VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ KATEDRA

Klaidų valdymo metodų įtakos programų sistemų atsparumui klaidoms tyrimas taikant agentais grįstą modeliavimą

Research on software error handling methods using agent based modelling

Magistro baigiamasis darbas

Atliko: Tomas Smagurauskas (parašas)

Darbo vadovas: doc. dr. Karolis Petrauskas (parašas)

Recenzentas: Prof. dr. (HP) Saulius Gudas (parašas)

Vilnius – 2018

Turinys

Įvadas

Norint sėkmingai sukurti programų sistemą yra reikalingas planas - programų sistemos architektūra. Programų sistemų architektūra suprantama kaip taikomų programų sistemos projektavimo ir aukšto abstrakcijos lygio realizavimo sprendimų visuma. Programų sistemų architektūra turi atsakyti į klausimus, kaip bus patenkinti sistemai keliami reikalavimai. Architektūra turi užtikrinti, kad nefunkciniai reikalavimai bus tarpusavyje suderinti. Norint pasiekti nefunkcinių reikalavimų suderinamumą, reikia atlikti kompromisus sprendžiant kaip, kokiu principu, kokia sistemos dalis bus įgyvendinta.

Norint palyginti programų sistemų architektūras reikalingi parametrai, pagal kuriuos jos bus vertinamos. Standartas ISO/IEC 25010 [dNor11] nusako nefunkcinius reikalavimus programų sistemoms, kurie nusako įvairius kokybinius kriterijus, kuriais pasižymi programų sistemos. Tarp šių kriterijų yra tokie kaip greitaveiką, atsparumas klaidoms, resursų naudojimas.

Klaidų valdymo šablonai nusako veiksmus atliekamus įvykus klaidai, kurie turi sumažinti arba panaikinti klaidos sukeltą žalą. Priklausomai nuo parinkto klaidų valdymo šablono, programų sistema pasižymės skirtingomis kokybinėmis charakteristikomis - kaip gerai klaida bus suvaldyta, kaip greitai veiks sistema ir kita.

Agentais grįstas modeliavimas yra sistemų modeliavimo būdas. Taikant agentais grįsto modeliavimo metodus galima modeliuoti programų sistemų architektūras. Teisingas programų sistemų architektūros parinkimas yra svarbus viso projekto sėkmei, todėl yra reikalingi būdai kurie leistų patikrinti architektūros tinkamumą. Modeliuojant programų sistemų architektūras pasitelkus agentus, galima gauti informaciją apie tai, kokių charakteristikų galima tikėtis iš architektūros. Esant metodikai, nusakančiai kaip modeliuoti klaidų valdymo šablonus architektūrose, būtų galima tiksliau parinkti tinkamiausią klaidų valdymo šabloną.

Magistro baigiamojo darbo tikslas yra sukurti metodą, kuris leistų palyginti klaidų valdymo metodus programų sistemose, naudojant agentais grįstą modeliavimą. Darbe bus remiamasi [Mik14] ir [Sil16] aprašyta metodika, ją praplečiant klaidų valdymo galimybe.

Tikslui pasiekti reikalingos įgyvendinti užduotys:

1. Sukurti klaidų valdymo modeliavimo agentais metodą, kuriuo būtų galima modeliuoti programų sistemą, joje vysktančias klaidas ir tų klaidų valdymą.

2. Įvykdyti simuliacijas su sukurtu agentų modeliu, kombinuojant skirtingus klaidų valdymo metodus ir programų sistemų architektūras, siekiant ištirti jų poveikį sistemos elgsenai.

3. Gautus duomenis palyginti su realių sistemų duomenis, siekiant patikrinti jų validumą. Validavimas atliekamas sukuriant analogiškos minimalios sistemos realią versiją ir jos charakteristikas palyginant su gautomis agentų modelio simuliacijoje.

Šiame darbe bus pateiktas minėtasis modelis, leidžiantis modeliuoti klaidų valdymo metodus programų sistemose, taikant agentais grįstą modeliavimą, bei modeliui bus atlikta validacija. Modelis pateikiamas kaip agentų ir jų parametrų rinkinys, bei taisyklės, kaip agentai turi

komunikuoti tarpusavyje, bei naudoti minėtuosus parametrus. Su modeliu bus atlikta simuliacija ir pateikti jos rezultatai, bei tų rezultatų palyginimo kriterijai.

1. Literatūros apžvalga

1.1. Programų sistemų architektūra

Išaugęs informacinių technologijų prieinamumas sąlygojo organizacijos resursų planavimo (ERP), santykių su klientais valdymo (CRM) ir kitų informacinių sistemų (programų sistemų) paplitimą organizacijose. Organizacijoms konkuruojant, siekiant kuo geriau patenkinti klientų poreikius, tinkama informacinė sistema gali užtikrinti organizacijos pranašumą rinkoje. Didelėms organizacijoms, ypač tokioms kurios yra geografiškai išplitusios, turi sudėtingas tiekimo grandines ir kuriuose dirba daugybė darbuotojų, kyla iššūkių efektyviai suvaldyti visą organizacijos procesą. Informacinės sistemos gali organizacijai suteikti galimybes suvaldyti šiuos iššūkius.

Informacinės sistemos įgalina organizcijas efektyviau valdyti savo veiklą, taip leidžiant joms pasiekti didesnius pelnus [MSV03].

Siekiant sukurti kokybišką informacinę sistemą yra svarbu pasirinkti tinkamą architektūrą. Programų sistemų architektūra nurodo kokie bus elementai, kokios bus tų elementų atsakomybės ir kaip bei su kokiais kitais elementais jie sąveikaus. Architektūra yra projektuojama atsižvelgiant į tai, kokie yra keliami reikalavimai sistemai. Architekto tikslas yra taip suprojektuoti sistemą, kad visi sistemai keliami reikalavimai, tiek funkciniai, tiek nefunkciniai, galėtų būti įgyvendinti.

1.2. Nefunkciniai reikalavimai

Nefunkciniai reikalavimai apibrėžia, kokiomis kokybinėmis savybėmis programų sistema turi pasižymėti. Programų sistemų kokybinės savybės gali būti tokios kaip: greitaveika (kiek trunka užklausos apdorojimas), saugumas (kas ir kaip gali valdyti duomenis) ar plečiamumas (kaip gerai fizinės įrangos resursų pajėgumai koreliuoja su programų sistemos gebėjimu atlaikyti didesnį kiekį vartotojų užklausų ar pan). Yra ir daugiau kokybės charakteristikų. Industrijoje plačiai paplitęs standartas ISO/IEC 25010 [dNor11], kuris apibrėžia 8 charakteristikas ir 31 sub-charakteristiką. 8 pagrindinės charakteristikos yra:

1. Funkcinis tinkamumas
2. Vykdymo efektyvumas
3. Suderinamumas
4. Naudojamumas
5. Patikimumas
6. Saugumas
7. Palaikomumas
8. Perkeliamumas

Galima pastebėti, kad nefunkcinius reikalavimus būtų galima skirstyti į dvi papildomas kategorijas, pagal tai, kaip jie yra suformuluojami, - kiekybiškai išreiškiamas vykdymo metrikas ir kokybiškai išreiškiamus reikalavimus. Kiekybiškai išreiškiamo nefunkcinio reikalavimo pavyzdys galėtų būti sistemos greitaveika: sistema turi galėti apdoroti bent 100 užklausų per sekundę. Nefunkcinio kokybinio reikalavimo, kuris apibrėžia kokiomis kokybinėmis savybėmis turėtų pasižymėti sistema, pavyzdys galėtų būti suderinamumo reikalavimas: sistemą turi būti galima pritaikyti prie ateityje atsirasiančių operacinių sistemų. Kokybiškai išreikštus reikalavimus, dažniausiai, galima išreikšti grupe kiekybiškai išreikštų reikalavimų.

Kokybinių savybių įgyendinimas dažniausiai yra kompromisų paieška, tarp kelių skirtingų atributų [Bas12]. Pavyzdžiui: galima numanyti, kad saugumo ir greitaveikos reikalavimai nebūtinai bus paprastai tarpusavyje suderinami. Jeigu saugumo reikalavimas sako, kad visi veiksmai sistemoje turi būti išsaugomi audito tikslais, tai, žinoma, pareikalaus papildomų vykdymo resursų, dėl ko užklausos apdorojimo laikas sumažės.

Žinant, kad turi būti vykdomas informacijos apie užklausas saugojimas, galima būtų sugalvoti bent kelis būdus kaip tokį funkcionalumą realizuoti. Vienas būdas galėtų būti visa tai saugoti pridedant eilutę su įrašu įprastinėje mašinos failinėje sistemoje. Šis būdas būtų paprastas realizuoti, nes jis nereikalautų jokių papildomų bibliotekų ar kokios nors kitos papildomos įrangos. Žinoma, rašymas į failą dažniausiai būna gana brangi operacija mašinos resursų atžvilgiu. Taip pat šį reikalavimą galima realizuoti atliekant tą patį įrašo išsaugojimą duomenų bazėje. Šiuo atveju pati išsaugojimo operacija būtų pigesnė, nes brangiu rašymu į failinę sistemą pasirūpintų specialiai tam kurta duomenų bazių valdymo sistema, bet, ši sistema įneštų papildomo sudėtingumo ja naudojantis ir papildomų kaštų ją eksploatuojant.

Iš šio pavyzdžio matoma, kad skirtingi, to pačio reikalavimo įgyvendinimo būdai, gali turėti skirtingą poveikį kokybės charakteristikoms. Vienu atveju yra pasiekiama geresnė greitaveika, todėl tuo pačiu metu galima aptarnauti daugiau vartotojų naudojant tą patį resursų kiekį; kitu atveju yra išlaikomas sistemos paprastumas ir nepriklausomybė nuo išorinės programinės įrangos. Dėl šių priežasčių, programų sistemos architektūros parinkimas yra itin svarbus etapas, norint, kad sistema galėtų pasiekti visus jai keliamus tikslus.

1.3. Architektūros stiliai ir šablonai

Programų sistemų inžinerijoje, pasikartojančioms koncepcijoms programose nusakyti, dažnai yra naudojami architektūros stiliai ir šablonai. Šablonai yra tam tikri architektūrinių problemų sprendimo būdai, kurie yra apibrėžiami konkrečia struktūra ar veikimo specifika. Architektūros stiliai apibrėžia principus, kuriais remiantis yra kuriama sistema. Tokie principai gali nusakyti struktūros, komunikacijos, paskirstymo ar kitų architektūros sričių stilius.

Pirmą kartą pradėta kalbėti apie programinės įrangos architektūros šablonus dar 1987 metais [Smi87] - konferencijoje buvo paminėta perpanaudojamų architektūros abstrakcijų idėja. Idėja programų sistemų inžinerijos srityje prigijo ir toliau buvo plėtojama kitų autorių, - 1995 metais išleista viena iš žinomiausių šios srities knygų: “Design patterns: elements of reusable object-oriented software” [VHJ+95]. Architektūros šablonų sričiai plečiantis, sukurta ir aprašyta vis

daugiau šablonų. Šablonai tapo nebe tokie abstraktūs, kokie buvo aprašinėti pirmuosiuose šaltiniuose [VHJ+95], o labiau pritaikyti spręsti specializuotoms problemoms.

Programų sistemų architektūrų stiliai nusako sprendimus aukštesniame abstrakcijos lygyje, nei architektūros šablonai. Stiliai nusako kokiais principais ar koncepcijomis remiantis, bus parenkami šablonai ar kiti architektūriniai sprendimai. Architektūros stiliaus įgyvendinimui gali būti panaudoti keli architektūriniai šablonai. Toliau bus abžvelgti keli architektūros stiliai, kurie yra aktualūs šiame darbe.

1.4. Monolitinis diegimas stilius

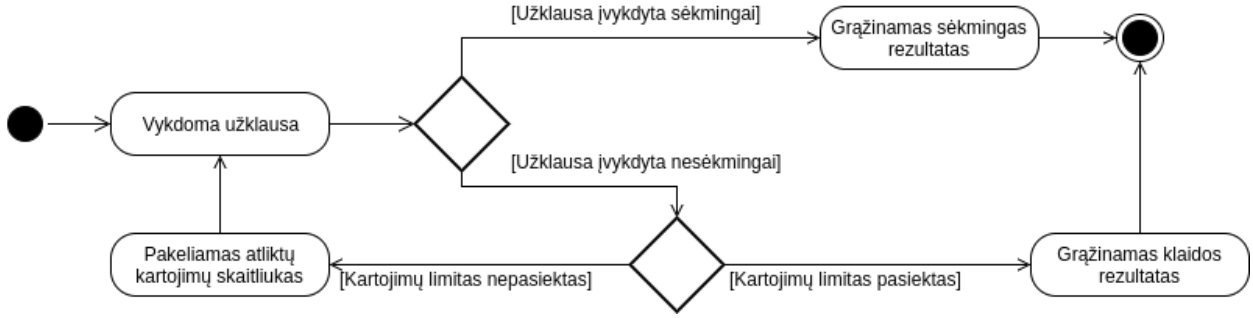
Monolitiniu diegimu yra laikomas toks architektūros stilius, kurio pagrindu sukurta programa gali būti diegiama tik kaip vienas nedalomas elementas. Pavyzdys tokio tipo programos galėtų būti, kai visa internetinė sistema, sukurta Java programavimo aplinkoje, kuri yra pateikiama kaip vienas archyvinis (WAR) failas. Tokios architektūros programos yra vystomos kaip vienas projektas, kuriame kažkoks modulinis skaidymas galimas tik to pačio projekto rėmuose. Tai, kad visa aplikacija yra vienoje vietoje, gali būti naudinga su ja dirbantiems programuotojams, nes visi reikalingi elementai ir kodas yra pasiekiami vienoje vietoje. Bet, projektui plečiantis, tai gali tapti nenaudinga, jeigu projektą taptų per sunku suprasti, dėl to, kad jis yra pateikiamas kaip vienas didelis modulis ir nėra aiškaus jo skirstymo elementais [NS14].

Monolitinio stiliaus programą yra lengva įdiegti, nes viskas yra viename modulyje. Tačiau, dėl to, kad viskas yra viename modulyje, monolitines programas yra sunku išplėsti (angl. scale). Plėtimas galimas tik diegiant identiškas programos kopijas, nėra galimybės išplėsti tik reikalingo modulio. Diegiant programos kopijas susiduriama su problema, kad visos programos, dėl to kad jos yra identiškos, dirbs su tais pačiais duomenimis ir didėjant saugomų duomenų kiekiui kiekvienas įdiegtas egzempliorius susiduria su vis didėjančiu duomenų krūviu. Taip pat, iškilus problemai su serverio pasiekiamumu, iškart tampa nepasiekiama visa sistema, nes ji yra leidžiama vienoje vietoje.

1.5. Paslaugų architektūra

Paslaugų architektūra (angl. services oriented architecture) yra programų sistemų architektūros dizaino stilius, kuriame visas sistemos funkcionalumas yra pateikiamas kaip paslaugos (angl. services). Sistema yra išskirstoma į mažesnius elementus pagal jų veiklos specifiką, kurie pateikia paslaugų priėjimo interfeisus išoriniams naudotojams. Paslaugų architektūros modelyje galima išskirti 3 aplikacijų tipus: paslaugų tiekėjas, paslaugų gavėjas ir paslaugų registratorius [ZCZ07]. Paslaugų tiekėjo programos vykdo tam tikras operacijas ir jas pateikia kaip paslaugas. Paslaugų gavėjas naudoja kitų programų teikiamas paslaugas. Paslaugų registratorius saugo informaciją apie prieinamas paslaugas ir pateikia ją kitoms programoms. 1 paveikslelyje yra vaizduojamas šių programų sąsajos pavyzdys.

Paslaugų architektūrą, kaip požiūrį į sistemos projektavimą, galimą išplėsti iki organizacijos veiklos architektūros.



1 pav. Klaidų valdymo metodo kartojant diagrama

Paslaugų architektūros pavydzys, kaip ji yra suprantama šiame darbe, pateiktas 1. Paslaugų architektūra modeliuojama kelių elementų sluoksnių rinkinys, susidedantis iš aukštesnio ir žemesnio lygmens paslaugų. Paslaugos gali kviesti tik tokio pačio arba žemesnio lygmens paslaugas, kitaip sakant, elementų architektūrą galima suprasti, kaip orientuotą grafą, neturintį ciklų.

Mikropaslaugų architektūra

Mikropaslaugų architektūra išskaido funkcinius elementus į nepriklausomas paslaugas. Mikropaslaugų architektūra yra viena iš paslaugų architektūrų įgyendinimo variacijų. Toks išskaidymas sumažina atskirų elementų tarpusavio sąryšius, nes yra pateikiami tik interfeisai bendravimui tarp elementų. Toks išskirstymas yra parankus, nes leidžia lengvai plėsti informacines sistemas, tačiau to pasekoje gali būti apsunkinamas jų vystymas. Į mikropaslaugų architektūrą galima žiūrėti kaip į idėjiškai priešingą monolitinei architektūrai, nes kol monolitinėje architektūroje visi elementai yra sutelkti viename projekte, mikropaslaugų architektūros atveju, elementai yra išskaidyti į nepriklausomus projektus (mikropaslaugas).

Mikropaslaugos prieigą dažniausiai pateikia naudojant interneto paslaugas arba žinučių perdavimo protokolus. Komunikacija gali būti užmezgama arba tiesiogiai su kokiu nors moduliu arba naudojant brokerį, kuris nukreipia užklausą į reikalingą elementą. Kadangi ryšiai yra užmezgami dinamiškai, galimos problemos dėl interfeisų suderinamumo. Tokios problemos ypač gali iškilti tais atvejais, kada prie tos pačios sistemos mikropaslaugų vystymo dirba didelis kiekis darbuotojų. Jeigu sistema yra pagrįsta mikropaslaugomis, dažnai būna pasiskirstoma, kad tam tikra tikra programuotojų komanda dirba su konkrečiu elementu ar elementų grupe. Kai skirtingoms komandoms reikia integruoti savo elementus tarpusavyje, gali iškilti nesuderinamumo problemų, jeigu komandos turėjo skirtingas tų elementų vizijas ir jų nebuvo suderinę tarpusavyje.

Problemos taip pat gali iškilti mikropaslaugų elementų atnaujinimo atvejais. Jeigu organizcija, vystanti mikropaslaugomis grįstos architektūros sistemą yra pakankamai didelė, gali būti, kad bus netinkamai sudokumentuojami, ar dėl kokių kitų veiksmų nesuderinimo, užmirštami sąryšiai esantys tarp kelių elementų. Tiksliai nežinant kurie elementai bus paveikti pakeitus elementą, nuo jo priklausantys elementai gali nebefunkcijonuoti korektiškai. Todėl gali kilti nesklandumų atnaujinant turimas mikropaslaugas, jeigu nėra užtikrinančiu procesų ar infrastruktūros, kuri galėtų užtikrinti tokios išskaidytos sistemos korektišką veikimą.

Jeigu yra sėkmingai išsprendžiamos problemos kurios kyla dėl sistemos išskirstymo, mikropaslaugų architektūra teikia tokių privalumų kaip: lengvas plečiamumas, galimybė

sėkmingai naudoti skirtingas technologijas tarpusavyje, sistemos atsparumas klaidoms. Kadangi mikropaslaugų elementai teikia smulkias aibes mikropaslaugų, yra lengva horizontaliai praplėsti sistemą, paleidžiant daugiau reikalingos mikropaslaugos egzempliorių. Taip pat, kadangi gali būti naudojama daug vienetų tos pačios paslaugos, galima nustatyti, kad kiekvienas egzempliorius dirbtų tik su tam tikru poaibiu duomenų.

Kiekvienas elementas gali būti sukurtas naudojant skirtingą technologiją, nes komunikacija yra vykdoma naudojant bendrinius interfeisus, kurie nėra priklausomi nuo konkrečios technologijos. Dėl šios priežasties kiekvienam elementui galima naudoti skirtingą technologiją, kuri gali būti tinkamesnė tos problemos sprendimui.

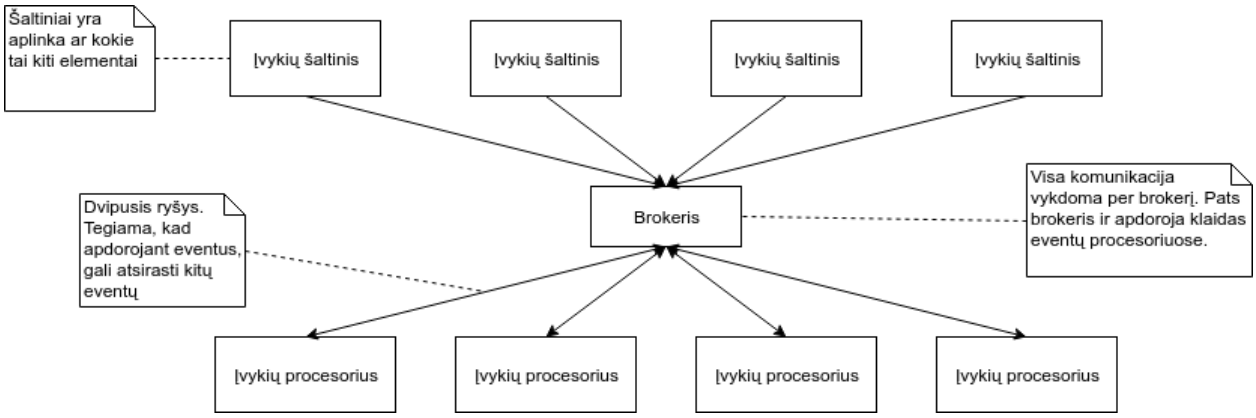
Neveikiant vienam mikropaslaugos egzemplioriui klaidos gali būti išvengta, jeigu yra paleista daugiau tos paslaugos egzempliorių - kiti egzemplioriai gali toliau funkcionuoti korektiškai. Dėl to, kad yra naudojami dinaminai ryšiai tarp mikropaslaugų, neveikiant vienam elementui, kurio nėra galimybes pakeisti kitu, veikiančiu, elementu, sutriktų tik jį kviečiančių kitų mikropaslaugų funkcionavimas, kitaip tariant - klaida būtų izoliuota.

