LATVIJAS UNIVERSITĀTE

DATORIKAS FAKULTĀTE

NEJAUŠDATU TESTĒŠANA TĪMEKĻA LIETOTŅU SASKARNĒM

BAKALAURA DARBS

Autors: **Patriks Misāns**

Studenta apliecības Nr.: pm17051

Darba vadītāja: Dr. sc. comp. Vineta Arnicāne

RĪGA 2021

ANOTĀCIJA

Visaptveroša testēšana ir kritisks solis programmatūras kvalitātes nodrošināšanā. Tīmekļa lietotnēs, līdzās vienības testiem, tiek izmantoti automatizēti un manuāli sistēmas testi. Šo testu rakstīšana un izpilde prasa nozīmīgus resursus. Nejaušdatu testēšana(*fuzz testing*) piedāvā risinājumus automatizētai testpiemēru ģenerācijai, bet šo metožu pielietojums tīmekļa saskarņu testēšanā ir maz pētīts.

Darbā tiek pētītas iespējas un grūtības nejaušdatu testēšanas metožu izmantošanā tīmekļa lietotņu nejaušdatu testēšanā. Ir piedāvātas pieejas un algoritmi nejaušināmas tīmekļa saskarnes reprezentācijas izgūšanai, testpiemēru ģenerācijai un rezultātu novērtēšanai. Tāpat tiek apspriesti šo metožu izmantošanas praktiskie apsvērumi.

Pētījuma rezultāts ir jauna tīmekļa nejaušdatu pētnieciskās testēšanas pieeja. Papildus izveidots apkopojums ar problēmām un potenciāliem risinājumiem nejaušdatu testēšanas izmantošanā.

Atslēgas vārdi: nejaušdatu testēšana, tīmekļa testēšana, grafu teorija

ABSTRACT

FUZZ TESTING WEB APPLICATION INTERFACES

Comprehensive testing is a critical step in ensuring software quality. Web applications unit tests alongside automatic and manual system tests. Writing and executing these tests take a considereable amount of resources. Fuzz testing provides a method for automaticaly generating test cases. The methods, however, are scarcely explored in the domain of web user interface testing.

This work explores the possibilites and difficuilties in applying fuzz testing methods for web interface testing. Approaches and algorithms for fuzzing web interfaces are provided. These include genereting a fuzzable representation of the interface, generating test cases and evaluating results.

The result is a novel approach for randomized exploratory web interface testing. In addition a gathering of problems and potential solutions for using fuzz testing in the domain of web interfaces is provided.

Key words: fuzz testing, web testing, graph theory

SATURA RĀDĪTĀJS

APZĪMĒJUMU SARAKSTS

Termins	Skaidrojums
Invariants	Vienmēr patiesa programmatūras īpašība.
Saskarnes stāvokļu grafs	Pilnīgi definēts grafs, kas apraksta saskarnē iespējamos stāvokļus un pārejas starp stāvokļiem.
Web	tīmekļa
Parsētājs	Programma, kas sadala ievadi atsevišķās daļās. Tipiski tālākai apstrādei, piemēram, kompilatorā.
Defekts	Nevēlama vai negaidīta sistēmas darbība.
Nejaušdati	Nejauši vai pseido-nejauši ģenerēti dati
CLI(*Command line interface*)	Komandrindas-veida saskarne.
GUI(*Graphical user interface*)	Saskarne, kas veidota no atsevišķiem grafiskiem elementiem
Robežnosacījums	Kritiskā vērtība, kas atdala divas datu klases
BFS(*Breadth-first-search*)	Grafa pārstaigāšanas metode
lēmumelements	Elements, pret kuru testa laikā tiek izpildīta darbība
URL(*Universal resource locator*)	Adrese, kas unikāli identificē resursu tīmeklī

IEVADS

Testēšana ir priekšnosacījums korektas programmatūras izstrādei. Efektīva testēšana notver kļūdas pirms tās kļūst par lietotājam redzamiem defektiem. Atrodot kļūdas agrāk programmatūras izstrādes dzīves ciklā, tiek samazināts nepieciešamais darbs kļūdas novēršanai[1]. Manuāla, cilvēku veikta, testēšana pieprasa daudz laika, un secīgi, līdzekļus. Tādēļ eksistē automatizētas testēšanas pieejas, kas aizvieto daļu manuālās testēšanas darbu ar automātiski izpildāmiem testēšanas skriptiem. Šādi skripti ir izmantojami gan funkcionālai, gan nefunkcionālai testēšanai. Tomēr šo skriptu manuāla rakstīšana un uzturēšana arī prasa lielu resursu ieguldījumu. Papildus, tas izvirza prasības pret testu rakstītāju prasmēm. Tādēļ populāri ir rīki, kas atklāj kļūdas un nefunkcionālas nepilnības bez pilnīgu testpiemēra soļu rakstīšanas. Piemēram, statiskās analīzes rīki kā *SonarQube* vai atmiņas analīzes rīki kā *Valgrind* dod pienesumu kvalitātes nodrošināšanā ar mazāku resursu ieguldījumu salīdzinājumā ar manuālo vai skriptēto testēšanu.

Šobrīd industrijā pētnieciskā testēšana, un bieži regresā testēšana tiek veikta manuāli. It īpaši nozīmīgs ir laiks, kas nepieciešams regresijas testēšanai. Laiks, kas nepieciešams pilnīgai regresijas testēšanai aug kopā ar saskarnes sarežģītību. Pēc autora uzskatiem daļu šī darba ir iespējams aizvietot ar rīkiem, kas izmanto nejaušdatu testēšanas metodes.

Nejaušdatu testēšana ir automatizētās testēšanas pieeja, kas izmanto nejauši vai pseido-nejauši ģenerētus datus kā ievadi programmatūrai. Šīs metodes darbina programmatūru ar šiem nejaušinātajiem datiem un salīdzina programmatūras rezultātus pret lietotāja vai globāli definētiem invariantiem un drošības prasībām. Šī metode ir sevi pierādījusi sistēmprogrammatūras testēšanas kontekstā[2].

Tomēr, saskarņu(*GUI*) nejaušdatu testēšana ir vēl attīstības stadijā. Interneta saskarņu kontekstā sevi ir pierādījušas automatizētās testēšanas pieejas, kas simulē lietotāja darbības. Tomēr šīs pieejas paļaujas uz lietotāja rakstītiem skriptiem ar ierobežotu lokālu nejaušināšanu. Piemēram, skriptā ievadlaukam norāda vērtību diapazonu, no kura izpildes laikā tiek izvēlēta vērtība. Tā ir laba prakse, bet lai aizvietotu daļu pētnieciskās un regresās testēšanas bez skriptu rakstīšanas, ir nepieciešams nejaušināti ģenerēt lietotāja darbības kā klikšķus.

Šajā bakalaura darbā tiks apskatītas vairākas eksistējošās nejaušdatu testēšanas pieejas, to potenciālo pielietojumu tīmekļa saskarņu nejaušdatu testēšanai, apskatītas potenciālās problēmas, kā arī piedāvāti risinājumi un algoritmi centrālo funkciju realizācijai. Darba mērķis ir kalpot kā teorētiska bāze tālāku, praktisku rīku izstrādei.

