VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS INSTITUTAS

Elixir proceso priežiūros įtaka išskirstytų algoritmų teisingumui

Impact of Elixir process Supervision on the correctness of distributed algorithms

Magistro darbo planas

Atliko: Matas Savickis (parašas)

Darbo vadovas: Karolis Petrauskas, Doc., Dr. (parašas)

Recenzentas: (parašas)

Vilnius – 2023

TURINYS

Įvadas

Šiais laikais kai kurios programų sistemos yra išskirstytos [**mcr**]. Tokios sistemos, kaip sufleruoja pavadinimas, yra kuriamos išskirstant skaičiavimo mazgus į atskiras, savarankiškas dalis [**coulouris2005distributed**]. Ne kaip monolitinės sistemos, išskirstytos sistemos gali veikti skirtingose serverinėse, kurios gali būti įvairiose geografinėse vietose [**shirriff2006method**]. Toks sistemos išskirstymas pasižymi šiomis savybėmis:

1. Pasiekiamumas (angl. *Availability*) – vartotojas gali pasiekti sistemą bet kuriuo metu [**180327**].

2. Patvarumas (angl. *Durability*) – išskirstytos sistemos užtikrina duomenų išlaikymą ir pastovų veikimą net jeigu ir vienas iš paskirstytos sistemos mazgų nustotų veikti dėl sistemos sutrikimų, sukeltų programos klaidų arba gamtos katastrofų, tokių kaip: gaisrai, potvyniai ir panašios nelaimės. Ši savybė taip pat užtikrina sistemos gebėjimą atsistatyti ir neprarasti duomenų po minėtų įvykių [**5470366**].

3. Plečiamumas (angl. *Scalability*) – didėjant vartotojų skaičiui bei programų sistemos kompleksiškumui išskirstytos sistemos užtikriną vertikalų (padidinti mazgo techninės įrangos galingumą) bei horizontalų (padidinti mazgų skaičių sistemoje) plečiamumą [**862209**].

4. Našumas (angl. *Efficiency*) – sistemos vartotojų skaičius dažniausiai būna nepastovus, jis kinta dienos metu arba atitinkamais metų periodais. Išskirstytos sistemos padeda užtikrinti našų įrangos resursų naudojimą sumažinant įrangos galingumą (kai vartotojų skaičius yra nedidelis) bei padidinant galingumą (kai sistemos apkrova padidėja).

Išskirstytas sistemas galima kurti naudojant bet kokią programavimo kalbą(ActiveMQ [**snyder2011activemq**] naudoja Java programavimo kalbą, Apache Spark [**spark2018apache**] naudoja Scala programavimo kalbą). Viena populiaresnių programavimo kalbų naudojamų išskirstytų sistemų kūrime yra Elixir. Elixir yra dinaminė, funkcinė programavimo kalba skirta kurti išskirstytoms sistemoms. Elixir kalba veikia naudodama Erlang BEAM VM virtualią mašiną kurios pagalba galima kurti mažo delsimo, išskirstytas, trigdžiams atsparias sistemas.

Norint užtikrinti kūriamos sistemos kokybę dažnai yra rašomi testai. Testai padeda atskleisti programos klaidas arba pasakyti ar naujas kodas nepaveikė seniau parašyto funkcionalumo [**819971**]. Tačiau net ir laikantis geryjų testavimo praktikų nepavyksta išvengti programos klaidų. Net ir paskyrus daugiau resursų testavimui sudėtinguose sistemose, pilnas sistemos testavimas yra neįmanomas [**sullivan2004software**]. Norint rasti subtilesnius sutrikimus pačioje sistemos architektūroje, tokius kaip dalinis mazgų neveikimas, lygiagrečių algoritmų klaidos naudojant keletą procesų, gedimų atsparumo ir atsistatymo po gedimo algoritmų klaidos bei kitiems kraštutiniams veikimo scenarijams [**newcombe2014use**] rasti ir išspręsti turime ieškoti kitų būdų. Vienas iš jų formalios specifikacijos.

Formalios specifikacijos yra matematinės technikos skirtos apibūdinti sistemų elgseną ir padėti kuriant jas naudojant griežtas ir veiksmingas priemones [**holzmann1995improvement**]. Turint sistemos formalią specifikaciją galima ja pasinaudoti vykdant formalų verifikavimą ir parodant,

kad algoritmas yra adekvatus pagal sukurtą specifikaciją. Sudarinėti formalią sistemos specifikaciją galima ir nepradėjus įgyvendinti sistemos, turint tik jos architektūrą. Formaliai verifikuota specifikacija suteikia informacijos apie architektūros korektiškumą ir įgalina objektyviai koreguoti sistemos architektūrą dar prieš pradedant ją įgyvendinti. Formalios specifikacijos sudaromos pasinaudojant tam tikromis kalbomis arba įrankiais. Viena iš tokių formalaus specifikavimo kalbų yra TLA$^+$ [**lamport2002specifying**].