1.6. Įvykiais grįstas dizainas

Įvykiais grįstas dizainas (angl. event driven design) yra architektūros stilius, kuriame komunikacija tarp elementų yra vykdoma pasitelkiant įvykius. Įvykiu yra laikomas pranešimas apie reikšmingą pasikeitimą vykdymo būsenoje [Cha06]. Toks komunikacijos vykdymas leidžia pasiekti mažą susiejimą (angl. coupling) tarp elementų. Elementas kuriame įvyksta tam tikras būsenos pasikeitimas išsiunčia informaciją apie įvyki visiems kitiems elementams, kurie laukia to tipo įvykio. Įvykio sukūrimas paprastai yra iškviečiamas kaip įvykio identifikatoriaus ir įvykio duomenų pora. Visi įvykio klausytojai yra notifikuojami apie įvykusį įvykį, taip pat perduodant siuntėjo pateiktus duomenis.

Architektūros šablonas, kuriame klausytojai gali užsisakyti tam tikro įvykius ir apie juos gauti notifikacijas yra vadinimas pateikimo-prenumeravimo šablonu (angl. publish-subscribe) [EFG+03]. Šiame šablone galima užsiprenumeruoti konkretų įvykį arba kokią nors įvykių grupę. Įvykiai įprastai yra identifikuojami jiems suteikiant pavadinimus. Taip galima prenumeruoti įvykius pagal tam tikrą pavadinimą, pavyzdžiui - “įvykis”. Jeigu yra keletas įvykių, pavyzdžiui - “įvykis.sukūrimas” ir “įvykis.ištrynimas”, juos abu būtų galima užprenumeruoti naudojant vardų šabloną. Šablonus galima nurodyti naudojant pakaitinį simbolį (angl. wildcard), pavyzdžiui žvaigždutę (“*”). Tokiu atveju prenumeruojant naudojant šabloną “įvykis.*” būtų užprenumeruojami visi įvykiai turintys pradžioje tekstą “įvykis.” ir bet kokią reikšmę toliau. Pateikto pavyzdžio atveju ir “įvykis.sukūrimas” ir “įvykis.ištrynimas” tenkina šį šabloną ir būtų sėkmingai užprenumeruota laukti šių įvykių.

Daugelis šiuo metu populiarų programavimo kalbų kaip Java, C#, Javascript turi kalbos lygyje įgyvendintą įvykių funkcionalumą. Analogiškai pateikimo-prenumeravimo šablonas gali būti implementuotas naudojant žinučių perdavimo protokolą kaip AMQP.



2 pav. Įvykiais grįsto dizaino architektūra

Šiame darbe įvykiais grįsto dizaino architektūra modeliuojama pagal [Ric15] knygoje aprašytą būdą, kur elementai gali ir apdoroti įvykius ir taip pat juos sukurti. Šie elementai vadinami įvykius procesoriais. Visi įvykių procesoriai gauna ir pateikia informacija apie įvykius per įvykių brokerį. Modeliuojant įvykių brokeris yra naudojamas kaip įeities elementas su kurio komunikuoja elementai įvykių procesoriai dvipusiu ryšiu.

1.7. Karkasai

Karkasai yra architektūros šablonai, kuriais naudojantis yra modeliuojamas ne lokalizuotos problemos, o visos sistemos struktūros sprendimas. Karkasai dažniausiai pateikia bendrą idėją, kaip bus įgyvendinta sistema, pavyzdžiui, kokia bus klasių struktūra kuriamoje programoje, kaip bus formuojami atsakymai į užklausas, kokie bus naudojami interfeisai bendrauti su kitomis sistemomis. Siekiant palengvinti darbą su karkasais yra parašytos bibliotekos, kurios implementuoja visą karkaso architektūrą, o norint jį naudoti, tereikia toliau plėsti savo aplikaciją naudojantis karkaso bibliotekos pateikiamomis gairėmis. Karkasinių bibliotekų naudojimas yra itin paplitęs, nes jos suteikia galimybę ženkliai paspartinti programinės įrangos kūrimo procesą.

1.8. Klaidos

Klaida yra laikoma, kai programų sistema pasiekia tokią vykdymo būseną, kada ji nebegali atlikti tolesnių operacijų. Klaidos dažniausiai kyla dėl defektų programinės įrangos kode [TLL+14] - defektas programinės įrangos kode sąlygoja, kad vykdomos operacijos pakeičia sistemos būklę į neplanuotą (neapibrėžtą), ko pasekoje veikimo metu įvyksta klaida ir sistema nebegali toliau tęsti savo veikimo. Defektai programinėje įrangoje paprastai atsiranda kūrimo fazėje. Nors ir yra sukurta daug metodikų, kurios siekia užkirsti kelią defektų sukūrimui arba sukurtus defektus eliminuoti ankstyvose programų kūrimo proceso fazėse, sukurti sistemos, kuri visiškai neturėtų defektų, yra neįmanoma. Kadangi sukurti programų sistemos, kuri visiškai neturėtų klaidų nėra įmanoma, o klaidos yra pagrindinė nepasiekiamumo priežastis, yra nagrinėjami alternatyvūs sprendimai padedantys užtikrinti sistemos pasiekiamumą. Klaidų valdymo šablonai yra vienas iš sprendimų, leidžiančių sumažinti vykdymo metu iškylančių klaidų sukeliamą žalą.

1.9. Klaidų sukeliami nuostoliai

Informacinėse sistemose įvykstančios klaidos, kada dalis ar visa informacinė sistema tampa nepasiekiama, sukelia nuostolius naudojančioms organizacijoms. Kokia dalis sistemos bus paveikta klaidos priklauso nuo sistemos architektūros ir kokie klaidų valdymo metodai yra taikomi joje. Seniau atliktos studijos rodo, kad valanda sistemos neveikimo, gali kainuoti nuo dešimčių tūkstančių iki milijonų [Pat+02]. Priklausomai nuo to, kokia organizacijos veiklos dalis ir kiek teikiamų paslaugų yra kompiuterizuota, gali daryti įtaką tam, kokius nuostolius sukelia organizacijos informacinių sistemų sutrikimai. Nuostoliai patirti dėl nekorektiškai funkcionuojančių informacinių sistemų nėra paprastai apskaičiuojami, nes informacinių sistemų problemos pasireiškia daugelyje sluoksnių organizacijos operacinių sluoksnių, pakeičiant jų įprasta darbo metodiką [Pat+02]. Jeigu organizacija teikia elektronines paslaugas, sistemai esant nepasiekiamai, ar kuriam iš jos moduliu funkcionuojant netinkamai, gali kilti kėblumų su paslaugų tiekimu, paslaugų naudotojai negalėtų jomis naudotis ar užsakyti naujų paslaugų. Sistemai pradėjus veikti nekorektiškai, dalį veiksmų organizacijai gali tekti atlikti rankiniu būdu, ko pasekoje atsirastų papildomos išlaidos žmogiškiesiems resursams.

1.10. Klaidų valdymas

Vienas iš klausimų, į kurį reikėtų atsakyti kuriant architektūrą, yra kaip bus valdomos vykdymo metu iškilusios klaidos. Kuriant ar renkantis klaidų valdymo šablonus yra taikoma prielaida, kad programų sistemos veikimo metu įvyks nenumatytos klaidos, dėl kurių nebebus galima tęsti operacijų ir todėl reikia pritaikyti metodą, kuris galėtų atlikti koreguojančius veiksmus, siekiant ištaisyti įvykusi klaidą. Tinkamai parinkti, kokiais metodais bus valdomos klaidos yra svarbu, nes nuo to priklauso tokios sistemos charakteristikos kaip: pasiekiamumas, duomenų integralumas, robastiškumas ir kitos. Kaip ir dauguma kitų architektūrinių sprendimų, renkant klaidų valdymo metodus yra atliekamas kompromisas tarp poveikio keletai kokybinių charakteristikų. Pavyzdžiui pritaikius nesėkmingos operacijos pakartojimo šabloną, kada nepavykus operacijai yra bandoma ją atlikti dar kelis kartus, tikintis kad vėlesni bandymai bus sėkmingi, iš vartotojo pusės užklausa matomai užtrunka ilgiau dėl to, kad yra atliekami kartojimo veiksmai; tuo tarpu jeigu tik gavus klaidą ji būtų propaguojama atgal vartotojui, tokia užklausą būtų įvykdyta greičiau. Pavyzdys parodo, kad vienas klaidų valdymo šablonas, lyginant su kitu, gali siekti to pačio tikslo (suvaldyti klaidą), bet turėti skirtingas kokybines charakteristikas.

Klaidų valdymo šablonai yra nagrinėjami ir literatūroje, yra išleistą juos aprašančių knygų [Han13]. Literatūroje ir kituose šaltiniuose [Han13][WHF93][C213] yra aprašomi tokie klaidų valdymo šablonai kaip:

1. Karantinas
2. Restartavimas
3. Sistemos atstatymas
4. Kartojimas

5. Atjungimas
6. Atstatymo taškas
7. Klaidos sprendėjęs
8. Duomenų atstatymas
9. Klaidų propagavimas
10. Mirusių žinučių eilė
11. Klaidos nutildymas

Šis sąrašas, žinoma, nėra baigtinis egzistuojančių klaidų valdymo šablonų sąrašas. Dalis šių klaidų valdymo šablonų, kurių veikimas yra pagrįstas išskirstytos architektūros ar asinchroninės komunikacijos principais, bus toliau nagrinėjami darbe.

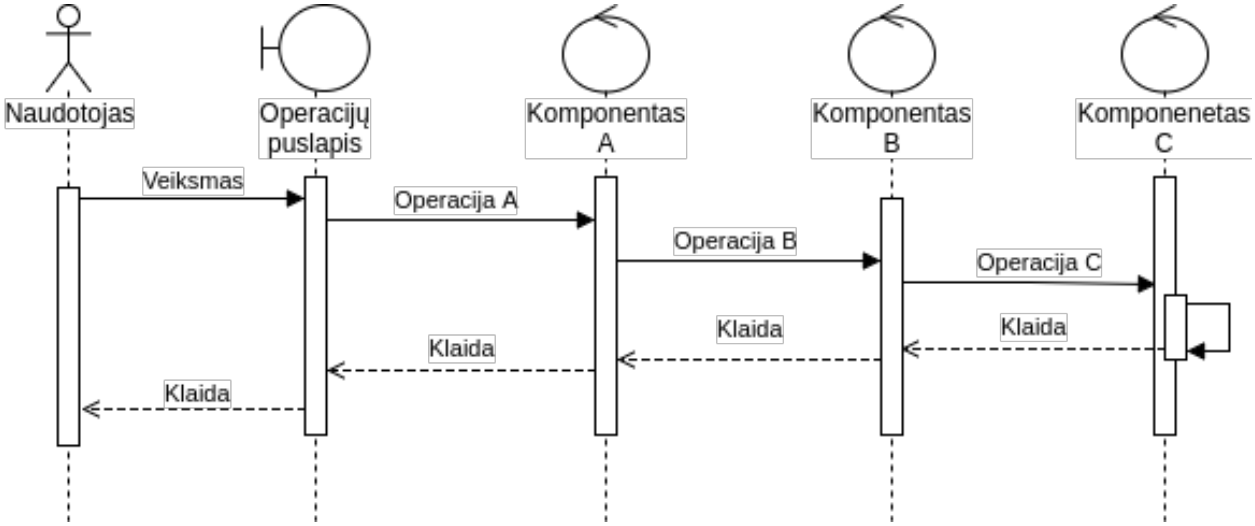
Klaidų valdymo šablonais yra sprendžiama, kokie veiksmai bus atliekami iškilus klaidai vykdymo metu. Dažniausiai klaidų valdymo mechanizmo tikslas yra bandyti išspręsti įvykusią klaidą taip, kad iš vartotojo perspektyvos operacija vistiek būtų įvykdyta. Dažniausiai klaidos kyla dėl to, kad programų sistema atsiduria tokioje būsenoje, kurioje niekas neplanavo, kad ji atsidurs [TLL+14], todėl dauguma klaidų valdymo šablonų grindžia savo veikimo principą bandant atstatyti sistemos būseną į validžią. Taip pat gali būti taikomi tokie metodai, kurie pasikliautų tuom, kad sistemos naudotojas, kurio inicijuota operacija patyrė klaidą, galės atlikti veiksmus, kurie tą klaidą išspręstų. Gali būti taikomi metodai, kurie taiko prielaidą, kad elemento, kuriame pradėjo vykti klaidos paprastai ir automatizuotai sutvarkyti nėra įmanoma ir jeigu tas elementas turi pakaitalą jam, tai visą srautą reikia nukreipti į pakaitinį elementą, o klaidingąjį atjungti, kol sistemos administratoriai išspręs jame kilusią problemą.

Kai kurie klaidų valdymo šablonai yra galimi tik pritaikius tam tikrus kitus architektūrinius principus (stilius) kuriamoje programų sistemoje. Pavyzdžiui: mirusių žinučių eilę (angl. dead letter queue) galima panaudoti tada, kai komunikacija programoje yra pagrįsta kokio nors tipo žinutėmis. Kiti klaidų valdymo stiliai remiasi tuom, ar komunikacija tarp elementų yra vykdoma sinchroniškai ar asinchroniškai. Sinchroniška komunikacija įprastai yra vykdoma tada, kada užklausos baigtis yra aktuali kvietusiojo elemento tolimesniam darbui, o asinchroniška komunikacija naudojama tada, kai kvietėjo elementas gali tęsti darbą, nepriklausomai nuo operacijos būsenos, ar ji baigta ar ne. Dėl šių priežasčių, reikėtų nagrinėti klaidų valdymo šablonus bendrame architektūros kontekste.

1.10.1. Klaidų propagavimas

Programų sistemos veikimo metu įvykus klaidai, klaidos nėra bandoma stabdyti, o ji yra perduodama aukštyn kvietimų medžiu, kol galiausiai gali pasiekti vartotoją. Veikimą iliustruoja 3 paveikslėlis. Šis angliškai dar yra vadinamas fail-fast. Klaida yra propaguojama naudotojui tais atvejais, kai yra turima prielaida, kad naudotojas gavęs informaciją apie klaidą pats sugebės

atlikti veiksmus, kurių pagalba pavyktų sėkmingai įvykdyti operaciją. Toks sprendimo būdas būtų veiksmingas, jeigu klaida kyla dėl netinkamų vartotojo įvesties duomenų. Atlikti tyrimai rodo, kad pateikus naudotojams pranešimą apie klaidas, tik maža dalis jų sugeba atlikti koreguojančius veiksmus, kurie leistų sėkmingai įvykdyti operaciją [EOK11]. Naudotojams yra sunku atlikti koreguojančius veiksmus nes klaidų pranešimai, ypač tais atvejais, kada įvyksta nenumatytos klaidos, būna labai abstraktūs. Norint sėkmingai interpretuoti tokius klaidų pranešimus reikia turėti arba užtektinai žinių apie informacinę sistemą su kuria yra dirbama arba žymių informacinių technologijų srities žinių, kurių paprasti naudotojai neturi.

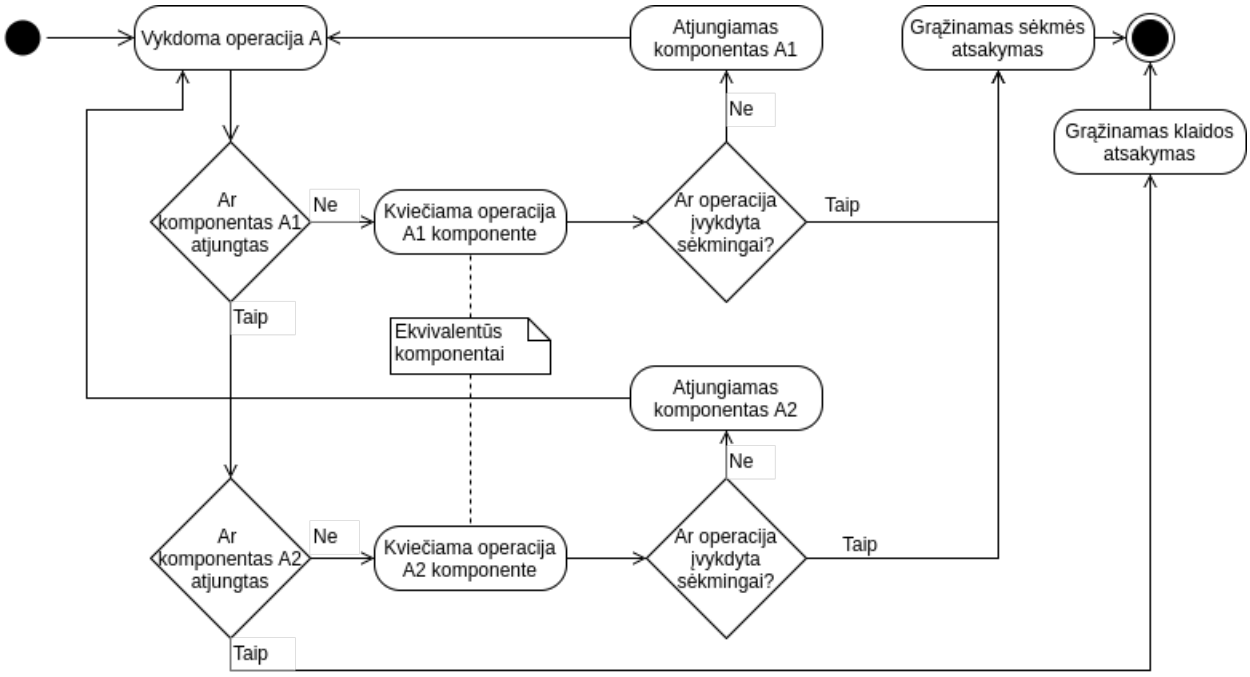


3 pav. Klaidų valdymo metodo propaguojant diagrama

1.10.2. Atjungimas

Naudojant atjungimo (angl. latch) klaidų valdymo šabloną, modulis, kuriame įvyksta klaidos yra išjungiamas, kad į jį nebebūtų kreipiamasi, o tos užklausos yra nukreipiamos į kitus to paties tipo modulius. Išjungtas modulis nevykdo jokio darbo iki kol jis vėl nėra įjungiamas. Atjungimo veikimą iliustruoja 4 paveikslėlis. Norint taikyti tokį klaidų valdymo šabloną, reikia kad sistemos architektūra palaikytų paslaugų (angl. services) duplikavimą, tai reiškia, kad modulis kuris teikia tam tikrą funkcionalumą, gali būti paleistas keliais egzemplioriais. Tokias sąlygas tenkina mikropaslaugų ar kokie kiti įvykiais grįsti (angl. event driven) ar išskirstytų architektūrų tipai.

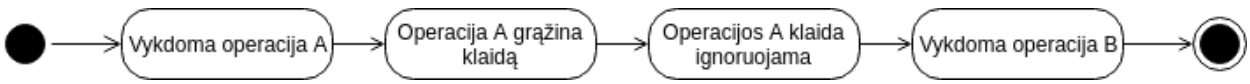
Atjungus modulį, situaciją turėtų įvertinti sistemos administratorius ir imtis atitinkamų veiksmų, kad modulis galėtų būti vėl paleistas. Toks klaidų valdymo šablonas yra tinkamas tuo atveju, kai klaida yra sąlygojama nevalidžios modulio būsenos. Jeigu klaida įvyksta dėl blogos naudotojo įvesties ar dėl užklausos nesuderinamumo su esamais duomenimis, toks šablonas klaidos neišspręstų.



4 pav. Klaidų valdymo metodo atjungiant diagrama

1.10.3. Klaidų nutildymas

Gavus klaidą, ji yra ignoruojama ir yra bandoma vykdyti veiklą toliau taip, lyg klaida nebūtų įvykusi. Metodas yra iliustruojamas 5 paveikslelyje. Toks metodas yra veiksmingas tais atvejais, kai klaidingai įvykusi operacija nėra būtina vartotojo inicijuotos operacijos įvykdymui. Tokį klaidų valdymo šabloną yra sunku pritaikyti operacijoms kurios dirba su duomenų apdorojimu, perdavimu ar saugojimu, nes tokio tipo operacijos paprastai yra kritinės korektiškam sistemos veikimui. Toks klaidų valdymo metodas yra tinkamas tais atvejais, kada operacija yra notifikuojamojo pobūdžio, pavyzdžiui - po sėkmingo duomenų išsaugojimo atvaizduoti vartotojui informacinį pranešimą. Tokiu atveju pagrindinis veiksmas - duomenų išsaugojimas - yra įvykdomos, o tik šalutinis veiksmas - vartotojo notifikavimas - yra neįvykdomos, o sistemos būsena (angl. state) išlieka korektiška, nes visos būseną keičiančios operacijos yra įvykdomos sėkmingai. Klaidų nutildymo metodas yra parankus tuom, kad jis turi labai mažus reikalavimus bendrai sistemos architektūrai, nes jis nepriklauso nuo tokių sistemos savybių kaip komunikacijos būdas ar įdiegimo metodo.



