VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

Magistro baigiamasis darbas

Paskirstytų sistemų kontraktai ir jų skaitmeninis įvertinimas remiantis modelių patikrinimo arba simuliavimo metodais

(Distributed system contracts and their quantitative assessment using model checking or simulation methods)

Atliko studentas
Karolis Kajėnas

(parašas)

Darbo vadovas
Prof. Linas Laibinis

(parašas)

Recenzentas:

(parašas)

Vilnius – 2018

Santrauka

Magistro baigiamojo darbo tema yra paskirstytų sistemų kontraktai ir jų skaitmeninis įvertinimas remiantis modelių patikrinimo arba simuliavimo metodais. Abstraktūs programinės sistemos modeliai (kontraktai) bei su jais susiję simuliaciniai sistemos prototipai yra labai dideli pagalbininkai architektui ankstyvuose sistemos kūrimo etapuose, nes juos kuriant ir analizuojant galima numatyti būsimus sistemos trūkumus bei apsispręsti dėl konkrečių sistemos įgyvendinimo sprendimų.

Šio baigiamojo darbo tikslas – įgyvendinti paskirstytos sistemos skaitmeninių charakteristikų įvertinimą remiantis simuliaciniais metodais. Tai gali būti pasiekta sukuriant automatizuotą procesą, kuris leistų paversti konceptualų (generinį) paskirstytos sistemos modelį į simuliacinį sistemos prototipą, tinkamą tokios sistemos skaitmeninių charakteristikų įvertinimui.

Šitas tikslas virto tokiais konkrečiais darbo uždaviniais:

- Sukurti konceptualaus modelio šabloną paskirstytoms sistemoms;
- Paversti šį modelį į simuliacinį sistemos modelį (prototipą), remiantis nustatytomis tokio vertimo gairėmis (žodynu);
- Įvertinti sistemos skaitmenines charakteristikas ir įvairias sistemos konfigūracijas simuliacijos metu stebimų metrikų pagrindu.

Darbo uždaviniai buvo sėkmingai įvykdyti, naudojantis pasirinktais įrankiais – Mermaid grafine aplinka konceptualaus modelio sukūrimui ir Simpy karkasu (realizuotu Python kalba) simuliacinių sistemos prototipų realizavimui ir įvertinimui. Sukurtas automatizuotas procesas yra lengvai plečiamas, pridedant naujus paskirstytų sistemų parametrus ar įvertinamas jų skaitmenines charakteristikas.

Raktažodžiai: paskirstytos sistemos, modeliavimas, sistemų simuliavimas, sistemų skaitmeninis įvertinimas, Python, SimPy, Mermaid, konceptualus modelis, simuliacinis modelis.

Summary

The topic of this Master ‘s thesis is distributed system contracts and their quantitative assessment using model checking or simulation methods. Abstract models (contracts) of software systems and the associated system simulation prototypes can significantly facilitate work of the system architects at early stages of system development, because creation and analysis of such models allows to reveal the drawbacks of a system to be constructed as well as to decide on concrete system implementation decisions.

The main goal of this work – to implement automated assessment of quantitative system characteristics using system simuliation methods. This can be achieved by creation of an automated process, which allows to convert a conceptual (generic) model of a distributed system into a system simulation prototype, suitable for quantitative assessment of quantitative characteristics of such a system.

This goal can be deconstructed into the following concrete work tasks:

- To create conceptual model template of distributed systems;
- To convert this model into a systems simulation model (prototype), using the established guidelines for such conversion;
- To perform system quantitative assessment using various system parameters and configurations.

The work tasks were successfully completed, relying on the following tools – the Mermaid graphical environment to support construction of a conceptual system model, and the Sympy discrete event simulation framework (implemented in Python) to create and assess a simulation prototype. The developed automated process is easily extendable, by adding new parameters or quantitative characteristics of distributes systems.

Keywords: distributed systems, modelling, system simulation, system quantitative assessment, Python, SimPy, Mermaid, conceptual model, simulation model.

Turinys

1. Įvadas

Darbas yra orientuotas į paskirstytas sistemas, jų kontraktų (modelių) kūrimą, tokių sistemų modelių validavimą ir verifikavimą, bei būdus paskirstytų sistemų skaitmeniniams įvertinimams gauti. Aprašytų sistemų atliekančių įvardintas funkcijas rinkoje nebuvo aptikta atliekant literatūros apžvalgą.

Abstraktūs programinės sistemos modeliai (kontraktai) bei su jais susiję simuliaciniai sistemos prototipai yra labai dideli pagalbininkai architektui ir visai jo komandai pirmuosiuose sistemos kūrimo etapuose, nes juos kuriant ir analizuojant galima numatyti būsimus sistemos trūkumus ar "silpnąsias vietas", dar prieš pradedant detalius pačios sistemos kūrimo darbus. Taip pat jie padeda aptikti tinkamas sistemos konfigūracijas (t.y., sistemos parametrų reikšmes), pagal kuriuos galima būtų toliau projektuoti sistemą.

Simuliacinių modelių kūrimo srityje yra akivaizdu, jog egzistuoja didelis trūkumas viešai prieinamų platformų, kurios leidžia kurti generinius sistemos modelius ir juos analizuoti, automatiškai transformuojant šiuos modelius į simuliacinius sistemos prototipus bei stebint ir kaupiant dominančias sistemos metrikas prototipo vykdymo (simuliacijos) metu. Šio darbo tikslas yra pademonstruoti, kaip grafinis paskirstytos sistemos konceptualus modelis (atspindintis tiriamos sistemos struktūrą ir kitus esminius reikalavimus) gali būti konvertuotas į programinį kodą, kuris ir būtų simuliacinis sistemos prototipas. Savo ruožtu, šis sistemos prototipas gali būti analizuojamas pasirinktų metrikų pagrindu.

Magistrinio darbo tikslas – įgyvendinti paskirstytos sistemos skaitmeninių charakteristikų įvertinimą remiantis simuliaciniais metodais. Tai gali būti pasiekta sukuriant automatizuotą procesą, kuris leistų paversti konceptualų (generinį) paskirstytos sistemos modelį į simuliacinį sistemos prototipą, tinkamą tokios sistemos skaitmeninių charakteristikų įvertinimui. Toks konceptualus modelis aprašo sistemos struktūrą, kartu su identifikuotais sistemos parametrais bei stebimomis sistemos metrikomis. Pirminis informacijos šaltinis yra sistemos reikalavimų dokumentas.

Šis bendras tikslas suskyla į tokius darbo uždavinius:

- sukurti konceptualaus modelio šabloną paskirstytoms sistemos;
- paversti šį modelį į simuliacinį sistemos modelį (prototipą), remiantis nustatytomis tokio vertimo gairėmis (žodynu);
- įvertinti sistemos skaitmenines charakteristikas ir įvairias sistemos konfigūracijas simuliacijos metu stebimų metrikų pagrindu.

Tam, kad įgyvendinti suformuluotus darbo uždavinius, mes remiamės tokiais pasirinktais įrankiais – Mermaid grafine aplinka konceptualaus modelio sukūrimui ir Simpy karkasu (realizuotu Python kalba) simuliacinių sistemos prototipų realizavimui ir įvertinimui.

Tolesnė darbo struktūra yra tokia. 2-as skyrius apžvelgs nagrinėtus literatūros šaltinius. 3-as skyrius paaiškins sukurto automatizuoto proceso pagrindus bei naudotas technologijas. Darbo praktiniai rezultatai bus pristatyti 4-ame skyriuje. Paskutinis 5-as skyrius apžvelgs pagrindines darbo išvadas.

2. Literatūros šaltinių apžvalga

Paruošta apžvalga susideda iš vienuolikos apžvelgtų šaltinių. Šaltiniai yra žymimi pagal jų išnašų numeravimą. Kiekviename iš šaltinių apžvelgiama tik reikšmingi ir informatyvūs faktai.

2.1 Paskirstytos sistemos

Paskirstytos sistemos yra dažniausiai sutinkamas sistemų tipas šių dienų kuriamose kompiuterizuotose sistemose. Šių sistemų pagrindinis privalumas yra galimas horizontalus plėtimas ir (dažnai integruotas) atsparumas gedimams. Pasirinkti šaltiniai, knygos [G14] ir [YX13], apžvelgia įprastas paskirstytas sistemas ir taip pat sparčiai plintantį jų porūšį – kiber-fizines sistemas.

2.1.1 „“Distributed systems: an algorithmic approach“ šaltinis

Šaltinis [G14] koncentruojasi į paskirstytų sistemų naudojimą ir pritaikymą. Tai yra bendrosios informacijos šaltinis apie paskirstytas sistemas. Šis šaltinis nepateikia informacijos apie sistemų kontraktus (modelius) ar jų sudarymą tarp sistemų komponentų ar atskirų sistemų.

Šaltinyje yra pabrėžiama, jog padidėjusi paskirstytų sistemų paklausa atsirado dėl sumažėjusios aparatinės įrangos kainos, komunikacinių technologijų pagerėjimo, masyvaus interneto naudojimo bei augančio programinės įrangos poreikio. Sparčiai populiarėjant debesijos kompiuterijai ir didelio duomenų kiekio technologijoms (angl. „big data“), paskirstytų sistemų svarba išaugo.

Knygoje yra lyginamos lygiagrečios ir paskirstytos sistemos. Autorius įvardina skirtumus, ką kiekviena iš šių sistemų nori įgyvendinti. Lygiagrečios sistemos paprastai koncentruojasi į našumą, kai tuo tarpu paskirstytos sistemos yra orientuotos į avarinių situacijų toleravimą ir komunikavimą tarp komponenčių.

Šaltinis įvardija daug priežasčių nulemiančių paskirstytų sistemų aktualumą ir neseniai išaugusį populiarumą. Viena iš priežasčių yra išaugęs tokių sistemų atsparumas, kuris tampa (dėl aparatinės įrangos tobulėjimo) pagrindinė jų charakteristika.

Paskirstytos sistemos taip pat turi savo trūkumų. Vienas iš jų yra sudėtingas skirtingų komponentų valdymas (remiantis įvairiais komunikaciniais mechanizmais) ir matomumas sistemoje. Matomumo problemą padeda spręsti tarpininkas (angl. „middleware“), kuris yra programinės įrangos sluoksnis tarp aplikacijos ir operacinės sistemos OSI lygio.

Paskirstytos sistemos susiduria su keletą problemų, kurios taip pat turi būti atspindėtos kuriant generinį sistemos modelį. Į šias problemas įeina pagrindinės komponentės išrinkimas, resursų dalijimasis, laiko sinchronizacija, globalios būklės išlaikymas ir aktyvių komponenčių kiekio valdymas.

Šio šaltinio analizės tikslas buvo bendrais bruožais susipažinti su paskirstytomis sistemomis, jų veikimo ir naudojimo principais. Tai leido suprasti, kokių komponenčių prireiks įgyvendinti generinį sistemos modelį.

2.1.2 „Cyber-physical systems“ šaltinis

Šaltinis [YX13] pristato sparčiai plintantį paskirstytų sistemų porūšį – kiber-fizines sistemas. Kiber-fizinės sistemos yra sistemos, kurių darbas yra priklausomas nuo kiber ir fizinių sistemos komponenčių, bei jų sąveikos. Šių sistemų kūrimas turi atsižvelgti į šias sąveikas ir persidengiantį funkcionalumą, kurį turi abiejų tipų komponentai. Į kiber-fizinių sistemų aibę įeina medicininiai prietaisai ir sistemos, automobiliai, robotika, aviacija ir kitos kritinės struktūros. Dėl tokių sistemų svarbos ir susijusių pavojų turi būti užtikrintas korektiškas jų veikimas.

Standartiškai kiber-fizinės sistemos yra sunkiai analizuojamos, kuriamos ir validuojamos. Taip yra dėl jų kūrimo proceso iššūkių. Kuriant kiber-fizinę sistemą, yra būtina atsižvelgti į komponenčių sąryšius, įskaitant tiek fizinių, tiek kiber komponenčių tarpusavio veikimą. Kiber-fizinės sistemos apjungia fizinių ir kiber komponenčių dinamiką. Siekiama pilnavertė integracija tarp šių skirtingų komponenčių paverčia jų modeliavimo, analizavimo, kūrimo ir validavimo darbą dideliu iššūkiu.

Paminėtos priežastys komplikuoja kiber-fizinių sistemų analizę. Tam tikroms kiber-fizinės sistemos komponentėms, ypač fizinėms, vienintelė galima jų abstrakcija yra simuliacinis modelis. Dėl šios priežasties pilnas tikrosios sistemos analizavimas praranda tikslą. Efektyvios simuliavimo technikos poreikis būtent kiber-fizinėms sistemoms analizuoti buvo aiškus jau seniai. Dauguma

dabartinių technikų remiasi supaprastintais (abstraktintais) kiber komponenčių prototipais, kurie yra vykdomi lygiagrečiai su fizinių komponenčių simuliavimu. Šis būdas nėra pats tinkamiausiais analizės tikslams.

Norint pašalinti esamus trūkumus, simuliacijų technikos turi modeliuoti pilną sistemos veikimą, įtraukiant abu fizinį ir kiber lygius. Siekiamybė yra analizę atlikti dar prieš atsirandant fiziniams komponentams (t.y, remiantis jų prototipais). Tai gali padėti pasiekti sistemos simuliavimas dar pilnajai sistemai neegzistuojant. Be to, aparatinė komponenčių virtualizacija turi didelį potencialą nustatyti sąsajas tarp fizinių ir kiber komponenčių.

2.2. Simuliavimo ir verifikavimo metodai

Simuliacinis modelis ir jo verifikacija bei validacija leidžia užtikrinti „teisingą“ modelio veikimą konkrečioje sistemos konfigūracijoje, susidedančioje iš konkrečių sistemos parametrų ir reikšmių. Pats modelis yra naudojamas gauti duomenis dominančius vartotoją apie simuliuojamą sistemą. Šaltiniuose yra analizuojama simuliavimo ir verifikavimo metodai, jų naudojimas, patikra ir modelio kūrimo procesas.

2.2.1. „An introductory tutorial on verification and validation of simulation models. “ šaltinis

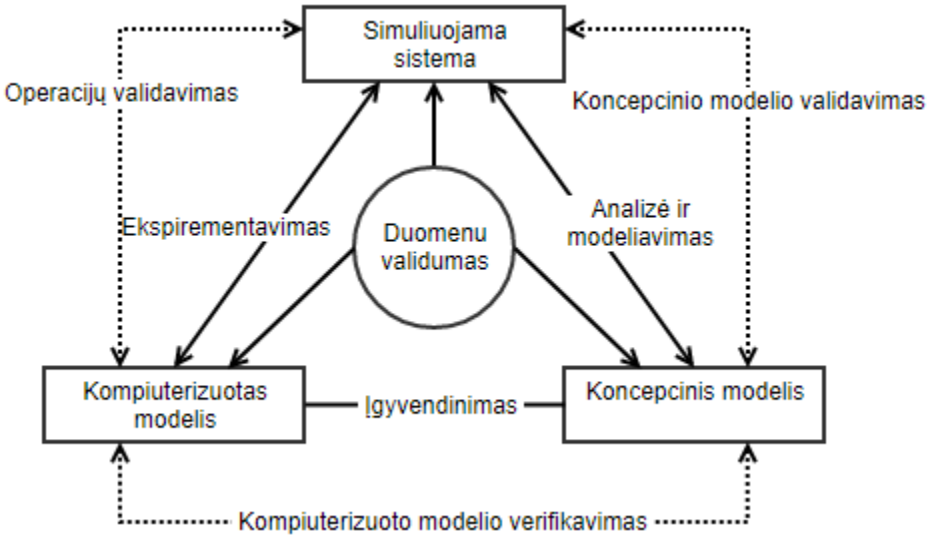
Šaltinyje [R15] yra apibrėžiamas sistemos validacijos ir verifikacijos sąvokos bei kuriamų sistemos modelių svarba. Jame yra grafiškai pavaizduotas verifikacijos ir validacijos sąryšis bei jų vaidmuo bendrame modelio kūrimo procese. Be to, yra nagrinėjamas modelio tikslumas ir ryšys tarp modelių ir sistemos dokumentacijos.

Verifikacija ir validacija yra glaudžiai susijusios su modelio rezultatų tikslumu tam tikram naudojimui ar tikslui. Literatūroje modelio verifikacija yra įvardijama, kaip „užtikrinimas, kad kompiuterinės programos modelis ir jo realizacija yra teisingi“, o validacija yra apibrėžiama kaip „pagrindimas, kad kompiuterinės programos modelis tam tikroje aplinkoje įgyja patenkinamus rezultatus, kurių buvo tikėtąsi modelio pritaikyme“. Šaltinyje didelis dėmesys yra skiriamas verifikacijai ir validacijai, remiantis sistemos simuliaciniais modeliais, kurie yra skirti aproksimuoti sistemos veikimą, pavyzdžiui, sistemos grąžinamus rezultatus. Svarbūs modelio kriterijai yra galimybė jį patikrinti (verifikuoti) ir lengvas jo panaudojimas. Modelio galimybė

patikrinti turi užtikrinti vartotojų pasitikėjimą modeliu ir jo teikiama informacija. Modelio panaudojimo lengvumas yra nustatomas pagal modelio ir jo naudojimosi instrukcijų paprastumą.

