VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

Funkcinio-reaktyvaus programavimo taikymas įvykių kaupimo sistemose

Functional Reactive Programming in Event Sourcing Systems

Mokslo tiriamasis darbas

Atliko: 1 kurso 12 grupės studentas
Žilvinas Kučinskas ..... (parašas)

Darbo vadovas:
Viačeslav Pozdniakov ..... (parašas)

Recenzentas:
prof. Rimantas Vaicekauskas ..... (parašas)

Vilnius, 2014 m.

Turinys

1 Magistro darbo objekto apžvalga bei tyrimo problemos aprašymas

1.1 Tyrimo objektas

Tyrimo objektas yra funkcinio-reaktyvaus programavimo bei įvykių kaupimo principai.

1.2 Darbo tikslai ir uždaviniai

Darbo tikslas yra pritaikyti funkcinio-reaktyvaus programavimo principus įvykių kaupimo sistemose taip, jog būtų išpildyti šie reikalavimai:

- įvykių kaupimo sistemos skaitymo modelis būtų be būsenos;
- įvykių kaupimo sistemos skaitymo modelis būtų kuriamas tik per įvykių kompoziciją;
- įvykių kaupimo sistemos skaitymo modelio programinis kodas būtų griežtai tipizuotas.

Siekiant šio tikslo, turi būti išspręsti šie uždaviniai:

- įrodyti, kad funkcinį-reaktyvų programavimą įmanoma taikyti įvykių kaupimo sistemose;
- sukurti konkretizuotą kalbą (angl. domain specific language), apjungiančią funkcinio-reaktyvaus programavimo bei įvykių kaupimo principus;
- aprašyti konkretizuotuos kalbos kūrimo metodiką, apibrėžti gautų rezultatų apribojimus, suformuluoti iškilusias problemas bei paaiškinti jų priežastis.

1.3 Tyrimo aktualumas

Funkcinis reaktyvus programavimas integruoja laiko tėkmę bei sudėtinius įvykius į funkcinį programavimą. Šis principas suteikia elegantišką būdą išreikšti skaičiavimus interaktyvių animacijų, robotikos, kompiuterinio vaizdavimo, vartotojo sąsajos ir modeliavimo srityse [Cza12, p. 4]. Pagrindinės funkcinio reaktyvaus programavimo sąvokos:

- signalai arba elgsena - reikšmės, besikeičiančios bėgant laikui;
- įvykiai - momentinių reikšmių kolekcijos arba laiko-reikšmės poros.

Funkcinis-reaktyvus programavimas įgalina apsirašyti elgseną deklaratyviai [EH97, p.1]. Elgsena ir įvykiai gali būti komponuojami kartu, išreikšti vienas per kitą. Funkcinis reaktyvus programavimas apibrėžia kaip signalai arba elgsena reaguoja į įvykius. [Ams, p. 1] Šį principą galima iliustruoti pavyzdžiu. Tarkime turime Excel lapą, kuriame yra trys laukai: darbuotojo pradirbtos valandos, valandinis užmokestis bei formulė, kuri paskaičiuoja konkretų darbuotojo atlyginimą. Darbuotojui pradirbus daugiau valandų, atnaujinamas pradirbtų valandų skaičius. Kartu atsinaujina ir pačios formulės reikšmė, tai yra konkretus užmokestis pakinta. Šiuo atveju įvykus reikšmės atnaujinimo įvykiui, nuo jos priklausomos formulės taipogi atsinaujina arba tam tikras įvykis iššaukia elgseną sistemoje.