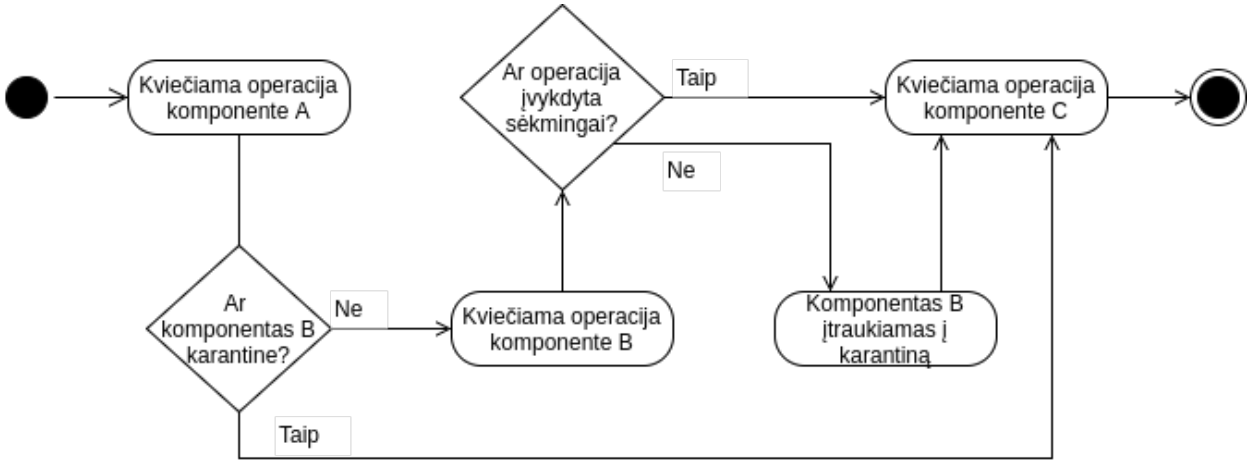
5 pav. Klaidų valdymo metodo nutildant diagrama

1.10.4. Karantinas

Karantino klaidų valdymo metodo tikslas yra izoliuoti nekorektiškai veikiantį modulį, kad jis negalėtų klaidų perduoti į kitus modulius. Metodo veikimas yra iliustruojamas 6 paveikslėlyje. Jeigu atliekant užklausas į tam tikrą modulį yra gaunamos klaidos, gali būti, kad tas modulis viduje atlieka neteisingas operacijas iki patekdamas į būseną nuo kurios nebegali tęsti savo veiklos (klaidos

būseną) ir apie tai pranešdamas kvietėjui. Klaidingos užklausos su nekorektiškais duomenimis gali sugadinti duomenis modulių, kurie funkcionuoja korektiškai. Tokiais atvejais nekorektiškai veikiančio modulio izoliacija padėtų išspręsti problemą dėl klaidų skleidimo į kitus modulius.

Norint įtraukti modulį į karantiną reikia nutraukti visas užklausas adresuojamas jam, kad jis nepradėtų vykdyti naujų operacijų, ir užtikrinti, kad visi iš to pačio modulio paleisti procesai būtų sustabdyti. Kai modulis generuojantis klaidą yra įtraukiamas į karantiną - į jį nebėra kreipiamasi ir yra bandoma vykdyti procesus tiesiog ignoruojant šio modulio teikiamą funkcionalumą. Karantinuotas modulis gali ir toliau vykdyti kažkokią veiklą, t.y. jis nėra pilnai išjungiamas. Šis metodas turi panašumų "atjungimo" metodo atžvilgiu, nes gali toliau vykdyti kažkokią veiklą, kuri galėtų padėti išspręsti modulyje kilusias problemas, tačiau "karantino" metodo atveju, tam tikras modulis yra tiesiog toliau ignoruojamas ir nebekviečiamas išvis, o ne tik kažkuris specifinis to modulio egzempliorius. Analogiškai kaip ir klaidų nutildymo atveju, toks valdymo metodas yra veiksmingas tuo atveju, kada modulio teikiamų operacijų pasiekiamumas nėra kritiškai svarbus naudotojo inicijuojamų operacijų sėkmei.



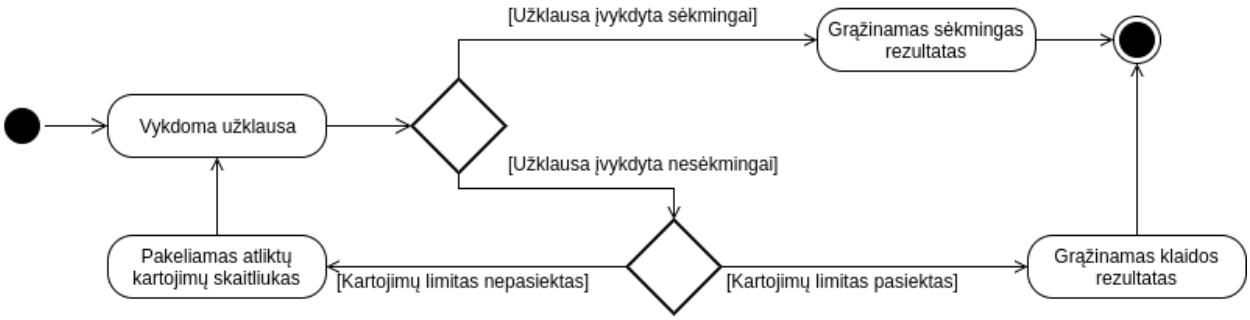
6 pav. Klaidų valdymo metodo karantinu diagrama

1.10.5. Kartojimas

Klaidų valdymui naudojant užklausos kartojimo metodą, nepavykusi užklausa yra papildomai kviečiama keletą kartų, tikintis, kad problema dėl kurios užklausa nebuvo įgyvendinta anksčiau išsispręs be kvietėjo įsikišimo. Metodo veikimas yra pavaizduotas 7 paveikslėlyje. Tokia veikimo prielaida yra teisinga, jeigu sukurta klaida įvyksta dėl kviečiamo elemento būsenos arba dėl problemų komunikuojant su elementu. Integracinių komunikacijų problemos ar laikinas integracinių elementų nepasiekiamumas yra labai tikėtinas. Ryšio sutrikimai, dažnai, susitvarko be tiesioginės žmonių intervencijos, todėl pabandžius atlikti tą pačią užklausą vėliau, yra galimybė, kad ji bus įvykdyta sėkmingai. Kadangi visos sistemos siekia palaikyti savo pasiekiamumą kuo aukštesnį, todėl tikėtina, kad sistemai, su kuria yra bendraujama integracijos tikslais, esant nepasiekiamai, jos funkcionalumas bus greitai atstatytas ir ji vėl taps pasiekiama.

Klaida gali būti sukeliama dėl siunčiamos žinutės turinio. Kadangi programos yra deterministinės, atlikus užklausą su tokiais pačiais duomenimis, bus gaunamas toks pats atsakymas.

Kai dėl pačios žinutės turinio yra sukuriama klaida, užklausos kartojimas nėra naudingas. Tokiais atvejais užklausos kartojimas tik sabotuoja sistemos veikimą ir iškelia būtinybę, kad užklausą būtų kartojama tik apibrėžtą kiekį kartų, siekiant išvengti vykdymo aklavietės. Keičiant kiekį kartų, kiek yra kartojama užklausa, galima derinti santykį tarp sistemos greitaveikos ir klaidų tolerancijos charakteristikų.



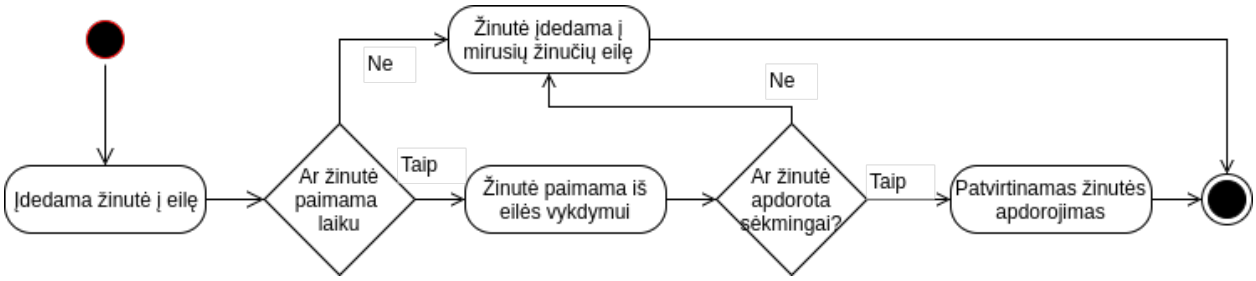
7 pav. Klaidų valdymo metodo kartojant diagrama

1.10.6. Mirusiųjų žinučių eilė

Mirusių žinių eilė (angl. dead letter queue) arba mirusių žinučių kanalas (angl. dead letter channel) yra klaidų valdymo šablonas, kuriame visos nesėkmingos žinutės yra talpinamos į specialią eilę. Metodo veikimas vaizduojamas 8 paveikslėlyje. Mirusių žinučių eilės pavadinimas yra kiek dažnesnis, jis kildinamas iš pažangaus žinučių eilės protokolo (advanced messaging queue protocol), nes jame žinutės yra talpinamos į eiles, kaip į tarpinius buferius, prieš nusiunčiant prenumeratoriams. Mirusių žinučių kanalas yra labiau generalizuotas pavadinimas, tinkamesnis tais atvejais kada yra kalbama konceptualiame lygmenyje apie šį klaidų valdymo šabloną, o kanalu vadinama bet kokia komunikacijos infrastruktūra.

Mirusių žinučių eilės klaidų valdymo šablonas yra tinkamas naudoti tais atvejais, kada užklausos rezultatas nėra būtinas tolimesniam programos vykdymui. Žinutės patalpintos mirusiųjų eilėje gali būti įvykdytos vėliau, kada, manomai, yra išspręstos problemos ar pasikeitusios sąlygos, dėl kurių žinutė negalėjo būti sėkmingai apdorotą iš karto. Taip galima užtikrinti, kad galiausiai visi reikalingi veiksmai bus atlikti arba, kad visiems integracijos taškams bus pranešta apie pasikeitimus įvykusius sistemoje.

Nagrinėjant konceptualesniame lygmenyje šį klaidų valdymo šabloną, vietoje eilės galėtų būti naudojama kokia nors kita užklausų talpykla. Natūralu, kad HTTP užklausų nepavyks sudėti į tą pačią eilę kaip ir AMQP žinučių. Tačiau iš abstrakčios perspektyvos tiek AMQP tiek HTTP žinutės yra kažkokie pranešimai, tik priderintas prie atitinkamo protokolo standarto, tai reiškia, kad panaudojus kokią nors tarpinę saugojimo priemonę galima pasiekti tą patį rezultatą.

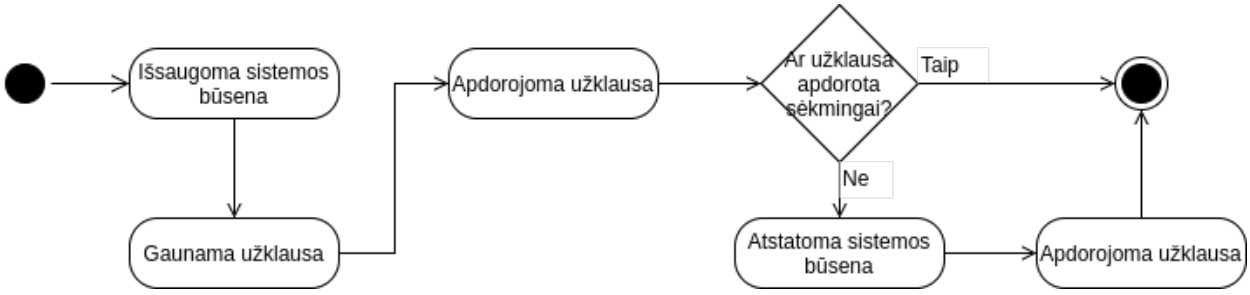


8 pav. Klaidų valdymo metodo mirusiųjų žinučių eile diagrama

1.10.7. Atstatymo taškai

Sukuriamas “atstatymo taškas”, kuriame yra išsaugoma visa esama programų sistemos būsenos informacija tam tikru laiko momentu taip, kad vėliau būtų galima būseną atkurti į išsaugotąją. Metodo veikimas yra vaizduojamas 9 paveikslėlyje. Saugojimo metodiką galima išskirti į du variantus: saugoti tik vykdomąją sistemos būseną arba saugoti vykdomąją sistemos būseną įskaitant programų sistemos naudojamą duomenų saugyklą. Šios dvi saugojimo strategijos skiriasi tuom, kad išsaugoti tik einamąją būseną yra palyginti paprasčiau ir ją galima greičiau atkurti. Išsaugoti duomenų bazės pakeitimus yra sunkiau, be to duomenų atkėlimas gali būti labiau komplikuotas, taip pat yra tikimybė, kad atstatinėjant duomenis duomenų bazėje jie bus sugadinti arba prarasti.

Atstatymo taškų klaidų valdymo metodika yra tinkama tais atvejais, kai klaida įvyksta dėl nekorektiškos sistemos būsenos. Jeigu klaidos yra sąlygojamos tik einamosios programų sistemų būsenos, jos išsaugotos kopijos atkėlimo turėtų pakakti, kad klaida nebepasikartotų. Būna tokių atvejų, kada klaida yra sąlygojama ne tik einamosios būsenos, bet ir saugomų sistemos duomenų. Tokiu atveju, norint atkurti sistemos veikimą atstatymo taškų principu, reikėtų atkurt ir duomenų bazės būseną į tą, kuri buvo atstatymo taško sukūrimo metu. Atkūrinėjant duomenų bazę, atsiranda pavojus sugadinti duomenis. Duomenų sugadinimo ar praradimo klausimas yra aktualus todėl, nes daug sistemų turi integracijas su kitomis sistemomis. Pakeitus duomenis vienoje sistemoje, tuo pačiu pakeitimai ar kiti veiksmai yra atliekami ir kitoje sistemoje. Grąžinus duomenis į senesniąją būseną vienoje sistemoje, gali būti neįmanomą atlikti analogiško veiksmo kitoje sistemoje, su kuria yra sukurta integracija. Dėl šių priežasčių, atstatymo taškų metodas gali būti netinkamas tais atvejais, kada programų sistema turi integracijas su kitomis sistemomis, kurių nėra galimybės valdyti. Šiame darbe nebus nagrinėjimas variantas su duomenų bazių duomenų atstatymu.



9 pav. Klaidų valdymo metodo atstatymo taškais diagrama

1.10.8. Klaidų valdymo metodų sąvybės

Šaltinis [Bas12] apibrėžia kategorijas, kaip galima skirtstyti klaidų valdymo metodus: metodai kurie tik nustato klaidas, metodai kurie atstato klaidų padarytą žalą ir metodai kurie užkerta kelią pakartotiniam klaidų įvykimui. Apibrendrinant aptartus klaidų valdymo metodus, pagal šaltiniuose [Han13][WHF93][C213] pateiktus klaidų valdymo metodų veikimo aprašymus galima nustatyti tam tikras savybes, kuriomis jie pasižymi. Išskirtinos savybės:

1. ***Atributas 1*** - Ar naudotojo užklausos rezultatas bus būtinai teisingas?
2. ***Atributas 2*** - Ar iškviestos procedūros rezultatas bus būtinai teisingas?
3. ***Atributas 3*** - Ar gali išspręsti klaidą realiu metu?
4. ***Atributas 4*** - Ar gali išspręsti egzemplioriaus idiopatinį nekorektišką veikimą?
5. ***Atributas 5*** - Ar gali išspręsti klaidas sukeltas išorinių priežasčių?

1 lentelė. Klaidos valdymo metodų veikimo atributai

Klaidos valdymo būdas	**Atributas 1**	**Atributas 2**	**Atributas 3**	**Atributas 4**	**Atributas 5**
Propagavimas	Taip	Taip	Ne	Ne	Ne
Atjungimas	Taip	Taip	Taip	Taip	Ne
Nutildymas	Ne	Ne	Ne	Ne	Ne
Kartojimas	Taip	Taip	Taip	Ne	Taip
Mirusiųjų žinučių eilė	Taip	Ne	Ne	Taip	Taip
Atstatymo taškai	Taip	Taip	Taip	Taip	Ne
Karantinas	Taip	Taip	Taip	Taip	Ne

Klaidų valdymo metodai turi savus atributus, kaip jie valdo klaidas. Klaidų valdymo įgyvendinimo atributai yra susiję jų kokybinėmis savybėmis ir gali padėti nutatyti prielaidas, kuriais atvejais koks klaidų valdymo būdas yra parankesnis.

1. ***Atributas 1*** - Ar įvykus klaidai bandoma kartoti kvietimą?
2. ***Atributas 2*** - Ar pakartotinai kviečiamas tas pats elemento egzempliorus?
3. ***Atributas 3*** - Ar klaida apdorojoma sinchroniškai?
4. ***Atributas 4*** - Ar bando keisti pačio elemento būseną?

2 lentelė. Klaidų valdymo metodų įgyvendinimo atributai

Klaidos valdymo būdas	Atributas 1	Atributas 2	Atributas 3	Atributas 4
Propagavimas	Ne	Ne	Taip	Ne
Atjungimas	Taip	Ne	Taip	Ne
Nutildymas	Ne	Ne	Taip	Ne
Kartojimas	Taip	Taip	Taip	Ne
Mirusiųjų žinučių eilė	Taip	Ne	Ne	Ne
Atstatymo taškai	Taip	Taip	Taip	Taip
Karantinas	Taip	Ne	Taip	Ne

Svarbu atsižvelgti į tai, kokiais būdais klaidų valdymo metodai bando pasiekti tikslą ir kaip tai yra implementuojama, norint gauti korektiškus rezultatus.

1.11. Klaidų valdymo metodų lyginimas

Siekiant atlikti skirtingų klaidų valdymo metodų palyginimą, reikalingi kriterijai, kuriais remiantis bus lyginama. Atliekant simuliavimą galima gauti kiekybinius rezultatus apie modelio veikimą. Tikslingas renkamų kiekybinių metrikų parinkimas yra svarbus norint sukurti modelį, kuris teisingai atliktų savo darbą. Darbe [HSS+93] yra pateikiamas skir tingų klaidų valdymo metodų lyginimas remiantis kriterijais: ar klaida buvo pastebėta, ar klaida buvo išspręsta, ar programa veikė po klaidos, ar klaida buvo teisingai nustatyta, kiek laiko sistema buvo nepasiekiama dėl klaidos, laikas per kiek buvo įvykdoma operacija be klaidos, laikas per kiek buvo įvykdoma operacija esant klaidai. Straipsnyje [XSS06], minimi kriterijai, kuriais gali skirtis klaidų valdymo sprendimai: resursų naudojimas, vykdymo laikas ir patikimumas. [dNor11] standarte apibrėžti nefunkciniai reikalavimai, tai pat mini tuos pačius kriterijus kaip prieš tai paminėti šaltiniai: korektiškumas, resursų naudojimas ir greitaveika. Kadangi

Apibrendinant šaltinius galima išskirti tokius kriterijus pagal kuriuos būtų galima lyginti klaidų valdymo metodus:

- Kaip greitai veikia metodas.
- Kaip kokybiškai metodas išsprendžia klaidas.
- Kiek resursų sunaudota darbui atlikti.

1.12. Agentais grįstas modeliavimas

Agentais grįstas modeliavimas yra vienas iš sistemų modeliavimo būdų. Naudojant agentais grįstą modeliavimą, galima daryti prielaidas, kaip tam tikra sistema, susidedanti iš daugelio elementų, elgtųsi tam tikromis sąlygomis. Agentais grįstas modeliavimas yra pritaikomas daugelyje tyrimų sričių tokių kaip biologijoje modeliuoti mikroorganizmų elgseną, socialiniuose moksluose modeliuoti individų elgseną ar fizikoje modeliuoti dalelių judėjimus [MN14].

Viena iš ypatybių, kas priverčia agentais grįstą modeliavimą išsiskirti iš kitų modeliavimo būdų yra tai, kad šiuo būtų yra modeliuojami visi sistemos nariai atskirai o ne kartu [CS99].

Kiekvienas agentas simuliacijoje priima asmeninius sprendimus, naudojant turimas žinias apie jį supančią aplinką, vietoje simuliacijos valdytojo, kuris žino visas simuliacijos detales. Tai leidžia sumodeliuoti autentiškesnes sistemas, - kurios teisingiau atspindi realybės elgseną.

Siekiant vykdyti agentais grįstą modeliavimą, yra sukurta programinės įrangos paketų, kuriuos naudojant galima apibrėžti agentų modelius sąlyginai greitai ir lengvai [Get08]. Kadangi apibrėžti agentų modelį simuliacijai yra greičiau nei kurti kokį nors realybę atitinkantį prototipą [RLJ06], tai yra labai naudingas būdas, siekiant palyginti keletą galimų programų sistemų architektūros sprendimų.

Agentais grįsto modeliavimo struktūra susideda iš agentų, sąveikų tarp agentų ir tų agentų erdvės [MN14]:

- Agentas turi apibrėžtas savybes ir taisykles, kaip reaguoja į jį veikiančius veiksnius (aplinką);
- Sąveika tarp agentų apibrėžia kaip vieni agentai reaguoja į kitus agentus;
- Agentų aplinka, kuri sąlygoja veiksnius veikiančius agentus;

Apibrėžiant šiuos elementus, galima nusakyti įvairias sistemas. Šiuolaikinių programų struktūras galima išreikšti agentais grįsto modeliavimo elementais (1 pav.), nusakytais C.M. Macal [MN14]: Klasės ar koks kitas lokalizuotas elementas yra ekvivalentus agentui. Sąveikos tarp agentų yra ekvivalenčios sąryšiams tarp programų sistemos elementų, pavyzdžiui, kai vienas elementas kviečią kitą elementą. Agentų aplinka atitinka programų sistemos vykdymo aplinkos būseną. Agentais grįstas modeliavimas programų sistemoms yra patrauklus pasirinkimas tuom, kad elementų veikimą nereikia įgyvendinti, o užtenka išreikštinai apibrėžti parametrus apie jų veikimą.