Simuliacinio modelio tikslas yra atsakyti ar modeliuojama sistema atitinka iš anksto suformuluotus kriterijus (reikalavimus ar tikslus). Modelio validumas turi būti vertinamas pagal kiekvieną tokį kriterijų. Simuliacinio modelio kūrėjai ir naudotojai, asmenys, priimantys sprendimus pagal modelio rezultatus, ir asmenys, įtakojami modelio rezultatų, yra suinteresuoti modelio ir jo rezultatų teisingumu kiekvienam iš suformuluotų kriterijų.

Simuliaciniam modeliui sukurti yra reikalinga nustatyti didelę aibę eksperimentinių sąlygų, kurios užtikrintų modelio veikimą. Modelis gali būti validus ir teisingas vienai sąlygų aibei, tačiau klaidingas kitai. Modelį galima vadinti validžiu, kai simuliuojant modelį su tam tikromis sąlygomis, modelio tikslumas neperžengia iš anksto žinomo tikslumo slenksčio. Slenkstis yra nustatomas pagal modelio paskirtį. Paprastai jis remiasi simuliaciniais duomenimis, kurie suteikia informaciją apie simuliuojamą sistemą. Modelio priimtina tikslumo amplitudė turi būti nustatyta dar prieš pradedant kurti modelį. Dažniausiai yra sukuriamos kelios modelio versijos, prieš gaunant priimtinus modelio rezultatus. Pagrindimas, kad modelis yra validus, pvz. tinkamas sistemos verifikavimui ir validavimui, yra modelio kūrimo proceso dalis.



1 pav. Minimizuota modelio kūrimo proceso diagrama

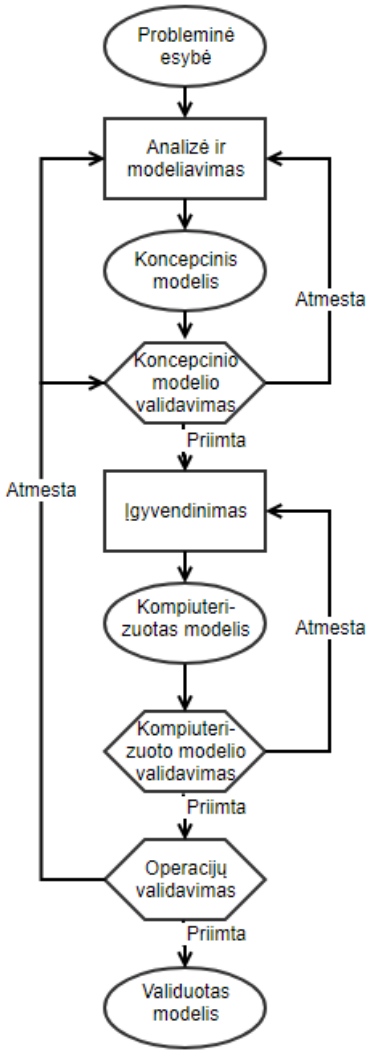
Peržiūrint minimizuotą modelio kūrimo proceso diagramą (1 pav.), sistema yra įvardijama, kaip probleminė būtybė (angl. „Problem Entity"). Konceptualus modelis yra matematinis/grafinis/loginis sistemos atvaizdavimas skirtas tam tikram eksperimentui. Kompiuterizuotas modelis yra įgyvendintas kompiuterinėje aplinkoje. Konceptualus modelis yra kuriamas per sistemos analizės ir modeliavimo žingsnius, tuo tarpu kompiuterizuotas modelis yra kuriamas per programavimo ir simuliavimo žingsnius. Išvados apie sistemą yra gaunamos atliekant kompiuterinius eksperimentus remiantis kompiuterizuotu modeliu.

Modelio kūrimo proceso diagrama (1 pav.) susieja tarpusavyje modelio verifikaciją ir validaciją. Konceptualaus modelio validacija yra įvardijama, kaip nustatymas ar konceptualaus modelio teoriniai spėjimai yra teisingi, atsižvelgiant į suformuluotus kriterijus (tikslus). Kompiuterizuoto modelio verifikacija yra aprašoma, kaip užtikrinimas, kad programinis konceptualaus modelio įgyvendinimas yra perteiktas teisingai. Operacijų validacija yra apibrėžiama, kaip patikrinimas, kad modelio rezultatai turi tinkamą tikslumą. Duomenų validumas yra aprašomas, kaip užtikrinimas, kad duomenys reikalingi modelio kūrimui, simuliavimui, vertinimui ir testavimui bei surinkti kaip gaunami rezultatai, yra tinkami.

Modelis turėtų būti kuriamas tam tikram tikslui ar specifiniam panaudojimui. Simuliacinis modelis turi būti kiek galima paprastesnis, kad atliktų tik reikalingas funkcijas, t.y., neturi būti nereikalingo funkcionalumo. Simuliacinis modelis taip pat yra struktūrinis modelis, tai reiškia, kad modelis turi atspindėti bazinę sistemos logiką ir tik reikalingiausius sąryšius sistemoje. Tuo struktūriniai modeliai skiriasi nuo empirinių modelių, kurie kuriami pagal galutinius duomenis, kaip regresiniai modeliai. Simuliacinio modelio kūrimas yra iteratyvus procesas, kur keletas modelio versijų yra išbandomi prieš galutinio ir naudojamo modelio sukūrimą.

Modelio kūrimo procesas turi apimti modelio verifikaciją ir validaciją. Sekant bendra paradigma, atvaizduota 1 pav., iteratyvus modelio kūrimo procesas parodytas 2 pav. Toks procesas gali būti rekomenduojamas naudoti validaus simuliacinio modelio kūrimui. Sistemos konceptualus modelis yra kuriamas pradinės problemos analizės metu ir tuomet tikrinamas konceptualaus modelio validacijos etape. Esant netikslumams, yra grįžtama į pradinį analizės žingsnį. Užtikrinus, kad modelio koncepcija yra validi, t.y., atitinka norimus validavimo rodiklius, galima pradėti sistemos prototipo kūrimą. Šiame žingsnyje turėtume gauti kompiuterizuotą modelį ir užtikrinti jo

verifikacijos galimybes, t.y. užtikrinti, kad verifikacijos tikslumas yra norimo lygio. Kitu atveju reikia grįžti su tikslu atlikti pakeitimus kompiuterizuotame modelyje. Kai kompiuterizuotas modelis yra priimtinas, yra atliekamas visos sukurtos sistemos (modelio) validavimas. Jei yra nustatyti neatikimai, yra vėl grįžtama į ankstesnius modelio kūrimo etapus. Jeigu modelis kaip sistema veikia darniai, tuomet modelis yra pripažįstamas kaip sėkmingai validuotas.



2 pav. Modelio kūrimo proceso diagrama

2.2.2. „Principles of modeling and simulation: a multidisciplinary approach“ šaltinis

Knyga [SB11] aprašo sistemos simuliacijų kūrimo pagrindus. Sistemos simuliacijos yra svarbios tik tam tikrame simuliaciniame kontekste, kuriam yra skirtas simuliacinis modelis.

Pavyzdžiui tyrinėjant dangaus kūnų trajektorijas ir susidūrimo taškus, simuliacijos padeda pagrįsti tyrinėjamąjį subjektą, šiuo atveju dangaus kūnų judėjimo charakteristikas. Simuliacinė aplinka turi kiek įmanoma imituoti tikras realioje sistemoje sutinkamas sąlygas.

Simuliacijos naudojamos procesų stebėjimui ir pastebėtų problemų identifikavimui. Dažniausiai jos turi konfigūruojamos per sistemos parametrus, kurių pagalba galima modifikuoti simuliacinę aplinką, priartinant ją prie tikrovės. Konfigūraciniai parametrai leidžia simuliuoti sistemą tam tikromis sąlygomis ir pagal tai nustatyti simuliuojamos sistemos ribas, trūkumus bei stipriąsias vietas, taip pat padeda ankstyvo dizaino fazėje nuspręsti dėl tolimesnių sistemos vystymo žingsnių.

Pavyzdžiui, simuliavimo metodai yra plačiai naudojami sistemose, skirtose oro prognozių generavimui. Tam yra naudojamas taip vadinamas „ensemble“ modeliavimo metodas. Orų sistema yra paprastai traktuojama kaip chaotiška, nes dėl galimų didelių rezultatų pokyčių dažnai tenka modifikuoti simuliacijos konfigūraciją. Tačiau ne visos simuliacinės sistemos kenčia nuo chaotiškumo problemos. Dalis sistemų pateikia pakankamai tolygius rodmenis, t.y. konfigūracija vienaip arba kitaip proporcingai koreliuoja su panašia konfigūracija. „Ensemble“ modeliavimas įtraukia statistinių duomenų naudojimą kuriant simuliacinią aplinką. Surinkus pakankamą kiekį statistinių duomenų yra formuluojamos prognozės priklausančios nuo statistinių duomenų. Šiuo atveju gaunami rezultatai yra gana tikslūs, nes rezultatų gavimui naudojamasi istoriniais duomenimis.

Naudojant modelius prognozuoti ateitį yra bandymas atsakyti „Kas bus, jei ... ?“, kur klausimo dalis „Kas“ yra nustatomas pagal simuliacijos konfigūraciją. Šiame kontekste, „Kas“ būtų dabartinės atmosferos ir oro kondicijos.

Simuliacijos taip pat gali būti naudojamos gauti galutinius rezultatus, t.y., galimus rytdienos orus priklausomai nuo šiandieninių oro charakteristikų. Sukūrus programinę įranga, simuliacija gali būti naudojama sugeneruoti konfigūracijai, kuri būtų skirta programinės įrangos testavimui.

2.2.3. „An introduction to verification and validation of simulation models. “ šaltinis

Straipsnyje [S13] yra teigiama, jog verifikacija ir validacija yra glaudžiai susijusi su modeliu ir jo validacijų teisingumu tam tikriems naudojamiems scenarijams. Modelio verifikacija ir validacija yra įvardijami, kaip „... užtikrinimas, kad kompiuterizuoto modelio programa ir jos įgyvendinimas yra teisingi.“. Neapibrėžtumas kyla iš žodžio „teisingi“, kadangi kiekvieno modelio veikimas yra skirtingas, kaip ir kiekvienos simuliuojamos sistemos. Dėl tos priežasties, žodžio „teisingas“ prasmė šiame kontekste yra glaudžiai susijusi su simuliacinio modelio kriterijais (tikslais). Šie suformuluoti tikslai apibrėžia verifikacijos ir validacijos teisingumą. Tuo tarpu modelio validacija yra apibrėžiama, kaip „... įrodymas, kad kompiuterizuotas modelis naudojamoje srityje pademonstruoja patenkinamą veikimo tikslumą,...“.

Svarbiausias modelio validacijos ir verifikacijos uždavinys yra pritaikyti tikrinimus kiekvienam naudojimo ir veikimo atvejui. Jeigu simuliacinio modelio tikslas yra atsakyti į tam tikrus su validacija susijusius klausimus, tai tikslai turi būti patenkinami kiekvienos patikros metu. Pavienė patikra paprastai yra neoptimali. Visi suinteresuoti asmenys turi būti linkę užtikrinti, kad modelis ir jo rezultatai yra teisingi.

Modelio verifikacija ir validacija yra kritinės grandys simuliacinio modelio vystymo procese. Kiekvienas simuliacinis projektas iškelia naują unikalų uždavinį susijusį su modelio verifikacija ir validacija konkrečiomis sąlygomis.

Dažniausiai reikia daugybės eksperimentinių sąlygų norint tinkamai apibrėžti modelio taikymo aplinką. Modelis gali būti teisingas vienai sąlygų aibei, tačiau tuo pačiu metu klaidingas kitai. Modelio validacija yra laikoma teisinga, kai modelio demonstruojamas tikslumas yra priimtinuose, iš anksto nustatytuose rėžiuose. Tikslumas reikalingas patvirtinti teisingą modelio veikimą. Priimtinumo kriterijai turi būti nustatyti dar prieš pradedant kurti modelį. Dažniausiai yra sukuriama keletas modelio versijų, kol pasiekiamas norimas rezultatas, kuris užtikrina modelio teisingumą. Verifikacja ir validacija įrodo, kad modelis veikia teisingai. Verifikavimas bei validavimas dažniausiai yra modelio vystymo dalis.

2.2.4. „Introduction to simulation“ šaltinis

Šaltinyje [WI15] simuliacija yra apibrėžiama, kaip yra vykdomas eksperimentas su modeliu. Modelio elgesys imituoja tam tikrus simuliuojamos sistemos veikimo aspektus ir vartotojo naudojimąsi simuliuojama sistema. Šaltinis yra orientuotas į diskrečių įvykių simuliavimą, koncepciją, struktūrą ir panaudojimą.

Šaltinyje yra įvardinama modelio sąvoka. Teigiama, jog tai yra esybė, kuri yra naudojama reprezentuoti kitą esybę. Paprastai modeliai yra supaprastinta abstrakcija, kuri skatina įgyvendinti tik tą funkcionalumą, kurio reikia patenkinti modelio atliekamą paskirtį, t.y., dalinis įgyvendinimas. Tai dažniausiai yra naudojama, kai tiesioginis objekto naudojimas yra per brangus tam tikrų resursų atžvilgiu. Taip pat, modeliai yra naudojami koncepcijų simuliavimui.

Anot autoriaus, simuliacija yra tam tikras simuliacinio modelio naudojimas, kurio paskirtis yra eksperimentinė arba praktinė. Praktikoje, simuliacija yra labai panaši į eksperimentinius bandymus, išskyrus tai, kad eksperimentas vyksta su tęstiniu modeliu, o ne su tikrąja sistema. Į simuliacijos procesą yra įtraukiama modelio sukūrimas, kuris imituoja norimą veikimą. Eksperimentavimas savo ruoštu yra modelio ir jo veikimo stebėjimas bei išvadų padarymas iš gautų duomenų. Daugumoje aplikacijų simuliacijai yra taip pat priskiriamas ir skirtingų validacijų bei dizaino palyginimų testavimas.

Simuliacijos ištekliai yra svarbi dalis, kuri turi tam tikrus apribojimus. Ištekliai yra dalijamiesi tarp esybių, jos turi laukti eilėje, kol užimti ištekliai atsilaisvins. Tai dažniausiai nutinka dėl kitų būtybių, kurios naudojasi ištekliais. Vieni iš pavyzdžių gali būti darbuotojai, mašinos, mazgai tarp komunikuojančių tinklų ir reguliuojamos sankryžos.

2.3. Konkretūs metodų taikymai

Poskyris yra koncentruotas į konkrečias naudojamas praktikas simuliuojant paskirstytas sistemas. Straipsniuose yra įvardijami skirtingi metodai, įrankiai, sistemų kategorijos ir naudojamos technologijos. Taip pat yra aprašytos naudotos praktikos taikomos tam tikroms sistemoms.

2.3.1. „Integrating Event-B modelling and discrete-event simulation to analyse resilience of data stores in the cloud.“ šaltinis

Šaltinio [LBPTEP14] tematika yra debesijos sistemos simuliavimas, naudojant diskrečių įvykių modeliavimo aplinką - „Event-B“ ir skaitmeninių rodiklių simuliavimą - „SimPy“. Šaltinyje yra diskutuojama nauda gaunama iš simuliacinio modelio kūrimo, susiejant jį su iškylančių sunkumais testuoti vartotojų naudojamoje aplinkoje.

Straipsnyje yra naudojamos dvi verifikavimo/simuliavimo aplinkos „Event-B“ ir „SimPy“. „Event-B“ yra skirta verifikuoti duomenų vientisumo ir kitas duomenų saugyklos charakteristikas kaip logines sąvybes. Tuo tarpu „SimPy“ yra skirtas simuliuoti sistemos veikimą, skaitmeniškai įvertinant sistemos našumo ir patikimumo rodiklius.

Debesijos sistema, kuri yra pagrindinis testavimo subjektas, turi atitikti daugybę reikalavimų, kuriais turi pasirūpinti sistemos kūrėjai. Taip pat yra paminėta, jog užtikrinti sistemos atsparumą turint dideles duomenų talpyklas debesijos aplinkoje, yra sudėtinga inžinerinė užduotis. Tą pasiekti, ankstyvaisiais sistemos kūrimo etapuose reikia numatyti naudojamas našumo ir mazgų gedimo atsparumo charakteristikas. Užtikrinti tokios sistemos korektišką veikimą galima tik simuliuojant sistemą, tačiau yra per brangu simuliavimą atlikti realioje sistemoje.

Straipsnyje apibūdinta debesijos sistema naudoja „WAL“ („write-ahead logging“) ir „DDS“ („distributed data source“). „WAL“ užtikrina galutinį skirtingų duomenų kopijų nuoseklumą, nepaisant to, kad transakcija yra užregistruojama prieš ją mėginant įrašyti į duomenų talpyklą. „DDS“ yra paskirstyta duomenų talpykla, kurios kontekste ir yra naudojama „WAL“, tam kad užtikrinti galutinį duomenų suvienodinimą tarp skirtingų duomenų kopijų.

Dalis straipsnio yra skirtas suprasti simuliavimo privalumus prieš analogišką realios sistemos testavimą. Vienas iš tokių privalumų yra tai, kad nereikia modifikuoti esamos sistemos, norint ją simuliuoti. Taip pat resursų naudojimas yra drastiškai mažesnis, nes visa reali sistema nėra naudojama. Vietoje to, jos resursų naudojimas yra simuliuojamas.