TLA$^+$ yra formalios specifikacijos kalba, kurią sukūrė Leslie Lamport [**lamport2002specifying**]. Leslie Lamport 1980 metais sukūrė laiko veiksmų logiką (angl. *Temporal Logic of Actions*) [**10.1145/177492.177726**] pasinaudodamas Amir Pnueli 1977 metais sukurta laiko logika (angl. *Temporal Logic*) [**4567924**]. 1999 metais Leslie Lamport naudodamasis laiko veiksmų logika sukūrė formalaus specifikavimo kalbą TLA$^+$ [**lamport2002specifying**].

TLA$^+$ kalba yra skirta kurti konkurencinių ir išskirstytų sistemų formalias specifikacijas ir jas verifikuoti. Naudojant TLA$^+$ galima specifikuoti šias išskirstytų sistemų savybes [**lamport2019safety**]:

1. Gyvumas – geri dalykai įvyksta programos vykdymo metu. Sistema galiausiai atliks jai paskirtą užduotį arba pateks į norimą būseną.

2. Saugumas – blogi dalykai neatsitiks programos vykdymo metu. Sistema nesustos veikti dėl netikėtai iškilusios klaidos.

Kadangi TLA$^+$ specifikacijos yra rašomos formalia kalba, tai leidžia patikrinti sukurtos specifikacijos saugumo ir gyvumo savybes.

Šias savybes mes galime patikrinti naudodamiesi TLC Model Checker (modelio tikrintoju). TLC yra išreikštinės būsenos (explicit-state) modelio tikrintojas, kurio paskirtis yra palaipsniui pasiekti visas galimas sistemos būsenas pagal nurodytą formalią specifikaciją [**yu1999model**]. Tačiau kartais pagal sukurtą formalią specifikaciją susidaro labai daug būsenų, kurias sistema gali pasiekti, todėl tampa nepraktiška naudoti TLC. Tokiu atveju galime naudotis TLA$^+$ specifikacijos įrodymo sistema TLAPS [**cousineau2012tla+**]. TLAPS yra įrodymų sistema skirta patikrinti TLA$^+$ įrodymus. Šios sistemos paskirtis yra patikrinti pateiktus teoremų įrodymus. Įrodžius teoremą laikoma, kad TLA$^+$ specifikacija yra korektiška.

Yra ir kitų formalaus specifikavimo kalbų kurias galėtume naudoti šiame darbe. Viena iš jų Z formalaus specifikavimo kalba [**O'Regan2017**], kuri sėkmingai buvo naudota specifikuoti UNIX failų sistemai [**bowen1996formal**], bei Oxfordo universiteto paskirstytų skaičiavimų projekte [**bowen1996formal**]. Dar viena formalaus specifikavimo kalba yra VDA [**bjorner1978vienna**], kuria buvo specifikuoti bendros atminties sinchronizavimo algoritmai [**Slaats1998**]. Tačiau šiam darbui buvo pasirinkta TLA$^+$ kalba dėl jos pritaikymo išskirstytoms sistemoms naudojant būsenų mašiną [**lamport2002specifying**], esamų sėkmingo pritaikymo pavyzdžių specifikuojant išskirstytas sistemas [**newcombe2014use**; **kfkTla**; **rafttla**; **jiryu2020extreme**] bei gausaus TLA$^+$ įrankių pasirinkimo.

Temos aktualumas bei naujumas

Darbo tikslas

Sukurti metodą kuris išskiria TLA$^+$ specifikaciją iš Elixir supervizoriaus programinio kodo.

Uždaviniai

1. Sukurti taisyklių rinkinį kuris išskirtų TLA+ specifikaciją iš Elixir supervizijos medžio kodo.
2. Pagal sudarytas taisykles įgyvendinti įrankį, kuris Elixir kodą verčia į TLA+ specifikaciją.
3. Įvertinti sugeneruotos specifikacijos teisingumą.
4. Surasti atviro kodo sistemų naudojančių Elixir supervizijos medį ir patikrinti to kodo teisingumą sugeneruojant TLA+ speicifikaciją.

Laukiami rezultatai

1. Elixir supervizoriaus kodo pavertimas į TLA$^+$ specifikaciją.
2. Įrodymas apie Elixir kodo pavertimo į TLA$^+$ specifikaciją adekvatumas.
3. Elixir kodo ir TLA$^+$ specifikacijos sutapimo įvertinimas.
4. Papildytas TLA$^+$ specifikacijos iš Elixir kodo generatorius.

Hipotezė

- H0 (Nulinė hipotezė) – mappinimas tarp Elixir supervizijos medžio kodo ir TLA+ specifikacijos nėra įmanomas.
- H1 (Alternatyvi hipotezė) – egzistuoja metodas, kaip mappinti Elixir supervizijos medžio kodą į TLA+ specifikaciją.