1. NEJAUŠDATU TESTĒŠANA(*FUZZ TESTING*)

1.1.Nejaušdatu testēšanas ideja

Nejaušdatu testēšana ir automātiskās testēšanas metode, kur testējamā sistēma tiek pakļauta nejaušām, nekorektām vai pseido-nejaušām ievadēm. Nejaušdatu testēšanu var veikt gan pēc melnās, gan baltās kastes testēšanas pieejas. Baltās kastes testēšanas gadījumā sistēmas struktūra tiek izmantota lai ģenerētu ievades, kas darbina lielāku testējamās sistēmas daļu. Piemēram, testējot kompilatoru tiek izmantoti programmēšanas valodas sintakses likumi lai ģenerētu ievades, kas ir sintaktiski korektas programmas. Tādejādi lielāka daļa testu tiek akceptēta no parsētāja(*parser*) puses. Testa rezultāts tiek noteikts pārbaudot sistēmu uz defektiem, izņēmumgadījumiem vai testētāju noteiktiem invariantiem. Piemēram, statusa koda.

Šīs testēšanas efektivitāte ir iekš tās augstās automatizācijas pakāpes un neatkarības no testa rakstītāja izdomas. Ģenerētie testa piemēri ļauj nosegt lielāku programmatūras daļu, un ir izpildāmi ātrāk par manuālajiem testiem. Šis ir īpaši efektīvi regresiju un robežnosacījumu(*corner-case*) kļūdu meklēšanā.

1.2.Strukturētie nejaušdati

Alternatīvi pilnīgi nejaušiem datiem, nejaušdatu testēšanā izmanto ar pseido-nejaušus datus, kas pakļaujas kādam protocolam, formātam vai citiem ierobežojumiem. Šie nosacījumi ir atkarīgi no testējamās programmatūras konteksta. Mērķis ir veidot nejaušdatus, kas ir pietiekami “korekti” lai darbinātu testējamos koda ceļus, bet pietiekami “nejauši” lai noklātu pēc iespējas lielāku testējamo koda daļu.

2. SASKARŅU NEJAUŠDATU TESTĒŠANA

Saskarņu nejaušdatu testēšana kā ideja jau eksistē. Pamatideja par nejaušinātiem datiem un salīdzināšanu pret rezultātu invariantiem ir paturēta no klasiskās nejaušdatu testēšanas. Lielākais jauninājums ir jauna koncepcija par to, kas ir uzskatāms par sistēmas ievaddatiem. Tradicionālā nejaušdatu testēšana tiek veikta vienības vai integrācijas testu līmenī. Tie ir programmātiski definēti pret vienību ar skaidru ievades/izvades interfeisu. Piemēram, *Linux* kerneļa utilītes tika testētas kā atsevišķas *komandrindas(command line interface, tālāk CLI)* programmas. *CLI* utilītes parasti aprobežojas ar vienkāršu pieprasījums-atbilde modeli. Pilni ievaddati ir specificēti pieprasījumā, un sistēma pēc darbības izpildes atgriež pilnu atbildi. Grafisko(*Graphical user interface, tālāk GUI*) programmu paradigmā darbības tipiski notiek kā soļu kopums. Šis fakts sarežģī nejaušdatu testēšanu gan ievaddatu specificēšanā, gan rezultātu nolasīšanā.

Tas nozīmē, ka *GUI* nejaušdatu testēšanā ir nepieciešams ģenerēt testpiemērus, kas spēj ne tikai aizpildīt ievadlaukus, bet arī darināt grafiskos elementus[4].

3. TĪMEKĻA SASKARŅU NEJAUŠDATU TESTĒŠANA

Līdzīgi *desktop GUI* saskarņu nejaušdatu testēšanai, *tīmekļa GUI* saskarņu nejaušdatu testēšanai jāizmanto konkrētās sfēras īpašības. Šajā sadaļā ir piedāvāta viena saskarnes reprezentēšanas metode, metodes nejaušo pārstaigāšanu ģenerēšanai un rezultātu novērtēšanai. Konceptu ilustrācijai tiek izmantots vienkāršotas *tīmekļa* saskarnes piemērs. Kritiskajiem algoritmiem ir piedāvāts viens realizācijas pseidokods.

Augstā līmenī piedāvātais pētnieciskās testēšanas risinājums sastāv no četrām fāzēm:

1. Saskarnes grafa konstruēšana fāze – no dokumenta objektmodeļa(*Domain object model, tālāk DOM*) tiek izveidots stāvokļu grafs, kas tiks izmantots testpiemēru ģenerēšanā.
2. Testpiemēru ģenerēšanas fāze – Izmantojot konstruēto saskarnes grafu tiek ģenerēti konkrētu testpiemēru soļi.
3. Testpiemēru darbināšanas fāze – Ģenerētie testpiemēri tiek automātiski darbināti.
4. Novērtēšanas fāze – darbināšanas rezultāti tiek novērtēti un apkopoti priekš gala lietotāja.

3.1. Tīmekļa saskarnes reprezentācija

Tīmekļa saskarnes veidojas pārlūkprogrammai attēlojot dokumenta objektmodeli (*Domain object model*), kas savukārt tiek veidots no *HTML* dokumenta satura mijiedarbībā ar iegultajiem skriptiem. Viena no saskarnes reprezentācijas iespējām ir skatīties uz dinamiskajiem objektmodeļa stāvokļiem kā grafu. Mezgli ir saskarnes stāvokļi(Piem. "lauki nav aizpildīti"). Šķautnes ir iespējamās darbības, ko varam veikt ar elementu(Piem. *click* notikums(*event*) uz pogas elementa). Darbības ar sistēmu maina sistēmas stāvokli. Pie šādas pieejas nejaušdatu ievaddatu ģenerēšana nozīmē dažādu grafa pārstaigāšanu ģenerēšanu.

Definēsim "stāvokli" kā visu objektmodelī pieejamo elementu stāvokļu apvienojumu. Elementa stāvokli definēsim kā elementa vērtību(skat. tālāko sadaļu par ekvivalences klasēm), tekstu un redzamību. Teorētiski elementa aspektus, ko iekļaut var ņemt patvaļīgi. Tomēr jāņem vērā, ka nederministisku aspektu iekļaušana ar tālāk aprakstīto algoritmu var ģenerēt bezgalīgu grafu. Papildus, svarīgs ir granularitātes līmenis. Par elementu var uzskatīt sarakstu vai katru individuālo teksta lauku sarakstā. Rodas nepieciešamība pēc iespējas lietotājam definēt šo granularitātes līmeni speciāliem elementiem.

Potenciāla problēma šai pieejai ir dokumenta objektmodeļa mainība. Interaktīvās tīmekļa lietotnēs saskarnes elementi bieži tiek dinamiski pievienoti caur *javascript* atbildot lietotāja darbībām ar saskarni. Tādēļ naivā pieeja nespētu ģenerēt nejaušdatus pilnai lietotnes uzvedības telpai. Piedāvātais risinājums ir dinamiski ģenerēt pilno uzvedības grafu caur saskarnes izlūkojošo darbināšanu – veikt nejaušas darbības ar sākotnēji ģenerēto grafu, un pievienot iepriekš neredzētos saskarnes elementus grafam.

Ir vairāki veidi kā mijiedarboties ar tīmekļa saskarni. Ir iespējams veidot HTTP pieprasījumus[4], bet šādu testi nav piemēroti dinamiskām, uz *javascript* bāzētām saskarnēm. Šajā darbā tiek izmantots uz lietotāja darbībām bāzēts modelis. Darbības, ko veiks sistēma, ir darbības ko var veikt lietotājs manuāli. Tā pati pieeja ir izmantota gala testu darbināšanai – testēšanas ietvars simulē lietotāja darbības ar saskarni.

Ne visi elementam piesaistītie notikumi ir daudzsološi. Ģenerētā grafa izmēru var samazināt caur lietotāja noteiktiem ierobežojumiem. Piem. tikai *click* notikumi uz pogām.