Simuliacinio modelio kūrime yra svarbu atsižvelgti į komponenčių kontraktus, jų vidinę logiką ir replikuoti visus galimus resursus. Šių charakteristikų replikavimas leidžia modelį ištestuoti įvairiomis sąlygomis, prie skirtingų sistemos konfigūracijų.

2.3.2. „A quantitative software testing method for hardware and software integrated systems in safety critical applications." šaltinis

Šaltinyje [TL14] yra teigiama, jog didžiausia dalis saugumą užtikrinančių sistemų yra kiber-fizinės sistemos, kurios sudarytos iš programinės ir tam tikros kontroliuojamos aparatinės įrangos. Tokios sistemos dažniausiai yra naudojamos ten, kur saugumas yra kritinės reikšmės. Tai galėtų būti atominės elektrinės valdymo blokas, cheminių procesų kontrolė, ir t.t. Šios sistemos yra skirtos aptikti potencialiai žmogaus gyvybei grėsmę keliančius įvykius ir vykdyti jų prevenciją, norint užkirsti kelią žmogaus sužalojimui ar žūčiai. Kiber-fizinės sistemos kiekybinio (angl. „quantitative“) patikimumo modeliavimas ir analizė dar vis sukelia daugybę iššūkių. Ypač tais atvejais, kai norima įvertinti sistemos patikimumo rodiklius.

Programinės ir aparatinės įrangos genda skirtingai. Taip yra dėl atskirų komponenčių veikimo principų. Aparatinė įranga jungia komponentes, kurios leidžia sistemai fiziškai funkcionuoti. Dažniausia problema yra tokios aparatinės įrangos susidėvėjimas ir tuo pačiu jų veikimo degradavimas. Tai su laiku susijusios problemos. Kai tuo tarpu programinės įrangos problemos gali būti įvairios, tačiau tarp šių problemų nėra įtraukiama fiziniai reiškiniai.

Kuriant programinės ir aparatinės įrangos patikimumo modelius yra daugybė faktorių, kuriuos derėtų apsvarstyti. Dėl fundamentalių patikimumo skirtumų tarp programinės ir aparatinės įrangos yra akivaizdu, jog modelis sukurtas aparatinei įrangai negali būti naudojamas programinei įrangai ir atvirkščiai.

Kuo toliau, tuo didesnę rolę sistemų kūrime užima programinės įrangos plėtojimas. Taip yra dėl to, jog dauguma kontrolės funkcijų remiasi generuojama ir programuojama logika, realizuota kaip programinis kodas. Tai gali būti sensorių parodymų apdorojimas ir kiti atliekami veiksmai, priklausomi nuo gautos informacijos.

Patikimumas yra vienas iš kiekybinės programinės įrangos metrikų. Ši metrika nurodo, kaip sistema turėtų veikti pagal jos nurodytą specifikaciją. Skirtingai nei aparatinė įranga, programinė įranga nesusidėvi. Yra daugybė programinės įrangos patikimumo modelių apibūdintų literatūroje.

Skaitine verte programinės sistemos patikimumą galima įvertinti 1 (100%), jeigu visi testai gražino teigiamą rezultatą. Kita vertus, realistiškas programinės įrangos produktas niekada nepasieks 100% patikimumo, nebent visos galimos variacijos su visais galimais įvesties ir išeities variantais yra testuojamos ir visi testai yra teigiami. Norint tai pasiekti, dažnai reikia labai didelio skaičiaus testų, kas yra neįmanoma pasiekti dėl kainos ir laiko kaštų. Todėl turi būti nustatytos taisyklės, apibrėžiančios testavimo pabaigą.

2.3.3. „Partial orders for efficient bounded model checking of concurrent software." šaltinis

Straipsnyje [JKT13] yra teigiama, jog dauguma programinės įrangos kūrėjų, kuriančių dideles, brandžias sistemas, nori optimizuoti sistemų našumą nekeičiant veikimo semantikos (funkcionalumo). Tai dažniausiai yra daroma atliekant pakeitimus išorinėse sistemose, kurios bendrauja su kuriama sistema, kurios logiką norima apsaugoti nuo pakeitimų. Tai sumažina, tačiau nepašalina galimybės pakeisti egzistuojančią sistemos logiką. Programinės įrangos kūrėjui atlikus pakeitimus, nėra galimybės patvirtinti, jog sistema veikia korektiškai. Nors patikrinti ar programa veikia greičiau nėra labai sudėtinga užduotis, tačiau atlikti tokią pačią patikrą dėl programos veikimo semantikos nėra trivialu.

Validacijos skyriuje yra tyrinėjama būdai, kaip užtikrinti abipusę patikrą. Yra aiškinamasi, kokios sąlygos parodo dviejų programų ekvivalentiškumą. Siūlomas patikros būdas nėra pilnai automatinis. Todėl yra būtina žmogiška (programuotojo) proceso priežiūra. Validavimo procesui vadovauja programuotojas ir nurodo kokios funkcijos yra testuojamos ir validuojamos. Taigi patikra gali būti pritaikoma, patikrinant tik dalį galimo komponentės veikimo. Validavimo pabaigoje yra pateikiami gauti rezultatai, pagal kuriuos vartotojas nusprendžia ar programa nepasikeitė. Dauguma atvejų yra galimos paklaidos, dėl šios priežasties maži netikslumai gali reikšti, jog sistemoje neįvyko loginių pakeitimų.

Validavimo proceso eigoje surinkta informacija yra pateikiama kaip suvestinė validavimo pabaigoje. Galima matyti gautus rezultatus ir programuotojo pasirinkimus. Taip yra sudaroma prevencija praleisti neteisingai veikiančias vietas ir įgalinti dvigubą patikrą.

Yra keli pagrindiniai iššūkiai, kuriant interaktyvią palyginimo validavimo sistemą. Pirmas yra sistemos kontraktų (modelių) pasikeitimas, kai atlikti pakeitimai negali būti palyginami arba kai funkcionalumas buvo pašalintas. Kitas iššūkis yra įgalinti realaus laiko patikrą ir stebėti programos būseną veikimo metu. Tai yra ypač sunkiai sprendžiama problema pasikeitus programos veikimui.

2.3.4. „Model Checking Web Applications" šaltinis

Šaltinyje [A15] modelio patikra yra apibrėžiama, kaip aibė technologijų naudojamų automatinei sistemos analizei. Formalus apibrėžimas yra – „Modelio patikra yra automatizuota technika, kuri duotam būsenos modeliui tikrina formalias ypatybes".

Modelio patikros įrankiai priima sistemos aprašą ir ypatybes. Sistema dažniausiai yra apibrėžiama, kaip baigtinės būsenos sistema ir jos savybės yra išreiškiamos, kaip laikinos loginės ("temporal logic") formulės. Modelio patikra verifikuoja ar sistemos savybės (išreikštos tokiu būdu) yra patenkintos visose sistemos būsenose. Jei tai nėra tiesa, tuomet modelio patikros įrankis paprastai pateikia pavyzdį, iliustruojantį kada ir kaip taip nutiko.

Praktikoje analizuojamas sistemos modelis yra smarkiai abstraktintas, todėl rezultatai nėra pilnutiniai. To pasekoje klaidos sistemoje gali išlikti net ir po verifikacijos.

2.4. Simuliavimo karkasas SimPy

Planuojamo naudoti karkaso dokumentacijos apžvalga, sąvokos ir trumpa informacija apie naudojimo scenarijus.

2.4.1 „Simpy" dokumentacijos šaltinis

SimPy [S17] yra diskrečių įvykių simuliacijos karkasas, kuris yra realizuotas kaip standartinis Python kalbos modulis. Sistema yra modeliuojama kaip procesų, bendraujančių diskretinių įvykių pagalba, rinkinys.

Karkasas turi sau būdingų sąvokų rinkinį, kurį privalu suprasti:

1. Aplinka (angl. „Environment“). Simuliacijos aplinka yra atsakinga už simuliacijos laiką, jo planavimą ir įvykių apdorojimą. Taip pat aplinkos pagalba yra leidžiama vartotojui žingsnis po žingsnio vykdyti simuliaciją.
2. Įvykis (angl. „Event“). SimPy įvykiai yra labai panašūs į ateityje vykstančius įvykius (angl. „futures“) ir pažadus (angl. „promises“). Klasė „Event“ SimPy yra naudojama aprašyti bet kokį įvykį. Yra kelios įvykių rūšių, tiesiogiai įtakojančių įvykių būseną ir jų tvarką.
3. Proceso sąveika (angl. „Process Interaction“). Diskrečių įvykių simuliacija yra įdomi savo sąveika tarp procesų. SimPy siūlo tris skirtingas sąveikas tarp procesų:
    a) laukti, kol kam nors taps reikalingas;
    b) laukimas, kol kitas procesas pasibaigs;
    c) pertraukimas kito proceso.
4. Bendri ištekliai (angl. „Shared Resources“). Ištekliai yra kitas būdas sąveikauti tarp procesų. Yra dvi kategorijos šių išteklių:
    a) ištekliai, kurie gali būti naudojami tik tam tikro kiekio procesų vienu metu.
    b) ištekliai, kurie modeliuoja gamybą ir suvartojimą.

Visus išteklių tipus sieją ta pati pagrindinė koncepcija – ištekliai yra susiejami su tam tikrais apribojimais. Procesas gali mėginti kažką paimti arba įdėti į prieinamus išteklius. Jeigu išteklių nėra, procesas turi atsistoti į eilę ir laukti.

Procesai šiame karkase yra aprašyti, kaip python generatoriaus funkcijos. Jos gali būti naudojamos, kaip pavyzdžiui, kūrimas aktyvių komponenčių lyg šie būtų vartotojas, transporto priemonė ar agentas. SimPy taip pat parūpina platų pasirinkimą dalijimąsi ištekliais, kurių pagalba galima modeliuoti apibrėžto didžio vienetus, kaip serverį, žinučių vamzdžius, ir t.t.

Karkasas leidžia simuliacijas paleisti realiu laiku ar rankiniu būdu žingsnis po žingsnio eiti per simuliuojamus įvykius.

Pažiūrėjus iš arčiau SimPy yra asinchroninis įvykių vykdytojas. Karkasas generuoja ir planuoja įvykių nutikimo seką. Įvykiai yra vykdomi pagal prioritetus, simuliacijos laiką ir pagal įvykio unikalų identifikatorių. Įvykiai taip pat turi ilgą sąrašą atgalinių veiksmų (angl.

„callbacks“), kurie yra įvykdomi kai įvykis yra iššaukiamas iš centralizuoto įvykių ciklo. Įvykiai taip pat gali turėti gražinamą reikšmę.

SimPy yra labai lankstus, kai kalbame apie simuliacijos vykdymą. Simuliaciją galima leisti, tol kol nebelieką įvykių, kol tam tikras įvykis nutiko, kol simuliacijos laikas prabėgo arba kol tam tikras įvykis nutiko. Taip pat simuliaciją galima tikrinti eilutę po eilutės parašyto programinio kodo ir sustabdyti po bet kurio įvykio.

2.5. Literatūros apžvalgos apibendrinimas

Paskirstytų sistemų panaudojimas yra labai platus ir šis sisteminis sprendimas yra taikomas vis dažniau. Paskirstytos sistemos, jeigu suprojektuotos teisingai, leidžia sistemai būti atspariai nuo fizinių negandų, visada būti pasiekiamomis, nepriklausomai nuo galimų trukdžių. Tačiau tokias sistemas yra sudėtingiau prižiūrėti, taip pat tai gali padidinti kaštus.

Plačiau nagrinėjamos yra debesijos paskirstytos sistemos, šios sistemos leidžia nešvaistyti resursų ir pasirinkus trečiąją šalį kaip sprendimų tiekėją ir nerimauti dėl infrastruktūrinių sprendimų. Tuo pačiu apžvelgtos ir galimos simuliacinio modelio kūrimo platformos kaip alternatyvos bei pasirinkama naudoti „Simpy“, kaip simuliacinio modelio kūrimo platforma.

Paminėta ir apžvelgta kiber-fizines sistemos, jų naudojimas ir iškylantys iššūkiai simuliuojant sistemas apjungiančias tiek fizinį tiek virtualų pavidalą. Šios sistemos didžiausi iššūkiai yra kontraktų kūrimas ir simuliavimas, taip pat fizinio ir virtualaus simuliavimo sujungimas į vieną simuliacinį modelį.

Svarbu paminėti, kad simuliacijos tikslumas negali būti visiškai tikslus ir visais atvejais atsiras paklaida, dėl tos priežasties yra apibrėžtos tenkinančio tikslumo ribos, ir jas pasiekus laikoma, jog simuliacija pavyko, simuliacija buvo sėkminga.

Rekomenduojamas simuliacinio modelio kūrimo procesas yra sudarytas iš konceptualaus modelio sukūrimo ir jo validavimo. Pagal konceptualų modelį kuriamas realus modelio įgyvendinimas ir atliekamas jo verifikavimas pagal specifikaciją.

Šaltinių apžvalga leido susipažindinti su egzistuojančiais kūrimo, testavimo, simuliavimo, validavimo būdais, egzistuojančiais skirtingų sistemų tipų modeliais, taip pat išsiaiškinti esmines sąvokas reikalingas atlikti tolimesniam darbo vystymui.

3. Teorinė dalis

Šiame skyriuje yra kalbama apie naudojamas technologijas, kurių reikia darbo tikslų įgyvendinimui. Be to, yra aprašomas siūlomas generinis automatizuotas procesas paskirstytos sistemos įvertinimui, įtraukiantis sistemos konceptualaus modelio kūrimą bei visus reikalingus žingsnius sukurti ir analizuoti simuliacinį sistemos prototipą.

3.1. Technologijų apžvalga

Šiame poskyryje visas dėmesys yra skiriamas naudojamoms technologijoms ir programavimo kalboms. Aprašoma kaip jos yra naudojamos ir koks jų funkcionalumas yra naudingas norint sukurti generinį (konceptualų) sistemos modelį bei jo simuliacinį prototipą.

3.1.1. „SimPy“ karkasas ir „Python“ programavimo kalba

Python - galinga ir patogi programavimo kalba, sparčiai populiarėjanti pasaulyje. Bene akivaizdžiausias šios kalbos privalumas yra milžiniškas prieinamų bibliotekų kiekis, padengiančias pačias įvairiausias reikmes (pvz,. matematinių procesų modeliavimą, įvairiapusį duomenų analizavimą bei vizualizavimą, ir t.t.). Tai leidžia supaprastinti ir pagreitinti programų kūrimą, kurti universalias, pagal poreikius pritaikytas programas. Python taip pat labai intuityvi programavimo kalba. Dėl to pradedantiesiems yra nesudėtinga pradėti mokytis programavimo, o labiau patyrę prie šios kalbos pripras akimirksniu.

Su kitomis programavimo kalbomis anksčiau susidūręs žmogus iškart pastebės - tą pačią užduotį galima atlikti šimtu skirtingų būdų! Taip yra todėl, kad Python savyje talpina tiek kitų populiariausių programavimų kalbų sintaksių ypatybes, tiek tūkstančius įvairiausių vidinių funkcijų. Be to, Python leidžia paprastai bei intuityviai užsiimti ir objektiniu programavimu.

Python - interpretuojama programavimo kalba, t.y. kodas analizuojamas programos vykdymo metu. Be milžiniško kiekio vidinių bibliotekų yra įtraukta ir daugybė betipių duomenų struktūrų. Minėti aspektai programuotojui suteikia didžiulę programavimo laisvę, leidžia sutrumpinti kodą bei programos rašymo laiką**.** Visi šie išvardinti privalumai padaro Python labai tinkamu pasirinkimu programinių sistemų prototipų kūrimui.

SimPy [S17] yra diskrečių įvykių simuliacijos karkasas, kuris yra realizuotas kaip standartinis Python kalbos modulis. Sistema yra modeliuojama kaip procesų, bendraujančių diskretinių įvykių pagalba, rinkinys. Pirmoji versija buvo išleista 2002 m. rugsėjį, o naujausia stabili versija buvo išleista 2016 m. rugpjūtį, taigi šis karkasas yra labai brandus.

Karkasas turi sau būdingų sąvokų rinkinį, kurį privalu suprasti:

1. Aplinka (angl. „Environment“). Simuliacijos aplinka yra atsakinga už simuliacijos laiką, jo planavimą ir įvykių apdorojimą. Taip pat aplinkos pagalba yra leidžiama vartotojui žingsnis po žingsnio vykdyti simuliaciją.
2. Įvykis (angl. „Event“). SimPy įvykiai yra labai panašūs į ateityje vykstančius įvykius (angl. „futures“) ir pažadus (angl. „promises“). Taip pat juos galima suprasti kaip sistemos generuojamus signalus ar trigerius. Klasė „Event“ SimPy yra naudojama aprašyti bet kokį įvykį. Yra kelios įvykių rūšių, tiesiogiai įtakojančių įvykių būseną ir jų generavimo tvarką.
3. Proceso sąveika (angl. „Process Interaction“). Diskrečių įvykių simuliacija yra įdomi savo sąveika tarp procesų. SimPy siūlo tris skirtingas sąveikas tarp procesų (kaip specialius įvykių tipus):
    d) laukti, kol kam nors taps reikalingas;
    e) laukti, kol kitas procesas pasibaigs;
    f) pertraukti kitą procesą.
4. Bendri ištekliai (angl. „Shared Resources“). Ištekliai yra kitas būdas sąveikauti tarp procesų. Yra dvi kategorijos šių išteklių:
    c) ištekliai, kurie gali būti naudojami tik tam tikro kiekio procesų vienu metu;
    d) ištekliai, kurie modeliuoja gamybą ir suvartojimą.