1.13. Agentais grįstas programų sistemų architektūrų modeliavimas

Moksliniuose šaltiniuose ir knygose programų sistemų architektūrų modeliavimas naudojant agentus nėra detaliai išnagrinėta tema, tačiau ši tema yra nagrinėjama A. Siliūno ir A. Mikoliūno magistriniuose darbuose [Sil16] [Mik14]. Šaltiniai [Sil16] [Mik14] teigia, kad norint sėkmingai modeliuoti programų sistemų architektūrą, reikia nustatyti, kokiame detalumo lygyje bus vykdomas modeliavimas. Tiksliau, reikia nustatyti, kas bus laikoma elementu, kokio dydžio programinio kodo abstrakcija bus modeliuojama. Modeliuoti galima nuo smulkiausio masto programų sistemos fragmento - kodo eilutės - iki didelio masto fragmento - modulio. Modeliuoti itin mažo masto elemento nėra prasminga, nes tada modelio sudarymas tampa labai ilgas ir gilinamasi į problemą nebeabstrakčiai, bet pradedamas tyrinėti įgyvendinimas. Pasirinkus per didelio dydžio elementą gali nebūti pasiektas pageidaujamas detalumas ir gauti simuliacijos rezultatai nebus užtektinai detalūs ar patikimi, norint atlikti vertinimą jais remiantis. Nusprendus kokios apimties struktūra bus laikoma elementu, apibrėžiamas kokie architektūros elementais bus modeliuojami. Vienas elementas atitiks vieną agentą modelyje. Šaltinis [Mik14] pateikia parametrus, kokius reikia apibrėžti norint agentu modeliuoti programų sistemų elementus tačiau nevisi jie yra tinkami šiam darbui. Toliau apibendrinami [Mik14] apibrėžti agentų parametrai ir jų panaudojamumas šiame darbe:

- Kaip elementas kvies savo naudojamus elementus
- Kiek minimaliai naudojamų elementų turi būti iškviesta
- Kiek maksimaliai naudojamų elementų gali būti iškviesta
- Kiek kartų naudojami elementai turi būti iškviesti
- Kiek kartų minimaliai naudojami elementai turi būti iškviesti
- Kiek kartų maksimaliai naudojami elementai turi būti iškviesti
- Kiek laiko truks elemento kvietimas
- Kiek maksimaliai užklausų galima vykdyti lygiagrečiai vienu metu
- Kokia tikimybė, kad elemento vykdymas pasibaigs klaida
- Kokia tikimybė, kad elementas yra nepasiekimas.

Šie parametrai apibrėžia tam tikras taisykles, kaip agentų elementai sąveikaus su kitais elementais.

Apibrėžus agentų aprašymo taisykles, reikia apibrėžti, kaip bus modeliuojami sąryšiai tarp agentų [Mik14]. Elementui reikia apibrėžti tokius parametrus apie jo ryšius su kitais elementais:

- Su kokiais kitais elementais bus sąveikaujama ir kokio tipo bus sąveikos: sinchroninės ar aschinchroninės.
- Ar elementui reikia laukti kol visos užklausos į kitus elementus bus baigtos, ar jis gali nelaukti kol jos bus pabaigtos.

Sistemos resursų panaudojimo modeliavimui reikia apibrėžti serverio kuriame modeliuojama programų sistema resursus ir elementų naudojamų resursų kiekius. Serverio resursams išreikšti reikalingi tokie parametrai:

- Kiek procesoriaus resurso turi serveris
- Kiek operatyvios atminties resurso turi serveris
- Kiek disko operacijų resurso turi serveris
- Kiek tinklo resurso turi serveris
- Kokia strategija yra atliekamos užduotys: ar naudojama eilė pirmas įeina pirmas išeina, ar naudojama eilės pirmas įeina paskutinis išeina, ar užduotys yra vykdomos atsitiktine tvarka.

Egzistuoja tokie atvejai, kada elementas kviečia kitą elementą, kuris atspindi išorinę sistemą, su kuria yra integruojamasi. Tokiu atveju, vistiek reikia sulaukti kada kviečiamas elementas baigs savo darbą, tačiau nereikia rūpintis to elemento naudojamais resursais ir jo implementacija. Tokio tipo elementams reikia aprašyti tokius parametrus:

- Elemento atsako laikas milisekundėmis
- Kokia tikimybė kad elementas yra nepasiekiamas
- Kokia tikimybė, kad elemento iškvietimas pasibaigs klaida.

Naudojant elemento atsakymo klaida tikimybės parametrą sumodeliuojamos visos klaidos. Vietoje milisekundžių naudojimo pateikiamo [Mik14], geriau būtų naudoti prie realybės nepririštus matavimo vienetus, kad būtų galima paprasčiau nustatyti modelio validumą ir išvengti daugiaprasmiškumų. Taip pat pats elementas turi parametrą, kuris nusako, kiek trunka elemento darbas, todėl išorines sistemas galima būtų modeliuoti pakeitus įprastais agentų elementais taip atsisakant šių papildomų parametrų.

Nusakyti elemento naudojamiems serverio resursams reikalingi parametrai:

- Kokiame serveryje elementas yra vykdomas (identifikatorius, jei yra keli serveriai)
- Kiek procesoriaus resurso reikalinga atlitki elemento darbui
- Kiek operatyvios atminties resurso reikia elemento darbui
- Kiek disko resurso reikia elemento darbui
- Kiek tinklo resurso reikia elemento darbui atlikti
- Kiek laiko reikia elemento darbui atlikti

Aprašyti agentų aplinkai tai pat yra naudojami agentai. Reikia nusakyti, kaip elgsis sistemos naudojai - kokį aplinkos poveikį jie atliks. Nuo naudotojų priklauso, kokiu dažnumu ir kaip bus pradedami kreipimaisi į elementus. Tam aprašyti reikalingi tokie parametrai:

- Koks bus įeities elementas (į kuri bus kreipiamasi pirmiausiai)
- Ar elementas bus pasiekiamas sinchroniškai ar asinchroniškai
- Ar užklausos bus generuojamas reguliariu ar atsitiktiniu intervalu
- Kokia tikimybė, kad per simuliacijos žingsnį bus sugeneruota užklausa
- Kas kiek simuliacijos žingsnių bus vykdomas užklausų generavimas
- Minimalus sugeneruojamų užklausų keikis
- Maksimalus sugeneruojamų užklausų kiekis
- Ar užklausai esant klaidingai ją yra bandoma ją kartoti
- Jeigu klaidinga užklausa yra kartojama - kiek kartų ji yra kartojama
- Po kiek laiko negavus atsakymo apie užklausos sėkmė, ji yra pradedama traktuoti kaip nesėkminga

Įeities elemento nėra būtina traktuoti kaip atskiro parametro, nes modeliuojant aplinką kaip įprastinį agentą, tio parametro tikslą galima išreikšti pasiekit naudojant įprastos agentų komunikacijos parametrus, aprašytus skyriaus pradžioje. Taip pat, minimi kartojimo parametrai yra nebeaktualūs, nes darbui reikalingas sprendimas, kurio dėka klaidų valdymo metodai būtų keičiami, o ne iš anksto numatyti elemente.

1.14. Modelio validavimas ir verifikavimas

Validavimas ir verifikavimas yra svarbi kiekvieno simuliavimo modelio dalis. Modelio verifikacija ir validacija yra atliekama modelio kūrimo proceso metu. Validavimas ir verifikavimas yra skirtas patikrinti, ar kuriamas modelis tiksliai atspindi modeliuojamą sritį [XKM+05]. [XKM+05] taip pat pateikia keletą atrinktų validavimo technikų, iš kurių programų sistemų modelių validavimui būtų galima pritaikyti vizualų validavimą ir validavimą skirtingais parametrais.

Vizualus validavimas yra atliekamas modeliuojamos srities ekspertui peržvelgiant sukurtą modelį. Ekspertas, pasitelkęs savo žinias, turėtų identifikuoti koncepcines modelio problemas. Vizualiam validavimui paprastai yra pasitelkiama animavima arba grafinio atvaizdavimo technika. Animavimo būdų yra atvaizduojamas modelio simuliacijos vykdymas. Grafinio atvaizdavimo būdų yra pateikiami modelio simuliacijos išvesties duomenys. Animavimo būdai labiau yra pritaikyti modeliams, kurie dirba su fizinių kūnų judėjimu, dėl to programų sistemų modeliams jie nėra labai naudingi. Matant grafiškai atvaizduotus simuliacijos sugeneruotus duomenis (programų sistemų architektūrų modeliavimo atveju tai gali būti: atsako laikas, klaidų skaičius ir pan.), ekspertas galėtų spręsti, ar jie yra adekvatūs.

Validavimas keičiant parametrus taikomas, kai turima duomenų apie tikros sistemos veikimą. Panaudojant skirtingus įvesties duomenis, analogiškus tikrai sistemai, žiūrima ar modelio simuliacijos sukuriami duomenys atitinka realiuosius. Panašiai kaip matematikoje aproksimuojant funkcijas - kuo daugiau gautos aproksimacijos (modelio simuliavimo) taškų sutampa su aproksimuojamos (realios sistemos) taškais, tuo labiau galima pasitikėti, kad gauta funkcija bus teisinga ir likusiuosiuse taškuose.

[Tro04] apibrėžia 3 aspektus, pagal kuriuos tikrinamas simuliacijos modelio validumas.

- Replikuojamas validumas - modelio simuliacijos metu gauti duomenys atitinka duomenis gaunamus iš analogiškos tikros sistemos.
- Nuspėjamumo validumas - duomenys gauti modeliuojant atvejus, kurie dar nėra įvykę pateikia tokius rezultatus, kokie bus įvykus modeliuojamai situacijai realioje sistemoje.
- Struktūrinis validumas - modelis ne tik pateikia rezultatus atitinkančius realią sistemą, bet ir juos gauna veikdamas principais, kurie atitinka realios sistemos veikimo principus.

Panaši nuomonė yra išreiškiama ir [Sar96] darbe. [Sar96] teigia, kad norint tinkamai validuoti modelį, reikėtų pritaikyti bent keletą iš pateiktų validavimo technikų:

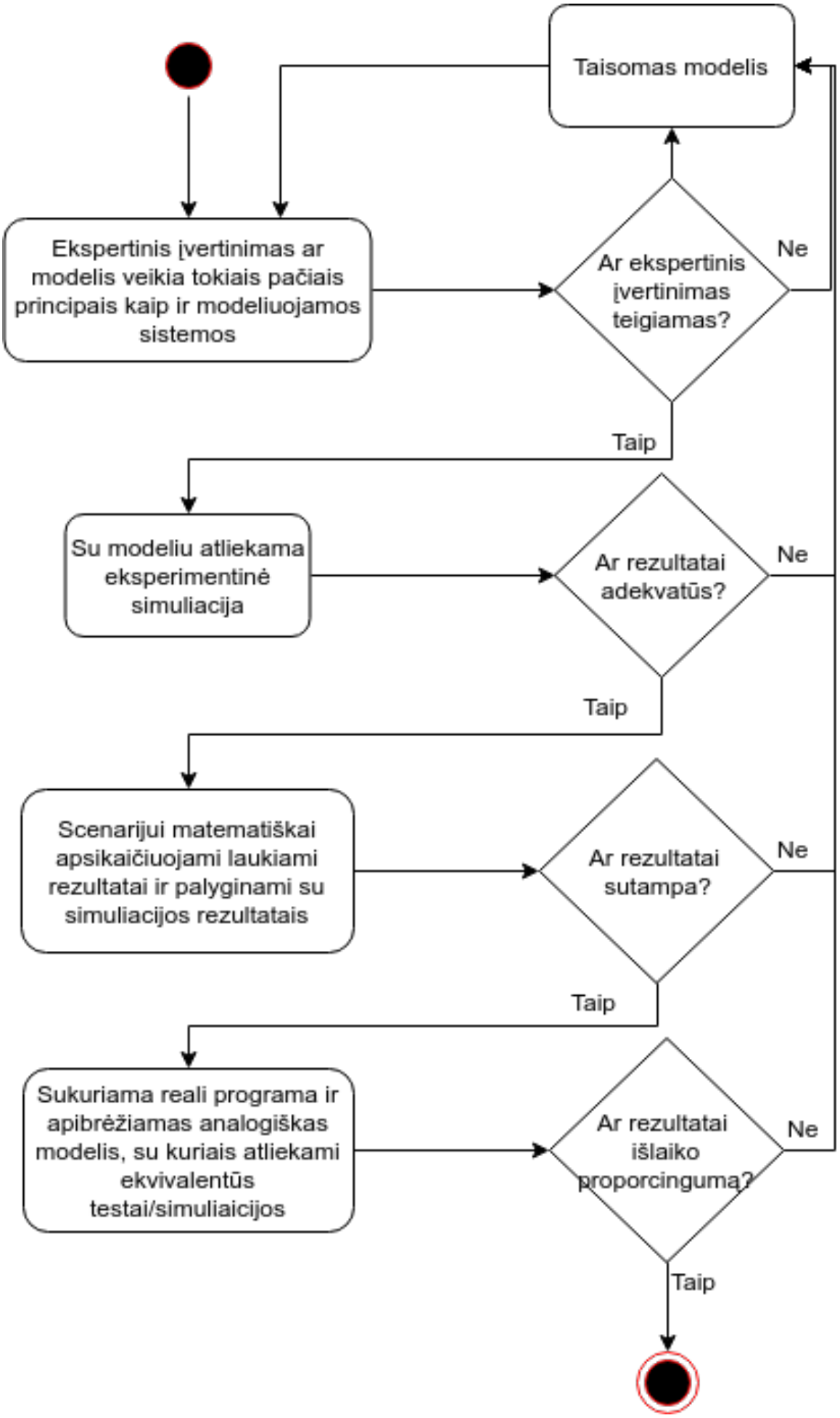
- Peržvelgiant modelio elgseną patikrinti ar veikimimo principai atrodo prasmingi ir adekvatūs.

- Grafinis (statistinis) rezultatų palyginimas su realia sistema. Grafinis palyginimas taikomas tam, kad tokiu būdu galima lengviau įžvelgti tam tikras koreliacijas, kurias galima praleisti taikant tik statistinę analizę.
- Palyginimas taikant hipotezes - ar hipotetiniai rezultatai atitinka modelio sugeneruotus realius rezultatus.
- Statistiškai agreguotų duomenų palyginimas.

Abu šaltiniai nusako panašius validavimo metodus. Šaltinis [Mar07] neišskirdamas konkrečių punktų, taip pat mini labai panašius validavimo aspektus kaip ir prieš tai minėti šaltiniai: validavimas lyginant rezultatus su realios sistemos rezultatais, rezultatų nuspėjamumas ir ekspertinė modelio peržiūra. Apibendrinus pateiktus šaltinius galima išskirti metodus, kurie bus taikomi šio modelio validavimui:

- Validuoti modelį palyginant rezultatus su realios sistemos rezultatais.
- Validuoti ar modelio elgsena yra panašį į modeliojamos sistemos elgseną.
- Validuoti ar modelio rezultatai yra nuspėjami.

Modelio validavimo proceso schema pateikiama 10 paveikslėlyje.



10 pav. Modelio validavimo schema

1.15. Programų sistemų veikimo modeliavimas

Programų sistemų operacinius nefunkcinių reikalavimų modeliavimo metodus galima skirtstyti į 3 pagrindines kategorijas [GP15] [BM99]:

- Analitiniai metodai
- Matavimais grįsti (empiriniai) metodai
- Simuliaciniai metodai

Analitiniai būdai yra atliekami modeliuojant sistemą pasitelkiant Markovo grandines, Petri tinklus ar kt. Analitiniai būdai tinkamesni modeliuoti sistemos kurios yra statinės, t.y. kai sistemos elementų būsena nekinta. Išskriaimi 3 pagrindiniai analitiniai būdai programų sistemoms modeliuoti:

- Markovo grandinės
- Petri tinklai
- Eilių teorija

Markovo grandinėmis sistema modeliuojama kaip orientuotas sistemos būsenų grafas, su taisyklėmis kaip naviguojama tarp tų būsenų. Petri tinkluose sistema yra apibrėžiamas kaip grafas, kur grafo elementai yra perėjimai tarp skirtingų būsenų. Markovo grandinės ir Petri tinklai turi problemą, kai reikia modeliuoti nehomogeniškas sistemas, pavyzdžiui, kai yra daug to pačio elemento tipo egzempliorių ir priklausomai nuo kiekvieno egzemplioriaus būsenos jis gali reaguoti į tokį patį stimulą kitaip [WPC06]. Eilių teorija modeliuoja elementus kaip eiles ir elementų ryšius, kaip manipuliacijas su kitų elementų eilėmis. Šis metodas turi trūkumą, kad reikia iš anksto žinoti tokias elementų metrikas kaip vykdymo trukmė, norint gauti adekvačius rezultatus [SG98].

Matavimais grįsti metodai vykdomi esant realizuotai programai ar kažkokiai jos matuojamai daliai, atliekant operacinius testus ir matuojant gaunamus rezultatus. Matavimais grįstas būdas nėra populiarus, nes norint jį atlikti reikalingi jau realizuota sistema, todėl jis labiau leidžia patikrinti esamą programą, nei daryti prielaidas apie tai kokios būtų jos charakteristikos.

Simuliavimo metodai išreikštinai modeliuoja sistemą, imituojant elementų darbą. Simuliavimo metai skiriasi tarpusavyje tuom, kuom naudojantis yra apibrėžiami sistemų modeliai. Vienas iš populiariausių modeliavimo metodų simuliacijai yra pasitelkiant UML diagramas [BM03] [Mar04]. Tačiau modeliavimo būdas naudojant uml turi problemą, kad juo galima apibrėžti modelis iki tokio detalumo, kiek leidžia UML. Modeliavimas pasitelkiant UML diagramas yra patrauklus, nes jos daugumai žmonių yra intuityvios ir lengvai suprantamos. Naudojant UML kyla problemos, kada reikia paruošti modelį apibrėžiant elementų veikimą žemame abstrakcijos lygyje, nes jeigu reikalingos labai detalios būsenų ir veiksmų sekos, išreikšti tai programiniu kodų gali būti paprasčiau nei, pavyzdžiui, sekų diagramomis. Taip pat, UML diagramomis gali būti sunku išreikšti kintančią elemento elgseną, priklausomai nuo elemento būsenos [Mar04].

Dabar esantys ir literatūroje aptarinėjami modeliavimo būdai yra labiau tinkami aukštesnio abstrakcijos lygmens modeliavimui, kada nėra aktualios elementų veikimo taisyklės detaliame lygmenyje.

2. Modeliavimas agentais

Šiame darbe kuriant agentų modelį remtasi [Mik14] ir [Sil16] darbais. Ankstesni darbai nagrinėjo bendrą sistemos ir jos būsenų modelaivimą naudojant agentus. Tuo tarpu šiame darbe kuriamas modelis papildomai nagrinėja klaidų valdymą elementuose, to pasekoje reikalingos apibrėžti sudetingesnės elementų taisyklės. Siekiant įgyvendinti tikslus modelyje yra naudojamos struktūros aprašomos tiek kaip agentai ir tiek kaip įprastiniai elementai. Modelyje yra naudojami 2 skirtingų tipų agentai:

- Aplinkos agentai - tai yra agentai kurie modeliuoja aplinką, šiuo atveju tai yra sistemos naudotojai. Šiais agentais yra modeliuojami išoriniai sistemos kvietimai. Šie agentai nesidalija savybėmis su elementiniais agentais. Aplinkos agentai renka atsakymus apie sėkmingai arba nesėkmingai įvykdytas užklausas ir kitą su vykdymo susijusią statistiką.

- Agentai elementai - tai yra agentai, kurie modeliuoja sistemos elementus. Jie yra sužadinami aplinkos agentų ir tada sąveikauja su kitais agentais elementais, o baigę darbą pateikia atsakymą aplinkos agentui.

Šiame darbe agentas yra suvokiamas kaip elementas kuris gali autonomiškai priimti sprendimus, kokius veiksmus jam reikia atlikti, remiantis pasiekiamais aplinkos duomenimis. Iš realizacijos perspektyvos, agentas yra specialaus tipo elementas, turintis metodą *Ženk* kuriuo yra įgyvendinamas simuliacijos žingsnis.

Kitaip nei [Mik14], tokie elementai kaip serveris nėra modeliuojami pasitelkiant agentus, nes jie neturi agentinių sąvybių - jie nereaguoja į aplinką ir patys iš savo turimų žinių nepriiminėja sprendimų, jie tik reaguoja į tiesiogines kitų agentų komandas.

Elemento tipo agentu galima modeliuoti programų sistemų elementą nepriklausomai nuo abstrakcijos lygio, svarbiausia, kad tarp visų elementų tipo agentų modeliuojamas abstrakcijos lygis butų toks pats. Elemento agentu gali būti visa programų sistema ir tada būtų modeliuojami ryšiai ir klaidų valdymas bendraujant tarp skirtingų sistemų, kas būtų vieno iš aukščiausių abstrakcijos lygių modeliavimas. Taip pat šiuo modeliu galima modeliuoti žemesnius abstrakcijos lygius, kaip komunikaciją tarp skirtingų klasių.

3. Modelis

Elemento agento parametrus (properties) galima skirstyti į 2 kategorijas - konfiguracinės sąvybės ir vykdymo laiko būsenos parametrus.

Konfiguraciniai parametrai reikalingi nusakyti kaip elementas elgsis ir kaip priiminės sprendimus. Konfiguraciniai parametrai turi būti tokie, kad juos keičiant, būtų galima sumodeliuoti betkokio tipo reikalingą elementą.

Vykdymo laiko parametrai yra reikalingi vykdymo laiko būsenai modeliuoti - išsaugoti loginį veiksmų vientisumą tarp simuliacijos žingsnių. Sukūrus elemento agento egzempliorių konfiguracinės savybės nekinta, o vykdymo laiko savybės kinta.

Taip pat yra pateikiami išskirti parametrai, kurie yra naudojami modeliuojant klaidų valdymo metodus. Šie parametrai taip pat susideda iš konfigūracinių ir vykdymo laiko parametrų.

3.1. Klaidų modeliavimas

Klaidos yra nenumatytas programų sistemos būsenos pasikeitimas [SLK08], dėl kurio programų sistema nebegali korektiškai veikti. Klaidos modeliuojamos nustatant parametrą, kuris žymi tikimybę, kad vykdant simuliacijos žingsnį įvyks klaida. Daroma prielaida, kad sistemose klaidos visiškai atsitiktinai neįvyksta, - jeigu tartume, kad klaidos tikimybė atliekant užklausą yra 1%, tai mažai tikėtina, kad paėmus nuoseklią 100 užklausų rezultatų imtį būtų po 1 klaidą. Dažniausiai klaidos yra sukeliamos kažkokio pasikeitimo duomenyse, konfigūracijoje ar kokiame kitame elemento aplinkos atribute, ir klaidos yra vykdomos tol, kol elementas neišeina iš šios būsenos. Galima numanyti, kad elementas patenka į būseną, kurios metu negali veikti korektiškai atsitiktinai. Todėl klaidų modeliavimas elemente, kada jos yra labiau tikėtinos apibrėžtuose perioduose labiau atspindėtų realybę.