3.1.1. Tīmekļa saskarnes reprezentācijas izgūšana no DOM.

DOM ir pilns, pārlūkprogrammas veidots, saskarnes stāvoklis konkrētajā brīdī[5]. Tas ir brīvi pieejams pārlūkprogrammā caur *javascript*. Ir 2 iemesli kāpēc šī reprezentācija ir jāapstrādā pirms tālākas testpiemēru ģenerācijas:

1. Objektmodelis ir mainīgs. Mēs vēlamies ģenerēt piemērus, kas testē visu saskarnes plūsmu, nevis vienu soli.
2. Objektmodelis satur ļoti daudz informācijas, kas padara to nepiemērotu stāvokļa definēšanai.(Skatīt sadaļu "Stāvokļa granularitāte")

3.1.1.1. Stāvokļu grafa izgūšanas algoritms.

Kā iepriekš minēts, sākotnējais lapas stāvoklis var neietvert visu saskarnes mainību, jo parasti tīmekļa saskarnes ir realizētas vairākos soļos. Faktiski pēc lapas ielādes pieejams ir sākotnējais stāvoklis – pirmā grafa virsotne. Pārējās virsotnes un šķautnes starp tām ir iespējams iegūt veicot grafa pārstaigāšanu. Zemāk aprakstīts vienkāršs algoritms pilnā stāvokļu grafa izveidošanai.

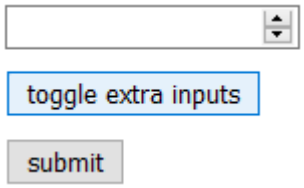
Pseidokods:

```
StateDict = {}

TransitionList = []

CurrState = DOM.getElements

StateDict[Hash(CurrState)] = (CurrState, exhausted=false, CurrState.Transitions)

While( StateDict not exhaused and TransitionList < MAX_SIZE) {

        Perform a transition from StateDict # spert soli

        NewState = Dom.getElements

        TransitionList.add((CurrState, NewState, action)) # pievienot pāreju

        StateDict.remove(action)

}
```

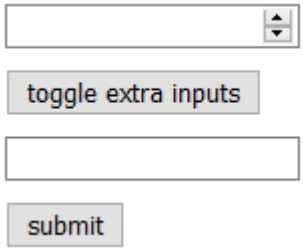
Algoritma pamatideja ir bezgalīga BFS(*breadth first search*) veikšana līdz ir pabeigta pilna pārstaigāšana vai sasniegts grafa izmēra ierobežojums. Rezultātā ir iegūta vārdnīca ar grafa virsotnēm, un saraksts ar šķautnēm. Rezultātā ir iegūta viena no grafa reprezentācijām.

3.1.1.2. Stāvokļu grafa izgūšanas algoritma darbināšanas piemērs.

Ilustrēšanai algoritms ir pielietots ļoti vienkāršai saskarnei. Saskarnē sākotnēji ir redzams 1 skaitlisks ievadlauks un 2 pogas. Viena poga iesūta rezultātu, bet otra parāda vēl vienu, sākotnēji paslēptu, tekstuālu ievadlauku.

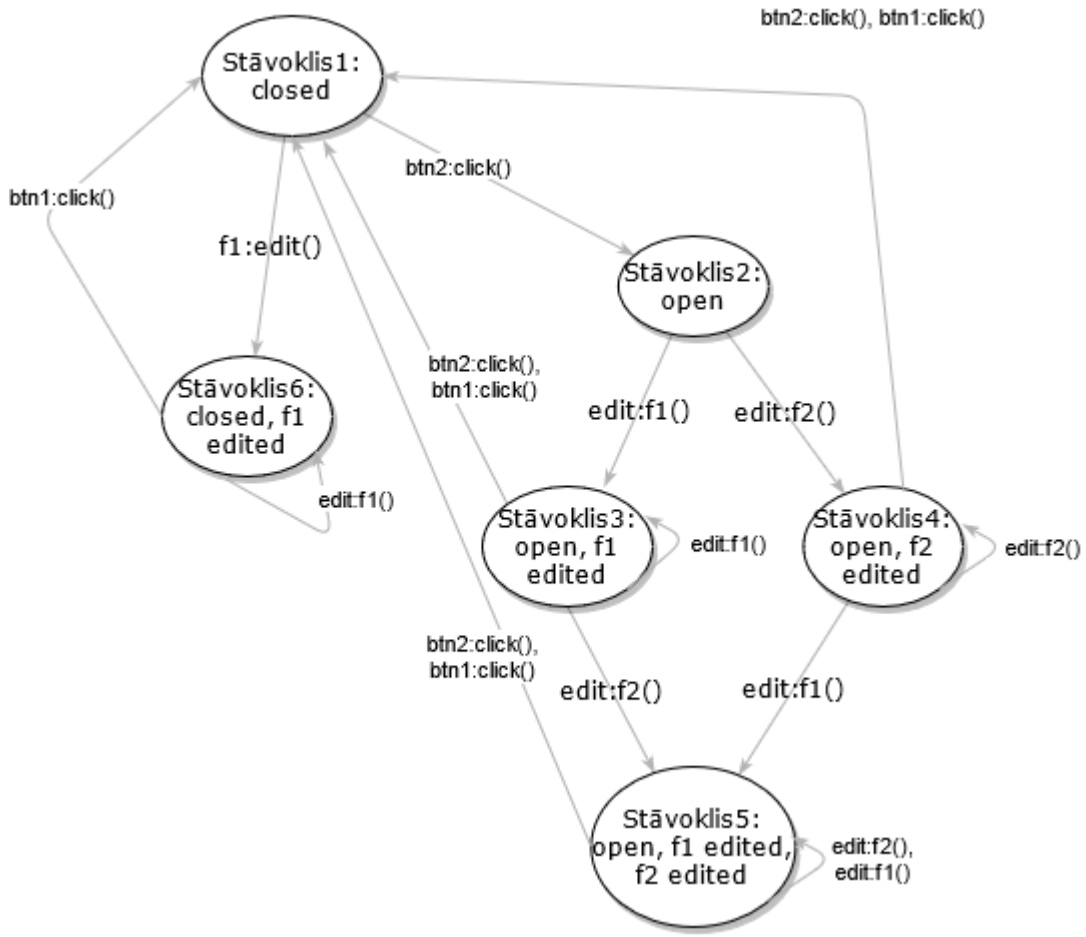


1. att. tīmekļa saskarnes 1. stāvoklis



2. att tīmekļa saskarnes 2.stāvoklis

Rezultējošais grafs notver šo mainību. Svarīgs aspekts ir reprezentētā plūsma. Ne visi stāvokļi ir sasniedzami no pārējiem stāvokļiem, bet visi stāvokļi ir sasniedzami sekojot pārejām no sākotnējā stāvokļa.



3. att. tīmekļa saskarnes stāvokļu grafa vizualizācija

3.1.1.3. Semantiskā HTML nozīme.

Augstāk dotajā piemērā kā elementi tika ņemti HTML ievades(*input*) elementi. Praksē, semantiski nozīmīgu elementu(kā *input* vai *button*) vietā reizēm tiek izmantoti semantiski neitrāli elementi(kā *div* vai *span*), kas savas funkcijas veic caur speciāli uzdevumam veidotiem apdarinātājiem(*handlers*). Praktiskā implementācijā šis aspekts būtu jāņem vērā. Semantiskais HTML nosaka kādus notikumus izmantot grafa pārstaigāšanai - ir skaidrs, ka semantiskiem pogas elementiem ir vērts sekot *click* notikumiem. Bez semantiskiem elementiem sekojamo elementu

izvēle ir sarežģītāka. Sekojot visiem iespējamiem notikumiem tiks ģenerēts pārlieku liels saskarnes grafs. Nesekojot nesemantisku elementu notikumiem daļa saskarnes mainības netiek pārbaudīta. Potenciāls risinājums ir pašu notikumu apdarinātāju(*handlers*) analīze. Mainīti apdarinātāji ir norāde uz notikumiem ko ir vērts izsekot.