Visus išteklių tipus sieja ta pati pagrindinė koncepcija – ištekliai yra asocijuojami su tam tikrais apribojimais. Procesas gali mėginti kažką paimti arba įdėti į prieinamus išteklius. Jeigu išteklių nėra, procesas turi atsistoti į eilę ir laukti.

Procesai šiame karkase yra aprašyti, kaip Python generatoriaus funkcijos. Jos gali būti naudojamos, kaip pavyzdžiui, sukurti aktyvioms sistemos komponentėms tokioms kaip vartotojas, transporto priemonė ar agentas. SimPy taip pat parūpina platų pasirinkimą kaip dalytis sistemos

ištekliais, pvz. galima modeliuoti apibrėžto dydžio vienetus, kaip buferius, žinučių vamzdžius, ir t.t.

Karkasas leidžia sistemos simuliacijas paleisti realiu laiku (apribojant jų maksimalų kiekį bei kiekvienos jų laiką) arba rankiniu būdu, žingsnis po žingsnio einat per simuliuojamus įvykius.

Pažiūrėjus iš arčiau, SimPy yra asinchroninis įvykių vykdytojas. Karkasas generuoja ir planuoja įvykių nutikimo seką. Įvykiai yra vykdomi pagal prioritetus, simuliacijos laiką ir pagal įvykio unikalų identifikatorių. Įvykiai taip pat gali turėti gražinamą reikšmę.

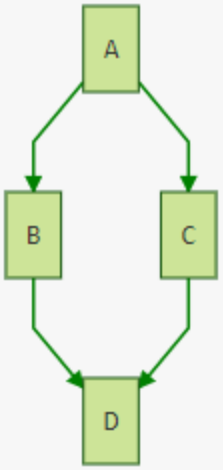
SimPy yra labai lankstus, kai kalbame apie simuliacijos vykdymą. Simuliaciją galima leisti, tol kol nebelieka įvykių, kol tam tikras įvykis nutiko, arba kol simuliacijos laikas prabėgo. Taip pat simuliaciją galima tikrinti eilutę po eilutės parašyto programinio kodo ir sustabdyti po bet kurio įvykio.

Dėl tokio savo lankstumo, brandumo, patogumo ir paprastumo SimPy ir buvo pasirinktas simuliacinio sistemos prototipo kūrimui.

3.1.2. „Mermaid“ vizualizacija

Grafinis sistemos modelis (diagrama) yra svarbi priemonė komunikacijai ir informacijos apie paskirstytą sistemą dalijimuisi. „Mermaid“ [M17] vizualizacijos kalba leidžia sukurti paprastų paskirstytų sistemų diagramas. Paprasta sintaksė nulėmė šios vizualinės kalbos pasirinkimą. Sintaksė yra labai paprasta ir remiasi sistemos blokų ir jų sąryšių aprašu rodyklių pagalba (-->).

```
graph TD;
    A-->B;
    A-->C;
    B-->D;
    C-->D;
```



3.1. pav. „Mermaid“ sintaksės ir atvaizdavimo pavyzdys

Paveiksle (2.1. pav.) yra parodyta panaudota sintaksė ir grafinis jos atvaizdavimas. Tai yra paprastas pavyzdys, tačiau galima įvairiai grupuoti sistemos komponentes ir taip sukurti komponenčių spiečius ar komponentes, kurios savyje turėtų kitas mažesnes komponentes. Galiausiai grafiškai atvaizduojama visa paskirstytos sistemos diagrama.

Šis įrankis darbe yra naudojamas tam, kad sukurti sistemos konceptualų modelį, kuris bus tiesiogiai naudojamas kuriant simuliacinį sistemos prototipą.

3.2. Diskrečių įvykių simuliacija

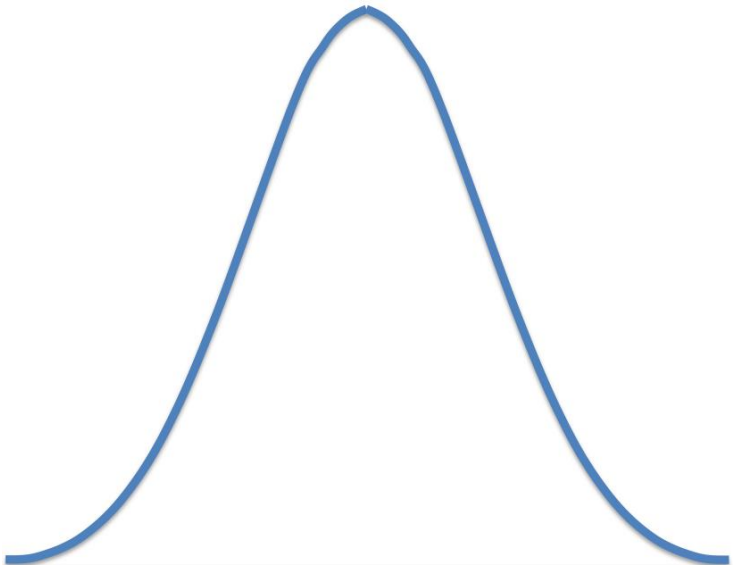
Simuliacinio sistemos prototipas ir jo veikimas remiasi sistemoje generuojamais diskrečiais įvykiais. Tai yra sistemos veikimo modelis, kurį galima įvardinti, kaip diskrečių įvykių seką laike. Kiekvienas įvykis nutinka tam tikru metu ir pakeičia sistemos būseną. Tarp iš eilės

einančių įvykių jokie sistemos būsenos pokyčiai neturi įvykti, dėl šios priežasties simuliacijos metu galima šokinėti tarp iš eilės einančių įvykių. Taip yra sutrumpinamas simuliacijos laikas ir įgalinamas būsenos sekimas simuliacijos metu.

Diskrečių įvykių simuliacijos ciklą būtu galima apibrėžti taip:

1. Pradžia:
    a. Inicijuojama ciklo pabaigos sąlyga;
    b. Inicijuojamos sistemos stebimos metrikos (tam tikri būsenos kintamieji ar jų išraiškos);
    c. Inicijuojamas laikrodis, dažniausiai simuliacijos pradžioje būna 0;
    d. Sukuriamas ir įdedamas pirmasis įvykis į įvykių sąrašą.
2. Ciklas:
    a. Kol pabaigos sąlyga nėra tenkinama, tol ciklas yra tęsiamas;
    b. Laikrodžio „laikas“ yra keičiamas pagal sekančio įvykio vykdymo laiką;
    c. Įvykdomas įvykis ir ištrinamas iš įvykių sąrašo;
    d. Galimai generuojamas naujas įvykis ar įvykiai, kurie įtraukiami į sąrašą;
    e. Atnaujinamos ir išsaugomos stebimų metrikų reikšmės.
3. Pabaiga:
    a. Generuojama metrikų ataskaita.

Jei įvertinti sistemos modeliui prireikia daug simuliacijų vykdymų, atitinkamos metrikų ataskaitos yra apjungiamos ir tada analizuojamos.



3.2. pav. Tikimybinio sistemos parametro (normalusis) reikšmių pasiskirstymas

Simuliuojamos sistemos rezultatai tiesiogiai priklauso nuo pasirinktų sistemos parametrų (tokių kaip komponenčių skaičius, buferio dydis, ir t.t.). Kai kurie sistemos parametrai (pvz., ateinančių užklausų dažnis, tikėtinas apdorojimo laikas) gali būti išreikšti tikimybiškai, t.y., kaip tikimybinis reikšmių pasiskirstymas. Jei tikimybė yra vienoda, konkreti reikšmė gali būti sugeneruota naudojantis atsitiktinių skaičių generavimu iš užduoto intervalo. Kitais atvejais, reikšmė yra gaunama pagal duotą tikimybinį skirstinį (žr. Pav. 3.2). Python turi bibliotekines funkcijas (žr., pvz., [P17]) įvairiems skirstiniams padengti. Toks reikšmių generavimas užtikrina tikslesnius simuliacijos rezultatus.

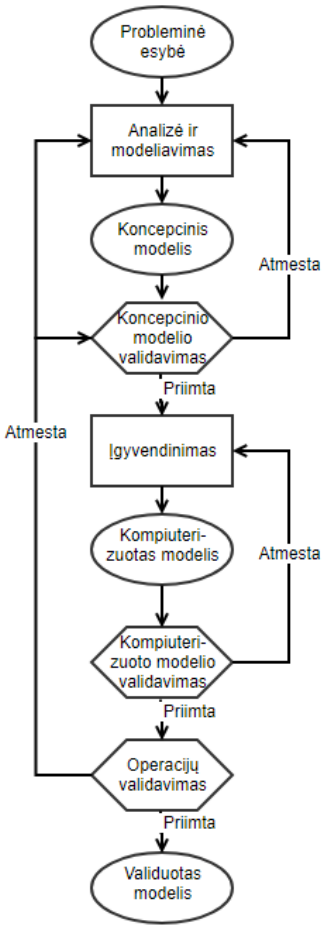
Toks simuliacinis sistemos prototipas yra palaikomas SimPy karkaso ir dėl savo prigimties puikiai tinka naudoti paskirstytų sistemų simuliacijoms. Viena to priežasčių yra tai, kad didžioji dalis paskirstytų sistemų veikimo laiko yra išnaudojama komponenčių komunikacijai, kuris simuliacijoje nėra įskaičiuojamas į simuliacijos vykdymo laiką, kaip tai yra daroma kitų simuliacinių metodikų (daugiau apie tai, žr. [R15]).

3.3. Sistemos modeliavimo, prototipo kūrimo ir įvertinimo procesas

Šiame poskyryje trumpai aprašysime visą procesą, padengiantį paskirstytos sistemos modeliavimą, prototipo kūrimą ir įvertinimą. Taip pat aptarsime konceptualaus modelio elementų konvertavimą į Python programinio kodo komponentes.

3.3.1. Procesas

Grafiškai šis procesas gali būti atvaizduotas taip (Pav. 3.3):



3.3. pav. Sistemos modeliavimo ir įvertinimo proceso diagrama

Konceptualus sistemos modelis yra kuriamas ir validuojamas bendro paskirstytos sistemos aprašo ar reikalavimų dokumento pagrindu. Tada šis modelis konvertuojamas į sistemos prototipą (kompiuterizuotą modelį), kuris yra validuojamas jį simuliuojant (vykdant) ir tuo pačiu stebint iš anksto žinomas sistemos metrikas (taip pat besiremiančias sistemos reikalavimais).

3.3.2. Vizualizacija ir jos vertimas į sistemos simuliacinį prototipą

Vizualiniai konceptualaus modelio elementai yra kuriami naudojant „Mermaid“ technologija. Pastarieji elementai yra tada verčiami į prototipo kodą, kurio pagalba yra atliekamos simuliacijos.

„Mermaid“ yra supaprastinta vizualizacijos technologija, kuri leidžia atvaizduoti komponentes ir jų tarpusavio sąryšius, tačiau per tuos pačius elementų tipus. Dėl šios priežasties elementai ir jų paskirtis yra atskiriami pagal jų pavadinimą, nurodytą konceptualiame modelyje. Analogiškai, komponenčių sąryšiai yra identifikuojami pagal vizualizacijoje nurodytą sąryšį.

Įgyvendinant transliaciją tarp “Mermaid” diagramų ir prototipo Simpy kodo, komponentės paprastai virsta SimPy procesais, tuo tarpu sąryšiai dažnai tampa Simpy tarpusavyje naudojamais ištekliais, bendras priėjimas prie kurių ir sudaro sąryšį. Komponentės esančios kitų komponenčių viduje tampa funkcijomis, kurios yra paeiliui kviečiamos išorinės komponentės SimPy proceso.

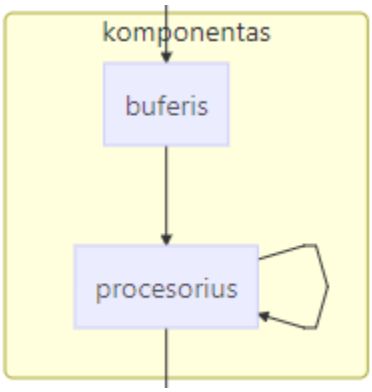
4. Praktinė dalis

Šiame skyriuje yra aprašyti praktiniai rezultatai.

4.1. Vizualių konceptualaus modelio elementų sąryšis su paskirstytų sistemų elementais

Kaip jau buvo minėta ankstesniame skyriuje vizualizacijai yra naudojama „Mermaid“ technologija. Taigi šis poskyris yra orientuotas į paskirstytos sistemos komponentų susiejimą su vizualiais elementais konceptualaus modelio diagramoje.

Elementai patalpinti kitame elemente yra atliekantys savo darbą viename procese (iš „SimPy“ skyriaus), kaip matyti 4.1. pav.



4.1. pav. Elemento vizualizacija „Mermaid” aplinkoje

Šiuo atveju *buferio* ir *procesoriaus* elementai veikia nuosekliai viename procese, nes yra patalpinti juos apimančiame didesniame elemente (pvz., sistemos komponentėje). Taigi perduodama informacija, toliau žinutė, pasiekusi *buferio* elementą bus saugoma jame. Tuo tarpu *Procesoriaus* elementas pasiima naują žinutę iš *buferio* elemento, kai baigia darbą su prieš tai buvusia žinute arba kai *buferio* elementas praneša *procesoriaus* komponentui apie egzistuojančias neapdirbtas žinutes.

Bendru atveju, konceptualaus sistemos modelis (grafinė diagrama) susidės iš aibės tokių procesų, nuosekliai veikiančių proceso (komponentės) viduje ir tuo pačiu potencialiai lygiagrečiai veikiančių tarpusavyje, sinchronizuojančių savo veiklą žinučių pagalba.

```
subgraph komponentas

buferis1[buferis]-->procesorius1[procesorius]
procesorius1[procesorius]-->procesorius1[procesorius]
end
```

4.1. prog. kodas. Naudotas sugeneruoti 4.1. pav. segmentą

Mažiausio elemento vieneto, „blokelio“, atliekama paskirtis yra nustatoma pagal jo rodomą pavadinimą. Pavadinimas yra įrašomas į laužtinius skliaustus, o unikalus elemento identifikatorius yra užrašomas prieš laužtinius skliaustus. Taigi šiuo atveju *buferio* elementas turi žinoti apie naudojamą duomenų saugojimo kolekciją, kurią ir naudoja žinutėms saugoti, kol *procesoriaus* elementas geba jas apdoroti. Savo ruoštu *procesoriaus* elementas turi taip pat galėti modifikuoti žinučių kolekciją, nes šis elementas apdoroja (pašalina) žinutes.

4.2. Rezultatai

Šiame poskyryje dėmesys yra skiriamas gautiems rezultatams ir jų pristatymui. Taip pat aprašoma, kaip pastarieji buvo gauti ir kokia yra jų nauda tolimesniam tiriamojo darbo plėtojimui.

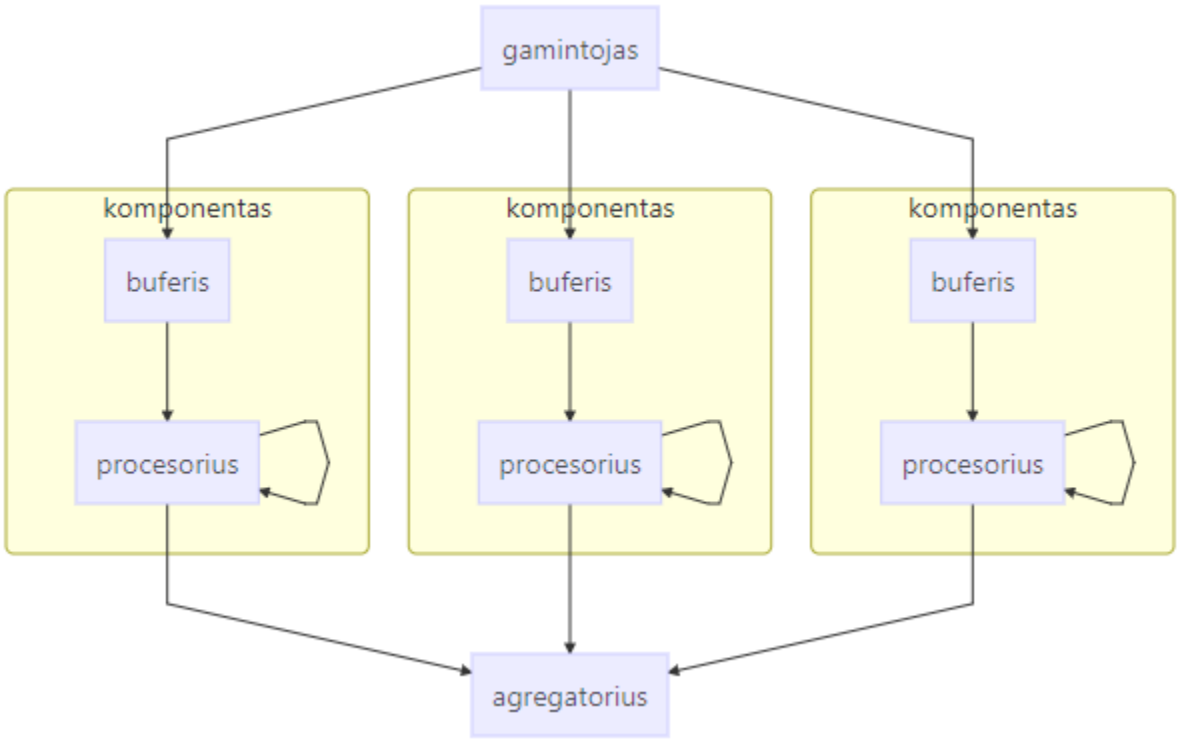
4.2.1. Konceptualaus sistemos modelis

Konceptualaus tiriamos paskirstytos sistemos modelis buvo sukurtas, siekiant praktiškai ištyrinėti SimPy karkaso galimybes įgyvendinant bazines paskirstytos sistemos funkcijas, t.y. realizuojant horizontalų sistemos plėtimą (dinaminį komponentų kiekį) ir paralelų operacijų vykdymą.

