiai susiję su programinės įrangos kokybe. Tyrimai rodo, kad daugiau nei pusė programinės įrangos kūrimo kaštų yra skiriama testavimui, o kritinėse sistemose šis skaičius yra dar didesnis (2). Kadangi programinė įranga vis labiau skverbiasi į žmonių kasdienį gyvenimą, reikia rasti būdų, kaip sumažinti testavimo kainą nepakenkiant testavimo efektyvumui.

Vienas iš tokių būdų – sumažinti testavimo priklausomybę nuo žmogaus darbo, pasitelkiant automatinio testavimo metodus (3). Automatizavus programinės įrangos testavimą, atsiranda ir galimybė tokius testavimo metodus teikti kaip internetinę paslaugą: žmogus į tinklapį įkelia norimą ištestuoti programą, o po kelių minučių ar valandų gauna išklotinę su rastomis klaidomis.

1.1.2. Mobiliųjų programėlių testavimas

Mobiliųjų programėlių testavimas – tai procesas, kai testuojama programinė įranga, kuri yra sukurta nešiojamiesiems įrenginiams (4). Mobiliųjų programėlių testavimas skiriasi nuo kitų programų testavimo, nes turime atsižvelgti ne tik į funkcinius ir vartotojo sąsajos reikalavimus, bet ir į įrenginio parametrus, ekrano dydį, platformą, ryšio problemas ir t. t. Be to, nuolat išleidžiamos naujos operacinių sistemų versijos, nauji įrenginiai, o tai dar labiau apsunkina mobiliųjų programėlių testavimą (5).

Galima išskirti keturias populiarias mobiliųjų programėlių testavimo metodikas (6):

- emuliatoriumi paremtas testavimas;
- įrenginiu paremtas testavimas;
- testavimas debesijoje;
- minia paremtas testavimas.

Emuliatoriumi paremto testavimo metu naudojamas mobiliojo įtaiso emuliatorius, kuris sukuria mobilaus įrenginio virtualiąją mašiną asmeniniame kompiuteryje. Tai pigiausias būdas, nes nereikia pirkti realaus įtaiso. Kita vertus, susiduriama su naudojamo emuliatoriaus apribojimais – daugelis emuliatorių nepalaiko rankų gestų bei specifiniam įtaisui skirtų funkcijų. Be to, emuliatorius dažniausiai kuriamas tam tikram įtaisui ar platformai, todėl sunkiau testuoti programėles, skirtas daugeliui platformų ar įtaisų.

Įrenginiu paremto testavimo metu perkami realūs mobilieji įrenginiai. Toks testavimas kainuoja brangiau, nei emuliatoriumi paremtas testavimas, tačiau leidžia ištestuoti specifiniam įrenginiui skirtas funkcijas bei programėlės veikimą realiomis sąlygomis.

Testavimo debesijoje idėja – sukurti mobiliųjų įrenginių debesiją, kuri teiktų testavimo paslaugas plačiu mastu. Tokio metodo privalumas – mažesnė testavimo kaina didelės apimties programoms, nei

įrenginiu paremto testavimo bei efektyvesnis įvairių bandymų palaikymas mobiliuosiuose įrenginiuose.

Minia paremto testavimo metu programėlę testuoja laisvai samdomi testavimo specialistai arba vartotojų bendruomenė. Šio metodo pliusas – nereikia investuoti į realius įtaisus, testuojama realioje aplinkoje, tačiau rizikuojama, kad nukentės testavimo kokybė.

1.1.3. Vartotojo sąsajos testavimas

Šiais laikais grafinė vartotojo sąsaja yra pagrindinis vartotojo sąveikos su programine įranga būdas. Ji leidžia patogiai naudotis programa ir yra svarbi jos dalis – vartotojo sąsajai gali būti skiriama net iki 60 % programos kodo (7).

Deja, dažniausiai pasitaikanti vartotojo sąsajos testavimo metodika yra rankinis testavimas. Automatiniuose testuose dažniausiai naudojama įrašo atkūrimo metodika, kai testuotojas naudojasi programa, o testavimo įrankis įrašo jo veiksmus ir juos išsaugo. Vėliau testuotojas gali paleisti įrašytą testą, kuris identiškai atkartoja jo atliktus veiksmus. Kai kurie tokio tipo įrankiai leidžia pakeisti įrašytas įvestis, taip leisdami sudaryti daugiau testavimo scenarijų. Šis procesas užima daug laiko tiek testų sukūrimui, tiek jų vykdymui. Alternatyva – programinės įrangos beta versijos išleidimas, leidžiant vartotojams patiems atlikti dalį testavimo.

Dar viena bėda, su kuria susiduriama vartotojo sąsajos testavime – testavimo įrankis gali aptikti tik išorinę programos būseną (8). Pavyzdžiui, išsaugant failą galima patikrinti ar parodytas pranešimas, kad failas išsaugotas, bet negalima patikrinti ar failas iš tikrųjų išsaugotas. Dėl šios priežasties, tokius testavimo įrankius reikėtų naudoti kartu su kitais, kurie gali patikrinti vidinę programos būseną.

1.1.4. Vartotojo sąsajos automatinio testavimo metodai

Pirmasis analizuojamas metodas (9) sprendžia mobiliųjų programėlių, sukurtų *Google Android* platformai, vartotojo sąsajos automatinio testavimo problemą. Šis metodas remiasi peržiūros robotu (programa, imituojančia programėle besinaudojantį žmogų ir gebančia ją išvaikščioti), kuris automatiškai sudaro programėlės vartotojo sąsajos modelį ir iš jo išgauna testavimo scenarijus, kurie gali būti vykdomi automatiškai. Peržiūros robotas sudaro vartotojo sąsajos medį, kuriame lapai atspindi vartotojo sąsajos langus, o šakos – perėjimus tarp jų. Šis metodas skirtas rasti veikimo metu pasitaikančioms klaidoms ir aptikti vartotojui matomus pasikeitimus po programos versijos atnaujinimo.

Šį metodą vykdo įrankis A^2T^2, parašytas *Java* kalba ir susidedantis iš trijų komponentų: *Java* kodo kontrolės komponento, vartotojo sąsajos peržiūros roboto ir testavimo scenarijų generatoriaus. *Java* kodo kontrolės komponentas tikrina *Java* kodą ir aptinka programos veikimo metu atsiradusias klaidas. Peržiūros robotas atsakingas už anksčiau aprašyto vartotojo sąsajos medžio sudarymą ir naudoja *Robotium* karkasą. *Robotium* karkasas leidžia analizuoti *Android* programėlės komponentus. Testavimo scenarijų generatorius atsakingas už testavimo scenarijų sukūrimą – tai *Java* testavimo metodai, kurie gali įvykdyti įvykių seką ir patikrinti ar įvyko klaida bei patikrinti ar sąsajos elementai pasikeitė nuo pirminių, išgautų sudarinėjant vartotojo sąsajos medį.

Antrasis metodas (10) skirtas testuoti *Android* programėlėms per jų vartotojo sąsają. Jis paremtas automatiniu programėlės vartotojo sąsajos ištyrimu, panaudojant įrankį *AndroidRipper,* siekiant sistemingai išvaikščioti programą, generuojant ir vykdant testavimo scenarijus, aptikus naujus

įvykius. Testavimo scenarijus sudaro įvykių sekos, paleidžiamos per programėlės vartotojo sąsajos valdiklius. Šis metodas leidžia keisti testavimo įrankio elgseną per tam tikrus parametrus, priklausomai nuo testuojamos programėlės ir testavimo tikslų. Taip pat aptinkamos ir vykdymo metu pasitaikančios programėlės klaidos. *AndroidRipper* įrankis sukurtas panaudojant *Robotium* karkasą, remiantis *Android Instrumentation* klase.

Trečiasis metodas (11) skirtas vartotojo sąsajos klaidoms aptikti *Android* programėlėse, taikant dinaminės analizės požiūrį ir sujungiant automatinio testavimo scenarijų ir atsitiktinių įvykių generavimo įrankius bei žurnalo failų analizę.

Iš pradžių, pasinaudojant programėlės išeities kodu, sugeneruojami testavimo scenarijai. Tam tikslui panaudojamas *Java* testavimo scenarijų generavimo įrankis *JUnit*. Taip pat panaudojamas automatinis įvykių generavimo įrankis *Monkey*. Kadangi dauguma programėlių laukia tam tikrų veiksmų vartotojo sąsajoje, kad pereitų iš vienos būsenos į kitą, *Monkey* gali sukurti įvykius tiek atsitiktiniu, tiek nustatytu būdu ir perduoti juos programėlei. Kai testavimo scenarijus paleidžiamas, į sisteminį žurnalą registruojama detali programėlės informacija. Po kiekvieno scenarijaus vykdymo atliekama sisteminio žurnalo analizė, siekiant surasti potencialias klaidas.

Ketvirtasis metodas (12) skirtas automatizuotai vartotojo sąsajos defektų paieškai mobiliosiose programėlėse ir remiasi statinio testavimo metodika. Iš pradžių išgaunamas testuojamos programėlės naršymo modelis, tada programėlė paleidžiama didelio kiekio skirtingų nustatymų mobiliuosiuose įrenginiuose. Vykdymo metu, kiekviename įrenginyje padaromos kiekvieno programėlės vartotojo sąsajos lango nuotraukos, o po to atliekama tų nuotraukų analizė, siekiant aptikti defektus.

Naršymo modelis išgaunamas tiesiai iš programėlės išeities kodo. Aktualūs elementai – programėlių langai ir dialogai, valdiklių įvykiai, kurie vykdo perėjimus tarp langų, perėjimų apribojimai, kurie nustato kokie laukai turi būti užpildyti, kad įvyktų perėjimas ir mygtukai, jų pradiniai dydžiai ir pozicijos. Naršymo modelis pavaizduojamas kaip kryptinis grafas, kuriame langai yra grafo mazgai, o valdiklių įvykiai, sukeliantys perėjimus tarp langų, yra kryptinės to grafo rodyklės. Naudojantis šiuo modeliu išgeneruojami visi įmanomi programėlės vykdymo keliai bei įvesties veiksmai, kurie priverstų įvykti perėjimą iš vieno lango į kitą. Tada testuojama programėlė paleidžiama didelio kiekio mobiliuosiuose įrenginiuose, kuriems perduodami sugeneruoti įvesties veiksmai, verčiantys programėlę vykdyti pagal testavimo scenarijų. Kiekviename žingsnyje padaroma programėlės lango nuotrauka bei nuotraukos metaduomenyse užregistruojama to įrenginio informacija. Galiausiai, atliekama visų surinktų programėlės nuotraukų analizė, siekiant aptikti iš anksto apsirašytus defektus.

Penktasis metodas (13) skirtas nenuoseklių spalvų, dydžių ar pozicijų objektams ar šriftams aptikti. Šis metodas integruoja kompiuterinės regos algoritmą, kuris leidžia hierarchiškai aptikti vartotojo sąsajos netikslumus, nuo stambiausių iki smulkiausių. Šį metodą realizuojantis įrankis *UI X-Ray* susideda iš trijų modulių: nuotraukų surinkimo, vartotojo sąsajos netikslumų aptikimo ir interaktyvios ataskaitos generavimo modulio.

Nuotraukų surinkimo įrankis gali kiekvienam langui automatiškai atrinkti specifikaciją (dizainerių nupieštą eskizą). Kadangi programėlė dažniausiai turi šimtus eskizų, rankinis atrinkimas užima didelę dalį laiko, todėl šis modulis yra esminis įrankyje. Atrinkus specifikaciją, darbą pradeda netikslumų aptikimo modulis, kuris aptinka ir išmatuoja netikslumus, pasinaudodamas kompiuterinės regos analize. Šio modulio išvestis panaudojama interaktyvios ataskaitos sudarymo modulyje, kuris leidžia vartotojui peržiūrėti, redaguoti ir pakomentuoti visus rastus netikslumus.

1.1.5. Tekstinių defektų aptikimo metodai

Kadangi projekto metu bus susitelkta į mobiliųjų programėlių vartotojo sąsajos tekstinių defektų paiešką, toliau pateikiama lentelė (1 lentelė.) su tekstinių defektų aprašais ir siūlomomis metodų idėjomis jiems aptikti (14).