3.1.2. Stāvokļu grafa granularitāte.

Svarīgs apsvērums, ir nodefinēt, ko domājam ar vārdu "stāvoklis". Ja pieejam naivi un uzskatām jebkādu ievadi kā stāvokli, tad viena vienīga skaitliska vērtība var saturēt daudzus miljonus stāvokļu(HTML *input* elementa specifikācija nenosaka maksimumu[3]). Tas, protams, nav praktiski izmantojams. Otra galējība ir ierobežot elementa veidotos stāvokļus uz tikai diviem – tukšs un aizpildīts. Šāda pieeja ir praktiski pielietojama, bet atstāj lielu iespēju telpu neapskatītu. Tātad eksistē balanss starp grafa izmēru un noklāto funkcionalitāti.

Pēc autora domām, visdaudzsološākā pieeja ir izmantot datu ekvivalences klases ar iespēju pievienot lietotāja-definētas klases.

3.1.3. HTML elementu ekvivalences klases.

Ekvivalences klases sadala ievades datus semantiski atšķirīgās daļās. Piemēram, skaitlisku ievadlauku var sadalīt negatīvos un nenegatīvos skaitļos vai e-pasta lauka ievades var sadalīt sintaktiski korektos un nekorektos e-pastos. Tad testēšana notiek ar ievaddatiem, kas reprezentē katru ievaddatu klasi. Šīs klases var būt ierobežoti noteiktas no pašas tīmekļa lapas satura(ir iespējams balstīties uz DOM elementu ierobežojumiem, bet server-puses loģikas analīze ir ārpus darba vēriena). *Number* input laukus var sadalīt skaitliskos diapazonos, *text input* laukus var sadalīt pēc simbolu tipa un skaita. Ja tiek izmantoti HTML *input* lauku ierobežojumi arī ir nolasāmi no *DOM*. Šie ierobežojumi arī ir ekvivalences klašu šķirtne – *maxlength="10"* nozīmē, ka 10 simboli ir laba robežšķirtne šī lauka ekvivalences klasēm.

Šādi izgūtas klases var tikt izmantotas gan grafa ģenerēšanas solī(šķirtnes ir vērtības stāvokļi - visu skaitļu vietā n klases), gan testpiemēru ģenerēšanā(ģenerējam piemērus, kas izmanto vērtības no dažādām ekvivalences klasēm).

Daļa ekvivalenču klašu nav pieejama iekš *DOM*. Server-puses loģika ir pilnīgi nepieejama, un *javascript* analīze ir sarežģīts uzdevums. Šīs problēmas risinājums ir lietotāja-definētas ekvivalences klases. Lietotājam ir zināma testējamās sistēmas biznesa loģika.

3.2.Nejaušdatu ģenerācija reprezentētajai saskarnei

Nejaušdatu ģenerācijas mērķis ir ģenerēt testpiemēra soļus, kas maksimizē varbūtību atklāt defektus(vēlams defektus, kas ietekmētu īstu lietotāju darbību.) testējamajā programmatūrā. Ir vairāki aspekti, kas ietekmē šo varbūtību.

1. Līdzība ar īstu lietotāju darbībām.
2. Pēc iespējas lielākas sistēmas daļas darbināšana.

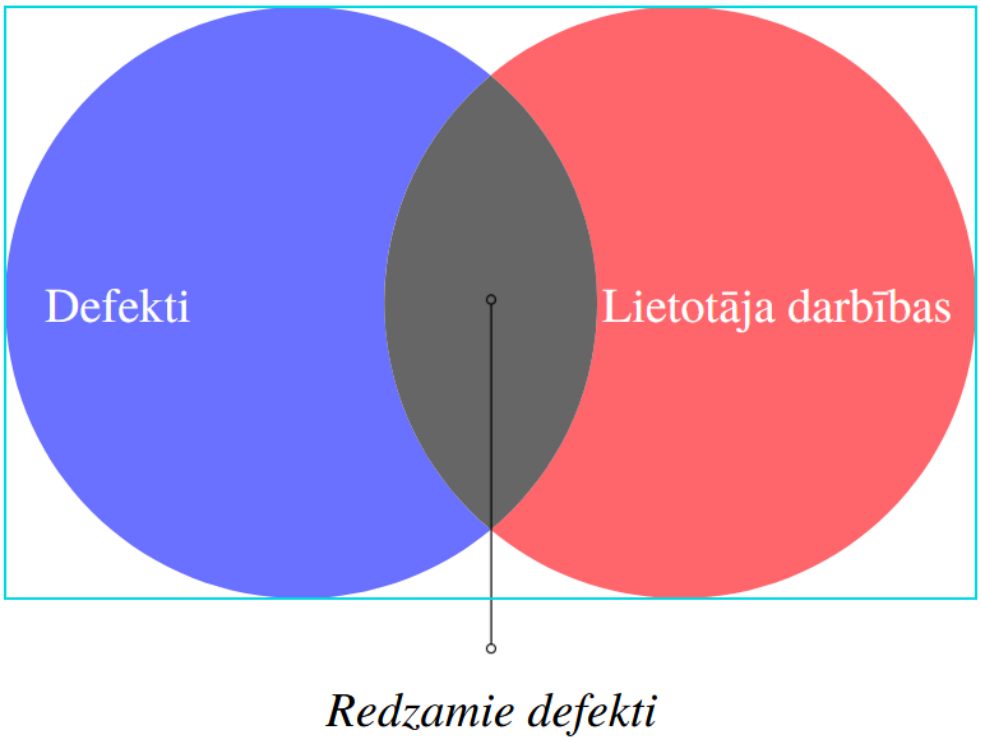
Zemāk detalizētāk aprakstīti apsvērumi testpiemēra soļu ģenerēšanai. Šajā kontekstā testpiemēra soļi ir secīgas darbības ar saskarnes grafā reprezentētajiem elementiem(klikšķi, dati/to ievade, u.c.).

3.2.1. Līdzība ar īstu lietotāju darbībām.

Ir aspekti kā laika nobīde starp darbībām, ievades precizitāte, u.c., kas atšķir automatizēto testu mijiedarbību ar sistēmu no dabiska lietotāja mijiedarbības. Ja šos apsvērumus neņem vērā, tiks ģenerēti daudz testpiemēri, kas atklāj kļūdas ko lietotāji nespētu replicēt. Savukārt kļūdas, kas ir manāmas tikai pie dabiskām laika nobīdēm netiks notvertas.

Vēl svarīgs aspekts ir testpiemēru mijiedarbības modelis ar sistēmu. Testpiemēriem ir vairāk kontroles pār testējamo sistēmu nekā dabiskam lietotājam. Piemēram, testēšanas ietvaram ir piekļuve pilnam lapas objektmodelim. Savukārt dabisks lietotājs(neizmantojot izstrādes rīkus) var veikt ievades un izsaukt notikumus tikai caur saskarni.

Eksistē rīki kas ieraksta un atkārto lietotāja ievades. Šajos ierakstos saglabājas arī laika nobīdes. Potenciāli tas ir risinājums iegūt laika nobīdes kas ir dabiskas konkrētajai sistēmai.