Įvykių kaupimo principo esmė – objektas yra atvaizduojamas kaip įvykių seka. Kaip pavyzdį tai galima parodyti remiantis banko sąskaita. Tarkime vartotojas, banko klientas, turi 100 litų sąskaitos balansą. Tarkime vartotojas nusipirko prekę už 20 litų, tada įnešė į savo sąskaitą 15 litų ir galiausiai nusipirko tam tikrą paslaugą už 30 litų. Akivaizdu, jog turint šią įvykių seką, galima atvaizduoti dabartinę objekto būseną - tai yra 65 litai vartotojo sąskaitoje. Įvykių kaupimo principas užtikrina, jog visi būsenos pasikeitimai yra saugomi įvykių žurnale kaip įvykių seka [Ver13]. Įvykių kaupimo principui yra būdinga, jog įvykių negalima ištrinti bei atnaujinti, duomenys yra nekeičiami, dėl to įvykių žurnalas yra sistemos gyvavimo istorija (tiesos šaltinis). Tačiau toks modelis turi ir trūkumų. Jis nėra pritaikytas patogiam užklausų rašymui. Iš įvykių srautų yra kuriamos projekcijos, skirtos konkretiems sistemoms poreikiams, pavyzdžiui: paieškai, klasifikacijai ar ataskaitų ruošimui.

Pritaikius funkcinį-reaktyvų programavimą įvykių kaupimo principu paremtose sistemose būtų galima modeliuoti ne tik momentinius įvykius, tačiau turėti ir jų istoriją. Yra poreikis sukurti konkretizuotą kalbą (angl. domain specific language), kuri įgalintų paslėpti įvykių žurnalą (arba duomenų saugyklą). Pastarosios naudotojas galėtų orientuotis į pačią sprendžiamos srities problemą, nekreipdamas dėmesio į žemesnio lygio realizacijos detales. Šiuo atveju būtų galima deklaratyviai (ką kažkuri programos dalis turi daryti) apsirašyti elgseną, nutikus įvykiui, kartu

su imperatyviomis (instrukcijos, kurios aprašo, kaip programos dalys atlieka savo užduotis) struktūromis.

1.4 Pritaikymo pavyzdys

Tarkime turime domeno sritį - bankininkystė. Turime įvykių srautą - vartotojų sukūrimas. Naudojant įsivaizduojamą Scala API galima sukurti vartotojų paieškos puslapį pagal vardą ir pavardę. Demonstacija pateikta 1 kodo pavyzdyje.

```
// stream model
case class CustomerCreate (val name: String, val surname: String, val personalNum:
    String)

val es = EventSourceConnection(url)
val createStream = Stream(es, "customerCreate")

class case CustomerModel(val name: String, val surname: String, val personalNum:
    String) extends ViewableModel

trait CustomerArgModel extends Arg2Model[String,String]{
  val name: Option[String]
  val surname: Option[String]
}

//args are passed on form view/post
val customerView = View(args: CustomerArgModel).foldLeft(
  (acc,event) => event match {
    case CustomerCreate(name, surname, personalNum) =>
      for {
        newName <- name
        newSurname <- surname
        newPersNum <- personalNum
        if (args.name == name && args.surname == surname)
      } yield CustomerModel(newName, newSur, newPersNum)
  }
)
```

```
// getting data for all Kucinskai
val specificData = customerView(None, Some("Kucinskas")):
    Option[List[CustomerModel]]
```

Kodo pavyzdys 1 - vartotojų paieškos puslapis naudojant įsivaizduojamą Scala API

Šiuo atveju veiksmai peržiūrint duomenis yra sumaišomi kartu su veiksmais gaunant įvykius. Verta pastebėti, jog lokali duomenų saugykla nebuvo paminėta arba apibrėžta. Pastaroji gali būti sugeneruota bei valdoma automatiškai.