1 lentelė. Tekstinių defektų ir siūlomų aptikimo metodų aprašymai

Nr.	Defekto aprašymas	Metodo idėja
1.	Tekstas nesulygiuotas su aplinkiniais valdikliais. Pvz.: tekstinio lauko etiketė vertikaliai nesulygiuota su tekstiniu lauku.	Tikrinamos visos teksto sritys. Jei teksto sritis nėra centruota su aplinkui esančiais valdikliais – tai defektas.
2.	Teksto dydis smarkiai skiriasi nuo aplinkinių valdiklių dydžio.	Tikrinamos visos teksto sritys, kiekvienai sričiai apskaičiuojamas teksto dydis. Jei teksto dydis skiriasi nuo aplinkinių valdiklių dydžio – tai defektas.
3.	Naudojamo įrenginio ekrane tekstas neįskaitomas.	Naudojamas lange turintis būti tekstas nustatyta kalba, kuris paimamas iš programėlės išeities kodo. Jei teksto nėra paveikslėlyje atpažintų tekstų rinkinyje – tai defektas.
		Tikrinamas kiekvienos teksto srities realus šrifto dydis, panaudojant įrenginio ekrano informaciją. Jei šrifto dydis mažesnis nei 2mm – tai defektas.
4.	Teksto spalva kertasi su fono spalva, tekstą sunku perskaityti.	Naudojamas lange turintis būti tekstas nustatyta kalba, kuris paimamas iš programėlės išeities kodo. Jei teksto nėra paveikslėlyje atpažintų tekstų rinkinyje – tai defektas.
		Naudojamas lange turintis būti tekstas nustatyta kalba, kuris paimamas iš programėlės išeities kodo. Jei lange turinčio būti teksto spalva pakankamai sutampa su lango fono spalva – tai defektas.
5.	Dalį teksto dengia valdiklis arba dalis teksto yra už ekrano ribų.	Naudojamas lange turintis būti tekstas nustatyta kalba, kuris paimamas iš programėlės išeities kodo. Jei teksto nėra paveikslėlyje atpažintų tekstų rinkinyje, tikrinami įvairūs to teksto poeilučiai. Jei poeilutis yra lange turinčių būti tekstų rinkinyje – tai defektas.
6.	Tekstas per ilgas atvaizduoti, todėl jis nukertamas ir atsiranda daugtaškiai.	Naudojamas lange turintis būti tekstas nustatyta kalba, kuris paimamas iš programėlės išeities kodo. Jei teksto nėra paveikslėlyje atpažintų tekstų rinkinyje, tikrinami įvairūs to teksto poeilučiai. Jei poeilutis yra lange turinčių būti tekstų rinkinyje ir baigiasi daugtaškiu – tai defektas.
7.	Tekstas klaidingai atvaizduojamas – trūksta raidžių, atvaizduojamos blogos raidės, raidės pakeičiamos klaustukais.	Tikrinamas paveikslėlyje atpažintų tekstų rinkinys. Jei atpažintas tekstas nepraeina rašybos patikros – tai defektas.
		Tikrinamas paveikslėlyje atpažintų tekstų rinkinys. Jei atpažintas tekstas turi klaustuko simbolį, kuris yra ne pirmas ir ne paskutinis žodyje – tai defektas.
8.	Tekstas nematomas.	Naudojamas lange turintis būti tekstas nustatyta kalba, kuris paimamas iš programėlės išeities kodo. Jei teksto nėra paveikslėlyje atpažintų tekstų rinkinyje – tai defektas.

9.	Skirtingi žodžiai vartojami apibūdinti tam pačiam objektui.	Naudojamas lange turintis būti tekstas nustatyta kalba, kuris paimamas iš programėlės išeities kodo. Tekstas tikrinamas pažodžiui. Jei žodis yra daiktavardis, o teksto rinkinyje yra jo sinonimas – tai defektas.
10.	Vartojami dviprasmiški terminai, tas pats terminas vartojamas skirtingiems objektams.	Naudojamas iš programėlės išeities kodo sudarytas masyvas, kuris turi tame lange turinčių būti vertimo žymeklių sąrašą. Iš lange turinčio būti teksto nustatyta kalba, kuris paimamas iš programėlės išeities kodo, atrenkami visi unikalūs žodžiai. Žodžiai išverčiami į kitas programėlėje vartojamas kalbas ir sudedami į kitus masyvus. Į vieną patikros masyvą sudedami visi programėlės lange esantys tekstai, kurie turi tikrinamą žodį. Į kitą patikros masyvą sudedami visi to lango lokalizacijos tekstai, kurie turi tikrinamą išverstą žodį. Jei patikros masyvai nesutampa – tai defektas.
11.	Vartojamas techninis žargonas.	Naudojamas lange turintis būti tekstas nustatyta kalba, kuris paimamas iš programėlės išeities kodo. Tekstas tikrinamas pažodžiui. Jei žodis yra daiktavardis, nėra jungtukas ir yra techniškas – tai defektas.
12.	Nepastovus rašymo stilius, tekste yra klaidų.	Tikrinamas visas lange atpažintas tekstas. Jei tekstas nepraeina rašybos arba gramatikos patikros – tai defektas.
13.	Tekstas per ilgas, kad būtų informatyvus.	Naudojamas lange turintis būti tekstas nustatyta kalba, kuris paimamas iš programėlės išeities kodo. Tekstas suskaidomas į sakinius. Tikrinamas kiekvieno sakinio ilgis. Jei sakinys per ilgas – tai defektas.
14.	Tekstas nesuteikia jokios prasmingos informacijos.	Naudojamas lange turintis būti tekstas nustatyta kalba, kuris paimamas iš programėlės išeities kodo. Tekstas suskaidomas į sakinius. Kiekvienas sakinys tikrinamas ar yra aiškus. Jei ne – tai defektas.
15.	Vartojamas įžeidžiantis tekstas.	Tikrinamas visas lange atpažintas tekstas. Jei tekstas įžeidžiantis – tai defektas.
16.	Vartojamas klaidinantis tekstas.	Tikrinamas visas lange atpažintas tekstas. Jei tekstas klaidinantis – tai defektas.
17.	Tekste vartojami per ilgi nurodymai, trūksta iliustracijų.	Tikrinamos visos teksto sritys. Jei teksto sritis užima visą lango dydį – tai defektas.
18.	Tekstą sunku suprasti dėl jo rašymo stiliaus.	Naudojamas lange turintis būti tekstas nustatyta kalba, kuris paimamas iš programėlės išeities kodo. Jei teksto skaitomumo indeksas yra lygus ar aukštesnis nei universitetinio lygmens – tai defektas.
19.	Klaidingas daugtaškių vartojimas. Daugtaškiai turėtų nurodyti nuorodą į papildomą langą, tačiau jo nėra.	Naudojamas lange turintis būti tekstas nustatyta kalba, kuris paimamas iš programėlės išeities kodo. Jei tekstas baigiasi daugtaškiu ir yra ant nuorodos, bet nuoroda niekur neveda – tai defektas.
20.	Neišversti pranešimai, naudojami vertimų žymekliai.	Tikrinamas visas lange atpažintas tekstas. Jei vertimo žymeklių sąraše yra atpažintas tekstas – tai defektas.
21.	Rodomas pranešimas kita kalba, rodomi pranešimai keliomis kalbomis vienu metu.	Naudojamas lange turintis būti tekstas nustatyta kalba bei lokalizuotų tekstų masyvas kurie paimami iš programėlės išeities kodo.

		Tikrinamas visas lange atpažintas tekstas. Jei lange turinčio būti teksto masyve nėra atpažinto teksto, bet atpažintas tekstas yra lokalizuotų tekstų masyve – tai defektas.

Daliai defektų siūlomi aptikimo metodai yra ganėtinai panašūs, o tai padidina tikimybę neteisingai klasifikuoti aptiktą defektą. Kai kurių siūlomų metodų įgyvendinimas ganėtinai miglotas. Pvz.: žmogus gali atpažinti ar tekstas yra klaidinantis, o kompiuteriui tą padaryti yra sunkiau, nes nėra apibrėžto metodo tam. Tokiu atveju gali padėti mašininis mokymasis, jei randamas tinkamas duomenų rinkinys mokymui.

Kita vertus, dalis defektų gali būti aptinkama pasinaudojus įprastinėmis vaizdo ir teksto analizės technikomis ir gali būti įgyvendinami.

1.2. Egzistuojančių rinkoje priemonių palyginimas

Šiuo metu rinkoje nėra priemonių, kurios automatiškai aptiktų tekstinius defektus mobiliųjų programėlių vartotojo sąsajose.

Viena iš alternatyvų – realaus lango vaizdo palyginimas su lango eskizu. Tai leidžia padaryti *Android Studio* programavimo aplinka, naudojant *Layout Inspector* funkciją (15). Šis būdas turi savo trūkumų: norėdami ištestuoti visus programėlės langus, turėtume kiekvienam iš jų pasidaryti eskizą, o ir pats palyginimas nėra automatinis – eskizas uždedamas ant realaus lango vaizdo, o skirtumus turime pamatyti patys. Be to, šis būdas leistu aptikti tik akivaizdžiai matomus defektus.

Android Studio taip pat siūlo ir vartotojo sąsajos testavimo karkasą *UI Automator* (16). Jis leidžia rašyti testus, kurie sąveikauja su programėlės vartotojo sąsaja, pavyzdžiui: rasti mygtuką „Nustatymai“, jį paspausti ir patikrinti ar atsidarė naujas langas. Naudojantis šiuo testavimo karkasu parašytais testais būtų galima bandyti patikrinti ar lange atvaizduojamas tekstas buvo išverstas teisingai, ar nuoroda su daugtaškiu tikrai kažkur veda.

Dar vienas potencialus testavimo būdas – naudotis *Google* siūloma *Firebase Test Lab* (17). Tai debesija paremta programėlių testavimo infrastruktūra, leidžianti testuoti *Android* ar iOS programėles daugelyje skirtingų įrenginių su skirtingais nustatymais ir peržiūrėti rezultatus – įskaitant vykdymo žurnalus, vaizdo įrašus ir nuotraukas. Kadangi galima testuoti realiuose įrenginiuose, atsiranda galimybė aptikti klaidas, kurios pasitaiko tik tam tikruose įrenginiuose ar su tam tikrais nustatymais. *Firebase Test Lab* vykdo *Espresso* ir anksčiau aprašytus *UI Automator* testus *Android* programėlėms arba *XCTest* testus iOS programėlėms.

Testuojant tik *Android* įrenginiuose veikiančią programėlę, galima testus generuoti automatiškai, pasinaudojant *Firebase Test Lab Robo* testais. Jie analizuoja programėlės vartotojo sąsajos struktūrą ir ją metodiškai išvaikšto, simuliuodami vartotojo veiksmus. *Robo* testas visada simuliuos tuos pačius veiksmus ta pačia tvarka, jei yra naudojamas tame pačiame įrenginyje su tokiais pat nustatymais, o tai leidžia patikrinti ar tarp programėlės atnaujinimų pavyko ištaisyti klaidas bei ar neatsirado naujų. *Robo* testai pildo vykdymo žurnalų failus, išsaugo langų nuotraukas ir sukuria vaizdo įrašą, pasinaudodami tomis nuotraukomis, kad parodytų atliktų veiksmų seką. Naudojantis šia informacija galima rankiniu būdu paieškoti vartotojo sąsajos defektų programėlėje.

Aprašytos priemonės palyginamos toliau esančioje lentelėje (2 lentelė.).

2 lentelė. Priemonių palyginimas

Kriterijus/Programa	**Android Studio**	**Google Firebase Test Lab**
Vartotojo sąsajos palyginimas su eskizu	+	-
Tekstinių defektų paieškos funkcija	-	-
Testai, sąveikaujantys su vartotojo sąsaja	+	+
Galimybė aptikti tekstinius defektus	Rašant testus, rankiniu būdu	Rašant testus, rankiniu būdu
Testavimui prieinami realūs įrenginiai	Realiai turimi	Visi *Google* siūlomi
Palaikomos platformos	*Android*	*Android*, iOS
Vartotojo sąsajos testų generavimas	-	+
Kaina	Nemokamas	Nemokama – iki 10 testų per dieną. Mokama – nuo 1$ per įrenginį per dieną.

1.3. Pasirinktas sprendimas

1.3.1. Atrinkti metodai ir technologijos defektams aptikti

Kadangi didžioji dalis aprašytų metodų buvo taikomi būtent *Android* platformai kurtoms programėlėms, nuspręsta susitelkti į defektų paiešką *Android* mobiliųjų programėlių ekrano nuotraukose.

Kaip pagrindinė įrankio programavimo kalba pasirinkta *Java*. *Android* programėlių kūrime ši kalba dominuoja, todėl jos panaudojimas leis *Android* programėlių kūrėjams patiems pagal savo reikmes papildyti ar pakeisti sukurtą analizės įrankį.

Nuspręsta toliau tęsti KTU Programų inžinerijos katedroje vykdomus darbus ir įgyvendinti ketvirtąjį aprašytą metodą – testavimo metodą, pagrįstą vaizdo analize, skirtą automatiškai nustatyti vartotojo sąsajos defektus, skirtus mobiliesiems įrenginiams (12). Programėlės naršymo modeliui sudaryti galima panaudoti įrankį *DroidBot* (18). Tai atvirojo kodo testinių įvesčių generatorius *Android* programėlėms, kuris gali sudaryti vartotojo sąsajos išvaikščiojimo grafą, o šiuo atveju būtent to ir reikia. Tada, naudojantis šiuo grafu, programėlę galima išvaikščioti ir surinkti kiekvieno lango nuotraukas. Galiausiai, surinktas nuotraukas reikėtų perduoti į kuriamą įrankį, kuris atliktų analizę kiekvienoje nuotraukoje, ieškodamas joje iš anksto aprašytų defektų. Kiekvienam defektų tipui turės būti įgyvendintas jo aptikimo metodas. Nuspręsta įgyvendinti metodus 1.1.5 skyriuje aprašytiems defektams nr. 3, 6, 7, 12, 13, 15, 17, 20, 21, kadangi juos galima aptikti atliekant tik paveikslėlio analizę, o kiti defektai reikalauja ir programėlės išeities kodo analizės.