Iš 4.2. pav. matyti jog egzistuoja trijų tipų komponentai, *gamintojas, komponentas* ir *agregatorius*. Kiekvienas iš šių atskirų elementų veikia paraleliai vienas kito atžvilgiu, taigi operacijos, atliekamos kiekvieno iš išvardintų komponentų, nesidalija skaičiavimų galia ir turi sau dedikuotą centrinį apdorojimo vienetą.

Sistemos komponentės yra ne tik yra paraleliai veikiančios, tačiau jos yra taip pat plečiamos, t.y. pridedant papildomų *komponenčių* gaunama našesnė sistema. Taigi, teoriškai padvigubinus *komponenčių* kiekį, sistema turėtų būti beveik dvigubai našesnė, su išlyga, jog kitos sistemos grandys yra pajėgios apdoroti apkrovas.

Taigi 4.2. pav. yra pavaizduotas naudojamas konceptualus modelis, kurio pagalba tolimesniuose skyriuose bus kuriamas ir aprašomas simuliacinis sistemos prototipas bei gaunami skaitmeniniai įvertinimai, priklausantys nuo prototipo konfigūracijos (konkrečių sistemos parametrų reikšmių).



4.2. pav. Konceptualus modelis

4.2.2. Simuliacinio prototipo kūrimas

Prototipinio kompiuterizuoto modelio kūrimas yra konceptualaus modelio konvertavimas į programinį kodą, kuris atlieka sistemos simuliaciją su sistemos architektui svarbiomis simuliuojamomis sistemos charakteristikomis (t.y., sistemos parametrų reikšmėmis). Pavyzdžiui, su tam tikru pasirinktu *komponenčių* iš 4.2. pav. kiekiu. Keičiant šį parametrą, yra stebimas neapdorotų žinučių kiekis (t.y., stebima metrika) tam tikrais laiko intervalais.

Taigi pats kūrimo procesas vyksta pagal 4.3. pav. nurodytą diagramą. Neriant gilyn į kompiuterizuoto modelio konstrukciją, buvo išskirti kiekvienas iš konceptualaus modelio elementų į atskirą nedalomą vienetą, kurie gali būti sukuriami (ir, svarbiausia, vykdomi)

nepriklausomai vienas nuo kito. Į šiuos skirtingus elementus buvo išskirti *gamintojas*, *komponentas* ir *agregatorius*.

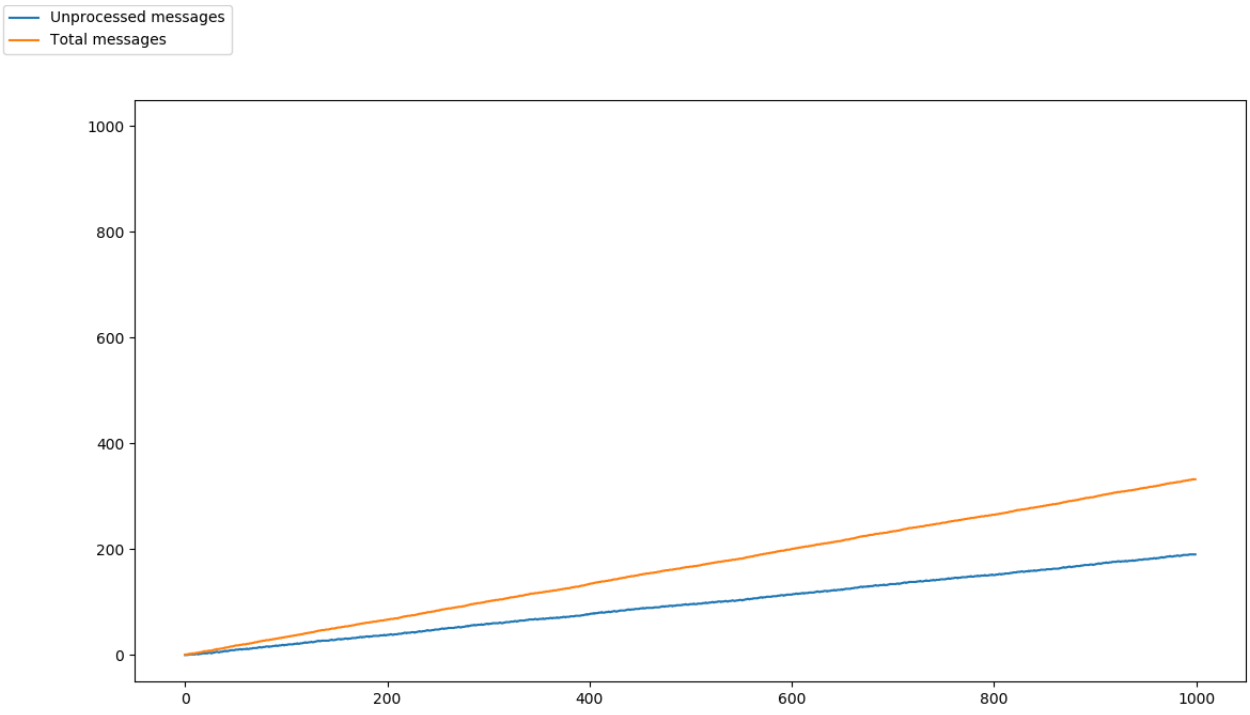
```
class Publication(object):
    def __init__(self, env, store):
        self.env = env
        self.store = store
        self.produced_messages = []

    def start_publishing(self):
        while True:
            print('Message sent to components %d' % self.env.now)
            self.store.put('msg')
            self.produced_messages.append(len(self.produced_messages) + 1)
            yield rand_latency(self.env, 20,40)
```

4.2. prog. kodas. naudojamas gamintojo elemento veikimui

Kaip jau buvo minėta, *gamintojo* elementas turi priėjimą prie žinučių buferio ir jį modifikuoja pridedamas žinutes, kai tuo tarpų *komponentės* elementas jas apdoroja ir ištrina. Taip pat verta paminėti, jog tam tikros sistemos parametrų reikšmės darbo yra generuojamos pagal tikimybinį skirstinį (3.2. pav.). Tokie parametrai yra atvykstančių naujų žinučių dažnis ir vidutinis žinutės apdorojimo laikas. Pirminiame prototipo variante, šios reikšmės buvo pasirinktos kaip konstantos arba generuojamos kaip pseudo atsitiktiniai skaičiai iš tam tikro intervalo (t.y., su vienoda tikimybe). Tokio generavimo pavyzdys yra funkcijos rand_latency iškvietimas (4.2. pav.) modeliuojantis pauzes tarp naujų žinučių (užklausų) atvykimo.

Prototipo kūrimas taip pat įtraukia metrikų stebėjimą simuliacijos metu ir jų atvaizdavimą simuliacijai pasibaigus. Grafinis metrikų atvaizdavimas leidžia lengviau analizuoti simuliacijos rezultatus bei sistemos parametrų reikšmes, kurios gali įtakoti našesnį sistemos veikimą.

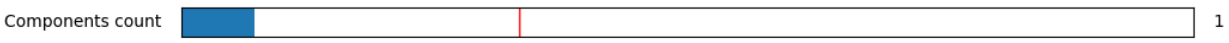
Pav. 4.3 rodo išsiųstų ir apdorotų žinučių kiekį tam tikrais laiko intervalais simuliacijos metu. Ši stebima metrika yra tiesiogiai generuojama dviejų žinučių apdorojimo *komponenčių* (4.2. pav.). Iš pavaizduotų rezultatų matyti, jog šis kiekis nėra pakankamas apdoroti visas *gamintojo* (4.2. pav.) sukuriamas žinutes ir atsiranda tiesinis neapdorotų žinučių augimas, todėl šiuo atveju reikia padidinti *komponenčių* (4.2. pav.) kiekį, norint patenkinti sistemos veikimo apkrovą.



4.3. pav. Simuliacijos rezultatų charakteristika

4.2.3. Simuliacinio prototipo konfigūravimas

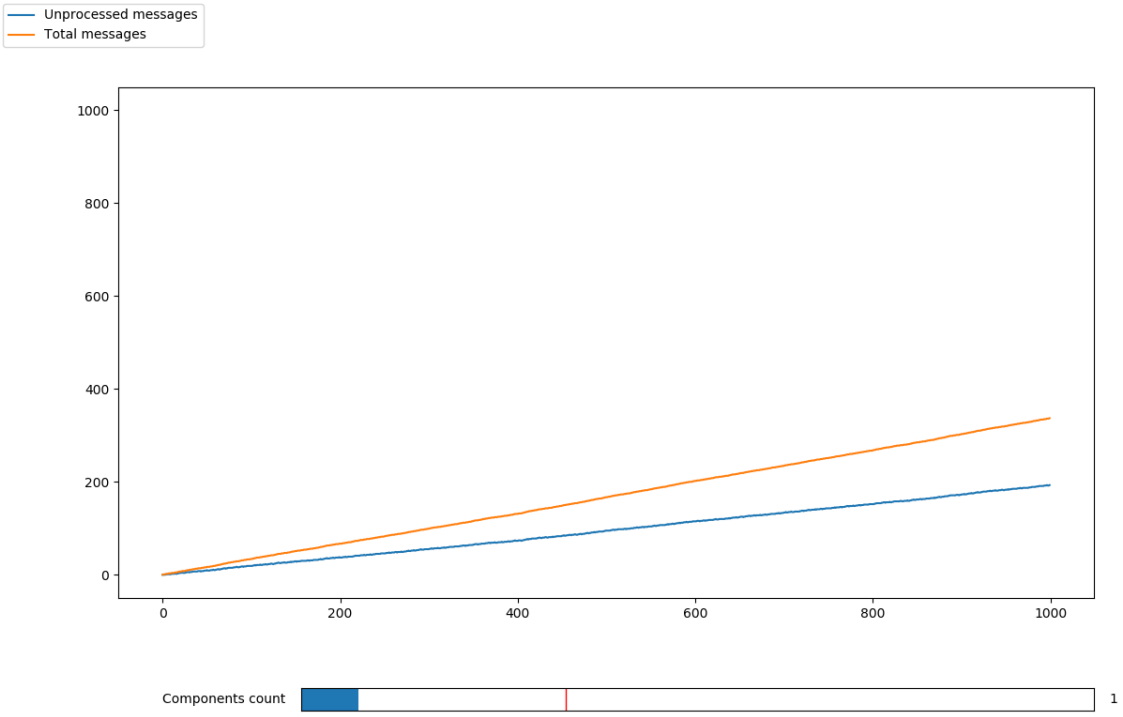
Pratęsiant praeitame skyriuje plėtotą mintį apie parametrų keitimą ir skirtingų charakteristikų gavimą, dabar aptarsime, kaip sukūrus prototipą jį galima modifikuoti, keičiant sistemos parametrų reikšmes. Prototipo simuliacijos yra vertinamos pagal stebimas metrikas, kurių reikšmės yra tiesiogiai priklausomos nuo einamųjų konfigūruojamų sistemos parametrų.



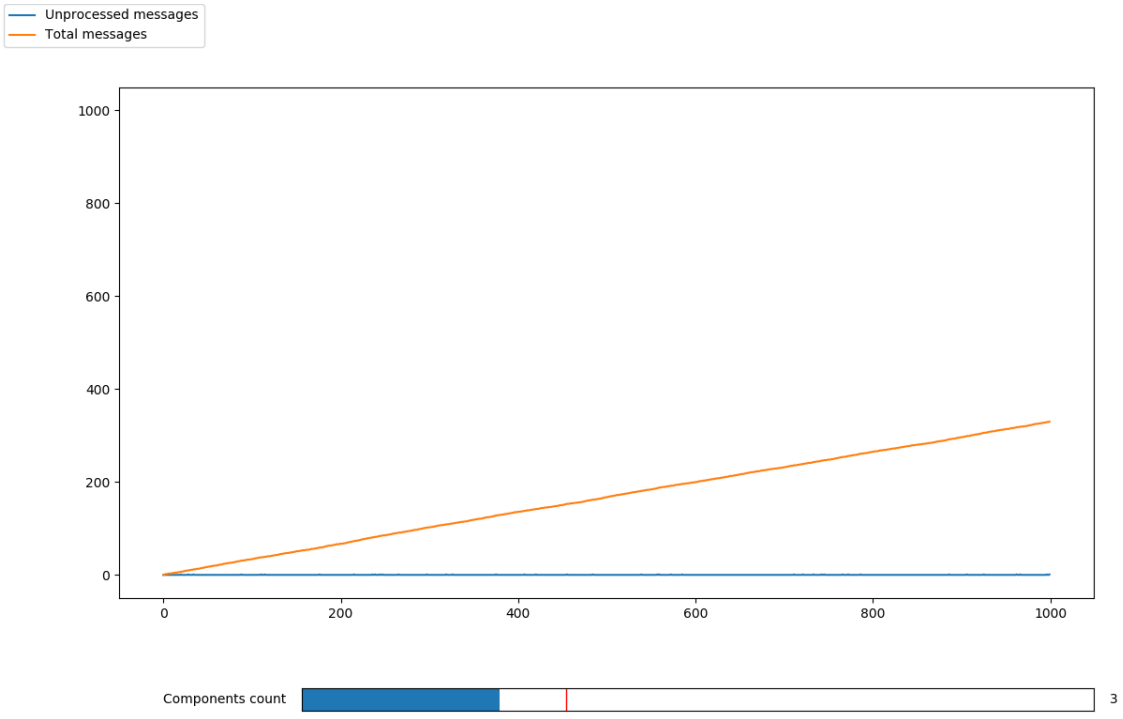
4.4. pav. Konfigūruojamas *komponenčių* (4.2. pav.) simuliacijos metu kiekis

Paveiksle (4.4. pav.) yra parodytas konfigūruojamas komponenčių skaičiaus slankiklis, kurio pagalba yra galima dinamiškai pakeisti šio esminio sistemos parametro reikšmę. Toks pakeitimas automatiškai paleidžia naują sistemos simuliaciją ir tuo pačiu pergeneruoja simuliacijos metrikas su nauja parametro reikšme. Parodytame paveiksle (4.4. pav.) parametro

reikšmė atitinka vieną apdorojančią komponentę (4.2. pav.), kas priveda prie lėčiausio galimo žinučių apdorojimo šioje sistemoje.



4.5. pav. Išsiųstu ir apdorotų žinučių charakteristika su viena *komponente* (4.2. pav.)



4.6. pav. Išsiųstu ir apdorotų žinučių charakteristika su trimis *komponentais* (4.2. pav.)

Grafinėje sąsajoje esantis slankiklis (4.4. pav.) matomas 4.5. ir 4.6. pav. yra nustatytas atitinkamai vienetui ir trejetui. Tai reiškia, jog 4.5. pav. yra tik viena aktyvi komponentė (4.2. pav.) apdorojanti žinutes, kai tuo tarpu tokių komponenčių 4.6. pav. yra trys ir tai atsispindi išsiųstų ir apdorotų žinučių stebimose metrikose. 4.6. pav. paleista simuliacija susidoroja su esamu *gamintojo* sukuriamu krūviu, o 4.5. pav. kur yra viena komponentė, ne.

Grafinėje sąsajoje atlikti pakeitimai sukuria įvykį (4.3. prog. kodas.), kuris paleidžia simuliaciją iš naujo su pakeistu parametru (4.3. prog. kodas.). Taip yra sugeneruojama nauja simuliacijos metrikos reikšmė.

```
components_slider.on_changed(run_sim)
```

4.3. prog. kodas. Įvykio prenumeravimas su paduodama nuoroda į metodą

Pakeistos reikšmės įvykio prenumeravimas nutinka įvykdžius 4.3. prog. kodą. Nutikus reikšmės pasikeitimui bus iškviečiama funkcija *run_sim* (4.3. prog. kodas) ir taip iš naujo sugeneruojama simuliacijos statistika, su nauja nurodyta reikšme.

5. Galutiniai tyrimo rezultatai

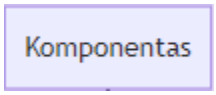
Tyrimas yra orientuotas į kuriamų paskirstytų sistemų kokybės gerinimą jas simuliuojant ir „laužant“ su tai sistemai būdingais parametrais. Parametrai, kontraktai ir kitos simuliacinio modelio dalys yra vienodai svarbios bendram simuliacijos tikslui pasiekti. Esant validžiam simuliaciniam modeliui galima pradėti sistemos tyrinėjimą ir pokyčius su skirtingais parametrais.