Antruoju pavyzdiniu atveju turimas vartotojo balanso(įplaukiančios/išplaukiančios lėšos) įvykių srautas ir norima gauti vartotojo, kurio asmens kodas yra *39008226547*, einamosios savaitės sąskaitos balansą. Demonstacija pateikta 2 kodo pavyzdyje.

```
val duration = 1.weeks
val personalNum = "39008226547"
val balanceStream = Stream(es, "customerBalance")
val notOlderThanOneWeek =
    for {
        event <- balanceStream
        filtered  <- event.filter(_.personalNum == personalNum
            && (DateTime.now - _.timeStamp) >= duration)
    } yield  filtered
val sum = notOlderThanOneWeek.toList.sum
```

Kodo pavyzdys 2 - vartotojo einamosios sąskaitos balansas naudojant įsivaizduojamą Scala API

Šiuo atveju *event* kintamasis yra galimai įvykių srauto monada (terminas vartojamas funkciniame programavime, kilęs iš kategorijų teorijos ir turi savas taisykles), o *filtered* kintamasis - duomenų saugyklos monada. Bendruoju atveju skirtingos monados tarpusavyje nesiderina [KW92]. Dėl to reikia išsiaiškinti - ar yra įmanoma ir kaip šias monadas suderinti.

1.5 Tyrimo metodika

Darbo tikslui pasiekti tiriamojoje dalyje bus pasirinkta konkreti funkcinė programavimo kalba (pvz.: Haskell, Scala) bei aprašoma kūrimo metodika.

1.6 Laukiami rezultatai

Magistrinio darbo metu planuojama išnagrinėti funkcinio-reaktyvaus programavimo ir įvykio kaupimo principus, įrodyti, jog šie principai gali būti panaudoti kartu bei suderinti, sukurti konkretizuotą kalbą (angl. domain specific language), apjungiančią šiuos principus, bei aprašyti kūrimo eigos metodiką, apibrėžti gautus rezultatus, suformuluoti apribojimus, iškilusias problemas bei paaiškinti jų priežastis.

2 Literatūros analizė

2.1 Funkcinis-reaktyvus programavimas

2.1.1 Įvadas

Pagrindinė funkcinio programavimo abstrakcija yra funkcija, kuri priima tam tikrą įvestį, o jos rezultatas yra tam tikra išvestis. Funkcijos įvestis bei rezultatas gali būti kita funkcija. Funkcinėje programavimo kalboje funkcijos yra pirmos eilės reikšmės.

Priešingai, labiau populiarus imperatyvaus programavimo modelis priima sakinius arba veiksmus kaip pirminius programos konstravimo blokus, kurie modifikuoja būseną. Tokios programos turi nuoseklų kontrolės srautą ir reikalauja samprotavimo apie pašalinius efektus. Iš prigimties jie yra atsparūs kompoziciniam samprotavimui.

Netgi funkcinėse programavimo kalbose, reaktyvios programos yra paprastai parašytos imperatyviu stiliumi, naudojant žemo lygio bei ne paruoštų komponentų abstrakcijas įskaitant atgalinius skambintojus arba objektu paremtais įvykių dorokliais. Tai pririša interaktyvumo modelį prie prie žemo lygio realizacijos detalių tokių kaip laikas bei įvykių apdorojimo modeliai.

Anot [Ams], funkcinis reaktyvus programavimas, toliau vadinamas tiesiog FRP reiškia, jog modelis turi išlaikyti funkcinio programavimo charakteristikas (pavyzdžiui, primityvios kalbos konstrukcijos turi likti pirmosios klasės) įtraukiant reaktyvumą į kalbos modelį. Funkcinis-reaktyvus programavimas yra būdas modeliuoti reaktyvų (besikeičiantis laiko tekmėje bei reaguojantis į išorinį stimulą) elgesį visiškai funkcinėse programavimo kalbose. FRP leidžia deklaratyviu ir paprastu būdu modeliuoti sistemas, kurios turi reaguoti į duomenis bėgant laikui.

2.1.2 Pagrindinis tikslas

Pagrindinis funkcinio-reaktyvaus programavimo tikslas [Ams]:

- saugus programavimas - kompiliatorius turi kiek įmanoma patikrinti programų korektiškumą;
- efektyvus programavimas - programos turėtų veikti realiu laiku, todėl efektyvios ir optimizuotos operacijos yra būtinos;
- komponavimas - suteikti kompozicines ir aukšto lygio abstrakcijas, jog būtų galima kurti reaktyvias programas. FRP leidžia kurti programas iš smulkesnių programų, o ne orientuotą į problemą, vientisą kodą.