Paveikslėlyje esantys elementai bus analizuojami pasinaudojant *Google Cloud Vision API* paslauga (19). Ji siūlo galingus, iš anksto apmokytus mašininio mokymosi modelius per REST ir RPC sąsajas. *Google Vision* galima priskirti nuotraukoms etiketes ir suskirstyti jas į milijonus iš anksto nustatytų kategorijų, aptikti objektus ir veidus, skaityti spausdintą ir ranka rašytą tekstą ir vystyti nuotraukų katalogo metaduomenis. Šio projekto metu naudosimės teksto atpažinimo nuotraukoje funkcija – *Vision API* naudoja OCR aptikti tekstui nuotraukoje ir gali atpažinti virš 50 kalbų tekstą bei analizuoti įvairių tipų failus.

1.3.2. Įgyvendinimo problemos

Š. Packevičius ir kt. kai kuriuos tekstinių defektų aptikimo metodus aprašė ganėtinai miglotai (14), todėl kyla rizika, kad jų įgyvendinti nepavyks. Be to, dalis tekstinių defektų, aprašytų 1 lentelėje, yra ganėtinai panašūs vienas į kitą, o jų aptikimo metodai irgi dalinai sutampa, todėl padidėja rizika aptiktus defektus klasifikuoti neteisingai.

Tekstinių defektų paieškos metu gali pasitaikyti ir tokios ypatingos situacijos, į kurias reikėtų atsižvelgti:

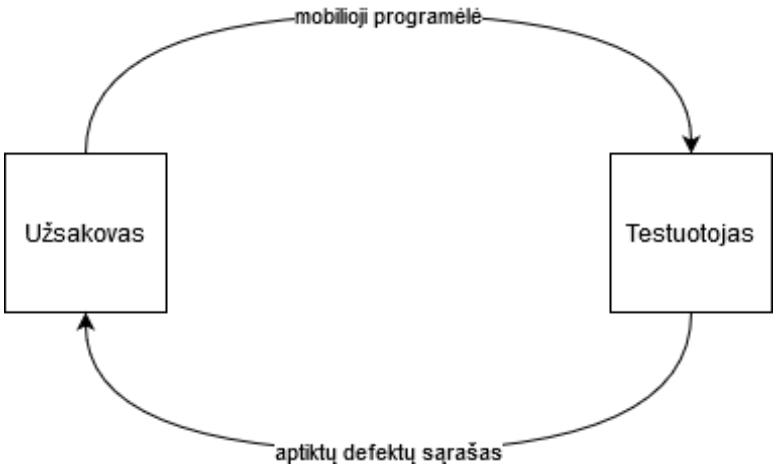
- programėlės lango nuotraukoje gali būti matoma reklama, kuri gali būti kitos kalbos nei programėlė, naudoti kitas spalvas ir t. t.;
- ieškant lokalizacijos klaidų, kyla rizika jas priskirti neteisingai kai tas pats žodis vartojamas keliose kalbose, o taip pat ir dėl panaudoto žargono.

2. Projektinė dalis

2.1. Veiklos sudėtis

2.1.1. Veiklos kontekstas

Įprastai testavimas vyksta taip: testavimo užsakovas pateikia testuotojui mobiliąją programėlę, testuotojas ją ištestuoja ir pateikia užsakovui aptiktų defektų sąrašą (1 pav.).



1 pav. Veiklos kontekstas

2.1.2. Veiklos padalinimas

Duomenų srautai detalizuojami toliau esančioje lentelėje (3 lentelė.).

3 lentelė. Veiklos padalinimas

Eil. Nr.	Įvykio pavadinimas	Įeinantys/Išeinantys informacijos srautai
1	Mobilioji programėlė	Mobiliosios programėlės išeities kodas, diegimo failai
2	Aptiktų defektų sąrašas	Programėlėje rastų defektų sąrašas

2.2. Sistemos funkcinis aprašymas

Šis projektas yra skirtas tekstinių defektų paieškai mobiliosiose programose analizuojant programos paveikslėlius. Naudojantis šiuo įrankiu galima sumažinti mobiliųjų programėlių testavimo trukmę ir jose esančių tekstinių defektų kiekį.

Testavimui galima paduoti tiek pavienes nuotraukas, tiek nuotraukų aplanką, tiek internete patalpintą nuotrauką (per jos nuorodą). Programėlėje galima keisti šiuos nustatymus:

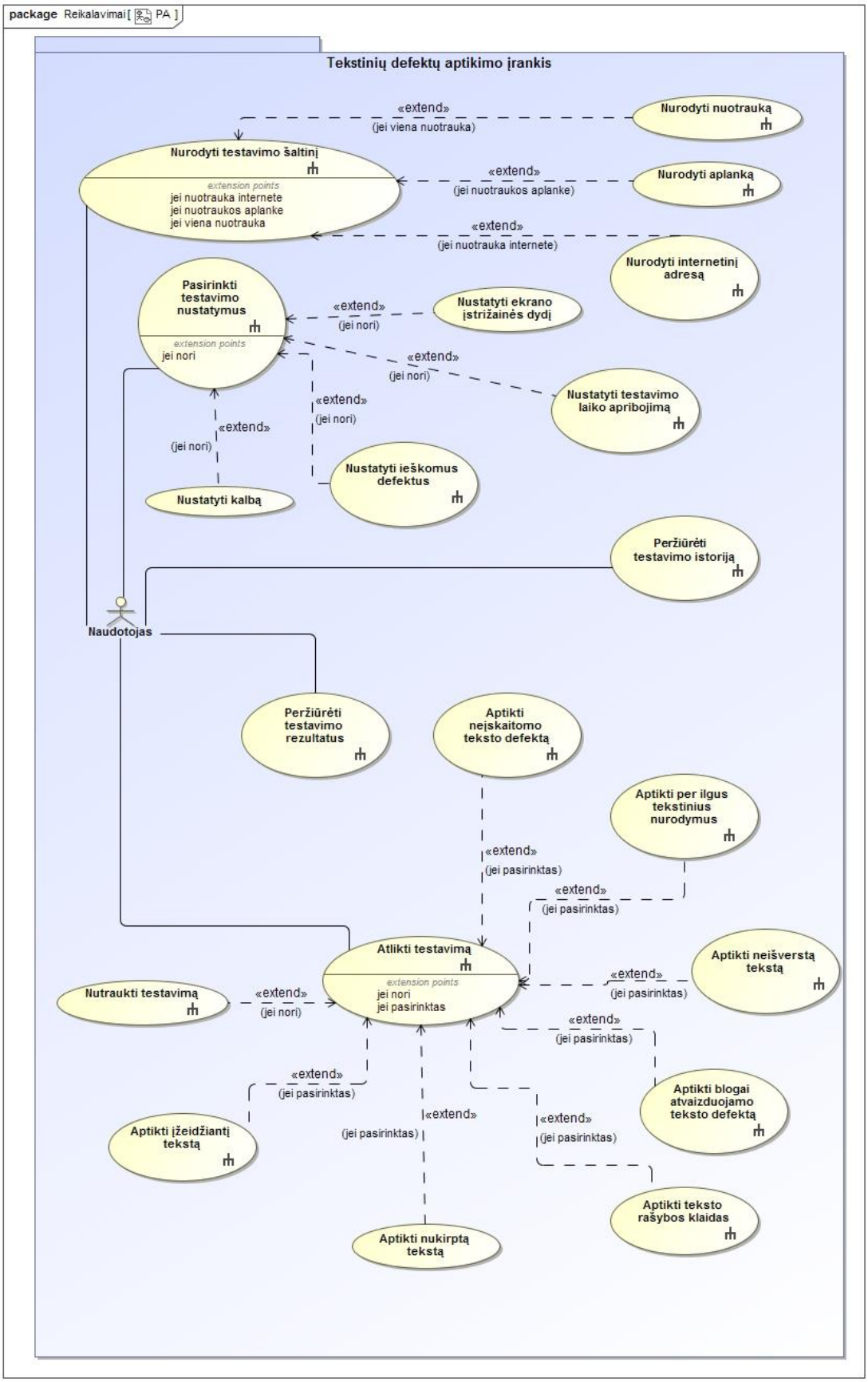
- nurodyti testuojamo įrenginio ekrano įstrižainės dydį;
- pasirinkti kokių defektų bus ieškoma testavimo metu;
- įvesti maksimalią testavimui skirto laiko trukmę;
- pasirinkti programos kalbą (lietuvių arba anglų).

Dabartinė įrankio versija geba aptikti šiuos tekstinius defektus:

- neišversto teksto defektas;
- neįskaitomo teksto defektas;
- tekstas su rašybos klaidomis;
- per ilgi tekstiniai nurodymai;
- blogai atvaizduojamas tekstas;
- nukirptas tekstas;
- įžeidžiantis tekstas.

Atlikus defektų paiešką, jos rezultatai yra įrašomi į failą, taigi yra galimybė peržiūrėti testavimo istoriją.

Ribas tarp sistemos ir vartotojo nusako toliau pateikta panaudojimo atvejų diagrama (2 pav.).



2 pav. Panaudojimo atvejų diagrama

2.3. Funkciniai reikalavimai

Šiame skyriuje pateikiami įrankiui iškelti papildomi funkciniai reikalavimai:

- reikalinga galimybė nutraukti analizę nepriklausomai nuo to ar ji baigta;
- naudotojui turi būti galima pasirinkti tiek savo kompiuteryje, tiek internete esančią nuotrauką, tiek visa aplanką su nuotraukomis;
- baigus testavimą turi būti rodomi ne tik visi aptikti defektai, bet ir kokiose nuotraukose jie buvo aptikti;
- testavimo rezultatai turi būti išsaugomi;
- nutraukus testavimą turi būti parodomi iki tol surinkti rezultatai.

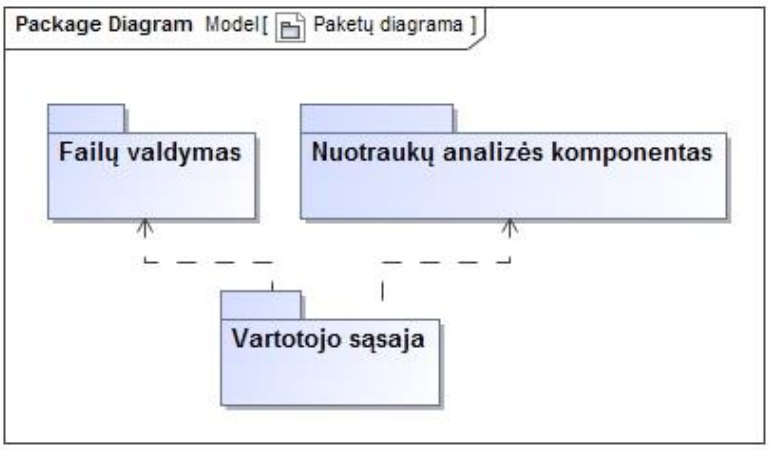
2.4. Sistemos apribojimai

Toliau pateikiami įrankiui iškelti nefunkciniai reikalavimai.

- Reikalavimai sistemos išvaizdai:
  - vartotojo sąsaja turi būti neįkyri.
- Reikalavimai panaudojamumui:
  - įrankiu turi būti lengva naudotis žmonėms su baziniais IT įgūdžiais;
  - įrankis turi būti prieinamas dvejomis kalbomis – lietuvių ir anglų;
  - įrankiu galima naudotis žmonėms, kurie nėra apmokyti juo naudotis;
  - naudotojams rodomi pranešimai turi būti suprantami plačiai auditorijai.
- Reikalavimai vykdymo charakteristikoms:
  - vieno defekto paieška vienoje nuotraukoje negali vykti ilgiau nei 10s;
  - įvykus klaidai tam tikro defekto paieškoje, darbas negali būti nutraukiamas – turi būti pereinama prie sekančio defekto paieškos;
  - įrankis turi galėti atlikti testavimą su 100 nuotraukų;
  - įrankis turi galėti atlikti testavimą su nuotrauka, kuri patalpinta internete.
- Reikalavimai saugumui:
  - įrankis turi nevykdyti testavimo su ne nuotraukų failais.

2.5. Sistemos statinis vaizdas

Sistemos išskaidymas į paketus pateikiamas toliau (3 pav.).