5.1. Esybių žodynas

Didelės paskirstytos sistemos yra sudarytos iš daugybės komponentų. Komponentai yra bene svarbiausia paskirstytos sistemos dalis, jie dažniausiai atlieką daugiausiai laiko reikalaujančias užduotis, kurios transformuoja, manipuliuoja duomenis, taip generuodamos rezultatus.

Simuliacinio modelio automatizuotam generavimui yra reikalingas tiksliai aprašytas konceptualus modelis ir tikslus simuliacinės aplinkos koncepcijų žodynas į simuliacinio modelio komponentus. Šią informaciją naudoja programinis kodas atliekantis automatinį simuliacinio modelio generavimą iš konceptualaus modelio.

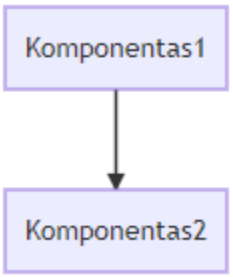
Automatinio simuliacinio modelio generavimo žodynas:

1) 

- komponentas visada turi savo veikimo ar aptarnavimo dažnį (angl. service rate), tai yra neatsiejama komponento dalis, kuri nusako, kokiu greičiu komponentas atlikts jam paskirtas užduotis. Jeigu komponentas atlieka žinučių apdorojimą, tai nuo aptarnavimo dažnio priklausys, kokiu intervalu jis apdoroją vieną žinutę. Taigi simpy terminais komponentas visada turės *timeout* su normaliuoju tikimybių parametro reikšmės pasiskirstymu, tai ir yra kitamas aptarnavimo dažnis.

2)  - sąryšiai nurodo komponentų tarpusavio sąsajas. Paskirstytos sistemos turi daugybę komponentų, komponentams reikia komunikuoti tarpusavyje, tam yra naudojamos

žinutės. Vienas komponentas kuria žinutes, kitas tuo tarpu jas priima ir apdoroja tam tikru intervalu ar aptarnavimo dažniu. Žinutės simuliacinėse aplinkose yra vadinamos ištekliais. Simpy turi trijų tipų išteklius, vienas iš jų yra *store*, kuris yra naudojamas, kaip žinučių talpykla ar keitimosi mechanizmas.

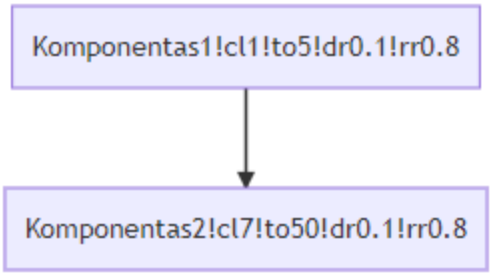
3) 

- minimalus paskirstytos sistemos modelis. Paskirstytos sistemos esybę simpy aplinkoje atitinka *environment*. Taigi šiuo atveju šis minimalus paskirstytos sistemos modelis atliks savo darbą perduodamas žinutes tarp komponentų pagal standartinį laiko intervalą su tikimybinių paskirstymu. Taigi konkretizuojant *Komponentas1* kurs žinutes standaritinių laiko interavalu su tikimybinių paskirstymų, *Komponentas2* apdoros po vieną žinutę standartiniu laiko intervalu su tikimybinių paskirstymų. Neegzistuojant tikimybiniam paskirstymui kiekviena šio modelio simuliacija baigtųsi vienodu rezultatu, tačiau tikimybinis paskirstymas prideda tikro pasaulio nenuspejamumą, pasireiškianti kaip tinklo sulėtejimus ar kitas technines kliūtis.

Esybių žodynas yra skirtas paviršutiniškai supažindinti su konceptualaus modelio automatizuotų kūrimu.

5.2. Konceptualus modelis (parametrai)

Konceptualus modelis yra automatizuoto simuliacinio modelio kūrimo pradžia. Pagal konceptualų modelį programinė įranga sukuria simuliacinį modelį, kuri vėliau galima manipuliuoti ir stebėti gaunamus simuliacijos rezultatus.



5.2.1. pav. Minimalus konceptualaus modelio pavyzdys.

graph TB

Komponentas1!cl1!to5!dr0.1!rr0.8-->Komponentas2!cl7!to50!dr0.1!rr0.8

5.2.1. prog. kod. 5.2.1 pav. minimalaus konceptualaus modelio grafiko kodas.

Konceptualaus modelio parametrai prasideda po pirmojo *!* ženklo ir toliau yra pridedami vienas po kito. Parametrų formatas - !<*pavadinimas*><*reikšmė*>. Taigi turint parametrą *to* (angl. timeout) ir priskiriant *50*, kaip šio parametro reikšmę - konceptualiame modelyje tai atrodys - *!to50*.

Šiuo metu egzistuojantys ir programai suprantami parametrai:

1) *cl (!cl2)* (angl. clone) - klonavimo parametras atlieka parametrų dauginimo funkciją. Pati pirma simuliacija bus atlikta su šiuo atveju dviem tokiais pačiais komponentais, kurie atliks žinučių apdorojimą, tik dvigubai greičiau nei vienas toks komponentas. Šį parametrą galima keisti programos gyvavimo metu, taigi jis yra naudojamas tik pirmai simuliacijai, iki tol kol yra pakeičiamas.
2) *to (!to10)* (angl. timeout) – apdorojimo dažnis yra naudojamas komponentų veikimo greičiui unifikuotais laiko vienetais išreikšti. Šiuo metu simuliacija tęsiasi 1000 laiko vienetų. Taigi turint apdorojimo dažni 10, komponentas atliks 100 ėjimų, vienos simuliacjos metu. Šiuo parametrų yra valdomas komponento darbo greitis, tai dažniausiai yra siejama su komponento vienos žinutės apdorojimo laiku.
3) *dr (!dr0.01)* (angl. death rate) – komponentės mirties dažnis. Šis parametras yra naudojamas simuliuoti komponento galimybę išsijungti ir nebeatlikti savo funkcijos, pavyzdžiui ne apdoroti žinučių. Šiuo atveju 0.01 yra parametro reikšmė, kuri indikuoja, jog šis parametras gali beveikti su 1% galimybę. Pagal tikimybių teoriją šis komponentė kas 100 ėjimų turėtu pradėti nebeveikti.

4) *rr (!rr0.8)* (angl. recovery rate) – komponentės prisikėlimo dažnis yra naudojamas nustatyti, kokia tikimybė yra komponentei pradėti savo darbo po mirties scenarijaus. Taigi jeigu komponentė numirė ir nebe atlieka savo funkcijos kitą savo eilę jį turi galimybę šiuo atveju 4/5 kartų prisikelti ir pratęsti savo darbus.

Naudojant 5.2.1. pav. ir jo pirmąjį komponentą, kuris yra *Komponentas1!cl1!to5!dr0.1!rr0.8*. Galime konkrečiai įvardinti kokiomis charakteristikomis šį komponentė pasižymės. Taigi ši komponentė bus vieną, jį gamins žinutes kas penkis laiko vienetus, kurias apdoros jos žinučių laukiančios komponentės. *Komponentas1* gali nustoti dirbti su 10% galimybę, tai yra kas 10 ėjimą šį komponentė turėtu įeiti į mirties būseną. Esant mirties būsenai šį komponentė turi 80% galimybę atsistatyti į veikimo režimą.

Konceptualiame modelyje parametrai atlieką svarbų vaidmenį aprašydami, kaip komponentės elgsis simuliacijos metu. Pristatytas tokių parametrų sąrašas yra pradinis ir gali būti toliau plečiamas. Automatizuoto vertimo proceso realizacija yra tokia, kad toks plėtimas daugeliu atvejų susives į naujų funkcijų ar metodų pridėjimą ir nesukels esminių sunkumų.

5.3. Simuliacinis modelis

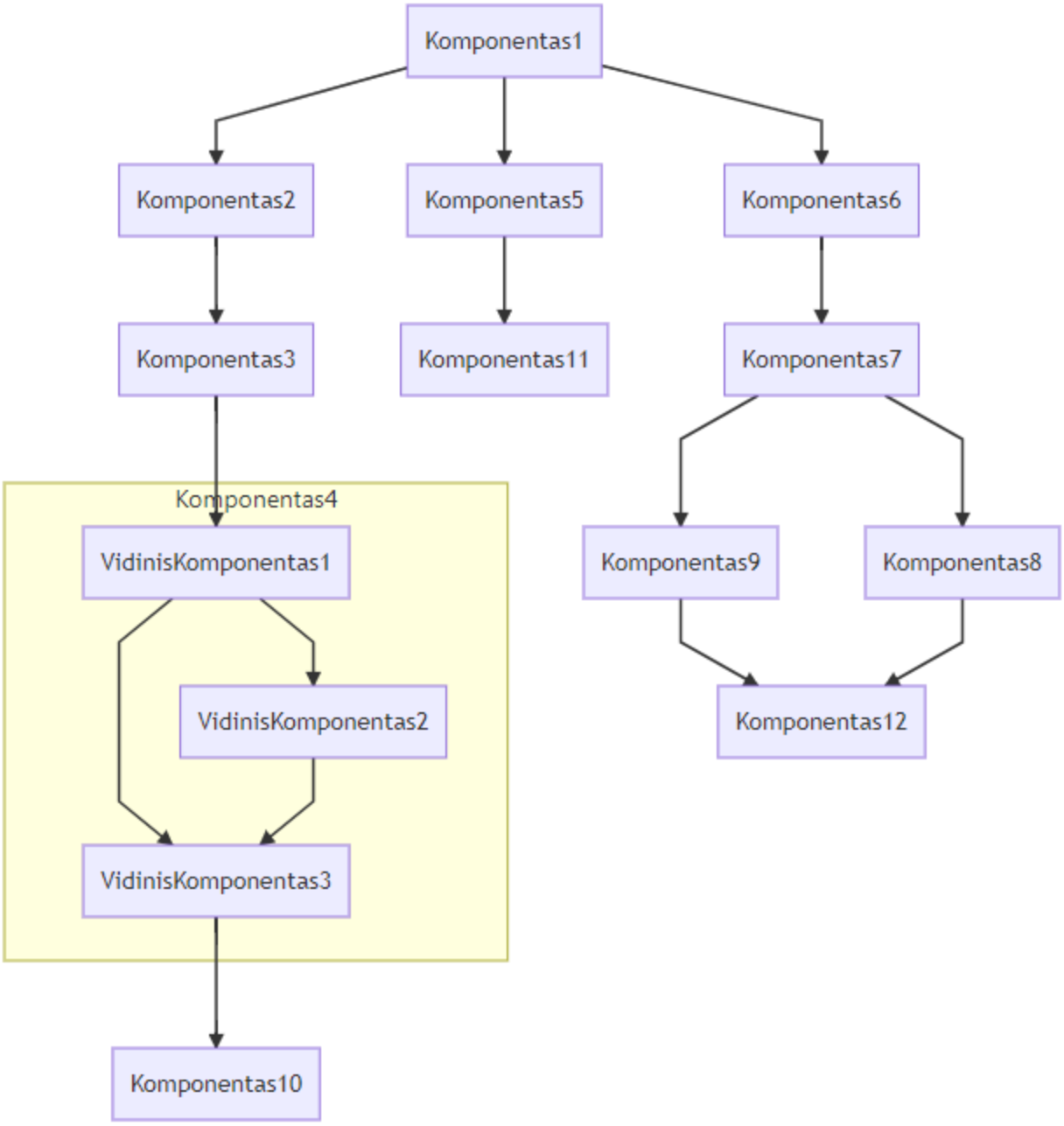
Simuliacinis modelis yra gautas rezultatas iš konceptualaus modelio. Kuris yra naudojamas simuliacijai atlikti. Simuliacinis modelis stebi tiek įvesties tiek išvesties parametrus, kurie simuliacijos pabaigoje yra pavaizduojami ir leidžia vartotojui spręsti apie simuliacijos prasmę ir naudą.

5.3.1. pav. pavaizduotas plikas konceptualus modelis, be jokių parametrų, tik struktūra, be jokių parametrų simuliacinio modelio automatinis generatorius priims sprendimus už vartotoją, kaip komponento apdorojimo laiką, visos simuliacijos laiką, kiek komponentų bus sukurta šiai simuliacijai. Nebus atvaizduojama jokių pasirenkamųjų komponentų parametrų. Paprastai tariant tai bus statinis simuliacinis modelis.

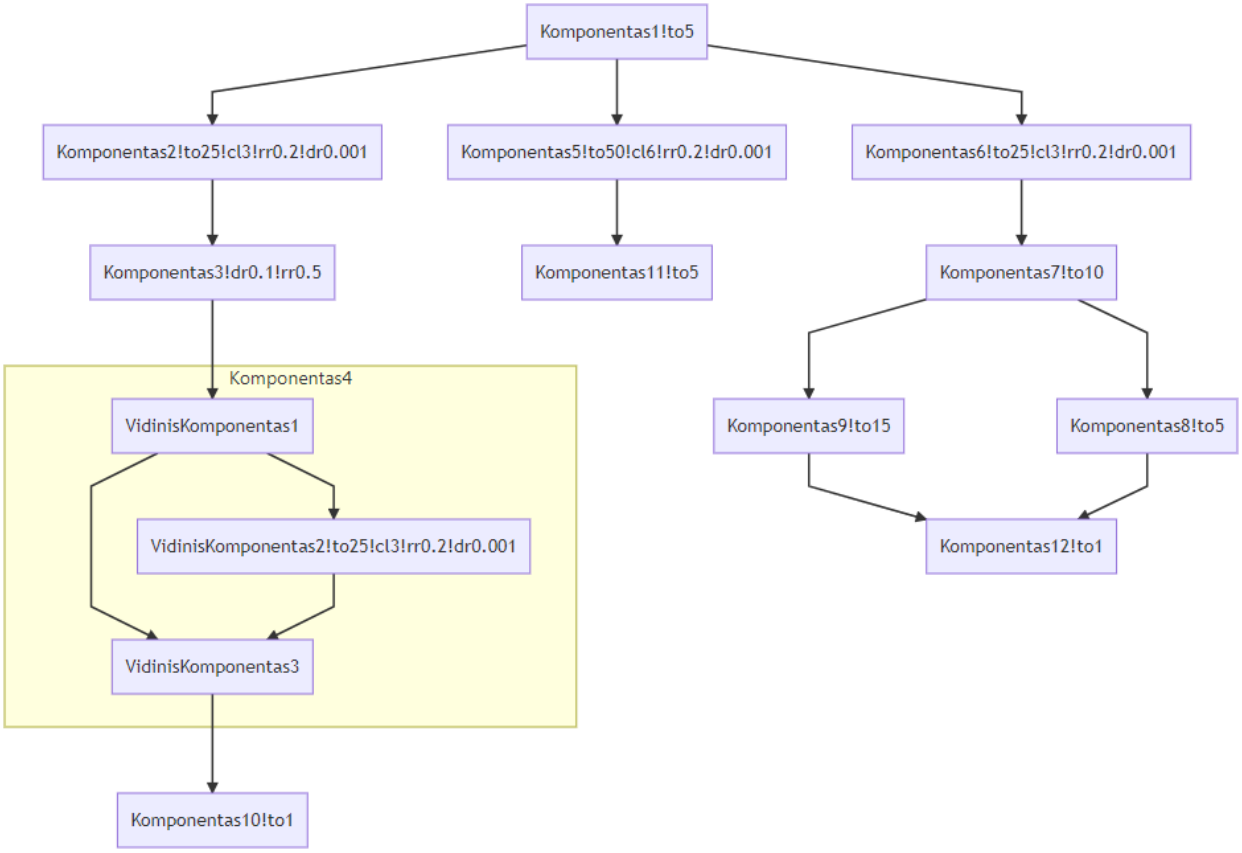
5.3.2. pav. yra pavaizduotas pilnas konceptualus modelis su visais parametrais, kuriu dėka simuliacinio modelio generatorius žino, kaip reikia sukurti ir vėliau manipuliuoti simuliaciją iš konceptualaus modelio. Taigi vienas pradinis *Komponentas1!to5* generuoja žinutes, kas penkis

laiko vienetus ir vėliau išsiunčia tas pačias žinutes kiekvienam jo besiklausiančiam komponentui. Šių žinučių klausosi komponentai 2, 5 ir 6. Kurie turi savo taisykles, kaip jie turi veikti, daugintis ir atsistatyti nelaimės atveju.