Siekiant taip modeliuoti klaidas yra įvedama ir naudojama koncepcija *sugedimas*. Tai reiškia, kad elemento tipo agentas turi loginio tipo parametrą su reikšmėmis taip arba ne, kuris nusako ar elemento agentas yra sugedęs. *Sugedęs* elemento agentas modeliuoja elementą su būsena, kada jis negali korektiškai apdoroti užklausų. Priklausomai nuo to ar elemento agentas yra sugedęs ar ne, yra naudojami kitokie koeficientai (kitokios tikimybės), kad elemento grąžins klaidos atsakymą. Norint prasmingai sumodeliuoti klaidas, kai elemento agentas nėra sugedimo būsenoje, tikimybė, kad jam pateiktos užklausos atsakymas pasibaigs klaida turi būti labai mažas, artimas 0, o agentui sugedus jis turėtų būti aukštas, - nuo kelių procentų iki 100%, taip, kad žymi dalis užklausų pasibaigtų klaida.

Kitas klaidų tipas, kuris yra modelyje, yra kylantis dėl nepakankamų modeliuojamo serverio resursų arba blogos konfiguracijos. Jeigu elementas negali pasiekti nei vieno elemento, kurį jam yra būtina išvkiesti apibrėžtą laiko tarpą, laukiantysis elementas grąžina klaidos atsakymą. Tokie atvejai įvyksta, kada yra blogai nurodytas reikalingo išvkiesti elemento tipas ir tokių iš viso nėra, arba visi reikalingi elementai yra išjungti, arba visi reikalingi elementai dirba.

3.2. Konfigūraciniai parametrai

- *Tipas* - agento elemento tipas. Naudojamas siekiant sugrupuoti tokius pačius elementus. Jeigu yra keli egzemplioriai to pačio elemento, paprastai kviečiantysis elementas negali pasirinkti kuris konkretus egzempliorius bus iškvietas. Šia savybe yra nurodomas tipas, ir kai elementas nori kviesti kitą elementą, pagal tipą jam yra atsitiktinai atrenkamas egzempliorius, kuris bus iškviestas.
- *KlaidosTikimybė* - tikimybė, kad elemento darbo žingsio metu įvyks klaida [0..1].
- *GalimiKviestiElementai* - elementų tipai (kurie yra apibrėžti su savybe *tipas*), kuriuos gali šis elementas kviesti.
- *MaksimalusKviečiamųElementųKiekis* - minimalus elementų kiekis, kuris turi būti iškviestas elemento darbo metu.
- *MinimalusKviečiamųElementųKiekis* - maksimalus elementų kiekis, kuris gali būti iškviestas elemento darbo metu.
- *Timeout* - po kiek žingsnių yra timeoutinama. Tai įvyksta tuo atveju, kada elementas dar turi neiškviestų elementų, kuriuos jis turi iškviesti, kad įvykdytų verslo transakciją, tačiau negali pasiekti laisvo jam reikalingo elemento egzemplioriaus.
- *TikimybėSugesti* - Tikimybė, kad elementas simuliacijos žingsnio metu suges. Sugedimas yra speciali elemento būsena, kurios metu elementas turi didesnę klaidos tikimybę nei įprastai.
- *TikimybėSusitaisyti* - tikimybė, kad elementui esant sugedusiam, jis susitaisys, tai yra, jo klaidų sukėlimo tikimybė grįš į pradinę vertę.
- *KlaidosTikimybėSugedus* - tikimybė, kad kvietimas baigsis su klaida, kai elementas yra sugedimo būsenoje.
- *VykdymoTrukmė* - kiek laiko trunka elemento vykdymas. Nurodomas tik konkrečiai šio elemento darbui reikalingas laikas, neatsižvelgiant į jo kviečiamų elementų vykdymo laiką.
- *ArKritinis* - tikimybė, kad elemento įvykdymas yra privalomas verslo transakcijos sėkmei. Jeigu elementas atsako su klaida, ar ir verslo transakcija atsakymas bus nesėkmingas.
- *TikimybėKviestiKitusElementus* - tikimybė [0..1], kad elementas bandys kviesti kitą elementą. Parametras naudojas modeliuojant įvykiais grįsto tipo architektūras, kada tas pats elementas, kuris apdoroja įvyki, gal sukurti kitą įvykį. Pagal nutylėjimą reikšmė yra 1.
- *KlaidosValdymoBūdas* - *enum* tipo parametras, kuris nurodo, koks klaidos valdymo būdas turi būti naudojamas, jeigu šio elemento iškviestas kitas elementas grąžina klaidos atsakymą.
- *Serveris* - nuoroda į serverio resursą, kuriame šis elementas turi veikti. Norėdamas pradėti darbą, elementas turi užtikrinti, kad šis serveris gali alokuoti jam reikalingus resursus, o baigdamas darbą, elementas turi pranešti serveriui, kad šis atlaisvintų jo resursus.

3.3. Vidiniai elemento parametrai, leidžiantys valdyti elemento būseną vykdymo metu

- *VykdymoSkaitliukas* - kiek žingsnių, elemento darbo laiko šiame kvietime yra įvykdyta (žr. konfiguracinį parametrą *VykdymoTrukmė*)
- *SpecialausDarboSkaitliukas* - skaitkliukas skirtas skaičiuoti darbui, kuris yra priskiriamas tam tikrais atvejais. Skiriasi nuo *VykdymoSkaitliukas* tuom, kad šis yra vykdomas pačioje elemento darbo žingsnio pradžioje ir nėra būtinas.
- *TimeoutSkaitliukas* - skirtas sekti, ar nesibaigė laukimui skirtas laikas, kai yra laukiama kol kuris nors iš reikalingų iškviesti elementų taps prieinamas.
- *IškviestiElementai* - sąrašas *IškviestasElementas* tipo elementų, atspindinčių elementus kurie perėjo iš laukiamos iškviesti būsenos ir jau yra šio elemento iškviesti. Detalesnis aprašymas pateikiamas skyriuje „Elementų klaidų valdymo metodika“.
- *Kvietėjas* - elementas, kuris iškvietė šį elementą. Reikalinga, kad būtų žinoma kokiam elementi pateikti atsakymą.
- *ArDirba* - nusako ar elementas šiuo metu jau yra iškviestas kito elemento ir dirba. Naudojamas tam, kad kiti elementai neišvkiestų pakartotinai šiuo metu jau išvkiesto elemento.
- *ArIšjungtas* - požymis, kad elementas yra išjungtas ir kad jis neturėtų būti kviečiamas kitų elementų.
- *ArKarantine* - požymis, kad elementas šiuo metu yra karantine. Elementas karantine nėra išjungtas, bet kiti elementai nesikreips į elementą esantį karantine.

3.4. Mašinos resursų modeliavimas

Kompiuterio (mašinos), kuriame teoriškai veikia modeliuojama sistema, resursus modeliuoti yra aktualu, nes skirtingi klaidų valdymo metodai, turi skirtingus poreikius mašinos resursams. Modeliuojant nustatomas resursų kiekis gali leisti teisingiau parinkti fizinę įrangą reikalingą leisti sistemai. Debesų kompiuterijos atveju, kai kaina yra skaičiuojama pagal sunaudotų resursų kiekį, galima būtų daryti geresnes prielaidas, kiek koks klaidų valdymo metodas kainuotų [AFG+10].

Modelyje yra modeliuojami procesoriaus, darbinės atminties, tinklo ir standžiojo disko resursai. Kompiuterio resursai modeliavimas yra įgyvendintas pasitelkiant elementą (ne-agentą) kuris yra pateikiamas agentams kaip konfiguracinis parametras. Elemento agentas, norėdamas pradėti darbą, kreipiasi į savo serverio resursą, prašydamas jam alokuoti tam tikrą keikį resursų, o baigęs darbą, kreipiasi į serverio resursą pranešdamas, kad šis gali atlaisvinti resursus. Jeigu elemento agentui kreipusis į serverio resursą, šis nustato, kad nėra užtektinai laisvų resursų reikalingų elemento agento darbo pradžiai, šis nėra iškviečiamas ir yra bandoma jį išvkiesti kitą žingsnį, tikintis, kad atsirado užtektinai laisvų resursų.

3.5. Aplinkos modeliavimas

Aplinka modeliuojama specialiu aplinkos agento tipu. Aplinkos agentas skiriasi nuo elementinių agentų tuom, kad jis negali būti iškviestas kitų agentų ir turi specialius parametrus, kuriais apibrėžiama kaip jis iškvietinės elementinius agentus. Parametrai nusako kokių kitų tipų agentus elementas turi kviesti ir kokia intervalu jie bus kviečiami. Remiantis [Mik14] darbu, naudojamos 2 elementų kvietimo strategijos: atsitiktinai pagal tikimybės parametrą ir pastovūs kvietimai tam tikru intervalu.

Aplinkos agento naudojami parametrai:

- *GalimiKviestiElementai* - galimų kviesti elementų tipų (žr. elementų agentų parametrą *Tipas*) sąrašas.
- *TikimybėIškviestiElementą* - vertė tarp 0 ir 1 nusakanti tikimybę, kad simuliacijos žingsnio metu turi būti iškviestas elementas.
- *KvietimoStrategija* - kaip bus kviečiami agentai, galimos reikšmės - *Atsitiktinai* arba *Periodiškai*

Kadangi aplinka irgi yra simuliuojama agentu, - aplinkos agente yra kviečiamas *Ženk* metodas, kuriuo yra atliekamas vienas simuliacijos žingsnis. Veiksmai ir jų tvarka, kas yra vykdomo *Ženk* metodo metu:

1. Sugeneruojamas atsitiktinis dydis tarp 0 ir 1
2. Patikrinamas ar sugeneruotas dydis yra didesnis už *TikimybėIškviestiElementą* parametro reikšmę. Jeigu taip - reiškiasi kad nebuvo pataikyta į teigiamą tikimybės rėžį ir žingsnio simuliavimas yra baigiamas.
3. Išrenkamas atsitiktinis elemento tipas iš *GalimiKviestiElementai* sąrašo. Ir bandomas gauti laisvas jo egzempliorius.
4. Jeigu laisvo egzemplioriaus nepavyksta gauti - žingsnio simuliavimas yra baigiamas.
5. Iškviečiamas gautas elemento egzempliorius kreipiantis į jo metodą *kviesti*, jeigu elementas nėra karantine; jeigu elementas yra karantine ir jis nėra kritinis, tada yra tęsiama simuliacija toliau; priešingu atveju yra atsakoma kvietėjui su klaida. Statistikos elemente pradedama nauja sesija, jai suteikiamas unikalus identifikatorius.

3.6. Elementų veikimo modeliavimo metodika

Elementai turi 3 viešai prieinamus metodus, kurie yra naudojami jo darbo metu: metodas iškviesti (inicijuoti darbą) elementą, metodas priimti atsakymą iš kito elemento ir metodas įvykdyti simuliacijos žingsnio darbui. *Kviesti* metodas yra kviečiamas, kai vienas elementas kviečia kitą elementą. *Kviesti* metodo tikslas yra atstatyti elemento būseną į pradinę ir paruošti duomenis, kurie bus naudojami šios užklausos metu. Metodas *Ženk* yra kviečiamas kiekvieno žingsnio metu

ir juo yra modeliuojama elemento veikla simuliacijos žingsnio metu. Metodas *PateiktiAtsakyma* yra kviečiamas kai agentas nori atsakyti jį iškvietusiam agentui.

- Veiksmai ir taisyklių patikrinimai eilės tvarka, kurie yra vykdomi metodo *kviesti* metu:

    1. Jeigu *ArDirba* savybės reikšmė yra *tiesa*, atsakoma, kad kvietimas yra negalimas
    2. Bandoma alokuoti elementui reikalingus resursus kreipiantis į serverį nurodytą *Serveris* parametre. Jeigu resursų nepavyksta alokuoti - atsakoma, kad kvietimas negalimas.
    3. *ArDirba* savybė nustatoma į *tiesa*
    4. *VykdymoSkaitliukas* reikšmė nustatoma į 0
    5. *Kvietėjas* reikšmė nustatoma į iškvietusį elementą
    6. Gaunamas atsitiktinis skaičius *X* iš rėžio tarp minimalios ir maksimalios galimų kviesti elementų reikšmių.
    7. Jeigu *TikimybėKviestiKitusElementus* yra mažesnė už 1, tada yra sugeneruojamas atsitiktinis dydis ir palyginamas su *TikimybėKviestiKitusElementus* reikšme; jeigus sugeneruotas dydis yra didesnis, tada gražinama tuščia kviečiamų elementų aibė; priešingu atveju yra išrenkamami *X* elementų iš galimų kviesti elementų aibės į kviečiamų elementų aibę, leidžiant pasikartojimus ir išsaugant juos į *KviečiamiElementai*.

  Veiksmai vykdomi elemento agento simuliacijos žingsnio metu:

    1. Jeigu parametro *ArIšjungtas* reikšmė yra *tiesa* sugeneruojamas atsitiktinis dydis ir palyginamas su parametro *TikimybeSusitaisyte* reikšme. Jeigu palyginimo sąlyga tenkinama yra inicijuojamas elemento susitaisymas: parametro *ArIšjungtas* reikšmė yra nustatoma į *netiesa* ir parametro *KlaidosTikimybe* reikšmė nustatoma į jos pradinę reikšmę (kokia buvo sukūrus elemento agentą).
    2. Patikrinama parametro *ArDirba* reikšmė; jeigu ji yra *netiesa* - agento darbas yra baigiamas šio simuliacijos žingsnio metu.
    3. Jeigu *SpecialausDarboSkaitliukas* parametro reikšmė yra didesnė už nulį, tada jo reikšmė yra sumažinama vienetu ir baigiamas elemento darbas šio simuliacijos žingsnio metu.
    4. Patikrinama tarp *IškviestiElementai* parametro elementų ar egzistuoja bent vienas, kurio parametrai tenkina sąlygą:

       $$ArIškviestas \wedge ArAtsakėSuKlaida \wedge \neg ArKlaidaApdorota$$

       Jeigu atsiranda elementas tenkinantis šią sąlgygą, jo elementui iškviečiamas klaidos valdymo metodas (žr. skyrių „Elementų klaidų valdymo metodika“). Baigiamas elemento darbas šio simuliacijos žingsnio metu.

5. Patikrinama tarp *IškviestiElementai* parametro elementų ar egzistuoja bent vienas, kurio parametrai tenkina sąlygą:

$$ArIškviestas \wedge ArKlaidaApdorota \wedge ArAtsakėSuKlaida$$

Jeigu egzistuoja bent vienas toks elementas, tai su pirmuoju iš jų tai elementas pateikia jį iškvietusiam elementui atsakymą, kad įvyko klaida ir baigia darbą.

6. Patikrinama sąlyga:

$$\exists(Elementas.ArDirba \wedge ArIškviestas \wedge \neg ArAtsakė)$$

Jeigu sąlygą yra patenkinama, reiškiasi, kad yra iškviestas elementas kuris šiuo metu vykdo darbą ir reikia laukti jo darbo pabaigos. Baigiamas agento darbas šio simulacijos žingsnio metu.

7. Patikrinama sąlygą:

$$TimeoutSkaitliukas > timeout$$

Jeigu ji yra patenkinama, vadinasi buvo pasiektas *timeout* periodas. Atsakoma kvietusiam elementui su klaidos atsakymu, baigiamas šio elemento darbas.

8. Patikrinama sąlyga:

$$\exists(Elementas = \emptyset \vee (\neg ArIškviestas \wedge \neg ArDirba))$$

Jeigu sąlyga yra tenkinama, tai su pirmuoju elementu tenkinančių salygą yra atliekama tokie veiksmai:

   (a) *Elementas* parametrui yra bandoma gauti ir priskirti elementą pagal *Tipas* reikšmę.
   (b) Jeigu elemento gauti nepavyksta - *TimeoutSkaitliukas* reikšmė yra pakeliama vienetu ir baigiamas agento darbas šio simuliacijos žingsnio metu.
   (c) Kviečiamas elementas nurodytas *Elementas* parametre.
   (d) Jeigu kvietimas nepavyksta - *TimeoutSkaitliukas* reikšmė yra pakeliama vienetu ir baigiamas agento darbas šio simuliacijos žingsnio metu.
   (e) *ArIškviestas* reikšmė nustatoma į *tiesa*.
   (f) *TimeoutSkaitliukas* reikšmė nustatoma į 0.
   (g) Baigiamas agento darbas šio simuliacijos žingsnio metu.

9. Patikrinama sąlyga:

$$VykdymoSkaitliukas < VykdymoTrukme$$

Jeigu sąlyga yra patenkinama tada *VykdymoSkaitliukas* reikšmė yra padidinama

1 ir agentas pabando sugesti generuodamas atsitiktinį dydį ir jį lygindamas su *TikimybeSugesti* parametro reikšme.

10. Atsakoma kvietusiam elementui su reikšme sugeneruota atsižvelgiant į *KlaidosTikimybe* parametro reikšmę.
11. Baigiamas elemento darbas.

1. Jeigu *ArIšjungtas* reikšmė yra *tiesa* (kas reiškia kad elementas yra *sugedęs*) tada elementas yra *sutaisomas* (*ArIšjungtas* savybės reikšmė nustatoma į *netiesa* ir *KlaidosTikimybė* reikšmė nustatoma į tokią, kokia buvo elemento sukūrimo metu)
2. Jeigu *ArDirba* savybės reikšmė yra *netiesa* elemento agento darbas šio žingsnio metu yra baigiamas.

Darbo baigimui yra naudojami 2 skirtingi terminai - vienas nusako agento darbo baigimą, o kitas nusako elemento darbo baigimą. Agento darbo baigimas reikškiasi, kad yra pabaigiamas *Ženk* metodo vykdymas. Elemento darbo baigimas reiškia pasikeitimus elemento būsenoje:

- *ArDirba* parametro reikšmė nustatoma į *netiesa*.
- Atitinkamame serveriui yra pranešama, kad reikia atlaisvinti šio elemento naudojamus resursus.
- *VykdymoSkaitliukas* reikšmė nustatoma į 0.
- *TimeoutSkaitliukas* reikšmė nustatoma į 0.
- Pašalinami elementai iš *IškviestiElementai* sąrašo.

Veiksmai vykdomi *PateiktiAtsakyma* metodo metu:

1. Elemento, kuriam yra atsakoma, iškviestų elementų sąraše atsakantį elementą atitinkančiame įraše pažymima *ArAtsakė* kaip *tiesa*
2. Elemento, kuriam yra atsakoma, iškviestų elementų sąraše atsakantį elementą atitinkančiame įraše pažymima *ArAtsakėSuKlaida* atitinkamai į tai, koks buvo atsakymas.

3.7. Elementų klaidų valdymo metodika

Klaidų valdymui yra sukurti vykdymo laiko parametrai, kurie kontroliuoja, kokie veiksmai yra atliekami su iškviestais elementais. Šių parametrų reikšmių kombinavimas elemento agento simuliacijos žingsio metu leidžia suderinti kitų elementų kvietimus ir įgyvendinti klaidų valdymo metodus. Kvietimo ir klaidų valdymo modeliavimui naudojami parametrai:

- *Tipas* - iškviesto elemento tipas.
- *Elementas* - nuoroda į iškviestą elementą.

- *ArIškviestas* - ar elementas jau buvo iškviestas.
- *ArAtsakė* - ar elementas jau pateikė atsakymą į užklausą.
- *ArAtsakėSuKlaida* - ar atsakymas buvo su klaida.
- *ArKlaidaApdorota* - ar jau buvo kvietas klaidų valdymo metodas įvykusiai klaida apdoroti.
- *KlaiduValdymoBūdas* - nuoroda į objektą, kuris valdo elemente įvykusias klaidas.

Klaidų valdymo metodai yra realizuojami kaip objektai turintys metodą *ValdytiKlaida*. Kadangi visi klaidų valdymo metodai yra iškviečiami kreipiantis į metodą *ValdytiKlaidą*, juos galima laisvai keisti, nekeičiant pačio elemento įgyvendinimo. Klaidos valdymo būdo metodika yra aprašoma kaip veiksmai, kurie yra atliekami *ValdytiKlaidą* metode. Metodas *ValdytiKlaidą* yra kviečiamas iš elemento agento *Ženk* metodo, kada yra tenkinama sąlyga:

$$\{x \in \mathit{KviečiamiElementai} \mid \mathit{x.ArAtsakė} \wedge \mathit{x.ArAtsakėSuKlaida} \neg \mathit{x.ArKlaidaApdorota}\}$$

Jeigu klaidos valdymo metodui yra reikalingi specialūs parametrai, tai jie yra saugomi klaidų valdymo metodo objekte. Toliau pateikiami veiksmai kurie turi būti atlikti *ValdytiKlaidą* metode, priklausomai nuo to, koks yra naudojamas klaidų valdymo būdas. Apostrofu yra žymima nauja reikšmė, kuri tampa po metodo įvykdymo.

3.7.1. Propagavimas

$$\mathit{ArKlaidaApdorota}' = \mathit{tiesa}$$

3.7.2. Atjungimas

$$\begin{aligned}
\mathit{Elementas.ArIšjungtas}' &= \mathit{tiesa} \\
\mathit{ArAtsakėSuKlaida}' &= \mathit{netiesa} \\
\mathit{ArIškviestas}' &= \mathit{netiesa} \\
\mathit{ArKlaidaApdorota}' &= \mathit{netiesa} \\
\mathit{ArKlaida}' &= \mathit{netiesa} \\
\mathit{ArAtsalė}' &= \mathit{netiesa}
\end{aligned}$$

3.7.3. Karotjimas

Šiame metode yra naudojamas šiam metodui specialūs parametrai:

- *KartojimųKiekis* - kiek pakartotinai kartų bus bandoma iškviesti klaida atsakantį elementą.

- *PakartojimųSkaitliukas* - kiek kartų buvo bandyta iškviesti klaida atsakantį elementą.