3 pav. Sistemos išskaidymas į paketus

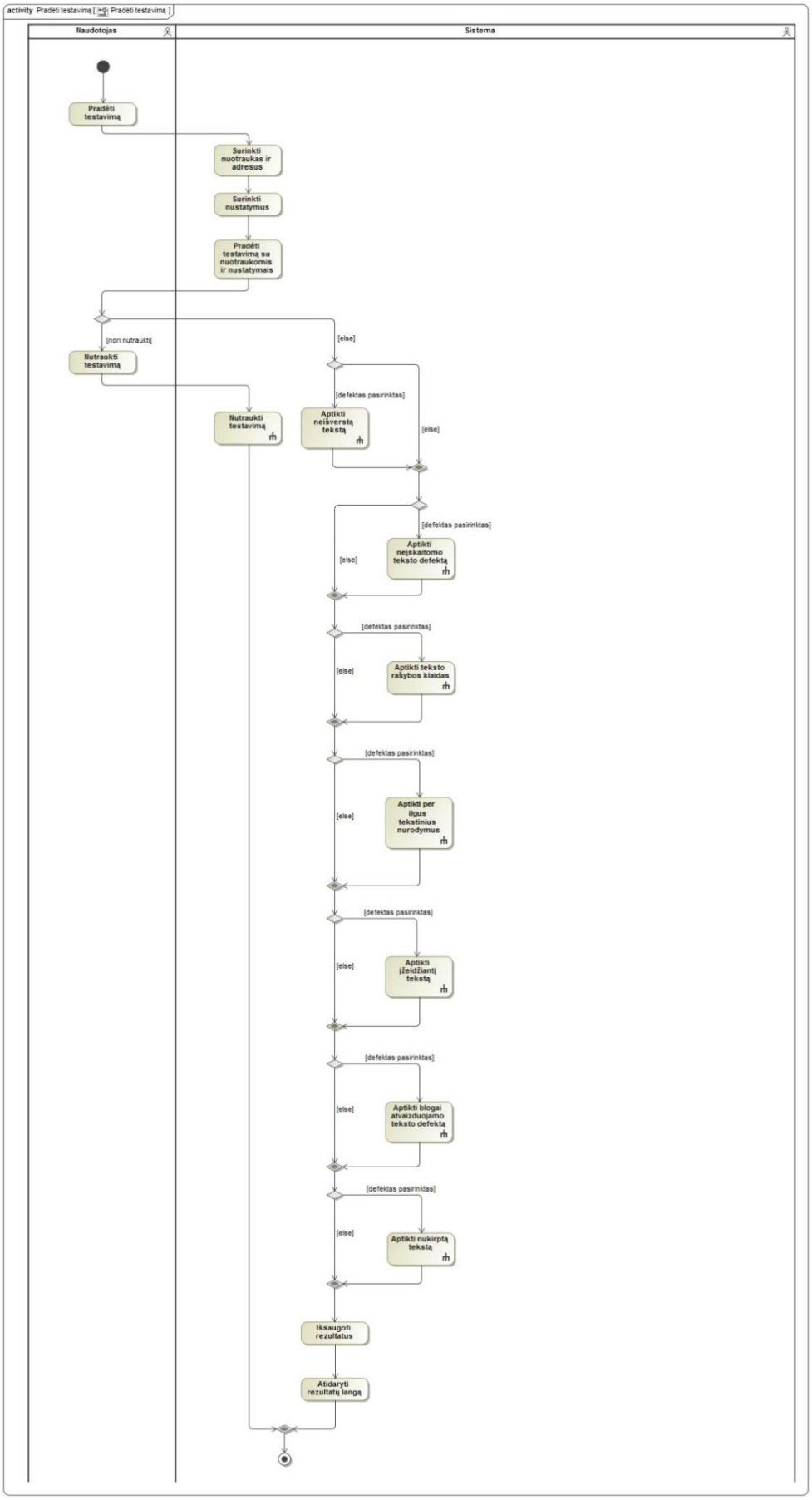
Paketas „Vartotojo sąsaja“ skirtas klasėms, realizuojančioms programos vartotojo sąsaja, paketas „Failų valdymas“ skirtas klasėms, dirbančioms su failais – nuskaitančioms nuotraukas, o paketas „Nuotraukų analizės komponentas“ apima nuotraukų analizės komponentą, su kuriuo dirba programa. Šis komponentas savyje apima pagrindinę projekto logiką – analizuoja nuotraukas, iš nuotraukų išgauna tekstus, analizuoja tekstus, pateikia rastus defektus.

2.6. Sistemos dinaminis vaizdas

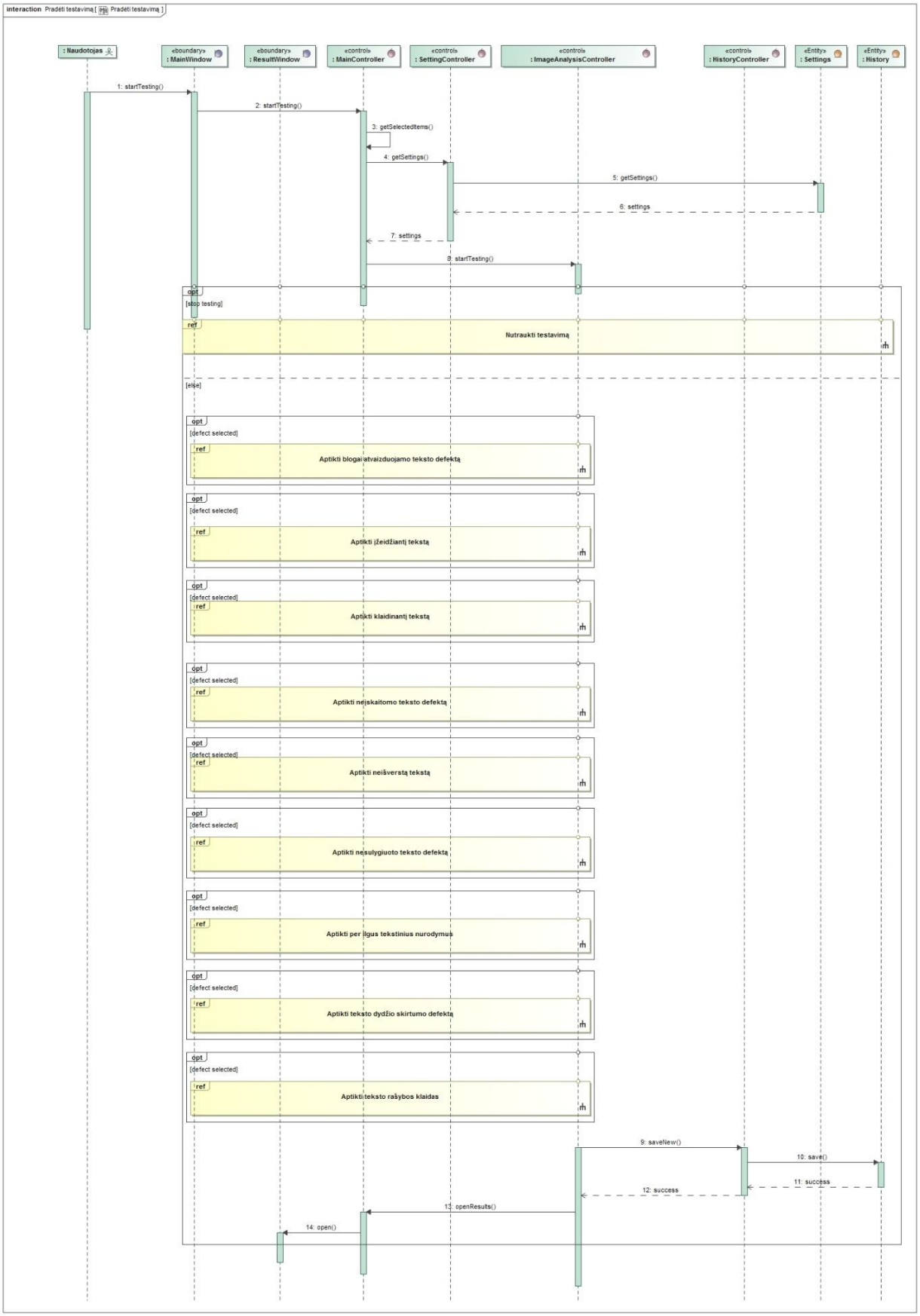
Dinaminį sistemos vaizdą pasirinkta pateikti veiklos ir sekų diagramomis, kurios parodo aktualiausių panaudojimo atvejų vykdymo scenarijus.

Failų valdymo posistemė skirta testavimo failų surinkimui. Ji realizuoja funkcijas, susijusias su testavimo šaltinio pasirinkimu. Nurodant testavimo šaltinį galima rinktis failą, aplanką arba įvesti internetinį adresą. Pasirenkant failą patikrinama ar tai nuotrauka ir jei taip – jis pridedamas į testavimui skirtų failų sąrašą. Pasirenkant aplanką einama per jo viduje esančius failus, tikrinami jų tipai ir jei tai nuotrauka – failas pridedamas į testavimui skirtų failų sąrašą. Įvedant internetinį adresą jis pridedamas į testavimui skirtų adresų sąrašą.

Vartotojo sąsajos posistemė suteikia vartotojui grafinę programos sąsają ir leidžia atlikti nustatymų pasirinkimo, rezultatų peržiūros, testavimo pradėjimo ir sustabdymo veiksmus. Pasirenkant testavimo nustatymus galima pakeisti testavimo laiko apribojimą bei testuojamų defektų sąrašą. Taip pat naudotojas gali peržiūrėti savo vykdytų testavimų istoriją bei detalius pasirinkto testavimo rezultatus. Be to, naudotojas gali pradėti (4 pav., 5 pav.) ir nutraukti tuo metu vykdomą testavimą.



4 pav. PA „Atlikti testavimą“ veiklos diagrama



5 pav. PA „Atlikti testavimą“ sekos diagrama

Nuotraukų analizės komponentas atsakingas už konkrečių defektų aptikimą nuotraukose. Šis komponentas gali aptikti neišverstą (7 pav.), neįskaitomą (8 pav.), blogai atvaizduojamą (12 pav.), įžeidžiantį (11 pav.), nukirptą (13 pav.) tekstą bei teksto rašybos klaidas (9 pav.) ir per ilgus tekstinius nurodymus (10 pav.).

2.7. Išdėstymo vaizdas

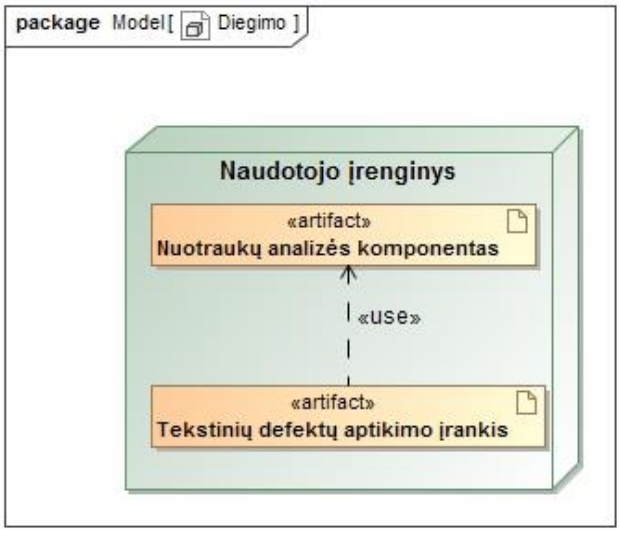
Sistema diegiama tiesiai į naudotojo asmeninį kompiuterį, diegimo diagrama pateikiama toliau (6 pav.).

Sistemai keliami minimalūs reikalavimai – ne mažesni nei sistemos kūrimui naudojamo kompiuterio:

- *Intel i5* 4 branduolių 2.5 Ghz procesorius;
- 8GB darbinės atminties;
- *NVIDIA GeForce GTX 860M* vaizdo plokštė su 2GB atminties;
- *Windows 8.1* operacinė sistema.