2.1.3 Sąvokos

Pagrindinės FRP sąvokos yra [EH97]:

- signalai arba elgsena - besikeičiančios laike reikšmės;
- įvykiai - kolekcija momentinių reikšmių arba laiko-reikšmės poros.

FRP pasiekia reaktyvumą naudodamas konstrukcijas, kurios tiksliai apibrėžia kaip signalai arba elgsena pasikeičia reaguodami į įvykius. Tai yra pagrindinis būdas išreiškiant bei realizuojant elgseną. Kitu būdu, elgsena gali būti laiko semantinės funkcijos, kuriose laikas yra pakeičiamas kilus įvykiui [NCP02].

2.1.4 Sąvybės

Anot anksčiausios funkcinio-reaktyvaus programavimo formuluotės [EH97], pagrindinės sąvybės, kuriomis pasižymi FRP:

- elgsenos arba signalų modeliavimas bėgant laikui,
- įvykių, kurie turi baigtinį skaičių atsitikimų daugelyje laiko taškų, modeliavimas,
- perjungimas (angl. switching) - sistema gali pasikeisti dėl atsitikusių įvykių,
- analizės detalių, tokių kaip reaktyvaus modelio įvykių ėmimo dažnis, atskyrimas.

2.1.5 Modeliavimo privalumai lyginant su žemo lygio prezentacijos detalėmis

Anot [EH97] modeliavimo privalumai prieš prezentacijos detales (pavyzdžiui nuotraukos pozicijos valdymas cikle) yra panašūs į funkcinės (arba galima sakyti deklaratyvios) programavimo kalbos paradigmą ir apima aiškumą, kūrimo lengvumą, komponavimą ir švarią semantiką. Be šių yra programai būdingų privalumų, tam tikrais atvejais patrauklesnių iš programinės įrangos kūrėjo bei galutinio vartotojo perspektyvos. Šie privaluai apima:

- Kūrimas - turinio kūrimo sistemos natūraliai konstruoja modelius, nes šių sistemų galutinis vartotojas mąsto modelio terminais ir paprastai neturi nei noro nei patirties programavimo prezentacijos detalėse.

- Optimizuojamumas - modeliu paremtos sistemos turi prezentacijos subsistemą, kuri gali atvaizduoti bet kokį modelį, kuris gali būti sukurtas sistemoje. Egzistuoja daug galimybių optimizacijai, nes aukšto lygio informacijos detalės yra prieinamos prezentacijos subsistemai.

- Reguliavimas - prezentacijos subsistema gali lengviau apibrėžti detalių išsamumo lygio valdymą bei pavyzdžių ėmimo dažnį, būtiną interaktyvioms animacijoms, remiantis reginio sudėtingumo, mašinos greičiu ir apkrova ir t.t.

- Mobilumas ir saugumas - modeliavimo platformos nepriklausomumas palengvina mobilių aplikacijų, kurios yra įrodytai saugios WWW(World Wide Web) programos, konstravimą.

2.1.6 Modeliavimo esmė

Yra keturios pagrindinės modeliavimo idėjos [EH97]:

- Laikinas modeliavimas. Reikšmės, vadinamos elgesiu, kurios kinta bėgant laikui yra labiausiai dominančios. Elgesys yra pirmos klasės reikšmės ir sukurtos kompoziciškai. Lygiagretumas yra išreikštas natūraliai ir neišreikštinai. Pavyzdžiui, sekanti išraiška išreiškia animaciją (paveikslėlio elgesį), kas yra apskritimas ant kvadrato. Laiko taške t, apskritimas turi dydį sin t ir kvadratas turi dydį cos t.

```
bigger (sin time) circle 'over' bigger (cos time) square
```