Valdant klaidą yra patikrinama sąlyga siekiant patikrinti ar ji yra tenkinama. Jeigu tenkinama sąlyga:

$$PakartojimuSkaitliukas < KartojimuKiekis$$

tada atliekami veiksmai:

$$\begin{aligned} PakaratojimųSkaitliukas' &= PakartojimųSkaitliukas + 1 \\ ArAtsakėSuKlaida' &= netiesa \\ ArIškviestas' &= netiesa \\ ArKlaidaApdorota' &= netiesa \\ ArKlaida' &= netiesa \\ ArAtsakė' &= netiesa \end{aligned}$$

priešingu atveju atliekama:

$$ArKlaidaApdorota' = tiesa$$

3.7.4. Klaidų nutildymas

$$\begin{aligned} ArAtsakėSuKlaida' &= netiesa \\ ArKlaidaApdorota' &= tiesa \\ ArKlaida' &= netiesa \end{aligned}$$

3.7.5. Mirusių žinučių eilė

Šiam klaidų valdymo būdui naudojama papildoma globali struktūra: mirusiųjų žinučių eilė. Apdorojant klaidą prie *MirusiųjųŽinučiųEilė* elemento pridedamas nesėkmingai išvkiestas elementas. Taip pat nustatoma:

$$ArKlaidaApdorota' = tiesa$$

Vėliau, galima gauti nesėkmingai kviestus elementus iš *MirusiųjųŽinučiųEilės* ir juos pakartotinai kviesti.

3.7.6. Atstatymo taškai

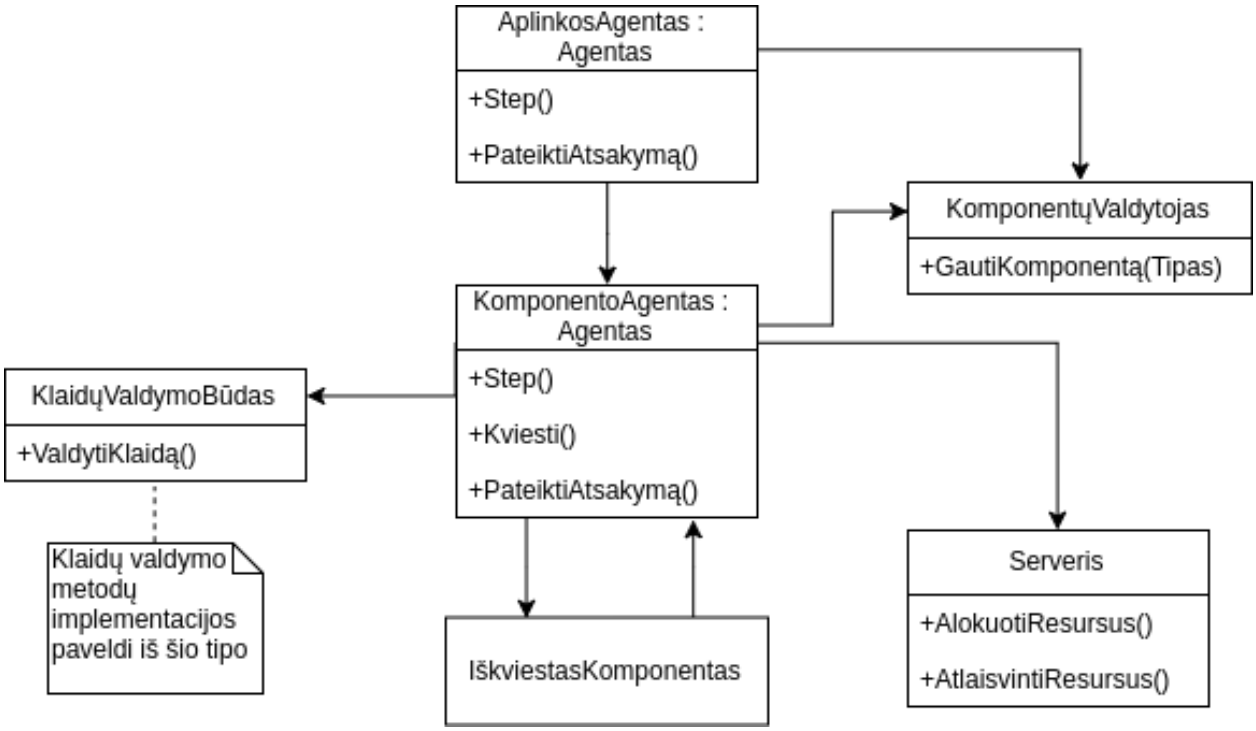
Šiame metode yra naudojami šiam metodui specialūs parametrai:

- *ĮkėlimoTrukmė* - kiek simuliacijos žingsnių bus skirta elemento atstatymui į pradinį tašką. Elemento atsatymas į tam tikrą būseną turėtų užimti kažkokį kiekį laiko, tam laukui apibrėžti naudojamas šis parametras.

Veiksmai atliekami bandant valdyti klaidą:

$$Elementas.SpecialausDarboSkaitliukas' = ĮkėlimoTrukmė$$
$$ArAtsakėSuKlaida' = netiesa$$
$$ArIškviestas' = netiesa$$
$$ArKlaidaApdorota' = tiesa$$
$$ArKlaida' = netiesa$$
$$ArAtsakė' = netiesa$$

3.8. Modelio struktūra

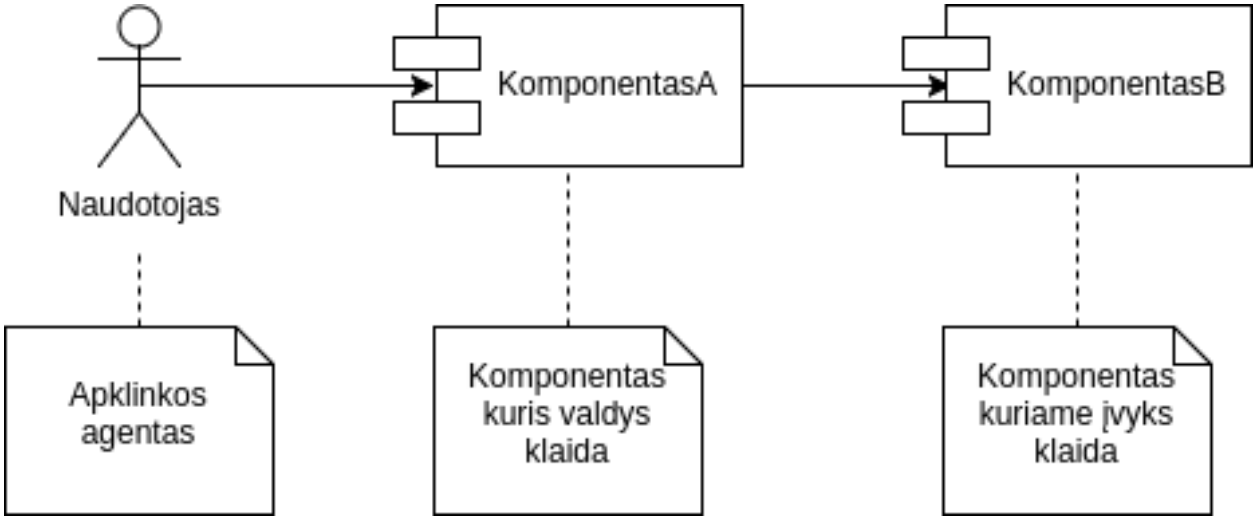


11 pav. Modelio struktūra

Bendrinė modelio struktūra pavaizduota 11 paveikslėlyje. *AplinkosAgentas* atlieka kvietimus *ElementoAgentas*, taip pradėdamas sesiją. *ElementoAgentas* ir *AplinkosAgentas* naudojasi *ElementųValdytojas* objektu, kad gautų jiem reikalingo tipo agento egzempliorių, kurį galėtų iškviesti. *ElementoAgentas* turi jam priskirtą *Serverį*, kurio resursais naudojasi, bei *KlaidųValdymoBūdas* tipo objektą, kuriame yra įgyvendinta metodika, kaip reikia valdyti klaidą, kurį įvyksta kviečiant elementus.

3.9. Modelio veikimo pavyzdzys

Siekiant pavaizduoti modelio veikimą, aprašomas pavyzdys, kaip keičiasi modelio būsenos, kai įvyksta klaida ir ji yra valdoma kartojimo metodu. Modeliuojama elementų struktūra susideda iš ElementoA ir ElementoB (struktūra pavaizduota 12 paveikslėlyje). Nagrinėjamame atvejyje aplinkos agentas iškviečia ElementąA, šis iškviečia ElementąB, kuriame įvyksta klaida. Įvykus klaida ElementasA ją bando suvaldyti atlikdamas kartojimą.



12 pav. Pavyzdinio modelio struktūra

Veiksmai ir būsenų pasikeitimai įvykstantys simuliuojant aprašytą scenarijų (1 sąrašo punktas atspindi 1 simuliacijos žingsnį):

1. *AplinkosAgentas* kreipiasi į *ElementųValdytojas* prašydamas jam duoti *ElementasA* tipo kvietimui paruoštą egzempliorių ir tada kreipiaisi į textitElementasA naudojant jo metodą *Kviesti*. *ElementasA* kreipiasi į *Serveris*, kuriam pasako alokuoti elementui reikalingus resursus. Sėkmingai įvykus alokaciji *ElementasA* išsaugo aplinkos agento elementą, kaip jį iškvietusį elementą. Prideda *ElementasB* į *KviečiamiElementai* sąrašą. Nustato *ArDirba* parametro reikšmę į *taip*.

2. *ElementasA* nustato, kad nelaukia jokio kito elemento atsakymo ir nesimuliuoja jokio darbo atlikinėjimo, todėl patikrina elementus esančius *KviečiamiElementai* sąraše ir nustato, kad *ElementasB* yra neiškviestas, nes yra tenkinama sąlyga:

$$\begin{aligned} &\mathit{IškviestasElementas.Elementas} = \emptyset \\ \vee(\neg &\mathit{ArIškviestas} \wedge \neg \mathit{IškviestasElementas.Elementas.ArDirba}) \end{aligned}$$

   ElementasA kreipiasi į *ElementųValdytojas*, kad gautų laisvą *ElementasB* egzempliorių, tada kviečia jo metodą *Kviesti*. *ElementasB* atlieka resursų alokavimą serveryje, nustato savo kvietėją, pakeičia *ArDirba* parametro reikšmę į *taip*.

3. *ElementasA* laukia *ElementasB* atsakymo, todėl nedaro nieko kito. *ElementasB*, kadangi

neturi jokių kitų elementų kuriuos jam reikėtų kviesti, simuliuoja savo darbą ir inkrementoja *VykdymoSkaitliukas* reikšmę.

4. *ElementasA* laukia *ElementasB* atsakymo, todėl nedaro nieko kito. *ElementasB* *VykdymoSkaitliukas* reikšmė pasiekia *VykdymoTrukmė* reikšmę, todėl *ElementasB* skaitosi baigęs savo darbą ir bando sugeneruoti atsakymą *ElementuiA*. Pasitelkiant atsitiktinių dydžių generavimą *ElementasB* nustatoma, kad atsakymas bus klaidingas. Pateikiamas atsakymas *ElementasA* indikuojant, kad jis baigėsi klaida. *ElementasB* baigia darbą. *ElementasA* nustatoma ties iškviestu elementu *ElementasB* reikšmės *ArAtsakė* į *tiesa* ir *ArKlaida* į *tiesa*.

5. *ElementasA* tikrinant iškviestus elementus nustatoma, kad yra atsakęs elementas, kurio atsakymas yra su klaida, bei klaida yra neapdorota (*ArKlaidaApdorota* reikšmė yra *netiesa*). *ElementasA* kreipiasi į klaidų valdymo būdo metoda *ValdytiKlaidą*, kurį šiuo atveju implementuoja *ValdymasKartojant* klasė. Klaidų valdymo metodas pakeičia reikšmės *ArAtsakė* į *netiesa*, *ArIškviestas* į *netiesa*, *ArKlaidaApdorota* į *netiesa*, bei nustato savo pakartojimų skaitliuko reikšmes.

6. Klaidos valdymo būdas taip nustatė reikšmes, kad pagal taisykles kaip veikia *ElementoAgentas* egzempliorius *ElementasA*, nustatoma, kad *ElementasB* dar nebuvo iškviestas (vėl tenkinama praeito kvietimo žingsnyje aptarta sąlyga) ir yra iškviečiamas vėl (taip pat, kaip ir ankstesniame žingsnyje).

7. *ElementasA* laukia *ElementasB* atsakymo, todėl nedaro nieko kito. *ElementasB* neturėdamas jokių kitų elementų kuriuos reikėtų kviesti inkrementuoja savo *VykdymoSkaitliukas* reikšmę.

8. *ElementasB* vykdymo skaitliuko reikšmei pasiekus *VykdymoTrukmė* nurodytą reikšmę, jis vėl vykdo atsakymą jį kvietusiam elementui *ElementasA*, šį kartą sėkmingai, užbaigia savo darbą.

9. *ElementasA* nustato, kad visi jam reikalingi iškviesti elementai jau buvo iškviesti ir pradeda simuliuoti savo darbą inkrementuodamas *VydymoSkaitliukas* reikšmę.

10. Baigus darbo simuliavimui skirtą intervalą, *ElementasA* kreipiasi į *AplinkosAgentas* metodą *PateiktiAtsakymą* baigdamas darbą.

Šiame pavyzdyje nusakyta, kokie veiksmai yra atliekami tipiniame simuliacijos scenarijuje pasitelkiant šiame skyriuje aprašytą modelį, skirtą klaidų valdymo simuliacijai. Pavyzdyje yra parodoma, kaip klaidų valdymo būdas yra įgyvendinamas pasitelkiant elemento agento parametrus, pagal kuriuos elementas sprendžia kaip vykdyti veiklą.

4. Modelio patikrinimas

Šiame skyriuje yra aprašomas modelio patikrinimas. Patikrinimui atlikti buvo vykdomi simuliuojami sumodeliuoti scenarijai, kurie, taip pat, buvo atkartoti realioje programoje. Iškelta hipotezė, kad keičiant scenarijus, atsako laikai ir kiti rezultatai turėtų išlikti proporcingi tarp realios programos ir modelio rezultatų. Skyriuje „Modelio validavimas ir verifikavimas“ pateikti 3 aspektai pagal kuriuos reikėtų atlikti modelio patikrinimą: rezultatų lyginimas su realia sistema, rezultatų nuspėjamumas ir modelio elgsenos lyginimas su realia sistema.

4.1. Rezultatų palyginimas su realia sistema

Atlikti 3 palyginimo atvejai (3 scenarijai) su realia sistema. Panaudoti scenarijai:

- Scenarijus I - 3 elementai tipo ’A’, 3 elementai tipo ’B’. ’A’ tipo elementai kviečia ’B’ tipo elementus.
- Scenarijus II - 3 elementai tipo ’A’, 3 elementai tipo ’B’, 3 elementai tipo ’C’. ’A’ tipo elementai kviečia ’B’ tipo elementus, ’B’ tipo elementai kviečia ’C’ tipo elementus.
- Scenarijus III - 3 elementai tipo ’A’, 3 elementai tipo ’B’. ’A’ tipo elementai kviečia ’B’ tipo elementus. ’B’ tipo elementai turi 10% tikimybė pateikti klaidos atsakymą. ’A’ tipo elementai bandys pritaikyti „kartojimo“ klaidos valdymo metodą iki 5 kartų.

Patikrinimas atliktas sukuriant programą „.NET Core“ aplinkoje. Dėl to, kad modelis naudoja sutartinius matavimo vienetus laikui apskaičiuoti, nuspręsta patikrinimo tikslams naudoti *bcrypt* [PM99] algoritmą darbui simuliuoti, dėl jo ilgo vykdymo laiko. 50 modelio žingsnių (laiko vienetų) atitinka vieną 1 *bcrypt* algoritmo įvykdymą.

3 lentelė. 1 scenarijaus rezultatai

Programos trukmė	268ms
Modelio trukmė	201 žingsnis
Trukmių santykis	$268 \div 201 \approx 1{,}333$

4 lentelė. 2 scenarijaus rezultatai

Programos trukmė	391ms
Modelio trukmė	288 žingsnis
Trukmių santykis	$391 \div 288 \approx 1{,}357$

5 lentelė. 3 scenarijaus rezultatai

Programos trukmė	271ms
Modelio trukmė	208 žingsnis
Trukmių santykis	$271 \div 204 \approx 1{,}302$

Visų 3 scenarijų rezultatų koeficientai yra labai panašūs: paklaida mažesnė už 4,5%, todėl galima teigti, kad modelis validžiai modeliuoja realias programas, bent tokiuose paprastuose scenarijuos.

4.2. Rezultatų nuspėjamumas

Rezultatų nuspėjimą galima atlikti matematiškai suskaičiuojant numanomas metrikas, tada patikrinant ar reali sistema gauna atitinkamus rezultatus. Poskyryje „Rezultatų palyginimas su realia sistema“ Aptarti 3 scenarijai. Pateikiamas matematiniu būdu nuspėjamas rezultatas trečiajam minėto poskyrio scenarijui, - kada dalyvauja 2 elementų tipai ir klaidos valdomos kartojant.

Naudojami kintamieji: *LaikasA* - elemento A vykdymo laikas, *LaikasB* - elemento B vykdymo laikas, *KoefB* - suminė elemento B iškvietimų tikimybė.

$$LaikasA + LaikasB \times KoefB = Laikas \tag{1}$$

$$100 + 100 \times \sum_{i=1}^{5} i(P(A_1) \times P(A_2|A_1) \times ... \times P(A_i|A_{i-1} \cap ... \cap A_1)) = Laikas \tag{2}$$

$$100 \times 100 \times 1{,}11111 = Laikas \tag{3}$$

$$Laikas \approx 211 \tag{4}$$

Gaitą rezultatą palyginus su anksčiau skyriuje gautu modelio rezultatu matomas stebimas tik 1,5% skirtumas, todėl galima teigti, kad modelio rezultatai yra nuspėjami.

4.3. Elgsenos palyginimas

Modelio elgsena buvo aptarta skyriuje „Modelio veikimo pavyzdzys“. Kaip minėta pavyzdyje, galima matyti, kad modelis susideda iš elementų individualaus darbo ir sąveikos su kitais elementais, kas atsipindi tikrų programų veikimo principus.

5. Simuliavimas. Metodų lyginimas

Šiame skyriuje pateikia, kaip vykdomas simuliavimas pasitelkiant darbe apibrėžtą modelį ir kokiais kriterijais remiantis lyginami rezultatai.

5.1. Vertinimo kriterijai

Simuliavimo metu renkami duomenys apie kiekvieną įvykdytą užklausą iš aplinkos (kvietėjo) agento perspektyvos. Atliekant simuliacijas yra surenkami kiekybiniai duomenys, kurie po to bus naudojami išvadoms atlikti. Naudojant surinktus duomenis galima analizuoti, kokią įtaką klaidos valdymo metodų ar architektūros parinkimas daro sistemos kokybiniams kriterijams.

Siekiant įvertinti skirtingų klaidų valdymo metodų efektyvumą yra renkami duomenys kiek vidutiniškai laiko trunka užklausos, kiek iš viso yra įvykdoma užklausų ir kiek procentaliai užklausų baigiasi sėkmingai. Trukmė yra skaičiuojama kaip žingsnių skirtumas tarp žingsnio kurio metu aplinkos agentas iškvietė elementą ir tarp žingsnio kada jis gavo atsakymą iš to elemento. Sėkmingomis užklausomis yra laikomos tos užklausos, kuriose buvo sėkmingai įvykdyti visi kritiniai kvietimai (tai reiškia, kad visi elementai kurie buvo kviesti ir kurių požymio *ArKritinis* reikšmė buvo tiesa atsakė sėkmingai). Pirminis kriterijus yra kiek užklausų buvo atsakyta sėkmingai, tada koks buvo vykdymo laikas. Nors nekritiniai kvietimai yra neprivalomi užklausos sėkmei, visais atvejais yra stengiamasi jų įvykdyti kuo daugiau.

Duomenys, kurie yra renkami kiekvieno kvietimo modelio simuliacijos metu ir pagal kuriuos skirtingi klaidų valdymo metodai yra lyginami:

- Ar atsakyta kvietėjui sėkmingai - aplinkos agentui pateikiamas loginis atsakymas - *taip* arba *ne* apie tai ar buvo sėkmingai įvykdyta užklausa. Pagal tai kiek atsakė sėkmingai ir nesėkmingai, gale simuliacijos yra suskaičiuojamas procentalus vidurkis.

- Per kiek laiko atsakyta kvietėjui - per kiek simuliacijos žingsnių aplinkos agentui buvo pateiktas atsakymas iš išvkiesto elemento.

5.2. Modelio paruošimas simuliacijai

Siekiant paruošti modelį simuliacijai, reikia sukurti visus reikalingus agentus bei kitus elementus ir sukurti reikalingus ryšius tarp jų. Sukūrus kiekvieną agentą, reikia jam sukurti ir priskirti klaidų valdymo metodo objektą. Priklausomai nuo to, kaip bus sukurti ryšiai tarp elementų ir kokie bus priskirti jų konifiguraciniai parametrai, galima sumodeliuoti skirtingus scenarijus.

Toliau pateikiamas pavyzdys, kaip sumodeliuoti primityvų scenarijų. Pavyzdinis scenarijus, panašiai kaip nagrinėtas skyriuje „Modelio veikimo pavyzdzys“, susideda iš 2 tipų elementų, tik šiuo atveju *ElementasB* yra 2 egzemplioriai. Vienas iš 2 egzempliorių visada grąžina klaidos atsakymą, o kitas egzempliorius visada veikia korektiškai.

Konfiguracija naudojama tokiam modeliui apibrėžti:
2 agentai su tipu „2“:

- *Tipas* = 2
- *VykdymoTrukmė* = 10
- Kiekvienu atveju skirtinga *KlaidosTikimybė* vertė:
    1. *KlaidosTikimybė* = 0
    2. *KlaidosTikimybė* = 1

1 agentas „1“ tipo:

1. *Tipas* = 1
2. *GalimiKviestiElementai* = $\left[2\right]$
3. *VykdymoTrukmė* = 10
4. 1 atveju *KlaidosTikimybė* = 0

Abejuose agentuose klaidos būtų nevaldomos.