Taip pat sistemoje turi būti įdiegta:

- JRE 1.8 arba naujesnė versija



6 pav. Sistemos diegimo diagrama

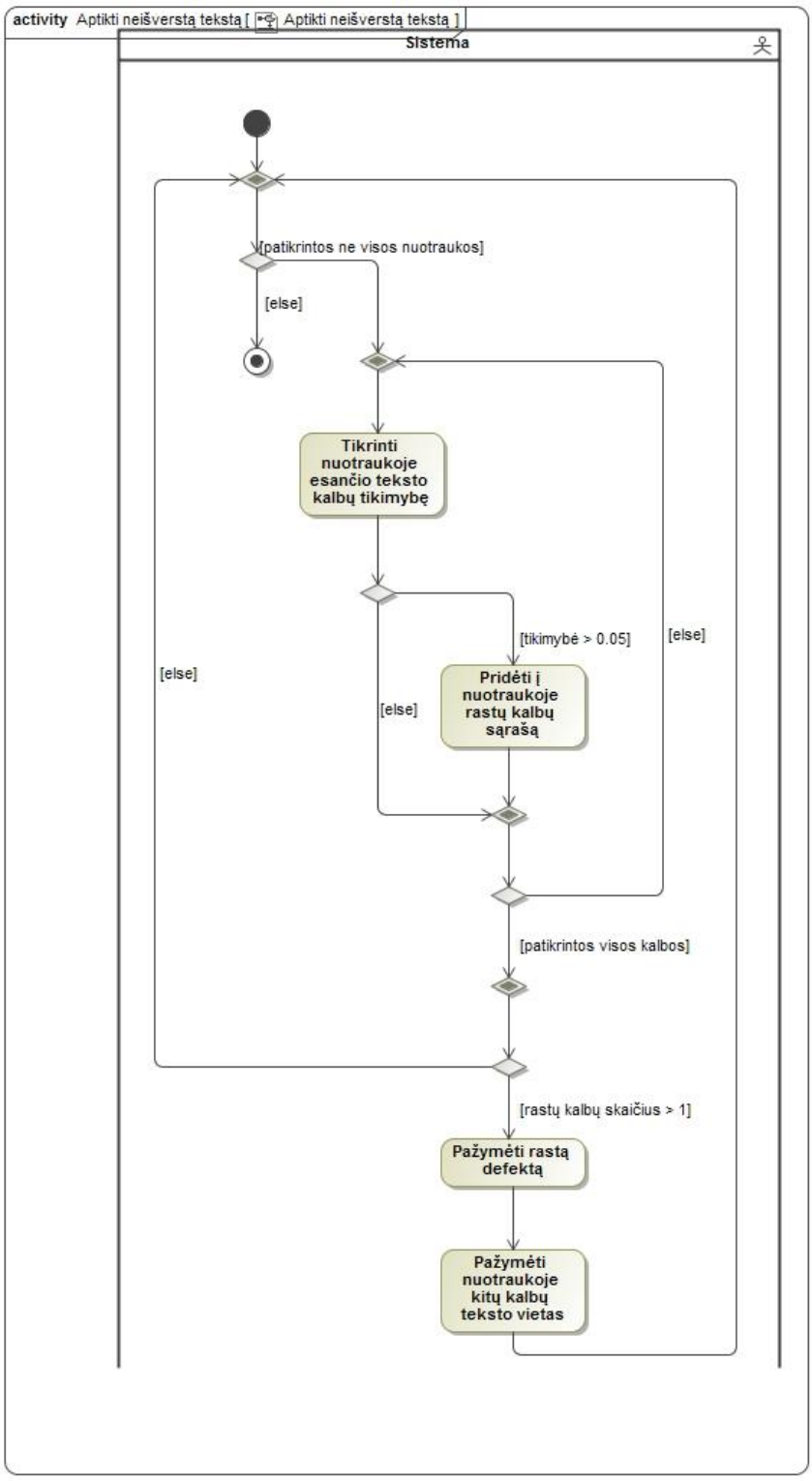
3. Tyrimo ir eksperimentinė dalis

3.1. Sukurti metodai

3.1.1. Neišversto teksto defekto aptikimas

Google Vision API nuotraukos analizės rezultatų rinkinyje gražina nuotraukoje aptiktas kalbas su tikimybėmis bei atskiruose teksto blokuose aptiktas kalbas su tikimybėmis. Pirmasis metodo variantas buvo tikrinti kalbų kiekį visoje nuotraukoje neatsižvelgiant į tikimybes, tačiau jis veikė blogai – buvo labai daug klaidingai aptinkamų defektų, nes nuotraukose buvo žodžiai, kurie skirtingose kalbose rašomi vienodai, todėl rezultatų rinkinyje atsirasdavo daugiau nei viena kalba.

Patobulintas metodo variantas – tikrinti kalbų kiekį, kurios turi daugiau nei 5 % tikimybę (7 pav.). Šiuo atveju metodas veikė geriau, tačiau vis dar yra nemažai klaidingai aptinkamų defektų. Defektas klaidingai aptinkamas kalbos pasirinkimo ekranuose, nes juose visada yra tekstas keliomis kalbomis, tačiau jis ten ir turi būti. Taip pat klaidingam aptikimui turi įtakos ir pačioje programėlėje esančių foninių ar naudotojo įkeltų paveikslėlių tekstas, kadangi paveikslėliai gali būti nesusiję su programėle, o *Google Vision* to neatskiria. Klaidingam veikimui įtakos turi ir ekrano nuotraukose matomos reklamos, kadangi reklamos kalba gali nesutapti su programėlės kalba, taigi rezultatų rinkinyje atsiranda papildoma kalba.



7 pav. PA „Aptikti neišverstą tekstą“ veiklos diagrama

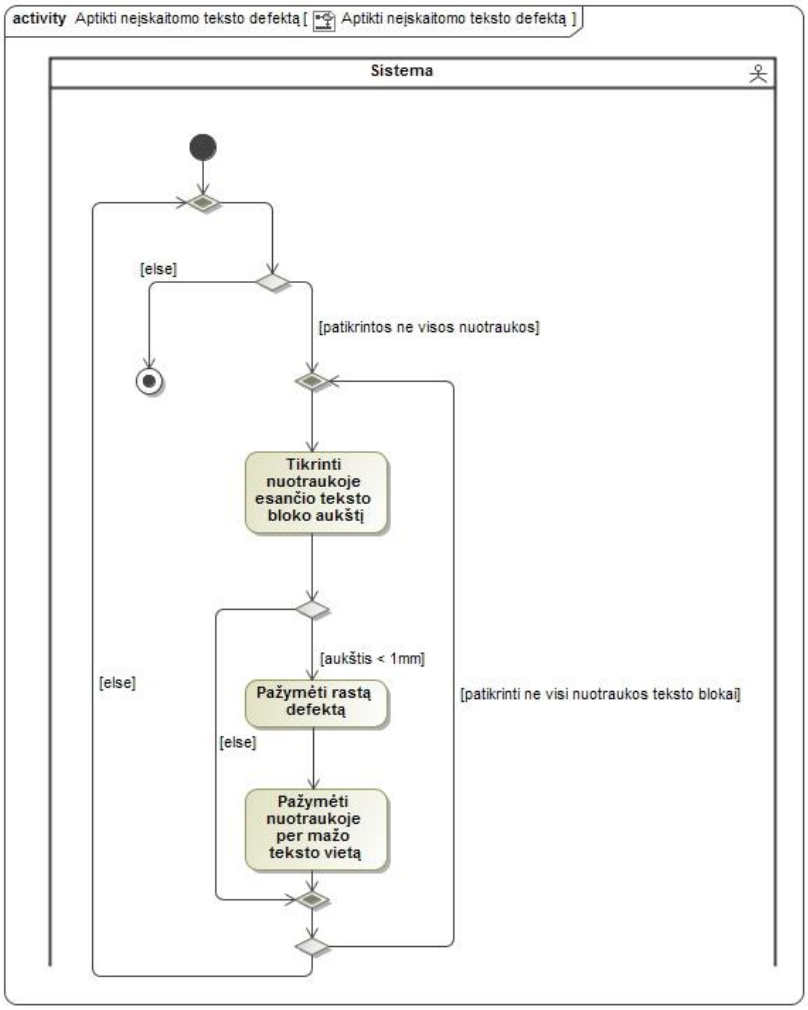
3.1.2. Neįskaitomo teksto defekto aptikimas

Pirmasis šio metodo variantas – išskaičiuoti teksto aukštį pikseliais. Įskaitomumui buvo pasirinkta 16 pikselių riba. Šis variantas nepasiteisino, kadangi įskaitomumui turi įtakos ir fizinis ekrano dydis bei įrenginio ekrano rezoliucija – 16 pikselių aukščio tekstas dideliame ekrane su maža rezoliucija užims gerokai daugiau vietos nei mažame ekrane su aukšta rezoliucija.

Antrasis metodo variantas (8 pav.) išskaičiuoja fizinį teksto aukštį pagal tokią formulę:

$$\frac{ekrano\ aukštis\ (mm) * teksto\ aukštis\ (px)}{nuotraukos\ aukštis\ (px)}$$

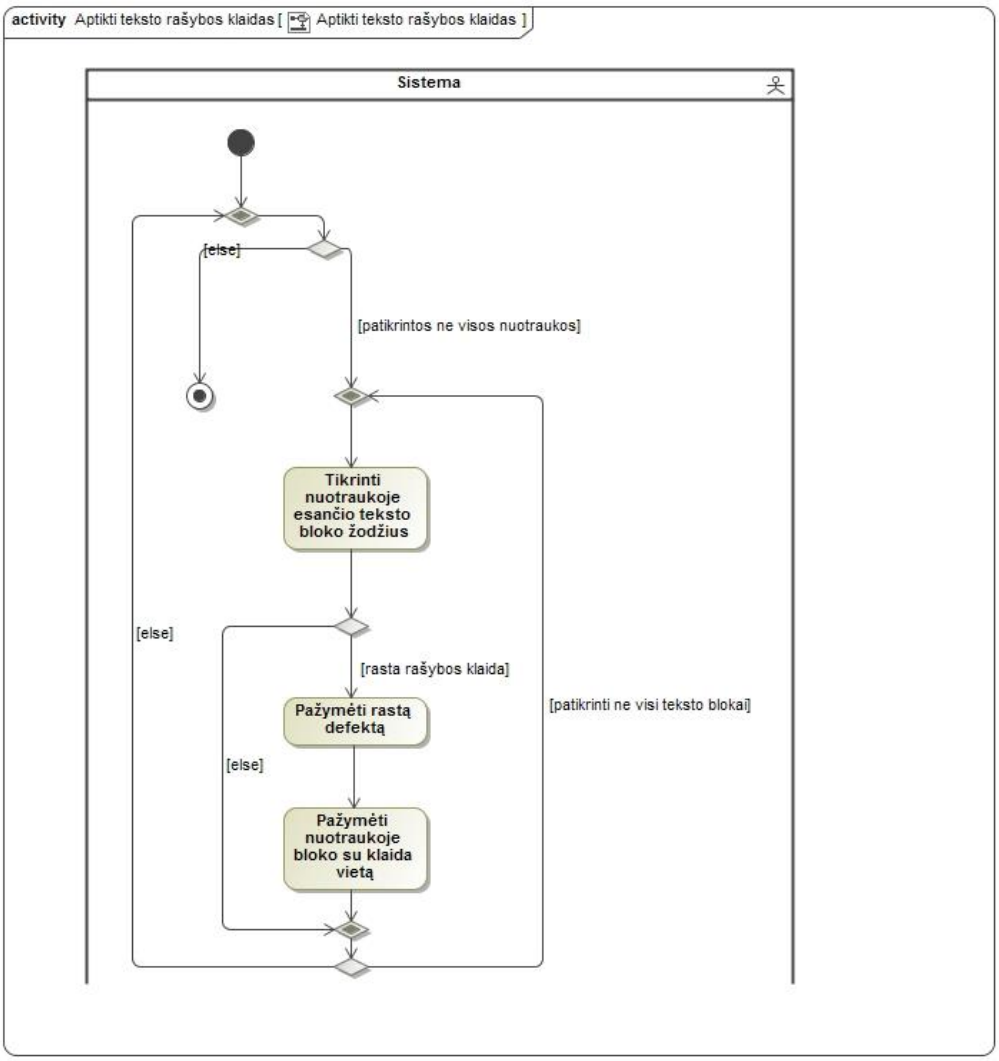
Šiuo atveju paeksperimentavus su keliomis skirtingomis reikšmėmis įskaitomumui pasirinkta 1.0 mm riba. Palyginimui – 5 colių ekrano įstrižainę turinčio įrenginio viršutiniame kampe esančio laikrodžio tekstas yra apie 2 mm aukščio, taigi, jei tekstas yra per pus mažesnis – jį laikome neįskaitomu.



8 pav. PA „Aptikti neįskaitomo teksto defektą“ veiklos diagrama

3.1.3. Rašybos klaidų teksto defekto aptikimas

Rašybos klaidoms aptikti panaudota *LanguageTool* biblioteka. Šios bibliotekos minusas – ji veikia tik su populiariausiomis kalbomis, todėl lietuviškas ar kitos nepalaikomos kalbos tekstas yra netikrinamas. Pirmas metodo variantas – tikrinti nuotraukoje esantį tekstą pagal nuotraukos kalbą su didžiausia tikimybe. Šis sprendimas nepasiteisino, kadangi nuotraukoje gali būti daugiau nei viena kalba. Patobulintame metodo variante (9 pav.) žodžiai yra tikrinami pagal to teksto bloko, kuriame jie yra, kalbą su didžiausia tikimybe.



9 pav. PA „Aptikti teksto rašybos klaidas“ veiklos diagrama

Šio metodo vykdymo trukmė yra ilgiausia iš visų – per valandą patikrinama apie 500 nuotraukų. Palyginimui – kitų defektų aptikimo metodai tokį nuotraukų kiekį patikrina per maždaug minutę. Kadangi eksperimentui nebuvo svarbu kiek rašybos klaidų yra nuotraukoje ir kokiose vietose jos yra, metodas buvo šiek tiek pakoreguotas – ieškoma tik pačios pirmos rašybos klaidos, o ją radus iškart pereinama prie kitos nuotraukos. Tai padėjo pakelti metodo greitaveiką iki maždaug 1500 nuotraukų per valandą.