4. att. attiecība starp defektiem un lietotāja darbībām

3.2.2. Pēc iespējas lielākas sistēmas daļas darbināšana.

Pirmkārt, testa piemēriem jābūt *pietiekami korektiem* lai tiktu darbinātas daļas aiz validācijas likumiem. Tas ir analoģiski kompilatoru testēšanai – ar pilnīgi nejaušiem datiem minimāla daļa testpiemēru darbina sistēmas daļas aiz parsētāja. Ar sintaktiski korektiem datiem tiek darbināta pilnīgāka sistēmas daļa. Papildus, tipiska saskarne satur vairākus ievadlaukus, katru ar saviem pieņemamiem datiem. Tāpēc varbūtība akceptēt ievadi ir kopējā varbūtība uzģenerēt pieņemamu ievadi katram laukam. P(n1)*P(n2)*…*P(nx). Pieaugot ievades lauku skaitam, akūti sarūk iespēja uzģenerēt pieņemamu ievadi.

Otrkārt, datiem jābūt pietiekami mainīgiem. Loģiski, ka dažādi ievaddati izmanto dažādus programmas izpildes ceļus. Datiem jābūt pietiekami mainīgiem lai noklātu programmas semantiskās datu klases.

3.2.3. Testpiemēros izmantoto datu ģenerēšana

Kļūdas bieži notiek pēc konkrētu datu ievades. Tāpēc efektīva ievadlaukos izmantoto datu ģenerēšana ir kritiska. Zemāk aprakstītas vairākas pieejas, to plusi un mīnusi.

3.2.3.1. Pilnīgi nejauši dati

Naivais risinājums ir padot pilnībā nejaušinātus datus. Pluss ir šīs pieejas vienkāršums. Koncepts ir viegla saprotams, un realizācija triviāla.

Tomēr ir vairāki apsvērumi, kas padara šo pieeju nepraktisku tīmekļa saskarņu testēšanai.

Pirmkārt, rodas nepieciešamība pēc ļoti liela testpiemēru skaita, jo dominējošā testpiemēru daļa nepakļaujas validācijas likumiem.

Otrkārt, jāņem vērā stāvokļu grafa granularitātes apsvērums(Skat. *Stāvokļu grafa granularitāte*). Pilnīgi nejaušu datu izmantošana bez pieļāvumiem noved pie nepraktiska izmēra stāvokļu grafa.

Treškārt, ģenerētie dati būs ļoti atšķirīgi no datiem, kas sagaidāmi no dabiska lietotāja.

3.2.3.2. Nejaušināti, bet strukturēti dati

Klasiskajā nejaušdatu testēšanā populāra pieeja ir ģenerēt datus, kas pakļaujas kādai noteiktai struktūrai. Kompilatoru testam ģenerē ievades, kas pakļaujas programmēšanas valodas sintaksei. Šī pieeja ir izmantojama vairākiem datu ievadlauku likumiem. E-pastu, personas kodu, telefona nr, u.c. laukiem ir konkrētas struktūras. Šīs struktūras var notvert ar regulārām izteiksmēm(*regular expressions*) vai datu-specifiskiem datu ģenerēšanas algoritmiem.

Šīs pieeja samazina, bet ne pilnībā atrisina vairākas pilnīgas nejaušdatu testēšanas problēmas. Ģenerētie testpiemēri darbinās lielāku testējamās sistēmas daļu(bet jāņem vērā, ka nekorekto ievažu apgabals paliek netestēts). Tāpat jāveic pieļāvumi stāvokļu granularitātes nodrošināšanai. Ģenerētie dati pakļaujas dabiska lietotāja ievažu sintaksei, bet ne semantikai.

3.2.3.3. Ekvivalences klases

Ekvivalences klases ļauj sadalīt ievadi semantiski nošķirtos apgabalos. Pēc sadalīšanas testpiemēru ģenerēšana nozīmē nejaušu datu izvēli no atsevišķajām ekvivalences klasēm.

Šī pieeja labi risina problēmas ar testpiemēru skaitu, granularitāti un līdzību dabiskai ievadei.

Mīnuss ir papildus darbs, kas jāveic definējot ekvivalences klases. Diemžēl ar objektmodelī pieejamo informāciju nepietiek lai automātiski definētu visas ekvivalences klases(skat. *HTML elementu ekvivalences klases*). Tas nozīmē, ka gala lietotājam jāveic papildus darbs definējot un uzturot sistēmai svarīgās ekvivalences klases.

3.2.3.4. Vērtību saraksti

Vienkāršs, bet darbietilpīgs risinājums ir manuāli definēt vērtību sarakstus. Pozitīvs ir vienkāršums un paplašināmība. Izmantojot semantiski nošķirtus datus var iegūt ekvivalenču klašu plusus bez nozīmīgi sarežģītākā klašu definēšanas darba.

Papildus manuālajam darbam ir mainības zudums. Salīdzinot ar ekvivalences klasēm, kuras klases ietvaros veic nejaušināšanu, konkrētie vērtību saraksti atstāj lielāku daļu testējamās sistēmas nenoklātu. Potenciāls risinājums mainības trūkumam ir nejaušināta saraksta vērtību mutēšana.

3.2.4. Darbību ģenerācija

Līdz šim ir apskatīti varianti ievadlauku datu ģenerēšanai. Pilniem testpiemēra soļiem ģenerētie dati ir jāapvieno ar stāvokļu grafa pārstaigāšanu. Definēsim "darbību" kā soli pa stāvokļu grafu. Tā var būt datu ievade vai notikuma(*event*) kā *click* darbināšana.

Darbību ģenerācijas mērķi ir identiski datu ģenerēšanai – veidot testpiemērus, kas maksimizē defektu atklāšanas varbūtību. Defekti var atklāties pēc vairāku darbību veikšanas, un dažādos stāvokļos. Kļūdas var rasties pēc darbību atkārtošanas, bet ir loģisks pieņēmums, ka defekta atklāšanas varbūtība no katras cikla atkārtošanas kļūst mazāka.

3.2.4.1. Darbību ģenerācijas algoritms

Pret darbību ģenerēšanas algoritmu tiek izvirzītas šādas prasības:

1. Ģenerētajām darbībām jābūt pēc iespējas atšķirīgām
2. Ģenerētajām darbībām jāapstaigā pēc iespējas lielāka grafa daļa
3. Ģenerētajām darbībām jāizmanto pēc iespējas vairāk pāreju
4. Ģenerētajām darbībām jāprioretizē jaunu stāvokļu apskate pāri ciklošanos
5. Jābūt patvaļīgam maksimālam darbību slieksnim
6. Jābūt patvaļīgam maksimālam testpiemēru slieksnim.

Piedāvātais algoritms izmanto alkatīgo paradigmu. Katrs ģenerētais testpiemērs izvēlēsies ceļu, kas maksimizē virsotnēm un šķautnēm piešķirto svaru. Pēc virsotnes vai šķautnes izvēles, tās vērtība tiks samazināta lai nodrošinātu darbību atšķirību.

3.2.4.1.1. Inicializācija

Katra šķautne tiek inicializēta ar vērtību *x,* katra virsotne tiek inicializēta ar vērtību *y.* Papildus, katrai virsotnei tiek dota vērtība *z* – no virsotnes sasniedzamo elementu skaits(mērķis ir prioretizēt virsotnes, kas "atver" ceļu pie vairāk virsotnēm).

Definēsim virsotnes "svaru"(tālāk, *weight*) kā sastāvdaļu svērto summu. weight = *0.33*x + 0.33*y + 0.33*z*. Mērķis ir neļaut inviduālai komponentei pilnībā nomākt pārējos aspektus. Katras šķautnes vērtība paliek *y.*

3.2.4.1.2. Iterācija

Vienā iterācijā tiek būvēts darbību(stāvoklisNo, stāvoklisUz, darbība) saraksts. Sākot no avota virsotnes, tiek veikta alkatīga pārstaigāšana ņemot darbību un solī sasniedzamo stāvokli ar maksimālo vērtību. Pēc soļa veikšanas izvēlētās virsotnes un šķautnes vērtība tiek samazināta par sabrukšanas īpatsvariem – *xdecay* un *ydecay*. *x = x*xdecay, xdecay<=1, y = y*ydecay, ydecay<=1*. Par "sabrukšanas īpatsvaru" definēsim vērtības daļu, kas tiek atņemta pēc pārstaigāšanas. Šis tiek atkārtots līdz sasniegts maksimālais soļu skaits. Rezultējošais darbību saraksts ir viena testpiemēra soļu saraksts. Mainītas tiek tikai virsotņu un šķautņu vērtības.