Esant tokiai konfiguracijai yra nesunku nuspėti, kad klaidų tikimybė turėtų būti apie 50%, o vidutinis atsako laikas apie 17 simuliacijos žingsnių (2 žingsniai kvietimui, 10 žingsnių 1 tipo elemento ir 10/2 2 tipo elemento). Paleidus 100000 žingsnių simuliaciją su tokia konfiguracija gaunami rezultatai:

- Viso sėkmių: 1399
- Viso nesėkmių: 1342
- Vidutinis atsako laikas: 17,17

Šiuo atveju verslo ir naudotojo užklausų parametrai sutampa, nes nėra naudojamas joks klaidų valdymo būdas. Dėl šios priežasties jie yra nepateikiami.

5.3. Modelio generavimas

Pasinaudojant agentais grįsto modeliavimo savybėmis, buvo sukurta programa, kuri leidžia generuoti modelius simuliacijai. Generuojanti programa leidžia apibrėžti agentus nurodant jų konfigūracinius parametrus ir egzempliorių kiekį arba įterpiant iš anksto apibrėžtos struktūros architektūrinį šabloną, tai leidžia pakankamai greitai apibrėžti įvairių elementų ryšių ir savybių architektūras.

6. Klaidų valdymo metodų vertinimas

Šiame skyriuje pateikiamas klaidų valdymo metodų vertinimas, kuris atliekamas pasiltekiant duomenis, gautus vykdant simuliacijas naudojant anksčiau pateiktą modelį. Pasitelkiant 2 modeliavimo scenarijus, kuriuose yra modeliuojamos minimalistinės paslaugų ir įvykiais grįstos architektūros, siekiama nustatyti koreliaciją tarp modelių elementų parametrų, juose naudojamų klaidų valdymo metodų ir rezultatų. Šiuose 2 scenarijuose atliekamos simuliacijos keičiant kombinuotai arba individualiai klaidos tikimybės, tikimybės sugesti, klaidos tikimybės sugedus ir tikimybės sugesti reikšmes. Išanalizavus gautus rezultatus siekiant nustatyti, prie kokių parametrų ir kokioje architektūroje, kurie klaidų valdymo metodai yra geresni. Žinant kokie klaidų valdymo metodai labiau tinkami su kokiais parametrais, šios žinios pritaikomkos parenkant klaidų valdymo metodus sudetingame modelyje ir palyginami simuliacijų rezultatai, kai nėra pritaikytos žinios iš analizės parenkant klaidų valdymo metodus ir kai jos yra pritaikomos.

6.1. Modelių analizė

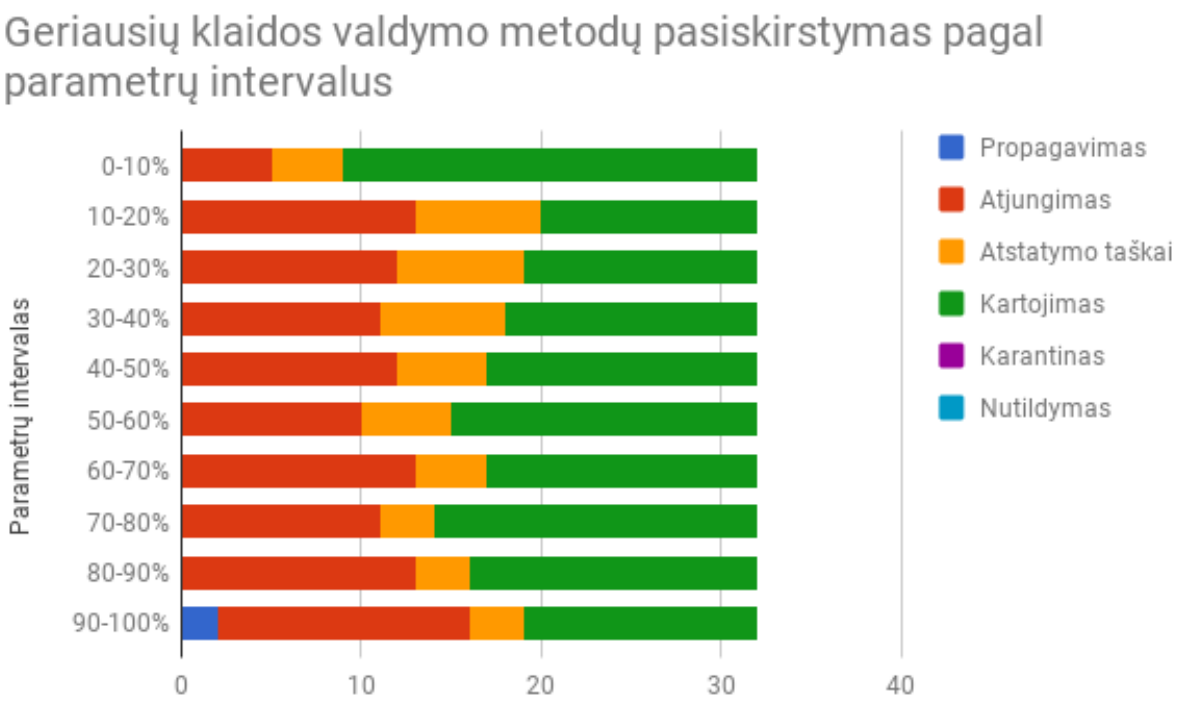
Įvykiais grįstos (schema **??** paveikslėlyje) ir paslaugų (schema **??** paveikslėlyje) architektūros atvejais atliktos eksperimentinės simuliacijos. Abiejų architektūrų apibrėžti taip, kad skirtingų elementų būtų pankankamai mažai, tam kad būtų galima lengviau nustatyti koreliacijas tarp elementų parametrų ir jų rezultatų. Taip pat abu modeliai suformuoti taip, kad būtų kiek galima ekvivalentiškesni atsižvelgiant į jų architektūrą, t.y. kad juose įvyktų toks pats kiekis kitų elementų kvietimų ir elementų pasiskirstymas priklausomai nuo jų parametrų būtų kiek galima vienodesnis, tik tai realizuojant kiekvienai architektūrai specifiniu būdu. Tai, kad architektūros yra labai panašios galima matyti iš simuliacijos rezultatų, kada yra naudojamas klaidų valdymo metodas jas propaguojant.

Eksperimentai atlikti įvykdant 100 000 žingsnių simuliacijas. Simuliuojami atvejai, kai parametro arba jų kombinacijų vertės yra didinamos 1% ir simuliuojant tada gautus modelius su visais tiriamais klaidų valdymo metodais. Pavyzdžiui, jeigu elementas turėjo 10% tikimybė įvykdti klaidai jo vykdymo metu, tada bus atlikta simulacija tą tikimybę pakėlus iki 11%, tada 12% ir t.t. įvykdant po 100 000 žingsnių simuliaciją su kiekvienu klaidų valdymo metodu. Parametrų reikšmės yra keliamos visame modelyje kartu vienu metu, tai reiškia, kad tiriant klaidos tikimybės įtaką yra iškarto pakeliama klaidos tikimybė 1% visiems elementams modelyje.

6.2. Modelių simuliacijos rezultatai

Buvo atliktos 30 skirtingų simuliacijų scenarijų. Jų rezultatai kaip sėkmingų užklausų dalies ir vidutinio vykdymo laikos grafikai yra pateikiami prieduose.

13 paveikslėlyje pateikiamas bendras geriausių klaidų vladymo metodų pasiskirstymas pagal parametrų reikšmių intervalus. Geriausių klaidų valdymo metodų pasiskirstymai esant vertinant tik tas simuliacijas kuriose buvo keičiamas konkretus parametras pateikiami prieduose esančiuose 16, 17, 18, 19 paveikslėliuose.



13 pav. Geriausių klaidų valdymo metodų pasiskirstymas pagal parametrų intervalus

Kartojimas dažniausiai užtikrina didžiausią sėkmės tikimybę (149 iš 300 13 paveikslėlyje pavaizduotų atvejų). Kartojimas gerai veikia tada, kada kviečiamas elementas turi mažą klaidos tikimybę, - tada kai 1 pakartojimo turėtų užtekti, kad pasiekti sėkmingą rezultatą. Kai yra mažos klaidos tikimybės (0-10%) kartojimas 21 iš 30 atvėjų duoda geriausius rezultatus. Tais atvejais, kada klaidos tikimybė yra didelė (dažniausiai tada, kada elementas yra sugedęs) kartojimas nevisada būna optimalus klaidų valdymo metodas, nes gali būti tik keliama vykdymo trukmė, galimai net nepasiekiant sėkmingo rezultato. Valdant klaidas kartojimu, kai yra didelė klaidos tikimybė kyla didėlio vykdymo laiko rizika, kai elementai yra išsidėstę keliais sluoksniai ir vis bando kartoti vienas kito užklausas. Tokiu atveju, jeigu yra vienas neveikiantis elementas kvietimų medžio apačioje, vykdymo laikas didės geometriškai, nes vienas elementas vis bandys kartoti kitą. Nors kartojimas dažniausiai užtikrina didžiausią sėkmės tikimybę, taip pat jis dažniausiai turi ir didžiausią vykdymo vykdymo laiką.

Kartojimas tinkamas:

- Kada klaidos tikimybė maža
- Elementas nėra linkęs sugesti

Kartojimas netinkamas:

- Elementai linkę sugesti
- Didelis kvietimų medis, kuriame visur naudojami kartojimai ir yra elementų su didele klaidos tikimybe

Propagavimas pasižymi mažiausia sėkmės tikimybe (žr. 13 paveikslėlį). Mažą sėkmės tikimybę sąlygoja tai, kad pats metodas niekaip klaidų nesprendžia, o gali tik padėti kitiems metodams tai padaryti, todėl jis priklausomas nuo to kokie klaidų valdymo metodai yra kvietimo hierarchijoje aukščiau jo. Jeigu kviečiami elementai yra nekritiniai propagavimas duos tokius pat rezultatus kaip ir karantinas ir nutildymas. Kritinių elementų kvietimo atveju propagavimas pats iš savęs klaidos tikimybe nusileistų karantinui ir nutildymui. Propagavimas turi pranašumą prieš karantinavimą, kad jis neišima kažkokio elemento tipo iš kvietimų medžio, taip, klaidos atveju, nesukurdamas papildomų klaidų, kurių ateityje butų galima išvengti. Propagavimą apsimoka naudoti tada, kada iš anksto galima nustatyti, kad elementui pačiam nepavyks išspręsti klaidos ir siekiant negaišti laiko tiesiog grąžinti vykdymą aukščiau kvietimo medžiu.

Propagavimas tinkamas:

- Kai žinoma, kad klaidos esamame elemente nebus išspręsta
- Kai žinoma, kad aukščiau kvietimo medyje esantys elementai turi optimalius klaidų valdymo metodus šio elemento šakai

Propagavimas nebus tinkamas visais kitais atvejais.

Atjungimas sėkmingumas koreliuoja su elementų tikimybe susitaisyti (žr. 19 paveikslėlį), kiek mažiau su klaidos tikimybe ir klaidos tikimybe sugedus. Prasčiausi rezultatai matomi matomi keičiant tikimybę sugesti. Atjungiant yra svarbu, kad atjunginėjami elementai turėtų užtektinai kitų egzempliorių, nes atjungus visus prieinamus egzempliorius bus visada gaunama klaida, todėl, kai elementai greitai susitaiso, juos galima išjunginėti, nes jie greitai grįš į prieinamų elementų aibę ir neturėtų būti jaučiamas jų egzempliorių trūkumas. Kada elementai turi žymiai didesnę tikimybę įvykti klaidai sugedus, nei įprastai, tokių elementų atjungimas duoda geresnius rezultatus. Keičiantis tikimybei sugesti, atjungimas duoda prastesnius rezultatus, nei kitų parametrų kitimo atvejais.

Atjungimas tinkamas:

- Esant santykinai didelei tikimybei susitaisyti.

Atjungimas netinkamas:

- Esant didelei tikimybei sugesti.

Atstatymo taškai tinkami kada elementas yra dažnai sugedęs - kai elemento tikimybė sugesti yra didelė. Priklausomai nuo to, kiek laiko užtrunka atstatyti elemento būsenai, šis metodas gali būti pranašesnis už kartojimą. Geri atstatymo taškų rezultatai matomi įvykiais grįstos architektūros atveju, kada keičiamos tikimybė sugesti ir klaidos tikimybė sugedus. Esant aukštai elemento tikimybei susitaisyti (žr. 19 paveikslėlį), atstatymo taškai nei viename iš scenarijų nebeduoda geriausių rezultatų.

Atstatymo taškai tinkami:

- Esant didelei tikimybei sugesti, ypač įvykiais grįstos architektūros atveju.

Atstatymo taškai netinkami:

- Ilgas atstatymo laikas palyginus su vykdymo laiku
- Esant didelei tikimybei susitaisyti.

Karantinas tinkamas tik tada, kada kviečiami elementai nėra kritiškai svarbūs užklausos sėkmei. Kai kvietimas yra kritiškai svarbus, karantinas nėra tinkamas klaidų valdymo būdas. Kai klaidos vyrauja tarp visų egzempliorių, t.y. jos būdingos pačiam elementui o ne vienam egzemplioriui.

Karantinas tinkamas:

- Kai elementas yra nekritinis
- Kai elementas turi aukštą klaidos tikimybę

Karantinas netinkamas:

- Kai elementas yra kritinis
- Kai elemento bendra vidutinė klaidos tikimybė (vidutinė klaidos tikimybė per laiko intervalą) maža

Nutildymas tinkamas tada, kada kviečiami elementai nėra kritiškai svarbūs užklausos sėkmei ir kai klaidos yra labiau atsitiktinės ir pavienės, nei stabilios ir prognozuojamos.

Nutildymas tinkamas:

- Kai elementas yra nekritinis
- Kai klaidos yra atsitiktinės

Nutildymas netinkamas:

- Kai elementas yra kritinis
- Kai tikimybė susitaisyti yra maža

6.3. Modelių analizės rezultatų pritaikymas didesniame modelyje

Siekiant patikrinti prieš tai padarytas išvadas gautas tiriant mažesnių modelių rezultatus, žinios bus pritaikytos parenkant klaidų valdymo metodus didesniame modelyje. Didesnis modelis susideda iš 50 skirtingų elementų tipų ir yra pavaizduotas 15 paveikslėlyje. Dauguma elementų ryšių yra paslaugų architektūros tipų, bet, taip pat, yra įvykiais grįstų architektūros elementų. Kiekvieno elemento yra tarp 40 ir 60 egzempliorių, o 30% elementų yra nekritiniai.

Pradžiai buvo įvykdyta simuliacija, kurioje buvo visuose elementuose taikomas propagavimo klaidų valdymo metodas, kas reiškė, kad klaidos niekur nebuvo valdomoas (rezultatai 6 lentelėje). Antram kontroliniam bandymui visiems elementams buvo nustatytas kartojimas kaip klaidų

valdymo metodas, nes anksčiau buvo nustatyta, kad kartojimas dažniausiai duoda santykinai geriausius rezultatus (rezultatai 7 lentelėje).

6 lentelė. Kontrolinės simuliacijos naudojant propagavimą rezultatai

Viso kvietimų	13274
Sėkmingų kvietimų %	39,2%
Vidutinis vykdymo laikas	180.06

7 lentelė. Kontrolinės simuliacijos naudojant kartojimą rezultatai

Viso kvietimų	8521
Sėkmingų kvietimų %	67,02%
Vidutinis vykdymo laikas	282

Apibrendrinant rezultatus (pateikta lentelėje) matoma, kad visur pritaikius kartojimą yra pasiekiami geresni rezultatai nei klaidų nevaldant (propaguojant). Lyginant propagavimą su kartojimu stebima didesnė sėkmingų užklausų dalis, didesnis vykdymo laikas ir mažesnis atliktų užklausų kiekis. Didesnis vykdymo laikas stebimas todėl, nes kartojimas trunka ilgiau nei tiesiog iškvietimas 1 kartą. Mažesnis užklausų skaičius sąlygojamas to, kad aplinkos agentai turi mažiau laisvų elementų kuriuos gali iškviesti, nes tie elementai jau dalyvauja ilgiau trunkančiose užklausose.

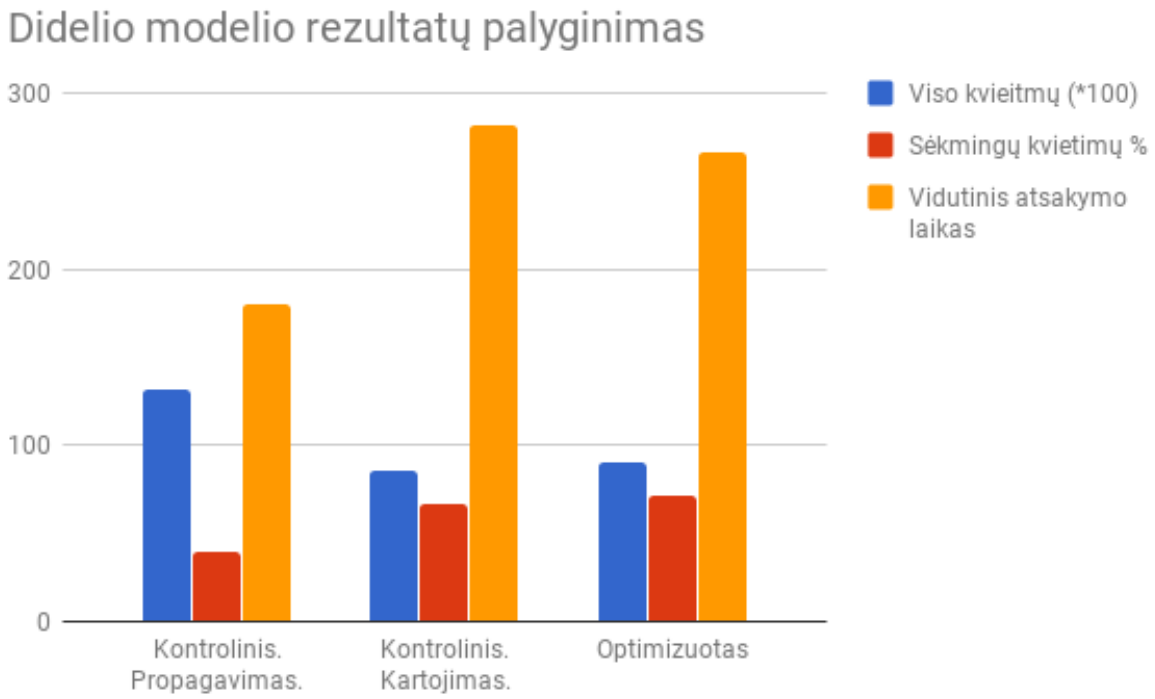
Siekiant atlikti klaidų valdymo metodų parinkimo optimizaciją, reikia įvertinti visus modelio elementus ir panaudojant prieš tai gautus rezultatus parinkti klaidų valdymo metodus subjektyviai kiekvienai situacijai. Klaidų valdymo metodų parinkimas vykdomas atsižvelgiant į tai, kokios yra kviečiamų elementų savybės. Apibendrinant anksčiau aprašytus pastebėjimus išskirtos kelios taisyklės, kuriomis bus remiamasi parenkant klaidų valdymo metodus. Šis taisyklių rinkinys, žinoma nėra baigtinis, bet yra užtektinas patikrinti atliktos analizės teisingumui.

- Jeigu klaidos tikimybė maža ir tikimybė sugesti maža arba klaidos tikimybė sugedus maža - klaidų valdymas kartojant
- Jeigu didelė tikimybė sugesti ir didelė klaidos tikimybė sugedus, bei vykdymo laikas ilgesnis už atstatymo laiką arba maža susitaisymo tikimybė - atstatymo taškai
- Jeigu elementas yra nekritinis ir jo klaidos bus tik pavienės - nutildymas
- Jeigu elementas yra nekritinis ir jo klaidos bus periodiškai pastovios - karantinas
- Jeigu klaidos tikimybė baža, bet didelė sugedus - atjungimas
- Jeigu elementui negalima pritaikyti jokių taisyklių - propagavimas

Taip parinkus klaidų valdymo metodus, buvo gautas toks klaidų valdymo metodų pasiskirstymas: kartojimas - 19, atstatymo taškai - 7, nutildymas - 2, karantinas - 2, atjungimas - 2. Specialiai parinkus klaidų valdymo metodus ir įvykdžius simuliaciją gauti rezultatai:

8 lentelė. Kontrolinės simuliacijos naudojant kartojimą rezultatai

Viso kvietimų	9036
Sėkmingų kvietimų %	71,76%
Vidutinis vykdymo laikas	266,19



14 pav. Didelio modelio simuliacijų rezultatai

Žvelgiant į optimizuoto ekperimento rezultatus 8 lentelėje ir į visų rezultatų apibendrinimą 14 paveikslėlyje matoma, kad optimizacija pagal surinktus pastebėjimus leido dar pagerinti rezultatus: įvykdyta daugiau kvietimų, užklausų vykdymo laikas buvo mažesnis ir daugiau užklausų baigėsi sėkme, nei tada, kai visi elementai naudoja kartojimo klaidų valdymo metodą.

Rezultatai

Magistrinio darbo tikslas buvo sukurti agentais grįsta modelį, kuriuo naudojantis būtų galima palyginti skirtingus klaidų valdymo metodus programų sistemose. Siekiant tikslo buvo gauti rezultatai.

- Sukurtas agentais grįstas modelis tinkamas klaidų modeliavimui:
  - Apibrėžti parametrai kurie yra aktualūs klaidų modeliavimui.
  - Apibrėžta metodika kaip modeliuoti klaidų valdymą.
  - Modelis patikrinas jį realizuojant ir rezultatus palyginant su analogiškų tikrų sistemų rezultatais.
- Atliktos simuliacijos su sukurtu modeliu:
  - Naudojant mažus įvykiais grįstų ir paslaugų architektūrų modelius nustatytos koreliacijos tarp modelio parametrų ir skirtingų klaidų valdymo metodų efektyvumo.
  - Nustatytos įžvalgos, prie kokių parametrų koks klaidų valdymo metodas yra labiau tinkamas.
  - Naudojantis gautomis įžvalgomis, parinkti klaidų valdymo metodai dideliam modeliui ir gauti geresni rezultatai lyginant su kontroliniais eksperimentais.