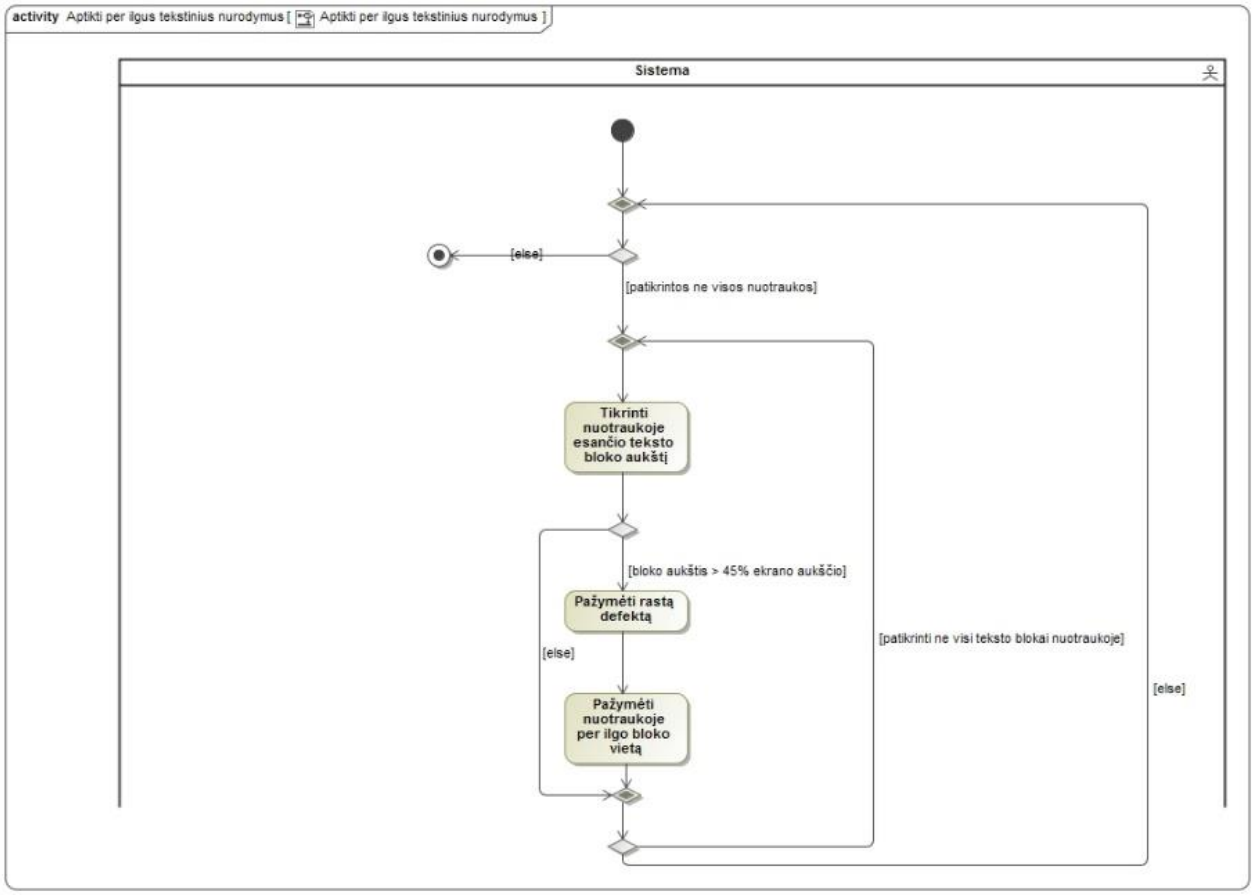
3.1.4. Per ilgų tekstinių nurodymų defekto aptikimas

Pirmasis metodo variantas – tikrinti teksto bloke esančių žodžių kiekį. Defektui aptikti pasirinkta 20 žodžių riba. Šis variantas nepasiteisino, nes yra per daug subjektyvus – anglų kalboje 15 žodžių gali reikšti tiek pat, kiek vokiečių kalboje reiškia 5 žodžiai. Antrasis variantas (10 pav.) – išskaičiuoti nuotraukoje esančių teksto blokų užimamą aukštį procentais, pagal formulę:

$$\frac{teksto\ bloko\ aukštis\ (px)}{ekrano\ aukšis\ (px)}$$

Defektui aptikti pasirinkta 45 % riba, nes reikia atsižvelgti į tai, kad nuotraukoje dažnai matoma telefono viršutinė pranešimų juosta bei apatinė valdiklių sekcija, todėl 45 % viso ekrano užimantis tekstas užims apytiksliai pusę pačios programėlės ekrano ploto.

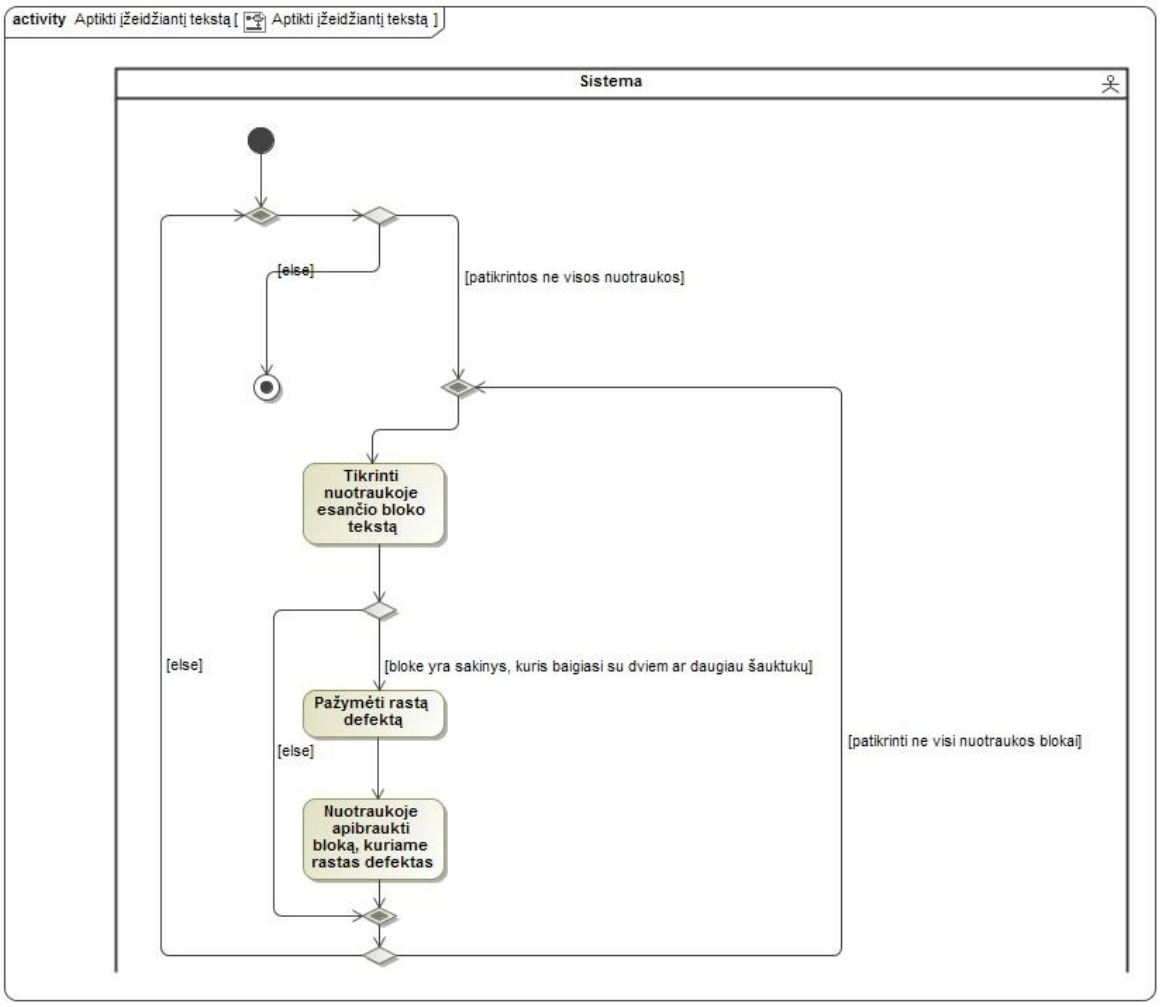
Tačiau eksperimento vykdymo metu pastebėta, kad jei tekstas pasuktas arba yra parašytas labai dideliu šriftu – jis aptinkamas kaip defektas, nors tas tekstas gali susidėti vos iš kelių žodžių. Taigi, galutinis metodo variantas papildomai vėl tikrina žodžių kiekį teksto bloke – jame turi būti bent 10 žodžių. Taip buvo atmetama nemaža dalis klaidingai aptinkamų defektų.



10 pav. PA „Aptikti per ilgus tekstinius nurodymus“ veiklos diagrama

3.1.5. Įžeidžiančio teksto defekto aptikimas

Pirmasis metodo variantas – ieškoti teksto blokų, parašytų vien didžiosiomis raidėmis arba turinčių daugiau nei vieną šauktuką sakinio gale. Buvo vadovaujamasi tokia logika, kad jei tekstas yra rašomas didžiosiomis raidėmis arba turi daugiau nei vieną šauktuką, naudotojas gali jaustis lyg ant jo būtų šaukiama ir dėl to įsižeisti. Didžiųjų raidžių paieška nepasiteisino, kadangi mobiliosiose programėlėse jos yra vartojamos daugelyje vietų, todėl dabartinė metodo versija ieško tik sakinių, besibaigiančių su daugiau nei vienu šauktuku (11 pav.).

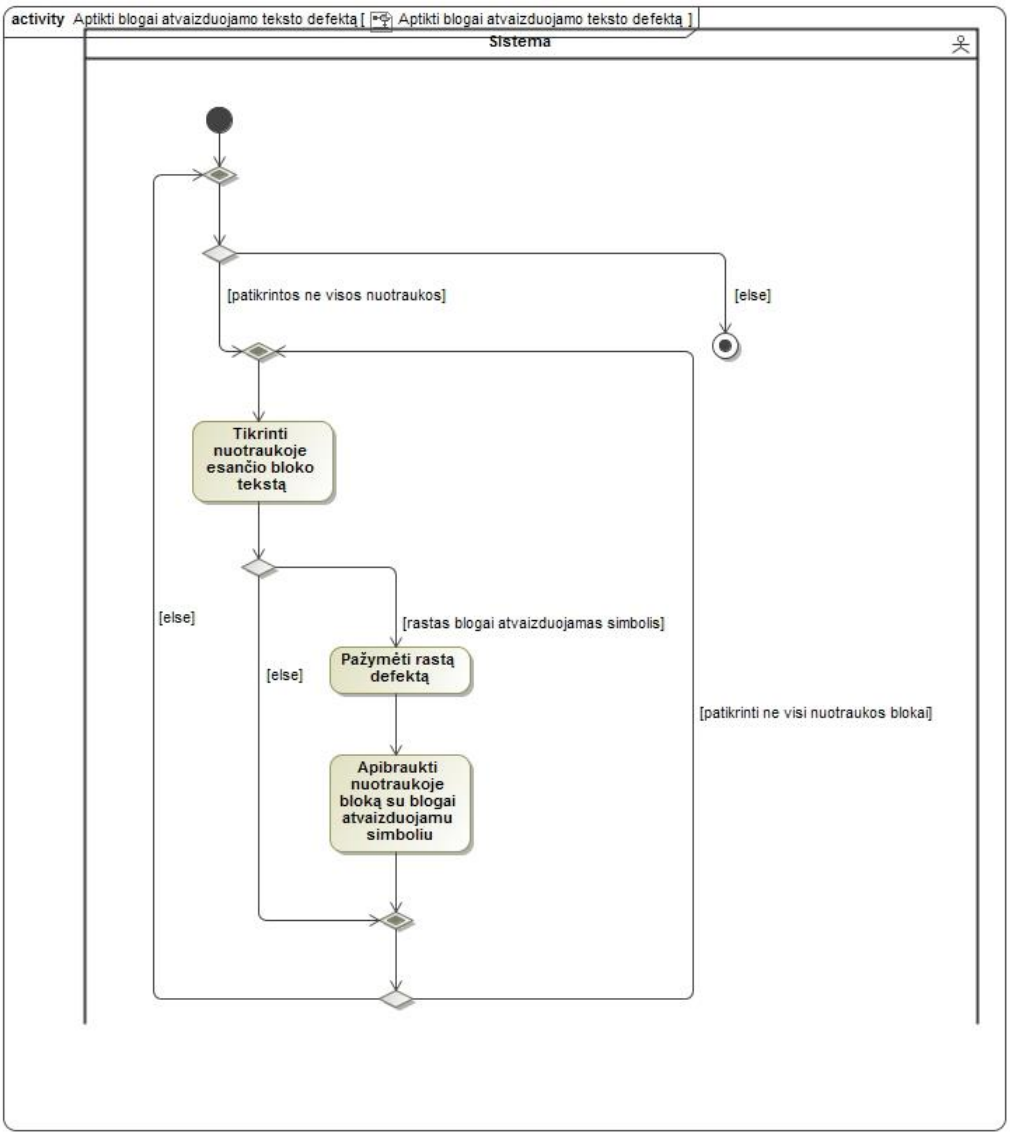


11 pav. PA „Aptikti įžeidžiantį tekstą“ veiklos diagrama

3.1.6. Blogai atvaizduojamo teksto defekto aptikimas

Bandant realizuoti šio defekto aptikimo metodą buvo pastebėta problema, kad *Google Vision* blogai atvaizduojamus simbolius (tokius kaip **?**, Ã, ‰) vis tiek bando interpretuoti kaip kokį nors realų simbolį (dažniausiai – panašiausią raidę), todėl dabartinė metodo versija (12 pav.) tikrina ar kurioje nors žodžio dalyje išskyrus pabaigą yra klaustukas. Vadovaujamasi tuo, kad *Android* programėlėje nesutapus teksto koduotei simbolis pakeičiamas į klaustuką.