Testpiemēru konstruēšana tiks turpināta līdz sasniegts maksimālais testpiemēru skaits.

3.2.4.1.3. Pseidokods

```
#hyperparameters

INITIAL_NODEVAL = 100

INITIAL_VERTICEVAL = 10

NODE_REACHABLE_WEIGHT = 5

NODE_DECAY = 0.9

VETICE_DECAY = 0.9

… # maksimālo soļu, testpiemēru skaiti

# init

Paths = [] # testpiemēru soļu saraksts

Nodes = [List of nodes]

Vertices = [List vertices]

Head = Head node

Foreach vertice in Vertices

    Vertice.value = INITIAL_VERTICEVAL

Foreach node in Nodes

    Node.value = INITIAL_NODEVAL

    Node.reachableWeight = ReachableNodeCount(node) * NODE_REACHABLE_WEIGHT

    Node.weight = 0.33*Node.vertices.count*INITIAL_VERTICEVAL + 0.33* Node.value +
0.33 + Node.reachableWeight

# iteration

While Paths.count < MAX_PATHS_COUNT
```

```
Steps = [] # (startNode, endNode, takenVertice)
CurrentNode = Head
While Steps.count < MAX_STEPS_COUNT
        targetNode = max(CurrentNode.vertices.endNode.weight)
        targetVertice = max(Vertices to targetNode.value)
        targetNode.value = targetNode.value * NODE_DECAY
        targetVertice.value = targetVertice.value * VERTICE_DECAY
        Steps.add((CurrentNode, targetNode, targetVertice))
```

3.2.4.1.4. Piezīmes

1. Virsotņu svara vienādojums izmanto cieti iekodētas vērtības(0.33). Praktiskā algoritmā optimālie īpatsvari būtu jānosaka eksperimentāli
2. Dotās hiperparametru vērtības ir provizoriskas.

3.3. Testu darbināšana

Automatizētai testēšanai ir nepieciešams rīks, kas tos darbina – tas ir, izpilda testpiemērus, novērtē rezultātus un ģenerē to atskaites. Klasiskai nejaušdatu testēšanai tradicionāli tiek izmantoti testēšanas ietvari kā *xunit*. Nejaušināmo datu ģenerēšana iekļaujas kā viens no testa soļiem. Analoģiski, tīmekļa saskarņu testēšanai ir iespējams izmantot automatizētos testēšanas rīkus, kas testē lietotni caur pārlūkprogrammas darbināšanu – *Selenium u.c.* Stāvokļu grafa un testpiemēru ģenerēšana var integrēties eksistējošajā testu kopā kā pirms-testu uzstādīšanas soļi.

Tomēr ir vairāki sarežģījumi, kas praktiskā risinājumā ir jāņem vērā.

3.3.1. Grafa atjaunināšana

Sākotnējais grafa ģenerēšanas algoritms var nenotvert absolūti visu stāvokli ja ir nepilnīgi nodefinētas/netiek izmantotas datu ekvivalences klases.

Pirmkārt, nepilnīgi definētas datu ekvivalences klases nozīmē, ka reālā testā izmantotie dati dos savādāku rezultāta stāvokli nekā grafa ģenerēšanas solī izmantotie. Tas nozīmē, ka plānā definētie soļi ir novirzīti no plānotā, vai pat nav iespējami(Piemēram, ja tiek paslēpta plāna izmantota poga/ievadlauks). Šie gadījumi ir jānosaka salīdzinot eksistējošo stāvokli ar sagaidāmo, un jāizceļ testa rezultātos.

3.3.2. Lielais testpiemēru skaits/izpildes laiks

Teorētiskais ceļu skaits caur *GUI* saskarni aug ātri kopā ar elementu skaitu. Tas ir tāpēc, ka ceļs ir permutācija no saskarnes darbībām. Katrs jauns elements pareizina kopējo iespējamo pārstaigājumu skaitu. Savā ziņā šis skaits ir ģeometriska progresija. Naivā pieeja cenšoties pilnībā nosegt visus iespējamos ceļus rezultētos nepraktiski lielā testpiemēru skaitā. Darbā aprakstītās metodes – ekvivalences klases, alkatīgs testpiemēru ģenerēšanas algoritms palīdz samazināt testpiemēru skaitu, bet tāpat ir sagaidāms, ka saskarnēm ar lielu elementu skaitu testu izpilde prasīs nozīmīgu laiku.

Blakus citām darbā apskatītām metodēm, ir divas ar izpildi saistītas nianses. Pirmkārt, testu izpildes soli var paralelizēt(tīmekļa testēšanas ietvari kā *Selenium* jau paredz šādu iespēju). Otrkārt, atšķirībā no vienībtestiem, šī metode nav pielāgota izpilde-test(*execute-test*) izstrādes modelim. Tā nav pielāgota būt bloķējošam solim nepārtrauktās integrācijas(*continuous integration*) līnijā.

3.4. Testa rezultātu novērtēšana – orākulu problēma

Nejaušdatu testēšanā testa rezultāta novērtēšana ir daudz sarežģītāka nekā skriptētos automatizētos testos. Konkrētu rezultātu salīdzināšanas ar etalonrezultātiem vietā nejaušdatu testēšanas vietā tiek izmantotas metodes, kas salīdzina rezultātus ar vispārējiem sistēmas invariantiem(Piemēram, a+b = b+a vai rēķina summa = rēķina summa bez PVN + PVN), heiristikas ap sistēmas darbību(Piemēram, server-puses pieprasījumi neatgriež 5** HTTP kodus) vai bāzētas uz anomāliju analīzi(Piemēram, pieprasījumi, kas aizņēma 100x ilgāk par sistēmas vidējo).

Šī ir dūmu, sistēmas vai pat ātrdarbības testēšana atkarībā no izvēlētajiem novērtēšanas kritērijiem.

3.4.1. Lietotāja doti invarianti

Invariants ir predikāts, kam jābūt vienmēr patiesam testējamās sistēmas kontekstā. Ja invariants testa laikā tiek apgāzts, var uzskatīt ka tests ir atklājis defektu.

Tīmekļa lietotņu saskarnes kontekstā invarianti ir atkarīgi no testējamās sistēmas sfēras. Invarianti bankas sistēmai būs stipri atšķirīgi no internetveikala.

Lietotāja invarianti arī var būt elementu eksistences nosacījumi. Piemēram, *DOM* ir pieejams elements ar id = #home. Šāda veida invarianti iegūst savu efektivitāti notverot negaidītus rezultātus. Piemēram, programmas tehniskās kļūmes gadījumā, daudzas saskarnes atver kļūdas skatu, kurā lēmumelements nav pieejams. Alternatīva pieeja ir pārbaudīt uz kļūdas lapas, vai kļūdu vēstošu elementu eksistenci ar identifikatoriem vai regulārām izteiksmēm(*regular expression*).

3.4.2. HTTP heiristikas

HTTP statusa kodiem ir semantiska jēga.

1. 1xx kodiem ir informatīva nozīme.
2. 2xx kodi apzīmē izdevušos operāciju.
3. 3xx kodi apzīmē pāradresāciju vai nepieciešamību pēc papildus darbības.
4. 4xx kodi apzīmē klienta kļūdu.
5. 5xx kodi apzīmē servera kļūdu.