Išvados

Parinkti tinkamus klaidų valdymo metodus programų sistemose yra sudėtinga ir sudėtingumas didėja priklausomai nuo sistemos dydžio. Naudojant agentais grįstus modelius, atliekant simuliacijas galima gauti prognozes apie skirtingų klaidų valdymo metodų eksploatacines savybes sistemoje. Modeliavimas naudojant yra pranašesnis už prototipavimą, nes yra pakankamai paprasta sukurti generatorius agentais grįstiems modeliams, kurių pagalba galima greitai apibrėžti didelius modelius.

Klaidos architektūriniuose elementuose dažniausiai įvyksta ne atsitiktinai, o nuspėjamai, t.y. egzistuoja konkretūs periodai, kada elementas yra nekorektiškoje būsenoje ir tada jis turi didesnę tikimybę patirti klaidą vykdymo metu. Dėl šios priežasties, modeliuojant elementus apibrėžiant tik paprastą klaidos tikimybę, nebūtų korektiškai atspindimos realios sistemos. Todėl modeliuojant reikia atsižvelgti ir į tikimybę komponentui patekti į klaidinga būseną, klaidos tikimybę esant joje ir tikimybę komponentui palikti šią būseną. Ne visi veiksmai, kurie yra vykdomi kvietimo metu yra būtini (kritiški), kad užklausa būtų sėkminga.

Iš simuliacijų rezultatų nustatyti parametrų išsidėstymai su kuriais kurie klaidų valdymo metai turėtų būti optimalūs:

- Kartojimas - kai elemento klaidos yra atsitiktinės ir jų tikimybė maža.
- Atjungimas - kai galima užtikrinti pakankamą veikiančių elemento egzempliorių kiekį ir klaidos tikimybė yra palyginti maža.
- Nutildymas - kai elementas nėra kritiškai svarbus užklausos sėkmei ir klaidos yra pavienės ir neprognozuojamos.
- Karantinas - kai elementas nėra kritiškai svarbus užklausos sėkmei ir nekoretiškas veikimas yra būdingas visiems to tipo elementams.
- Atstatymo taškai - esant didelei elemento tikimybei sugesti, ypač įvykiais grįstoje architektūroje.
- Propagavimas - kai joks kitas klaidų valdymo metodas nėra tinkamas.

Individualiai nenagrinėjant situacijos, klaidų valdymas kartojant dažniausiai bus tinkamas pasirinkimas, nes jis suteikia palyginti vienus iš geriausių galimų rezultatų nepriklausomai nuo elemento parametrų kombinacijos.

Skirtumų tarp skirtingų architektūrų ir klaidų valdymo metodų pastebėta tik su atstatymo taškų valdymo metodu, kada jis yra daug efektyvesnis įvykiais grįstoje nei paslaugų architektūroje. Kiti metodai abiejų architektūrų atveju pasižymi panašiomis savybėmis.

Pasitelkiant šiame darbe sukurtą modelį galima modeliuoti įvairias sistemas ir jose apsibrėžti įvairias programų sistemų architektūras. Naudojant darbe pateiktą klaidų valdymo modeliavimo metodiką galima sumodeliuoti daugelį klaidų valdymo metodų, net ir tų, kurie nėra aptariami darbe.

Literatūra

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph ir k.t. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[Bas12] Len Bass. *Software architecture in practice*. Pearson Education India, 2012.

[BM03] Simonetta Balsamo ir Moreno Marzolla. A simulation-based approach to software performance modeling. *ACM SIGSOFT Software Engineering Notes*, tom. 28 numeris 5, p.p. 363–366. ACM, 2003.

[BM99] Jan Bosch ir Peter Molin. Software architecture design: evaluation and transformation. *Engineering of Computer-Based Systems, 1999. Proceedings. ECBS'99. IEEE Conference and Workshop on*, p.p. 4–10. IEEE, 1999.

[C213] C2. *Exception Patterns*. 2013. URL: `http://wiki.c2.com/?ExceptionPatterns` (tikrinta 2018-01-13).

[Cha06] K Mani Chandy. Event-driven applications: costs, benefits and design approaches. *Gartner Application Integration and Web Services Summit*, 2006, 2006.

[CS99] Paul Coates ir Claudia Schmid. Agent based modelling, 1999.

[dNor11] Organización Internacional de Normalización. *ISO-IEC 25010: 2011 Systems and Software Engineering-Systems and Software Quality Requirements and Evaluation (SQuaRE)-System and Software Quality Models*. ISO, 2011.

[EFG+03] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui ir Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131, 2003.

[EOK11] Serge Egelman, Andrew Oates ir Shriram Krishnamurthi. Oops, i did it again: mitigating repeated access control errors on facebook. *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, p.p. 2295–2304. ACM, 2011.

[Get08] Adam Getchell. Agent-based modeling. *Physics*, 22(6):757–767, 2008.

[GP15] Deepali Gill ir Hari Mohan Pandey. Approaches for software performance modelling, cloud computing and openstack. *International Journal of Computer Applications*, 119(22), 2015.

[Han13] Robert Hanmer. *Patterns for fault tolerant software*. John Wiley & Sons, 2013.

[HSS+93] JJ Hudak, B-H Suh, DP Siewiorek ir Zary Segall. Evaluation and comparison of fault-tolerant software techniques. *IEEE Transactions on Reliability*, 42(2):190–204, 1993.

[Mar04] Moreno Marzolla. Simulation-based performance modeling of uml software architectures. *PHD, Université de Venise*, 2004.

[Mar07] Robert Ernest Marks. Validating simulation models: a general framework and four applied examples. *Computational Economics*, 30(3):265–290, 2007.

[Mik14] Algirdas Mikoliūnas. *Programų sistemų architektūrų tyrimas taikant agentais grįstą modeliavimą*. Magistrinis darbas, Vilnius University, 2014.

[MN14] Charles Macal ir Michael North. Introductory tutorial: agent-based modeling and simulation. *Proceedings of the 2014 Winter Simulation Conference*, p.p. 6–20. IEEE Press, 2014.

[MSV03] Vincent A Mabert, Ashok Soni ir Munirpallam A Venkataramanan. The impact of organization size on enterprise resource planning (erp) implementations in the us manufacturing sector. *Omega*, 31(3):235–246, 2003.

[NS14] Dmitry Namiot ir Manfred Sneps-Sneppe. On micro-services architecture. *International Journal of Open Information Technologies*, 2(9):24–27, 2014.

[Pat+02] David A Patterson ir k.t. A simple way to estimate the cost of downtime. *LISA*, tom. 2, p.p. 185–188, 2002.

[PM99] Niels Provos ir David Mazieres. A future-adaptable password scheme. *USENIX Annual Technical Conference, FREENIX Track*, p.p. 81–91, 1999.

[Ric15] Mark Richards. *Software architecture patterns*. O'Reilly Media, Incorporated, 2015.

[RLJ06] Steven F Railsback, Steven L Lytinen ir Stephen K Jackson. Agent-based simulation platforms: review and development recommendations. *Simulation*, 82(9):609–623, 2006.

[Sar96] Robert G Sargent. Verifying and validating simulation models. *Proceedings of the 28th conference on Winter simulation*, p.p. 55–64. IEEE Computer Society, 1996.

[SG98] Bridget Spitznagel ir David Garlan. Architecture-based performance analysis, 1998.

[Sil16] Audrius Siliūnas. *Programų sistemų, kuriose yra komponentų su būsenomis, architektūrų tyrimas taikant agentais grįstą modeliavimą*. Magistrinis darbas, Vilnius University, 2016.

[SLK08] Vilas Sridharan, Dean A. Liberty ir David R. Kaeli. A taxonomy to enable error recovery and correction in software. *Workshop on Quality-Aware Design*, 2008.

[Smi87] Reid Smith. Panel on design methodology. *ACM SIGPLAN Notices*, tom. 23 numeris 5, p.p. 91–95. ACM, 1987.

[TLL+14] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou ir Chengxiang Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 19(6):1665–1705, 2014.

[Tro04] Klaus G Troitzsch. Validating simulation models. *Proceedings of the 18th European Simulation Multiconference*, p.p. 98–106. Erlagen, Germany: SCS, 2004.

[VHJ+95] John Vlissides, Richard Helm, Ralph Johnson ir Erich Gamma. Design patterns: elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49(120):11, 1995.

[WHF93] Y-M Wang, Yennun Huang ir W Kent Fuchs. Progressive retry for software error recovery in distributed systems. *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, p.p. 138–144. IEEE, 1993.

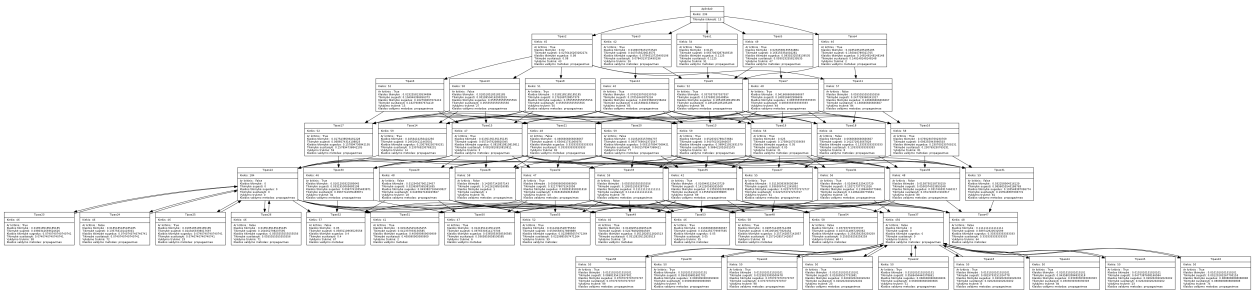
[WPC06] Wen-Li Wang, Dai Pan ir Mei-Hwa Chen. Architecture-based software reliability modeling. *Journal of Systems and Software*, 79(1):132–146, 2006.

[XKM+05] Xiaorong Xiang, Ryan Kennedy, Gregory Madey ir Steve Cabaniss. Verification and validation of agent-based scientific simulation models. *Agent-Directed Simulation Conference*, p.p. 47–55, 2005.

[XSS06] Zaipeng Xie, Hongyu Sun ir Kewal Saluja. A survey of software fault tolerance techniques. *University of Wisconsin-Madison/Department of Electrical and Computer Engineering*, 1415, 2006.

[ZCZ07] Liang-Jie Zhang, Hong Cai ir Jia Zhang. *Services computing*. Springer, 2007.

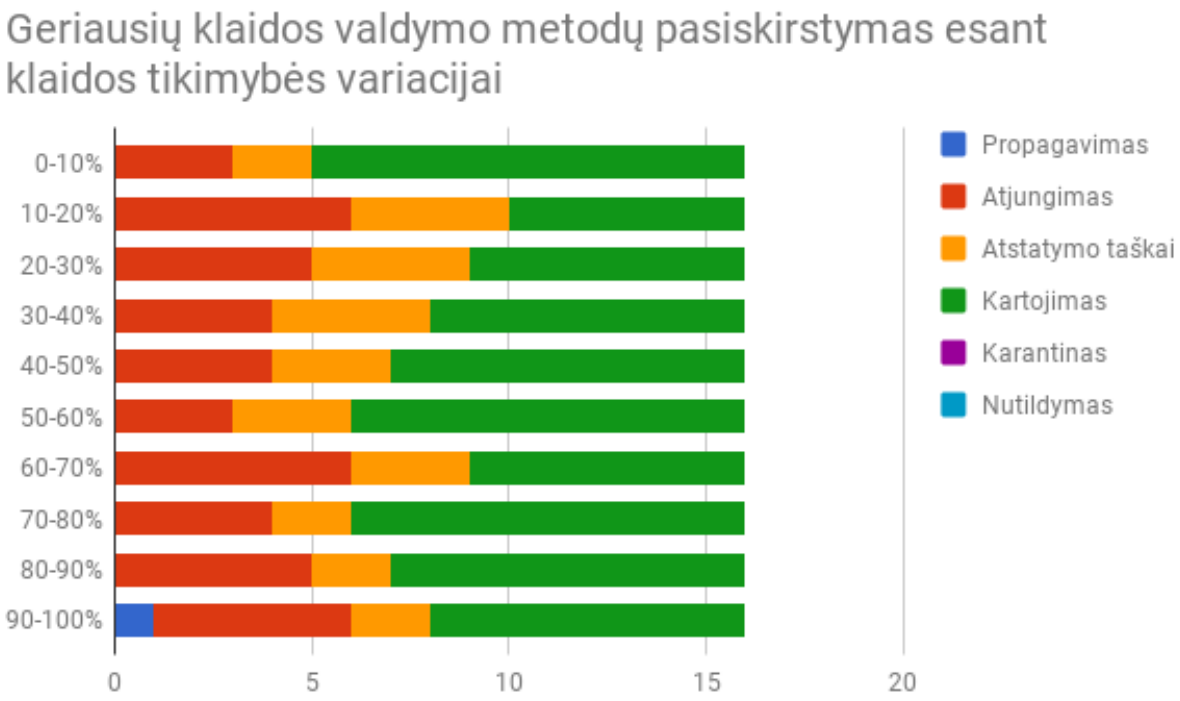
Priedas nr. 1
Didelis modelis naudotas patikrinimui



15 pav. Didelis modelis naudotas patikrinimui

Priedas nr. 2

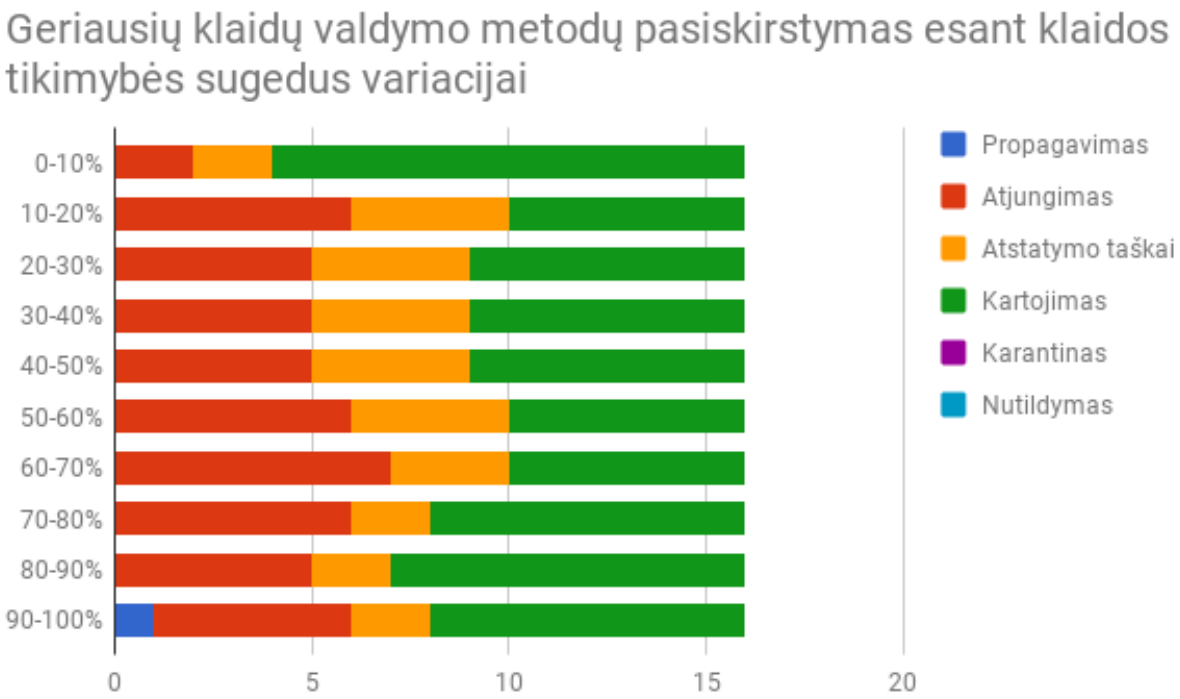
Geriausių klaidos valdymo metodų pasiskirstymas esant klaidos tikimybės kitimui



16 pav. Geriausių klaidos valdymo metodų pasiskirstymas esant klaidos tikimybės kitimui

Priedas nr. 3

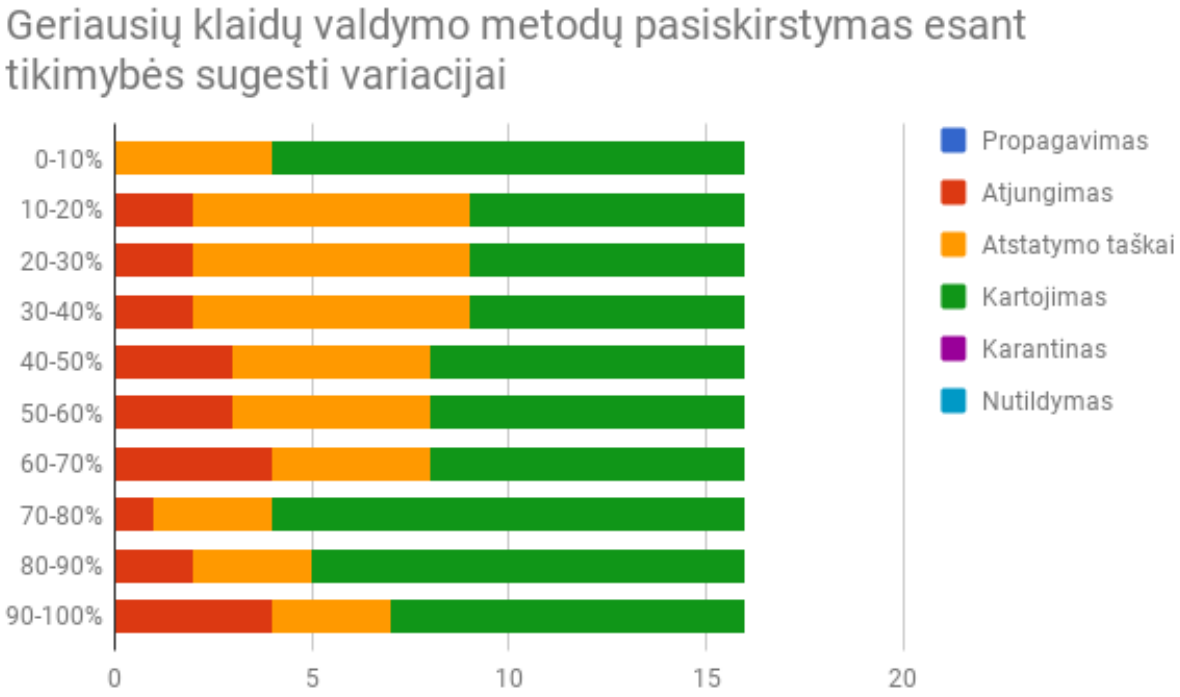
Geriausių klaidos valdymo metodų pasiskirstymas esant tikimybės sugesti kitimui



17 pav. Geriausių klaidos valdymo metodų pasiskirstymas esant tikimybės sugesti kitimui

Priedas nr. 4

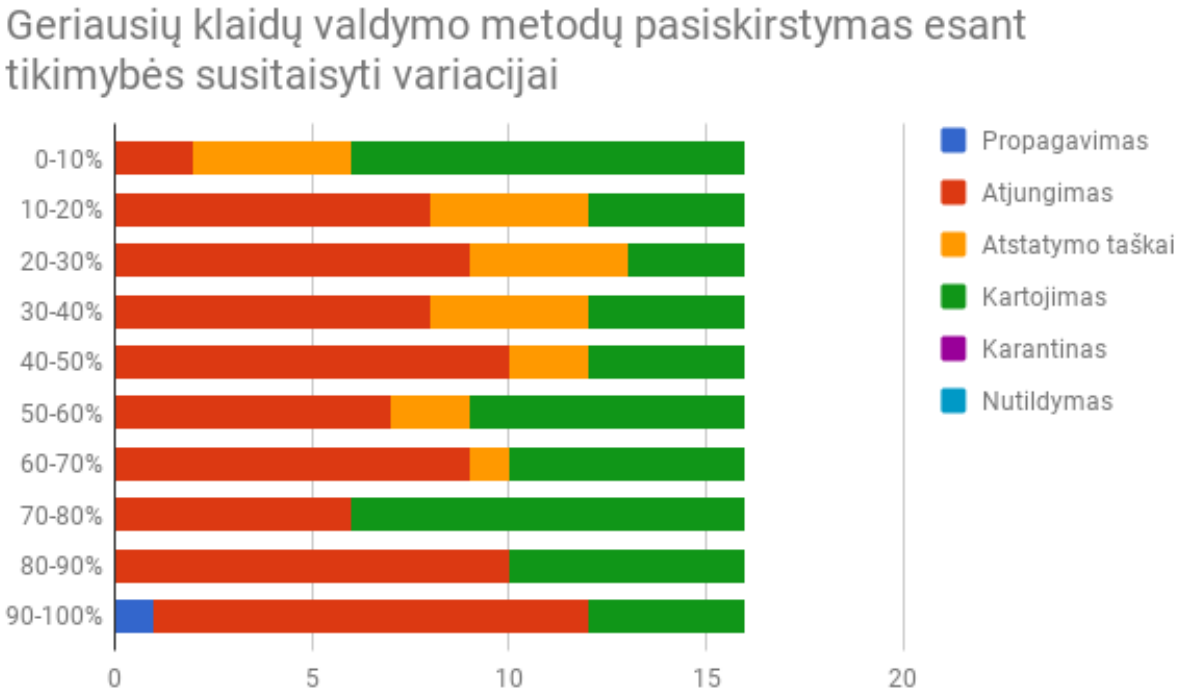
Geriausių klaidos valdymo metodų pasiskirstymas esant klaidos tikimybės sugedus kitimui



18 pav. Geriausių klaidos valdymo metodų pasiskirstymas esant klaidos tikimybės sugedus kitimui

Priedas nr. 5

Geriausių klaidos valdymo metodų pasiskirstymas esant tikimybės susitaisyti kitimui



19 pav. Geriausių klaidos valdymo metodų pasiskirstymas esant tikimybės susitaisyti kitimui