Tačiau buvo pastebėta, kad ieškant tik klaustuko, defektas yra klaidingai aptinkamas nuotraukose, kuriose yra nuoroda, nes nuorodoje parametrai pridedami po klaustuko, taigi papildomai buvo pridėtas filtravimas tekstui, turinčiam simbolius „www.“, „http://“, „https://“.

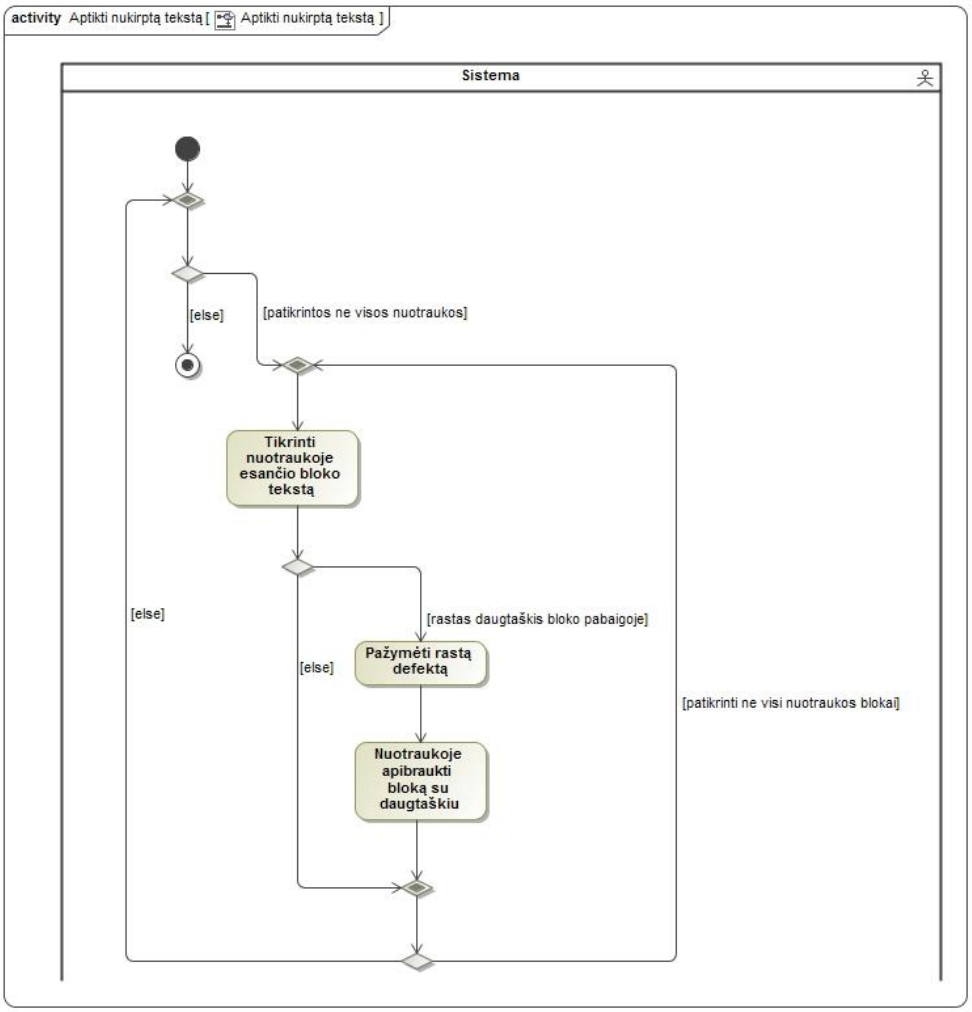


12 pav. PA „Aptikti blogai atvaizduojamo teksto defektą“ veiklos diagrama

3.1.7. Nukirpto teksto defekto aptikimas

Ieškant šio defekto tikrinama ar žodis baigiasi daugtaškiu (13 pav.). Tačiau šis metodas kai kuriuos defektus aptinka klaidingai, kadangi netikriname ar žodis iš tikrųjų yra nukirptas – tas daugtaškis gali būti tiesiog panaudotas sakinio gale kaip skyrybos ženklas. Problema su šiuo defektu yra ta, kad net ir nukirpus žodžio galūnę vis tiek galime gauti kitą pilną žodį, todėl defektas būtų neaptiktas. Turint prieigą prie programėlės išeities tekstų galima būtų gerokai patobulinti šio defekto aptikimą, kadangi tokiu atveju žodžius su daugtaškiais būtų galima palyginti su išeities tekstuose esančiais ir patikrinti ar jie iš tikrųjų nukirpti.

Taip pat atliekant testavimą buvo pastebėta, kad tekstas su daugtaškiu dažnai yra rodomas atliekant skaičiavimus ar kažką kraunant, pvz.: „Loading...“, „Please wait...“, todėl šie dažniausiai pasitaikantys variantai buvo nufiltruoti.



13 pav. PA „Aptikti nukirptą tekstą“ veiklos diagrama

3.2. Eksperimentas

3.2.1. Duomenys

Eksperimentui naudojamas apie 95 tūkst. ekrano nuotraukų turintis rinkinys, kuris surinktas iš 781 *Android* programėlių, panaudojant 12 skirtingų mobiliųjų įrenginių, turinčių skirtingas ekrano įstrižaines, rezoliucijas ir skirtingas kalbas (20). Šis nuotraukų rinkinys buvo rankiniu būdu peržiūrėtas ir suklasifikuoti visi nuotraukose rasti defektai. Peržiūros rezultatai pateikiami lentelėje toliau (4 lentelė.).

4 lentelė. Rankiniu būdu aptikti defektai

Defektas	**Kiekis**
Rašybos klaidos	11
Nukirptas tekstas	6268
Neišverstas tekstas	1251
Neįskaitomas tekstas	324
Blogai atvaizduojamas tekstas	3

3.2.2. Eksperimento eiga ir rezultatai

Pirmiausia, prieš atliekant eksperimentą, visos nuotraukos buvo nusiųstos į *Google Vision* API, o grąžinti rezultatai išsaugoti į JSON formato failus. Taip buvo daroma siekiant sumažinti užklausų kiekį, nes užklausos yra apmokestintos. Atlikus tai, įrankis buvo pakoreguotas, kad neatlikinėtų užklausų į *Google Vision*, o nuskaitytų rezultatus iš failo.

Prieš atliekant testavimą su visu nuotraukų rinkiniu, testavimas buvo atliekamas su mažomis jo dalimis, peržiūrint aptiktus defektus ir šiek tiek pakoreguojant metodus, siekiant pašalinti kuo daugiau klaidingai aptinkamų defektų. Atlikti pakeitimai ir galutinės metodų versijos aprašytos 3.1 poskyryje.

Tada buvo atliktas testavimas su pilnu nuotraukų rinkiniu, tačiau jis buvo išskaidytas į mažesnes dalis po maždaug 2000 nuotraukų, nes su didesniu jų kiekiu neužtekdavo kompiuterio atminties ir programa lūždavo. Pirmiausia visos nuotraukos buvo ištestuotos ieškant visų defektų išskyrus rašybos klaidų, o po to – ieškant tik rašybos klaidų defekto. Taip buvo daroma dėl jau minėtos problemos su rašybos klaidų defekto aptikimo metodo greitaveika. Rašybos klaidų defekto paieška visame duomenų rinkinyje užtruko beveik 50 valandų, o likusių defektų paieška – apie 4 valandas. Toliau pateikiami gauti rezultatai (5 lentelė.).

5 lentelė. Įrankio aptikti defektai

Defektas	**Kiekis**
Rašybos klaidos	52355
Nukirptas tekstas	4331
Neišverstas tekstas	43523
Neįskaitomas tekstas	33700
Blogai atvaizduojamas tekstas	24
Per ilgas tekstas	264
Įžeidžiantis tekstas	196

3.2.3. Rezultatų analizė

Šioje dalyje atliekamas rankiniu būdu aptiktų ir įrankio aptiktų defektų palyginimas (6 lentelė.).

6 lentelė. Aptiktų defektų kiekio palyginimas

Defektas	Rankiniu būdu aptiktas kiekis	Įrankio aptiktas kiekis
Rašybos klaidos	11	52355
Nukirptas tekstas	6268	4331
Neišverstas tekstas	1251	43523
Neįskaitomas tekstas	324	33700
Blogai atvaizduojamas tekstas	3	24
Per ilgas tekstas	-	264
Įžeidžiantis tekstas	-	196

Taip pat pateikiamas detalesnis įrankio aptiktų defektų išskaidymas į teisingai ir klaidingai aptiktus ir neaptiktus (7 lentelė.), darant prielaidą, kad rankiniu būdu ieškant defektų buvo pasiektas 100 %

tikslumas (nors taip nėra, nes rankiniu būdu ieškant defektų taip pat pasitaikė klaidingai aptiktų ir neaptiktų atvejų).

7 lentelė. Įrankio aptiktų defektų tikslumo analizė

Defektas	**Teisingai aptikti defektai**	**Teisingai neaptikti defektai**	**Klaidingai aptikti defektai**	**Klaidingai neaptikti defektai**
Rašybos klaidos	8 (73%)	41622 (44%)	52347 (56%)	3 (27%)
Nukirptas tekstas	593 (9%)	83843 (96%)	3738 (4%)	5806 (91%)
Neišverstas tekstas	935 (74%)	50125 (54%)	42588 (46%)	332 (26%)
Neįskaitomas tekstas	276 (80%)	60141 (64%)	33494 (36%)	69 (20%)
Blogai atvaizduojamas tekstas	0 (0%)	93953 (100%)	24 (0%)	3 (100%)

Rankiniu būdu aptiktų rašybos klaidų kiekis atrodo neįtikėtinai mažas. Tikrinant rankiniu būdu, aptiktų klaidų kiekis priklauso nuo to, kiek kalbų moka tikrintojas bei kaip gerai jas išmano. Galima daryti prielaidą, kad dalis klaidų buvo nepastebėta, nes tikrintojai nemokėjo tam tikros kalbos arba ją mokėjo nepakankamai gerai, kad pastebėtų rašybos klaidą. Kita vertus, įrankis klaidas aptiko per dažnai – maždaug kas antroje nuotraukoje. Tam įtakos turėjo ir tai, kad naudota biblioteka neatpažino techninių sąvokų, net ir tokių, kaip mobiliosiose programėlėse labai dažnai pasitaikančios „*Android*“, „*Bluetooth*. Be to, dažnai kaip klaida buvo pažymimas programėlės ar kūrėjų įmonės pavadinimas, nes jis buvo gramatiškai netaisyklingas. Žmogus tikrindamas tekstą žino, kad į tokias vietas reikėtų nekreipti dėmesio, tačiau įrankis tikrina visą matomą tekstą, neatsižvelgdamas į jokias kitas aplinkybes.

Rankiniu būdu ieškant nukirpto teksto defekto buvo priskaičiuojami ne tik tie tekstai, kurie buvo automatiškai sutrumpinti su daugtaškiu, bet ir tie tekstai, kurie išlysdavo už ekrano ribų ir buvo matoma tik pusę teksto, bei tie tekstai, kurie palįsdavo po mygtuku ar kitu elementu ir dalis teksto buvo užstojama, todėl rankiniu būdu aptiktas kiekis yra didesnis nei įrankio aptiktas kiekis, o įrankio klaidingai neaptiktų defektų kiekis išauga. Šio defekto paieškos algoritmas veikia gan gerai – defektas buvo klaidingai neaptinkamas tais atvejais, kai *Google Vision* neatpažino nuotraukoje esančio daugtaškio ir vietoje jo rezultatų rinkinyje grąžindavo vieną tašką arba negrąžindavo jokio simbolio. Taip pat pasitaikė ir klaidingo aptikimo atvejų, nes kaip buvo minėta aprašant algoritmą – netikrinama ar tekstas iš tikrųjų nukirptas, ar tiesiog baigiasi daugtaškiu. Dažniausiai pasitaikantys atvejai – „Loading...“ ir „Please wait..“ – buvo nufiltruoti.