4xx un 5xx kodu klases ir interesantas no testa rezultātu novērtēšanas skatpunkta. 5xx var norādīt uz kļūdu pieprasījuma apstrādē, un tāpēc ir pamatojums testa apgāšanai. 4xx ir vairāk trokšņains indikators, jo tas var norādīt uz korektu nepareizas ievades apstrādi. Šīs heiristikas noderīgums ir atkarīgs no testējamās sistēmas specifikas.

3.4.3. Nefunkcionālās īpašības

Pastāv iespēja izmantot sistēmas nefunkcionālās īpašības testa rezultāta novērtēšanai. Piemēram, pieprasījumi, kas pārkāpj ātrdarbības budžetu ir pamats izgāzt testu.

3.4.4. Regresā testēšana pret ierakstu.

Regresie testi nozīmē rīcību atkārtošanau lai pārbaudītu, ka programmas uzvedība nav neparadzēti mainīta. Mēs varam izmantot šo iezīmi testa rezultātu novērtēšanai – saglabāt bijušu testpiemēru veiksmīgas izpildes rezultātus un salīdzināt tos ar jauno izpildi. Novirze nozīmē potenciālu atkāpi no sagaidāmās uzvedības.

Darbā piedāvātās sistēmas kontekstā tas nozīmē saglabāt izmantoto stāvokļu grafu kopā ar ģenerētajiem testpiemēriem un atkārtot to izpildi.

3.4.5. Rezultāta attēlapstrāde un klasifikācija.

Manuālā pētnieciskā testēšanā nepieminēts pārbaudes kritērijs ir vai sistēma *izskatās pareizi*. Kritēriji nav stingri noteikti, bet rezultāti, kas atšķiras no sagaidītā ir pazīme, ka sistēmā vai testētāja mentālajā sistēmas modelī ir kļūda.

Pastāv iespēja izmantot manuālās pētnieciskās testēšanas ierakstus kā datu kopu klasifikatora izstrādei. Nianse ir pielietojuma plašumā – ģeneralizēt uz visu tīmekļa lietotņu kopu nav praktiski iespējams to lielās mainības dēļ. Klasifikators būtu jātrenē uz konkrētās testējamās datiem.

3.4.6. Novirzes no stāvokļu grafa

Savā ziņā defekti ir negaidīta sistēmas darbība. Tātad nonākšana stāvokļu grafa ģenerēšanas solī neredzētā solī ir indikators par negaidītu testa rezultātu.

3.4.7. Kļūdaini pozitīvi novērtējumi

Vairāki darbā piedāvātie risinājumi ir heiristikas. Ir sagaidāms, ka būs kļūdaini pozitīvi(*false-positive*) rezultāti. Kopā ar lielo testpiemēru skaitu, šis var radīt nopietnu trokšņa daudzumu testa rezultātos.

Viens risinājums ir izmantot novērtēšanas metožu kombinēšanu. Ja P(A) raksturo varbūtību saņemt kļūdaini pozitīvu vērtējumu ar metodi A, tad apvienojot ar otru metodi B, varam samazināt kļūdaini pozitīva vērtējuma varbūtību uz P(A|B) + P(B|A). Tomēr jāņem vērā, ka šādi palielinot specificitāti(spēju atrast pareizi negatīvos novērtējumus) neizbēgami tiks samazināta testa sensitivitāte(pareizi pozitīvo novērtējumu īpatsvars).

Savā ziņā piedāvātās metodes testu rezultāti būs tuvāki statiskās analīzes rīku rezultātiem. Liels apjoms potenciālu problēmu, kuru gala novērtējumam ir nepieciešams lietotājs.

3.4.8. Citas pieejas

Specifiskākos gadījumos ir pielietojami citi kreatīvi risinājumi. Mēģinot pārbaudīt pārlūkporgrammu paritāti mēs varam saukt jebkādu rezultātu atšķirību par potenciālu kļūdu. Varbūt mēs nezinām kādam būtu jābūt rezultātam, bet mēs varam pateikt, ka tiem būtu jāsakrīt.

3.5. Praktiskie apsvērumi

Ir vairākas potenciālās problēmas un apsvērumi, kas jāņem vērā piedāvātās sistēmas implementācijā.

3.5.1. Autorizācija.

Vairākums tīmekļa lietotņu slēpj daļu funkcionalitātes aiz autorizācijas. Papildus, autorizācija arī ir daļā no lietotnes saskarnes. Ja autorizācija ir daļa no testējamās plūsmas, jālieto datu nejaušināšanas risinājums ar korektas autorizācijas datiem. Piemēram, definēt korektos autorizācijas datus kā ekvivalences klasi priekš konkrētajiem ievadlaukiem.

Papildus apsvērums ir izslēdzošie stāvokļi ko rada autorizācijas drošības apsvērumi. Testējot autorizāciju ir sagaidāmi vairāki neizdevušies autorizācijas pieprasījumi. Ir laba drošības prakse aizliegt autorizācijas pieprasījumus pēc vairākiem neizdevušies pieprasījumiem. Ja testpiemēri nav izolēti viens no otra, iepriekšējie testpiemēri var ietekmēt nākošo testu rezultātu. Pēc autora uzskatiem labākais risinājums ir atdalīt autorizācijas testēšanu no kopējās sistēmas testēšanas.

3.5.2. Testpiemēru neatkarība

Svarīga efektīvu automatizēto testu īpašība ir to neatkarība. Viena testa izpildei nevajadzētu ietekmēt pārējo testu izpildi. Naivā implementācijā piedāvātā algoritma testpiemēri nav pilnīgi neatkarīgi, jo tie ietekmē vienotu datu glabātuvi. Ir paredzams, ka testpiemērs A maina datus uz kuriem paļaujas testpiemērs B. Tas var vai nebūt pieņemami atkarībā no testējamās sistēmas. Ir iespējams nodrošināt neatkarību notīrot un uzstādot vidi pēc katra testpiemēra izpildes. Tomēr jāņem vērā, ka katra testpiemēra izpildes laiks tiks pareizināts ar vides atjaunošanai nepieciešamo laiku. Ja tiek pielietota šī metode, un izpildes laiks ir svarīgs, ieteicams to kombinēt ar testpiemēru izpildes paralelizāciju. (Skat. *Izpildes paralelizācija*)

3.5.3. Testējamās telpas ierobežošana.

Darbā piedāvātais algoritms seko visām saskarnē atrodamām hiperssaitēm neatkarīgi no galamērķa *URL*. Ja sistēmā eksistē hipersaites uz ārējiem resursiem, šis rada situāciju, ka tiek ģenerēti testpiemēri priekš ārējām sistēmām, kas ir ārpus testējamās sfēras.

Risinājums ir definēt atļauto *URL* prefiksu sarakstu.

3.5.4. Potenciāls datu bojāšanai.

Darbā piedāvātā metode ir destruktīva. Ja caur saskarni ir iespējams labot/dzēst datus, vai veikt citas nevēlamas darbības, metode to darīs, un darīs to neparedzamos veidos. Tādēļ šis rīks ir izmantojams tikai izstrādes un testēšanas vidēs vai produktos kur paliekošas izmaiņas nav problemātiskas. Tas norāde uz vajadzību pēc pietiekami nobrieduša programmatūras izstrādes modeļa, kur vides ir viegli atjaunojamas, un nesatur piekļuvi pie neatjaunojamiem resursiem.