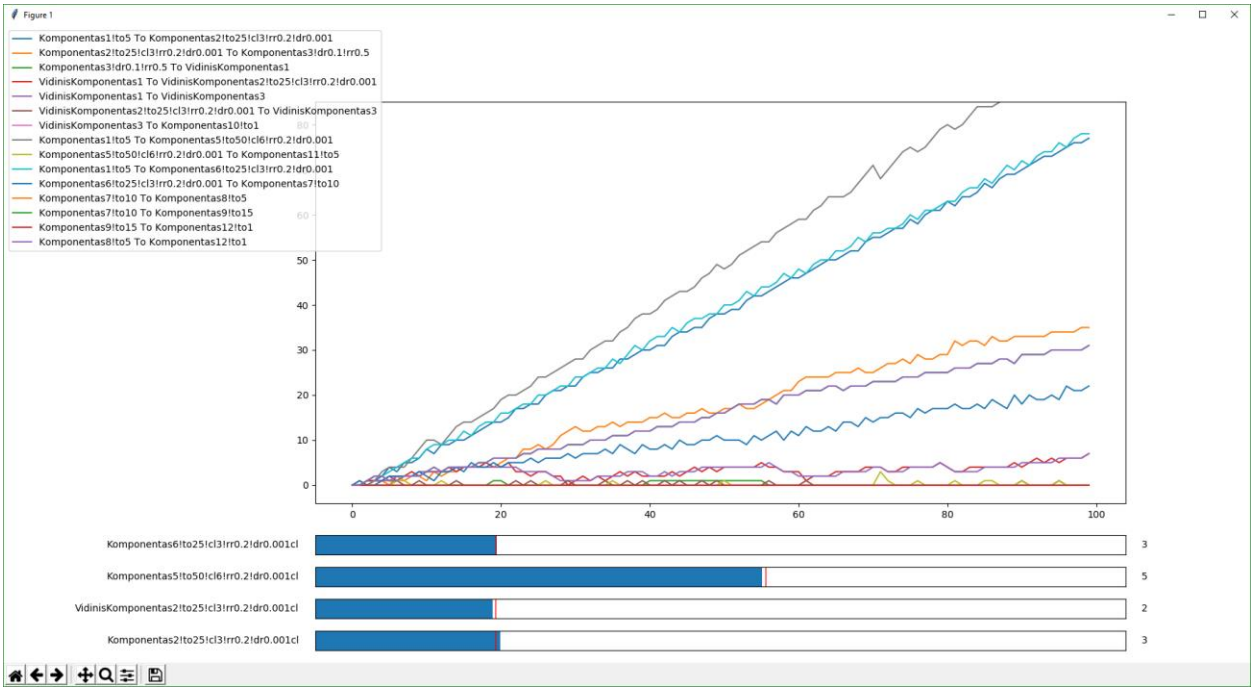
5.3.3. pav. matyti simuliacijos rezultatai, gauti atlikus vieną simuliaciją iš 5.3.2. pav. konceptualaus modelio gauto simuliacinio modelio. Apačioje yra keičiami dauginimosi parametrai, kuriuos keičiant simuliacija yra pergeneruojama, pagal nurodytus parametrus. Norint keisti apdorojimo ar kitas parametrus, reikia koreguoti konceptualų modelį.



5.3.1. pav. Plikas konceptualus modelis



5.3.2. pav. Pilnas konceptualus modelis



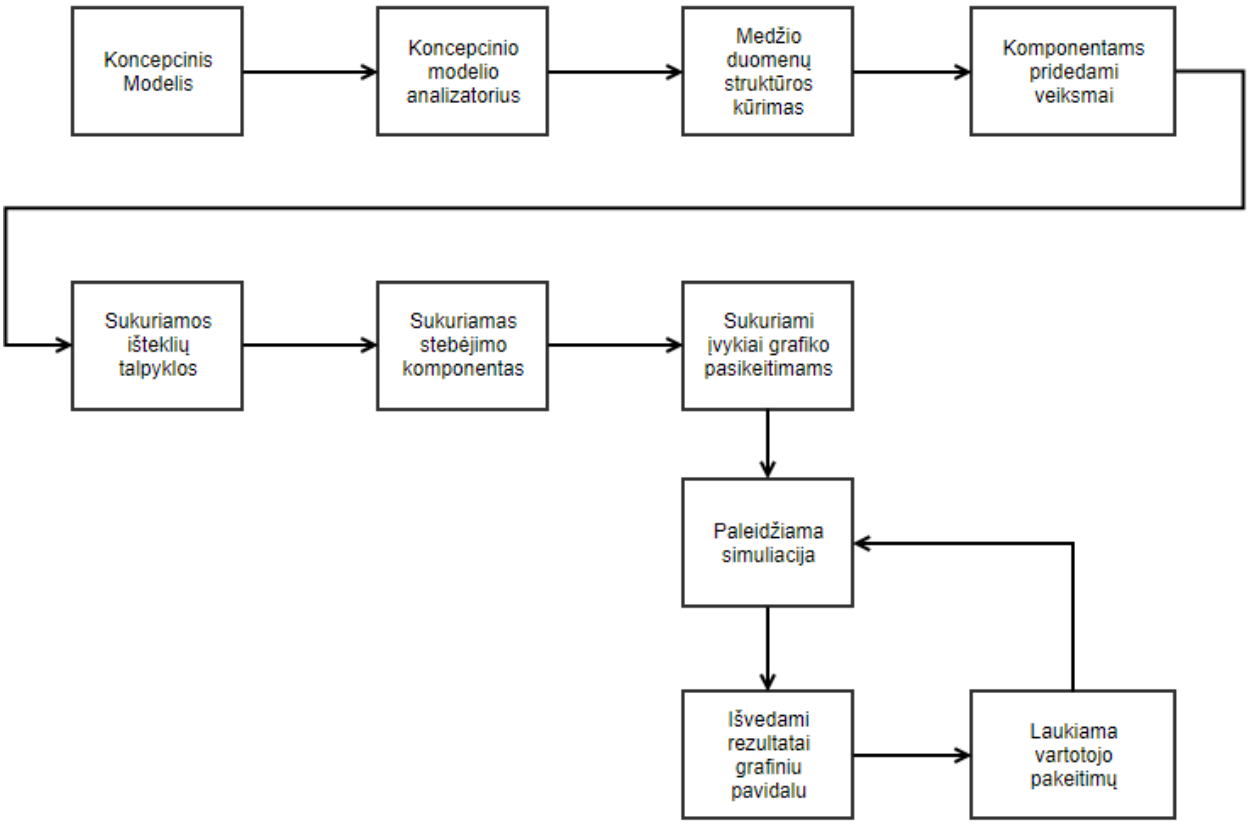
5.3.3. pav. Pavyzdinis simuliacijos rezultatas

Konceptualaus modelio konvertacija į simuliacinį modelį turi daug judančių dalių, kurias sunku vartotojui sekti vienu metu, tuo pasirūpina automatinis simuliacinio modelio generatorius ir sukuria tikslią konceptualaus modelio simuliacinę repliką.

5.4. Automatizuotas simuliacinio prototipo sukūrimo ir įvertinimo procesas

Simuliacinis modelis turi du skirtingus parametrų rinkinius, vienas yra įvesties, kuris yra gaunamas iš konceptualaus modelio, o kitas rinkinys yra išeities, tai tie parametrai kurie yra gaunami kaip simuliacijos rezultatai.

Šiuo atveju prototipinis simuliacinio modelio generatorius palaiko keturis įvesties ir vieną išvesties parametrus. Sukurtas prototipas pilnai atlieka savo funkciją, t.y. palaiko komunikaciją tarp komponenčių, leidžia manipuliuoti komponentėmis tiek per grafinę sąsają, tiek per patį konceptualų modelį, vertą paminėti, jog daugiau kontrolės, kaip simuliacija veiks yra gaunama kuriant konceptualų modelį.



5.4.1. pav. Generavimo ir veikimo diagrama.

Simuliacinio modelio kūrimo diagramoje (5.4.1. pav.) yra pavaizduotas kūrimo ir veikimo procesai. Taigi, norint sukurti simuliacinį modelį yra privaloma turėti konceptualų modelį, pagal kurį ir vyksta simuliacinio modelio generavimas. Toliau yra vykdoma konceptualaus modelio analizė, jos metu visa reikalinga informacija apie konceptualų modelį yra surenkama ir pagal tai yra lipdomas simuliacinis modelis. Sukūrus simuliacinį modelis jis iškart yra validuojamas ir gauti rezultatai yra atvaizduojami vartotojui.

Automatinė simuliacinio modelio generacija iš konceptualaus modelio buvo suplanuota tolimesniems tyrimo etapams, tačiau ne šio darbo apimtyje. Taigi pavyko įgyvendinti daugiau, nei buvo tikėtasi.

5.5. Pasiektų rezultatų suvestinė

Tyrimų metu buvo atlikti tokie darbai:

1) Atlikta išsami prieinamų šaltinių analizė;
2) Sukurtas minimalus konceptualaus modelio konvertavimas į simuliacinį modelį;
3) Išvesties ir įvesties parametrų (kontraktų) išgryninimas;
4) Grafinės sąsajos integracija;
5) Konceptualaus modelio analizę atliekančios programos sukūrimas;
6) Įgyvendintas prototipinis automatinis simuliacinio modelio generavimas ir duomenų struktūros kūrimas iš konceptualaus modelio.

Rezultatai 2), 3) ir 4) leido įgyvendinti pirmąjį suformuluotą darbo uždavinį „*sukurti konceptualaus modelio šabloną paskirstytoms sistemoms*“. Rezultatai 5) ir 6) leido pasiekti antrąjį darbo uždavinį „*paversti šį modelį į simuliacinį sistemos modelį (prototipą), remiantis nustatytomis tokio vertimo gairėmis (žodynu)“*. Kartu paėmus, pasiekti rezultatai 1) – 6), įgyvendinantys automatizuotą simuliacinio modelio kūrimą ir skaitmeninį įvertinimą remiantis konceptualiu paskirstytos sistemos modeliu, realizavo ir trečiąjį darbo uždavinį „*įvertinti sistemos skaitmenines charakteristikas ir įvairias sistemos konfigūracijas simuliacijos metu stebimų metrikų pagrindu“*.

6. Išvados

Paskirstytos sistemos tapo neatsiejama dalimi „enterprise“ programinės įrangos kūrime. Didelė dalis naujai projektuojamų sistemų būtent pasirenka paskirstytos sistemos modelį, vietoj monolitinio modelio. Šis pasirinkimas yra įtakotas šių dienų aparatinės įrangos galimybių ir vertinamo lankstumo. Tačiau paskirstytas sistemas yra sudėtingiau projektuoti nei monolitines sistemas, dėl papildomų judančių dalių, o taip pat reikalingos komunikacijos tarp komponenčių ar duomenų manipuliavimo skirtinguose duomenų saugyklose. Šią problemą sprendžiame šiame darbe.

Gebėti simuliuoti dar nesukurtą sistemą yra didelis pranašumas, nei mėginti atspėti, kaip sistema elgsis vėlyvuose sistemos vystymo etapuose. Taigi ne tik simuliacijos dėka galima sumažinti vėliau patiriamus kaštus dėl iškilusiu klaidų, kurios nebuvo pastebėtos ankščiau, bet ir palengvinti darbą programinės įrangos architektui suteikiant jam galimybę naudotis automatizuotu simuliacijos kūrimo procesu, kurį pavyko prototipo lygmenyje įgyvendinti šiame darbe.

Projektuojant sistemą dažniausiai yra pradedama nuo paviršutiniško sistemos eskizo, kuris turi būti pakankamai konkretus, kad būtu galima komunikuoti su suinteresuotomis šalimis. Detaliai aprašius sistemos eskizą, jis yra nesunkiai konvertuojamas arba jau tinkamas būti konceptualiu modeliu, kuris savo ruoštu yra naudojamas simuliacinio modelio generavimui ir validavimui.

Šis darbas pademonstravo, kad norint įgyvendinti automatizuotą simuliacinio modelio generavimą yra reikalinga atlikti eilė konkrečių žingsnių. Pirmiausia, šiam darbui atlikti buvo reikalinga perprasti naudojamą terminologiją, esamus tyrimus atliktus šioje srityje. Taip pat turi būti papildomai atlikta technologijų analizė ir pasirinkti optimaliausi technologiniai sprendimai įgyvendinti automatizuotam simuliacinio modelio generavimui. Toliau reikia išskirti, kaip technologinės sąvokos yra susiejamos su simuliaciniu modeliu, kaip jas naudoti ir integruoti į simuliacinį modelį. Pagaliau, reikia nubrėžti aiškią ribą tarp sistemos įvesties ir išvesties parametrų. Visi šie žingsniai buvo realizuoti magistro darbo metu, kurio pasiekti rezultatai parodo, kad toks automatizuotas sistemos validavimo procesas yra įgyvendinamas ir gali būti labai naudingas sistemos architektui.

Išvadų suvestinė:

- Rinkoje neegzistuoja viešai prieinamų automatiškai generuojančių simuliacinius modelius sistemų;
- Paskirstytu įvykių simuliavimo būdas efektyviai išnaudoja skaičiavimo išteklius;
- Automatizuotam simuliaciniam modeliui įgyvendinti yra reikalingos komponenčių ir sistemos parametrų vertimo gairės ir kiti papildomi žingsniai (įvardinti ankščiau);
- Darbo metu sukurtas automatizuotas procesas yra įrodymas, kad tokia automatizuota simuliacinius modelius generuojanti sistema (besiremianti konceptualiu sistemos modeliu) yra įgyvendinama.

7. Literatūros šaltiniai

[G14] Sukumar Ghosh. „Distributed systems: an algorithmic approach.” išleista “CRC press”, 2014.

[LBPTEP14] Linas Laibinis, Benjamin Byholm, Inna Pereverzeva, Elena Troubitsyna, Kuan Eeik Tan, and Ivan Porres. "Integrating Event-B modelling and discrete-event simulation to analyse resilience of data stores in the cloud." Publikuota “International Conference on Integrated Formal Methods”, Lapai 103-119. Išleista “Springer International Publishing”, 2014.

[WI15] K. Preston White, Jr., Ricki G. Ingalls. "Introduction to simulation." Publikuota “Winter simulation conference”, *2015 žiema*. Lapai 1741- 1755 ir 7-13. Išleista „IEEE press”, 2015.

[S13] Robert G. Sargent. "An introduction to verification and validation of simulation models." Publikuota “Simulation Conference (WSC)”, 2013 žiema. Lapai 321-327. Išleista „IEEE press“, 2013.

[YX13] Yang Gang, and Xingshe Zhou. "Cyber-physical systems." Išleista „World Scientific Publishing Company” 2013 birželio 19 d.

[TL14] Tanga, and Lixuan Lu. „A quantitative software testing method for hardware and software integrated systems in safety critical applications." University of Ontario Institute of Technology, Oshawa, ON, Kanada (2014).

[JKT13] Alglave, Jade, Daniel Kroening, and Michael Tautschnig. "Partial orders for efficient bounded model checking of concurrent software." Publikuota „International Conference on Computer Aided Verification”. Lapai 796-812. Išleista „Springer Berlin Heidelberg”, 2013.

[SB11] Sokolowski, and Catherine M. Banks, eds. „Principles of modeling and simulation: a multidisciplinary approach”. Išleista „John Wiley & Sons”, 2011.

[S17] Atviro kodo bendruomene. Simpy documentation simpy.readthedocs.io/en/latest/index.html ir simpy.readthedocs.io/en/latest/topical_guides/simpy_basics.html.

[R15] Robert G. Sargent. „An introductory tutorial on verification and validation of simulation models.” Publikuota “Proceedings of the 2015 Winter Simulation Conference”. Išleista „IEEE Press”, 2015.

[A15] Mohammed Yahya Alzahrani, „Model Checking Web Applications”. Heriot-Watt Universitetas. Išleistas 2015 žiema.