Neišversto teksto defekto aptikimo metodas klaidingai aptiko defektus daugelyje nuotraukų. Tam turėjo įtakos tai, kad programėlės nuotraukose matėsi reklamos, kurios ne visada buvo tos pačios kalbos, kaip pati programėlė. Taip pat defektai buvo klaidingai aptinkami programėlių, įmonių ar kitų dalykų pavadinimuose, nes tam pavadinimui gali būti priskirta kitokia kalba, nei likusiam programėlės tekstui. Pasitaikydavo atvejų, kai defektas buvo aptinkamas žodžiuose, kurie keliose kalbose užsirašo visiškai vienodai – tokiam žodžiui ne visada buvo priskirta tokia pati kalba, kokia priskirta kitam programėlėje esančiam tekstui. Be to, *Google Vision* aptikdavo tekstus ir nuotraukoje matomuose paveikslėliuose, kurie buvo panaudoti fone kaip dizaino elementas arba įkelti vartotojų ir nesusiję su pačia programėle, o ten esantis tekstas dažnu atveju buvo ne tos pačios kalbos, jame buvo ir įvairūs pavadinimai, o tai apsunkino kalbos nustatymą. Buvo pastebėta ir tai, kad defektas buvo

aptiktas visuose kalbos pasirinkimo ekranuose, nes ten visada yra daugiau nei viena kalba parašytas tekstas.

Neįskaitomo teksto defekto aptikimo metodas taip pat aptiko didžiulį kiekį defektų. Taip nutiko, nes kaip buvo aprašyta anksčiau, *Google Vision* nuskaito tekstą ir iš nuotraukoje esančių nuotraukų, o tokiu atveju jis dažnai yra labai mažas, nes ekrane gali būti matoma nuotraukos miniatiūra. Be to, pasirinkta defekto aptikimo riba yra labai subjektyvi – tekstas iš tikrųjų yra mažas, bet vienas žmogus jį gali laikyti jau per mažu, o kitas – dar ne.

Nuotraukose, kurios rankiniu būdu buvo klasifikuotos, kaip turinčios blogai atvaizduojamą tekstą, yra blogai atvaizduojami simboliai, kuriuos, kaip buvo minėta aprašant metodą, *Google Vision* interpretuoja į panašiausią raidę, taigi įrankis neaptiko defekto nei vienoje iš tų nuotraukų. Nuotraukose, kurias įrankis klasifikavo, kaip turinčias blogai atvaizduojamą tekstą, defektai irgi buvo aptinkami klaidingai, nes jose esantis tekstas baigdavosi klaustuku, o sekantis tekstas prasidėdavo be tarpo, todėl klaustukas atsirasdavo žodžio viduryje. Tam tikra prasme tai irgi yra defektas, bet ne toks, kurį turi aptikti šis metodas.

Per ilgo teksto rankiniu būdu ieškota nebuvo, todėl bus aptartas tik realizuoto metodo veikimas. Tekstas, kuris klasifikuotas kaip per ilgas, dažnu atveju iš tikrųjų užėmė daugiau nei pusę ekrano (dažniausiai visą), tačiau retais atvejais klaidingai buvo klasifikuotas tekstas, kuris buvo pasuktas šonu, nes jis užėmė didelę ekrano dalį – nuo ekrano apačios iki viršaus bei tekstas, turintis labai didelį šriftą. Pasitaikė ir klaidingo neaptikimo atvejų – kadangi algoritmas tikrina ar vienas teksto blokas užima didelį plotą, tai jei nuotraukoje yra daug teksto, tačiau tarp jo yra pakankami tarpai, kad jis būtų išskaidytas į atskirus mažesnius teksto blokus, jis nėra klasifikuojamas kaip defektas.

Įžeidžiantis tekstas taip pat nebuvo klasifikuojamas rankiniu būdu, todėl aptariamas tik realizuoto metodo veikimas. Algoritmas ieškojo teksto, besibaigiančio trimis ar daugiau šauktukų ir visose nuotraukose, kurios klasifikuotos kaip turinčios tokį defektą, iš tikrųjų buvo toks tekstas, taigi metodas veikia ganėtinai neblogai, tačiau yra daug potencialo jo tobulinimui.

3.2.4. Ateities darbai

Vienas iš būdų pašalinti didelį kiekį klaidingai aptinkamų defektų yra tikrinti ne tik programos paveikslėlį, bet ir programos išeities kodą. Viena iš didžiausių problemų yra ta, kad paveikslėlyje aptinkamas ir toks tekstas, kuris neturėtų būti tikrinamas: programos ekrane matomų foninių ar vartotojo įkeltų paveikslėlių tekstai, reklamų tekstai. Papildomai tikrinant ir programos išeities kodą, būtų galima atskirti ar aptiktas tekstas yra aktualus testavimui ir nereikalingą tekstą nufiltruoti.

Šis pakeitimas turėtų gerokai pagerinti neišversto teksto defekto aptikimo metodo veikimą. Taip pat šį metodą galima pagerinti sudarius kalbų pavadinimų žodyną ir nufiltruojant jame esančius žodžius – taip būtų išvengta defekto aptikimo kalbų pasirinkimo lange. Išeities kodo tikrinimas padėtų pagerinti ir nukirpto teksto defekto aptikimo veikimą – dabar netikrinama ar daugtaškis panaudotas teksto nukirpimui, ar kaip skyrybos ženklas, o turint prieigą prie išeities kodo, būtų galima atlikti tokį patikrinimą.

Rašybos klaidų defekto aptikimą galima būtų patobulinti pakeitus arba papildomai panaudojus dar vieną tikrinimo biblioteką, nes dabartinė veikia ne itin gerai – neatpažįsta mobiliosiose programėlėse dažnai vartojamų techninių sąvokų ir jas žymi kaip klaidą.

Neįskaitomo teksto defekto aptikimo metodo patobulinimui reikėtų atlikti bandymus su keletu mobiliųjų įrenginių, turinčių skirtingus ekrano įstrižainės pločius bei skirtingas rezoliucijas ir patyrinėti nuo kokios ribos tekstas gali būti laikomas neįskaitomas ir ar ta riba galioja visų pločių ekranams ir visoms ekrano rezoliucijoms. Jei riba yra kintanti, tada įrankį būtų galima papildyti pridedant šios ribos nustatymo funkcija. Taip būtų galima nustatyti ribą, kuri būtų tinkama tokiam mobiliajam įrenginiui, kurio nuotraukos yra testuojamos.

Dabartinė blogai atvaizduojamo teksto defekto metodo idėja yra netinkama ir veikia blogai. Kita idėja, kurią būtų galima išbandyti – panaudoti mašininį mokymąsi ir sukurti tinklą, skirtą blogai atvaizduojamų simbolių nuotraukoje aptikimui.

Per ilgo teksto defekto aptikimą galima pagerinti šiek tiek patobulinus dabartinį metodą: reikėtų susumuoti visų teksto blokų užimamą aukštį nuotraukoje, nes dabartinis metodas ieško vieno teksto bloko, kuris užimtų didelę dalį ekrano, tačiau jei ekranas yra užpildytas tekstu kuris suskirstytas į atskirus teksto blokus – defektas yra neaptinkamas.

Vienas iš būdų patobulinti įžeidžiančio teksto defekto aptikimą yra sudaryti žodyną su keiksmažodžiais ar kitais įžeidžiančiais ar nekorektiškais žodžiais ir jį panaudoti teksto tikrinime.

Išvados

1. Siūlomų tekstinių defektų aptikimo metodų analizės metu pastebėta, kad dalis defektų ir jų aptikimo metodų yra panašūs tarpusavyje, todėl padidėja rizika aptiktus defektus klasifikuoti klaidingai.
2. Rinkos tyrimo metu pastebėta, kad rinkoje nėra įrankio, gebančio automatiškai aptikti tekstinius defektus mobiliųjų programėlių vartotojo sąsajose.
3. Sukurtas įrankis, kuris aptinka tekstinius defektus mobiliosios programėlėse analizuodamas programėlės paveikslėlius ir septyni tekstinių defektų aptikimo algoritmai.
4. Atlikus tyrimą pastebėta, kad įrankis dažnai defektus aptinka klaidingai, tačiau klaidingo neaptikimo atvejų yra pakankamai nedaug: rašybos klaidų, neišversto ir neįskaitomo teksto defekto atvejais klaidingai neaptikta iki 27 % defektų.
5. Išanalizavus eksperimento rezultatus buvo pasiūlytos idėjos sukurtų metodų veikimo tobulinimui.

Literatūros sąrašas

1. 1. *Software testing research: Achievements, challenges, dreams.* **Bertolino, Antonia.** 2007 m., 2007 Future of Software Engineering, p. 85-103.
2. 2. *Testing: a roadmap.* **Harrold, Mary Jean.** 2000 m., Proceedings of the Conference on the Future of Software Engineering, p. 61-72.
3. 3. *Automated software testing as a service.* **Candea, George, Stefan Bucur, and Cristian Zamfir.** 2010 m., Proceedings of the 1st ACM symposium on Cloud computing, p. 155-160.
4. 4. *Mobile application testing and challenges.* **Nimbalkar, Ravi Ramchandra.** 2013 m., International Journal of Science and Research (IJSR), India, T. 2.
5. 5. *Mobile application testing—Challenges and solution approach through automation.* **Kirubakaran, B and Karthikeyani, V.** 2013 m., 2013 International Conference on Pattern Recognition, Informatics and Mobile Engineering, p. 79-84.
6. 6. *Mobile application testing: a tutorial.* **Gao, Jerry and Bai, Xiaoying and Tsai, Wei-Tek and Uehara, Tadahiro.** 2, 2014 m., Computer, T. 47, p. 46-55.
7. 7. *GUI testing: Pitfalls and process.* **Memon, Atif M.** 8, 2002 m., Computer, p. 87-88.
8. 8. *GUI testing using computer vision.* **Chang, Tsung-Hsiang and Yeh, Tom and Miller, Robert C.** 2010 m., Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, p. 1535-1544.
9. 9. *A gui crawling-based technique for android mobile application testing.* **Amalfitano, Domenico and Fasolino, Anna Rita and Tramontana, Porfirio.** 2011 m., 2011 IEEE fourth international conference on software testing, verification and validation workshops, p. 252-261.
10. 10. *Using GUI ripping for automated testing of Android applications.* **Amalfitano, Domenico and Fasolino, Anna Rita and Tramontana, Porfirio and De Carmine, Salvatore and Memon, Atif M.** 2012 m., Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, p. 258-261.
11. 11. *Automating GUI testing for Android applications.* **Hu, Cuixiong and Neamtiu, Iulian.** 2011 m., Proceedings of the 6th International Workshop on Automation of Software Test, p. 77-83.
12. 12. *The testing method based on image analysis for automated detection of UI defects intended for mobile applications.* **Packevičius, Šarūnas and Ušaniov, Andrej and Stanskis, Šarūnas and Bareiša, Eduardas.** 2015 m., International Conference on Information and Software Technologies, p. 560-576.
13. 13. *UI X-Ray: Interactive Mobile UI Testing Based on Computer Vision.* **Chen, Chun-Fu Richard and Pistoia, Marco and Shi, Conglei and Girolami, Paolo and Ligman, Joseph W and Wang, Yong.** 2017 m., Proceedings of the 22nd International Conference on Intelligent User Interfaces, p. 245-255.
14. 14. *Text Semantics and Layout Defects Detection in Android Apps Using Dynamic Execution and Screenshot Analysis.* **Packevičius, Šarūnas and Barisas, Dominykas and Ušaniov, Andrej and Guogis, Evaldas and Bareiša, Eduardas.** 2018 m., International Conference on Information and Software Technologies, p. 279-292.
15. 15. **Efficient Testing Android app – Tools - Android development.** ***AlexZHDev.*** **[Tinkle] [Cituota: 2019 m. 11 12 d.] https://alexzh.com/2018/12/10/efficient-testing-android-app-tools/.**

16. **16. UI Automator | Android Developers.** ***Android Developers.*** **[Tinkle] [Cituota: 2019 m. 11 12 d.] https://developer.android.com/training/testing/ui-automator.**
17. **17. Firebase Test Lab | Firebase.** ***Firebase.*** **[Tinkle] [Cituota: 2019 m. 11 12 d.] https://firebase.google.com/docs/test-lab.**
18. **18. honeynet/droidbot: A lightweight test input generator for Android. Similar to Monkey, but with more intelligence and cool features!** ***Github.*** **[Tinkle] [Cituota: 2019 m. 11 12 d.] https://github.com/honeynet/droidbot.**
19. **19. Vision AI | Derive Image Insights via ML | Cloud Vision API.** ***Google Cloud.*** **[Tinkle] [Cituota: 2021 m. 03 14 d.] https://cloud.google.com/vision.**
20. **20.** ***Automated Visual Testing of Application User Interfaces Using Static Analysis of Screenshots.*** **Packevičius, Šarūnas and Rudžionienė Greta and Bareiša Eduardas. 2020 m.**