3.5.5. Iekļaušana programmatūras būves procesā

Programmatūras izstrāde virzās uz nepārtrauktās integrācijas(*continous integration*) rīku universālu adopciju. Tipiski testēšana ir viens no programmas būves soļiem, uz ko gaida tālākie programmizlaides soļi. Tas rada prasības pret testu izpildes ātrumu. Ir sagaidāms, ka piedāvātais algoritms prasīs ilgu laika sākotnējai grafa ģenerēšanai dēļ lielā skaita virknēto HTTP pieprasījumu. Papildus rezultātu novērtēšana ir daļēji balstīta uz heiristikām, kas nav vēlams būves procesā dēļ tā trokšņainuma. Tāpēc nav vēlams veikt pilno pētniecisko testēšanu kā bloķējošu soli.

Pirmā alternatīva ir izmantot iepriekš ģenerētu stāvokļu grafu un testpiemērus priekš automatizēta regresijas testa.

Otrā alternatīva ir iekļaut risinājumu kā nebloķējošu soli. Analoģija šai pieejai ir statiskās analīzes rīki kā *SonarQube*. Testu/analīzes rezultāti tiek akumulēti un ir pieejami, bet tie tiek ievākti nebloķējot un neapstādinot pārējos programmas būves soļus.

3.5.6. Izpildes ierakstīšana

Tālākai problēmu reproducēšanai ir nepieciešams pēc testa izpildes saglabāt izpildītos testpiemēras soļus. Tas iekļauj sevī arī lietotāja imitācijai pievienotās laika nobīdes.

Papildus diagnostikai ir ieteicams saglabāt defektīvā stāvokļa ekrānšāviņu. Testā izpildītie stāvokļi var tikt glabāt pagaidu krātuvē un pieglabāt ja mēģinātais testpiemērs atklāja defektu.

3.5.7. Izpildes paralelizācija

Kritiskais algoritma laika patērētājs ir HTTP pieprasījumi grafa ģenerēšanas un testpiemēru izpildes soļos.

Grafa ģenerēšana viegli nepakļaujas vieglai paralelizācijai datu atkarību dēļ. Katrs solis paļaujas uz iepriekš ģenerētiem soļiem, tāpēc mēs nevaram veikt soļus ārpus kārtības.

Testpiemēru izpildes solis ir uzdevum-paralelizējams(*task parallelization*). Vienkārša pieeja ir uzskatīt katra testpiemēra izpildi kā neatkarīgu uzdevumu. Izmantojot šo pieeju ieteicams nodrošināt testpiemēru neatkarību(skat *testpiemēru neatkarība*).

3.5.8. Algoritma pielāgošana

Piedāvātais algoritms satur vairākus parametrus tā pielāgošanai konkrētam lietojumgadījumam.

Ekvivalences klases ir atkarīgas no biznesa loģikas.

Testa rezultātu invarianti ir atkarīgi no biznesa loģikas.

Atļautie url it atkarīgi no testēšanas sfēras.

3.5.9. Izpildes ierakstīšana

Katram testpiemēram nepieciešams saglabāt 3 rezultātus:

1. Izpildes rezultāti – saraksts ar apgāztām invariantiem/heiristikām
2. Veiktie soļi – testpiemēra izpildītās darbības
3. Ekrānšāviņi – rezultātu ekrānšāviņi lai atvieglotu lietotāja rezultātu verifikācijas darbu

REZULTĀTI

Ir izpētīts nejaušināmas tīmekļa saskarnes reprezentācijas izgūšanas uzdevums, un piedāvāts algoritms reprezentācijas izgūšanai no dokumenta objektmodeļa(*document object model*). Pieeja izmanto pakāpenisku saskarnes telpas izpēti caur dokumenta objektmodelī pieejamo elementu darbināšanu. Rezultāts ir pārstaigājams saskarnes stāvokļu grafs.

Ir apskatīts jautājums par saskarnes stāvokļu granularitāti un piedāvāti risinājumi vērtību sarakstu un ekvivalenču klašu formā. Risinājums ir cieši atkarīgs no testējamās sistēmas un izmanto cilvēka iesaisti.

Ir izpētītas pieejas testpiemēru izpildāmo soļu ģenerēšanai. Ir piedāvāts algoritms testpiemēru soļu ģenerēšanai caur alkatīgu saskarnes stāvokļu grafa pārstaigāšanu.

Ir aplūkotas grūtības un piedāvāti iespējamie risinājumi testa rezultātu novērtēšanai – lietotāja definēti invarianti, uz HTTP atbildes kodiem bāzētas heiristikas, u.c. Šis jautājums prasa tālāku izpēti.

Ir piedāvāts testu darbināšanas risinājums caur integrēšanos ar eksistējošiem pārlūkprogrammu darbinošiem ietvariem kā *Selenium.*

Nobeidzot, ir izceltas vairākas praktiskas problēmas, un risinājuma vieta kopējā izstrādes ciklā.

Pēc autora uzskatiem, tālāka izpēte ir jāveic testu rezultātu novērtēšanas jautājumā, un jāizstrādā prototips pret ko attīstīt grafa izgūšanas un pārstaigāšanas algoritums.

Kopumā mērķis izpētīt nejaušdatu testēšanu tīmekļa lietotņu kontekstā ir sasniegts. Nejaušdatu testēšanu tīmekļa lietotnēm ir izmantojama. Daudzsološākās ir regresiju testēšanas un pētnieciskās testēšanas automatizācijas iespējas.

SECINĀJUMI

Kopumā nejaušdatu testēšana ir pozicionējama kā papildinājums, bet ne aizvietojums eksistējošajām testēšanas metodēm. Piedāvātais risinājums ir analoģisks nejaušai saskarnes pētnieciskajai testēšanai. Nododot šo darbu automatizācijai, vairāk laika tiks atstāts skriptēto testu izstrādei.

Svarīgs punkts ir daļā procesa, ko nav izdevies pilnībā automatizēt. Kritiskie punkti ir ekvivalenču klašu un testa invariantu noteikšana. Abas šīs darbības prasa zināšanas par biznesa loģiku, kas nav nodiktējams rīka līmenī. Iespējams, risinājums ir caur ārēju specificēšanu vai pat testējamā koda(vai tā vienībtestu) analīzi.

IZMANTOTĀ LITERATŪRA UN AVOTI

1. Gerald D Everett; Raymond McLeod *Software testing : testing across the entire software development life cycle*, Wiley, 2007
2. **C.Carabas, M.Carabas**, Fuzzing the Linux kernel, Proceedings of the Computing Conference, 2017
3. HTML standarts
   Pieejams tiešsaistē: https://html.spec.whatwg.org/multipage/input.html#number-state-(type=number) (Skatīts 01.05.2021.)
4. A. Zeller, R. Gopinath, M, Böhme, G. Fraser, and Christian Holler: "Testing Graphical User Interfaces". "The Fuzzing Book", https://www.fuzzingbook.org/html/GUIFuzzer.html. (Skatīts 01.05.2021.)
5. Interneta resurss
   Pieejams tiešsaistē: https://web.archive.org/web/20170427220310/http://www.digital-web.com/articles/the_document_object_model/ (Skatīts 14.05.2021.)

Bakalaura darbs „Nejaušdatu testēšana tīmekļa lietotņu saskarnēm” izstrādāts LU Datorikas fakultātē.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti.

Autors: Patriks Misāns 26.05.2021.

Rekomendēju darbu aizstāvēšanai

Vadītāja: profesore Dr. sc. comp. Vineta Arnicāne 26.05.2021.

Recenzents: profesors Dr. math. Kārlis Podnieks

Darbs iesniegts Datorikas fakultātē 27.05.2021.

Dekāna pilnvarotā persona: vecākā metodiķe Ārija Sproģe

Darbs aizstāvēts bakalaura gala pārbaudījuma komisijas sēdē

07.06.2021. prot. Nr. 1B.

Komisijas sekretārs: I.Odītis