[M17] Mermaid flowcharts – basic syntax. Online: https://mermaidjs.github.io/flowchart.html

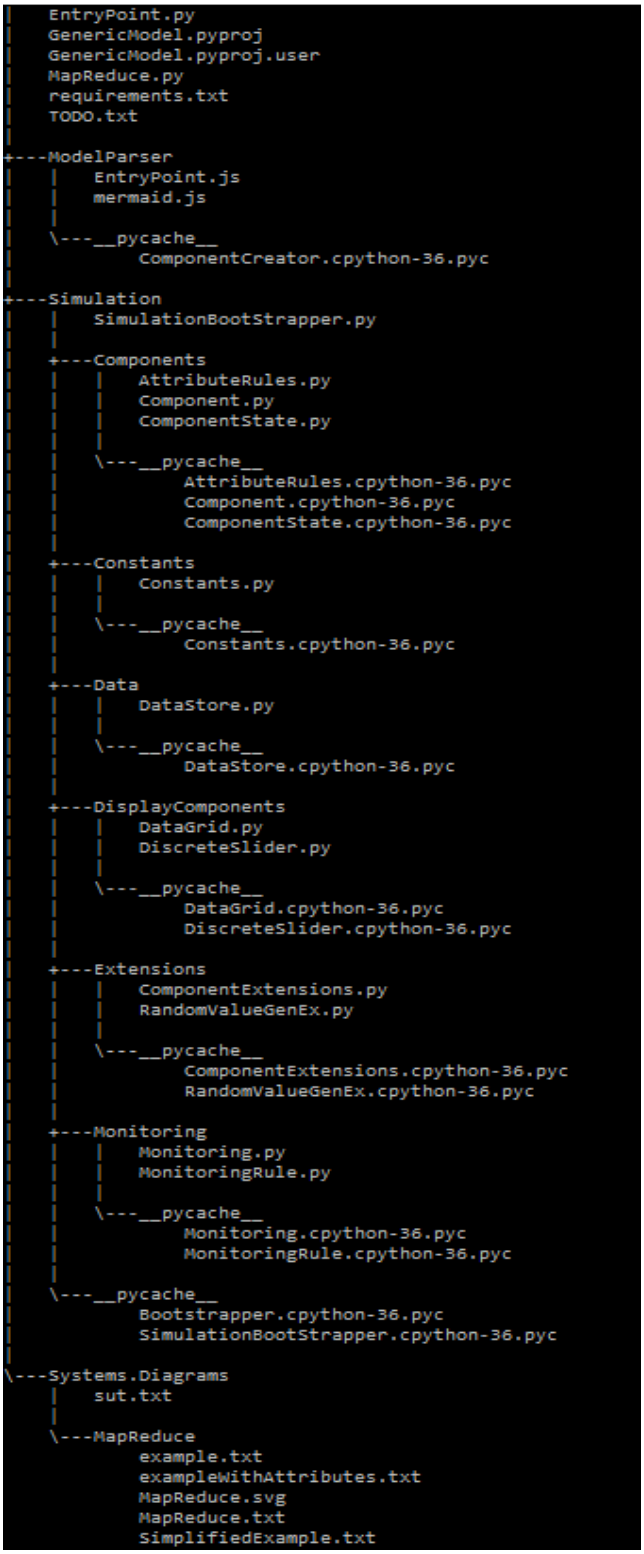
[S17] Simpy process interaction. Online: http://simpy.readthedocs.io/en/latest/simpy_intro/process_interaction.html

[P17] Python standard libraries: random variable generator. Online: https://svn.python.org/projects/python/tags/r32/Lib/random.py

[G18] Programinis išeities kodas. https://github.com/KarolisKaj/UniversityShared/tree/master/Master's/Code

8. Priedas

Programinio kodo failinė struktūra:



Programinis kodas:
ModelParser:
EntryPoint.js

```
function runParsing() {
    console.log('[JS] Creating Mermaid Instance...');
    var mermaid = require('./mermaid');
    mermaid.initialize({ theme: 'forest' });
    console.log('[JS] Mermaid Instance Created!');
    console.log("");

    console.log('[JS] Reading Model From File...');
    const fs = require('fs');
    var contents = fs.readFileSync('../Systems.Diagrams/sut.txt', 'utf8');
    console.log('[JS] Model read!');
    console.log("");

    console.log("-----------------MODEL--------------------");
    console.log(contents);
    console.log("-----------------MODEL--------------------");
    console.log("");

    console.log('[JS] Parsing Model...');
    var parsedGraph = mermaid.parse(contents);
    var vertices = JSON.stringify(parsedGraph.parser.yy.getVertices());
    var edges = JSON.stringify(parsedGraph.parser.yy.getEdges());
    var subGraphs = JSON.stringify(parsedGraph.parser.yy.getSubGraphs());
    console.log('[JS] Model Parsed!');

    var path = require("path");
    pythonEntryPoint = path.join(__dirname, '..', 'EntryPoint.py');

    console.log('[JS] Running Python Generic Model...');
    const { execFile } = require('child_process');
    execFile('C:/Program Files (x86)/Microsoft Visual
Studio/Shared/Python36_64/python.exe', [pythonEntryPoint, vertices, edges, subGraphs], {
maxBuffer: 1024 * 500 }, (error, stdout, stderr) => {
        if (error) {
            console.log('Failed to run python file.');
            throw error;
        }
        console.log(stdout);
        console.log("[JS] Simulation ended successfully!");
    });
}

runParsing();
```

mermaid.js:
Importuota biblioteka.

Simulation:
Components:
AttributeRules.py

```
import random
```

```
from Simulation.Constants.Constants import *

attribute_actions = dict([
    (death_rate, lambda rate: random.random() <= rate),
    (recovery_rate, lambda rate: random.random() <= rate)
    ])
```

Component.py

```
from Simulation.Components.AttributeRules import attribute_actions
from Simulation.Components.ComponentState import *
from Simulation.Constants.Constants import *

class Component(object):
    def __init__(self, name, attributes):
        self.name = name
        print("Initializing component named - " + self.name)
        self.actions = []
        self.attributes = attributes
        self.timeout_action = None

        self.component_state = ComponentState.Undefined
        self.recovery_rate_handle = None
        self.death_rate_handle = None

        self.model_component(self.attributes)

    def run(self):
        while True:
            self.component_state = self.new_state()
            if(self.component_state != ComponentState.Dead):
                for action in self.actions:
                    yield action()
            yield self.timeout_action()

    def add_action(self, action):
        self.actions.append(action)

    def set_timeout_action(self, action):
        self.timeout_action = action

    def get_name(self):
        return self.name

    def get_attrbutes(self):
        return self.attributes

    def model_component(self, attributes):
        if(recovery_rate in attributes):
            self.recovery_rate_handle = lambda :
attribute_actions[recovery_rate](float(attributes[recovery_rate]))

        if(death_rate in attributes):
            self.death_rate_handle = lambda :
attribute_actions[death_rate](float(attributes[death_rate]))

    def clone(self):
        component = Component(self.get_name(), self.attributes)
```

```
        component.set_timeout_action(self.timeout_action)
        for action in self.actions:
            component.add_action(action)

        return component

    def new_state(self):
        if(self.recovery_rate_handle is None and self.death_rate_handle is None): return
ComponentState.Alive
        if(self.component_state == ComponentState.Dead):
            if(self.recovery_rate_handle is None): return ComponentState.Dead
            return ComponentState.Alive if self.recovery_rate_handle() else
ComponentState.Dead;
        else:
            if(self.death_rate_handle is None): return ComponentState.Alive;
            return ComponentState.Dead if self.death_rate_handle() else
ComponentState.Alive;
```

ComponentState.py

```
from enum import Enum

class ComponentState(Enum):
    Undefined =1
    Dead = 2
    Alive = 3
```

Constants:

Constants.py

```
clone_attribute = 'cl'
death_rate = 'dr'
time_out = 'to'
recovery_rate = 'rr'

default_timeout = 10
default_monitoring_interval = 10
simulation_duration = 1000

startNode = "start"
endNode = "end"
linkText = "text"
```

Data:

DataStore.py

```
stored_components = None
stored_attributes = None

stored_vertices = None
stored_edges = None
stored_subgraphs = None

stored_stores = dict()
```

DisplayComponents:

DataGrid.py

```
import matplotlib

from matplotlib import pyplot as plt
from matplotlib.widgets import Slider, Button

from Simulation.DisplayComponents.DiscreteSlider import DiscreteSlider

class DataGrid(object):
    def __init__(self):
        print("Initializing DataGrid")
        self.parameters = []
        self.sliders = dict()
        self.next_slider_position = None
        self.fig = None
        self.main_plot_left = 0.1
        self.main_plot_bottom = 0.05
        self.main_plot_left_step = 0.1
        self.main_plot_bottom_step = 0.05

        self.slider_coordinates = [0.25, -0.03, 0.65, 0.03]
        self.slider_coordinates_step = [0, 0.05, 0, 0]


    def create_grid(self, data):
        matplotlib.rcParams['figure.figsize'] = (18, 9)

        fig, ax = plt.subplots()
        self.fig = fig
        plt.subplots_adjust(left=0.25, bottom=0.05)

        for metric in data:
            p1, = ax.plot(data[metric])
            self.parameters.append(p1)

        self.fig.legend(self.parameters, [metric for metric in data], loc='upper left')

    def update_data(self, data):
        for param, metric in zip(self.parameters, data):
            param.set_ydata(data[metric])

        self.fig.canvas.draw_idle()

    def add_changeable_slider(self, name, handle, default_value):
        components_axes = plt.axes(self.calculate_slider_coordinates())

        components_slider = DiscreteSlider(components_axes, name, 1, 10,
valinit=default_value)
        self.main_plot_bottom = self.main_plot_bottom + self.main_plot_bottom_step

        plt.subplots_adjust(left=0.25, bottom=self.main_plot_bottom)

        components_slider.on_changed(handle)
        self.sliders[name] = components_slider

    def calculate_slider_coordinates(self):
        slider_coordinates = []
```

```
        for current_coord, step in zip(self.slider_coordinates,
self.slider_coordinates_step):
            slider_coordinates.append(current_coord + step)

        self.slider_coordinates = slider_coordinates
        return slider_coordinates

    def show_grid(self):
        plt.show()
```

DiscreteSlider.py

```
import simpy
import matplotlib
from matplotlib import pyplot as plt
from matplotlib.widgets import Slider, Button

class DiscreteSlider(Slider):
    def __init__(self, * args, ** kwargs):
        self.inc = kwargs.pop('increment', 1)
        Slider.__init__(self, * args, ** kwargs)

    def set_val(self, val) :
        xy = self.poly.xy
        xy[2] = val, 1
        xy[3] = val, 0
        self.poly.xy = xy

        self.valtext.set_text('%u' % val)

        if self.drawon:
              self.ax.figure.canvas.draw()
        self.val = int(val)
        if not self.eventson:
              return
        for cid, func in self.observers.items():
            func(val)
```

Extensions:

ComponentExtensions.py

```
import simpy
import re

from Simulation.Constants.Constants import *
from Simulation.Components.Component import Component
from Simulation.Extensions.RandomValueGenEx import *

def create_components(edges, env, stores, attributes):
    components = dict()
    for edge in edges:
        add_new_key(edge[startNode], components, env, attributes)
        add_new_key(edge[endNode], components, env, attributes)
        add_actions(components, edge, stores, env)

    clone_components(components)
    return components

def add_new_key(name, components, env, attributes):
```

```
    if(not name in components):
        components[name] = Component(name, attributes[name])
        components[name].set_timeout_action(create_timeout_action(env, attributes[name]))


def clone_components(components):
    for component in components:
        components[component] = clone_component(components[component])

def clone_component(component):
    components = [component]
    attributes = component.get_attrbutes()
    for _ in range((int(attributes[clone_attribute]) - 1) if clone_attribute in
attributes else 0):
        components.append(component.clone())

    return components

def create_timeout_action(env, attributes):
    return lambda: generate_gauss_timeout(env, int(attributes[time_out]) if time_out in
attributes else default_timeout)

def add_actions(components, edge, stores, env):
    if (not edge[startNode] + ' To ' + edge[endNode] in stores):
        store = simpy.Store(env)
        components[edge[startNode]].add_action(lambda: store.put(edge[startNode] + "
produced message."))
        components[edge[endNode]].add_action(lambda: store.get())
        stores[edge[startNode] + ' To ' + edge[endNode]] = store

def get_attributes(edges):
    attributes = dict()
    for edge in edges:
         attributes[edge[startNode]] = extract_component_attributes(edge[startNode])
         attributes[edge[endNode]] = extract_component_attributes(edge[endNode])

    return attributes

def extract_component_attributes(name):
    matches = re.findall("(!([a-z]+)(\d+[\,\.]?\d*))", name)
    attributes = dict()
    for match in matches:
        attributes[match[1]] = match[2]

    return attributes
```

RandomValueGenEx.py

```
import random

def generate_gauss_timeout(env, mid_point):
    generated_timeout = random.gauss(mid_point, int((mid_point * 0.1)))
    return env.timeout(generated_timeout);
```

Monitoring:

Monitoring.py

```
class Monitoring(object):
```

```
    def __init__(self, monitoringRulesSet):
        self.monitoringRulesSet = monitoringRulesSet
        self.data = dict()

    def start_monitoring(self, interval_handle):
        while True:
            for rule in self.monitoringRulesSet:
                if(rule.get_name() in self.data):
                    self.data[rule.get_name()].append(rule.evaluate())
                else:
                    self.data[rule.get_name()] = []
                    self.data[rule.get_name()].append(rule.evaluate())

            yield interval_handle()

    def get_results(self):
        return self.data
```

MonitoringRule.py

```
class MonitoringRule(object):
    def __init__(self, name, rule):
        self.name = name
        self.rule = rule

    def evaluate(self):
        return self.rule();

    def get_name(self):
        return self.name
```

SimulationBootStrapper.py

```
import simpy

from Simulation.DisplayComponents.DataGrid import DataGrid
from Simulation.Monitoring.Monitoring import Monitoring
from Simulation.Monitoring.MonitoringRule import MonitoringRule
from Simulation.Extensions.ComponentExtensions import *
from Simulation.Constants.Constants import default_monitoring_interval,
simulation_duration
from Simulation.Data.DataStore import *

class SimulationBootStrapper(object):
    def __init__(self, vertices, edges, subgraphs):
        print("-----Initiliazing Bootstrapper-----")

        global stored_vertices
        stored_vertices = vertices
        global stored_edges
        stored_edges = edges
        global stored_subgraphs
        stored_subgraphs = subgraphs

        global stored_attributes
        stored_attributes = get_attributes(stored_edges)
```

```
        self.dataGrid = None

    def run_sim(self):
        global stored_stores
        stored_stores = dict()
        self.env = simpy.Environment()

        global stored_components
        stored_components = create_components(stored_edges, self.env, stored_stores,
stored_attributes)
        for index in stored_components:
            [self.env.process(component.run()) for component in stored_components[index]]

        monitor = Monitoring([self.create_monitor_rule(store) for store in
stored_stores])
        self.env.process(monitor.start_monitoring(lambda:
self.env.timeout(default_monitoring_interval)))

        self.env.run(until=simulation_duration)
        print("Simulation Finished...")
        self.create_dataGrid(monitor.get_results())

    def create_dataGrid(self, data):
        if(self.dataGrid is None):
            self.dataGrid = DataGrid()
            self.dataGrid.create_grid(data)
            self.create_dataGrid_dynamic_sliders()
            self.dataGrid.show_grid()
        else:
            self.dataGrid.update_data(data)

    def create_dataGrid_dynamic_sliders(self):
        for component in stored_attributes:
            for attribute in stored_attributes[component]:
                self.create_slider(component, attribute)

    def create_slider(self, component, attribute):
        if(attribute == clone_attribute):
            self.dataGrid.add_changeable_slider(component + clone_attribute, lambda x:
self.update_handle(component, clone_attribute, int(x)),
int(stored_attributes[component][clone_attribute]))

    def update_handle(self, component, name, value):
        global stored_attributes
        stored_attributes[component][name] = value
        self.run_sim()

    def create_monitor_rule(self, name):
        return MonitoringRule(name, lambda: len(stored_stores[name].items))
```

Root

EntryPoint.py

```
import argparse
import json

def addArgs():
```

```
    parser = argparse.ArgumentParser(description='Process some integers.')
    parser.add_argument('vertices')
    parser.add_argument('edges')
    parser.add_argument('subGraphs')
    return parser;

def getModelGraph(parser):
    args = parser.parse_args()
    vertices = json.loads(args.vertices)
    edges = json.loads(args.edges)
    subGraphs = json.loads(args.subGraphs)
    return vertices, edges, subGraphs

from Simulation.SimulationBootStrapper import SimulationBootStrapper
argsParser = addArgs()
bootstrapper = SimulationBootStrapper(*getModelGraph(argsParser))
bootstrapper.run_sim()
```

MapReduce.py

```
import simpy
import random

import matplotlib
from matplotlib import pyplot as plt
from matplotlib.widgets import Slider, Button
from Simulation.DisplayComponents.DiscreteSlider import DiscreteSlider

def latency(env, latency_duration):
    return env.timeout(latency_duration)

def rand_latency(env, min, max):
    latency_duration = random.randint(min, max)
    return latency(env,latency_duration)

class Monitor(object):
    def __init__(self, env, store, publisher):
        self.env = env
        self.store = store
        self.unprocessed_message_count = []
        self.message_count = []
        self.publisher = publisher

    def start_monitoring(self):
        while True:
            self.unprocessed_message_count.append(len(self.store.items))
            self.message_count.append(len(self.publisher.produced_messages))
            yield self.env.timeout(10)

class Component(object):
    def __init__(self, env, store):
        self.env = env
        self.store = store

    def start_processing_messages(self):
        while True:
            process_duration = 40
            message = self.store.get()
```

```
            print('Proccesed %s @ %d ' % (message, self.env.now))
            yield self.env.timeout(process_duration)
            yield rand_latency(self.env, 20,40)

class Publication(object):
    def __init__(self, env, store):
        self.env = env
        self.store = store
        self.produced_messages = []

    def start_publishing(self):
        while True:
            print('Message sent to components %d' % self.env.now)
            self.store.put('msg')
            self.produced_messages.append(len(self.produced_messages) + 1)
            yield rand_latency(self.env, 20,40)

class Receiver(object):
    def __init__(self, env):
        self.env = env

    def start_receiving(self):
        while True:
            print('Message received %d' % self.env.now)
            yield self.env.timeout(30)

matplotlib.rcParams['figure.figsize'] = (13, 8)

fig, ax = plt.subplots()
plt.subplots_adjust(left=0.10, bottom=0.20)

components_axes = plt.axes([0.25, 0.05, 0.65, 0.03])
components_slider = DiscreteSlider(components_axes, 'Components count', 1, 10, valinit=4)

p1, = ax.plot(range(1000))
p2, = ax.plot(range(1000))

def run_sim(val):
    env = simpy.Environment()
    store = simpy.Store(env)

    publication = Publication(env, store)
    receiver = Receiver(env)

    monitor = Monitor(env, store, publication)

    env.process(monitor.start_monitoring())

    env.process(publication.start_publishing())

    for i in range(int(components_slider.val)):
        env.process(Component(env, store).start_processing_messages())
    env.process(receiver.start_receiving())
    env.run(until=10000)

    p1.set_ydata(monitor.unprocessed_message_count)
    p2.set_ydata(monitor.message_count)
    fig.canvas.draw_idle()
```

```
components_slider.on_changed(run_sim)

fig.legend([p1, p2], ['Unprocessed messages', 'Total messages'], loc='upper left')
plt.show()
```