- Įvykių modeliavimas. Kaip ir elgesys, įvykiai yra pirmos eilės reikšmės. Įvykiai gali reikšti tam tikrus nutikimus realiame pasaulyje (pavyzdžiui, pelės mygtuko paspaudimas) arba predikatus paremtus animacijos parametrais (pavyzdžiui, artimumą arba susidūrimą). Tokie įvykiai gali būti sujungti su kitais iki norimo sudėtingumo taip atskiriant sudėtingą animacijos logiką į semantiškai turiningus, modulius konstravimo blokus. Pavyzdžiui, įvykis, aprašantis pirmą kairio mygtuko paspaudimą po laiko t0 yra paprasčiausiai *1bp t0*; aprašantis laiko kvadratą lygų penkiems yra *predicate (pow(time, 2) == 5 t0)* ir jų loginė disjunkcija *1bp t0 ./. predicate (pow(time, 2) == 5) t0*

- Deklaratyvus reaktyvumas. Elgesys dažnai yra natūraliai išreiškiamas kaip atsakas į įvykį. Bet netgi šis reaktyvus elgesys turi deklaratyvią semantiką dėl būsenos pasikeitimų, neretai įtraukiamų į įvykiais paremtą formalizmą. Pavyzdžiui, spalvos reikšmės elgesys, kuris periodiškai keičiasi iš raudonos į žalią su kiekvienu mygtuko paspaudimu gali būti aprašytas kaip paprastas pasikartojimas:

```
colorCycle t0 =
        red 'untilB' 1bp t0 *=> \t1 ->
        green 'untilB' 1bp t1 *=> \t2 ->
        colorCycle t2
```

- Polimorfinė medija. Laike besikeičiančių medijų (nuotraukos, video, garsas, 3D grafika) įvairovė ir šių tipų parametrai (erdvinės transformacijos, spalvos, taškai, vektoriai, skaičiai) turi savo pačių specialiai tipui skirtas operacijas (pavyzdžiui, nuotraukų sukimas, garso maišymas, skaitmeninė sudėtis), tačiau sutelpa į bendrinį elgesio ir reaktyvumo karkasą. Pavyzdžiui, 'untilB' operaciją naudojama prieš tai yra polimorfinė, tinkanti bet kuriai laike beisikeičiančiai reikšmei.

2.1.7 Skirtingi požiūriai

Anot [Ams], egzistuoja du bendri požiūriai į FRP:

- klasikinis FRP - elgesys ir įvykiai yra pirmos klasės ir reaktyvūs objektai;
- signalo-funkcijos FRP - elgesio ir įvykių transformatoriai yra pirmos eilės ir reaktyvūs objektai.

2.1.8 Klasikinis FRP

Anksčiausia ir dar vis standartinė FRP formuluotė [EH97] pateikia du primityvius tipų konstruktorius - Behavior (elgesys) ir Event (įvykis) - kartu su kombinatoriais, kurie pagamina šių tipų reikšmes. Lengviausias semantinis apibrėžimas šiems tipams yra pateiktas 3.

```
type Event a = [(Time, a)]
type Behavior a = Time -> a
```

Kodo pavyzdys 3 - klasikinio FRP semantiniai tipai

Kai šie du tipų konstruktoriai yra tiesiogiai išreikšti, sistema yra žinoma kaip klasikinė FRP sistema.

2.1.9 Klasikinio FRP istorija

Klasikinis FRP buvo originaliai parašytas kaip Fran (Funkcininė reaktyvi animacija) [EH97]. Fran yra karkasas, skirtas deklaratyviai specifikuoti interaktyvias animacijas. Fran atvaizduoja elgesį kaip dvi funkcijas, vieną nuo laiko iki reikšmės ir kitą nuo laiko intervalo (apatinė ir viršutinė laiko riba) iki reikšmės intervalo ir naujo elgesio. Įvykiai atvaizduojami kaip „tobulėjančios reikšmės", kurios imant su laiku pagamina žemesnią laiko ribą kitam atvejui, arba kitą įvykį jeigu jis iš tikrųjų įvyko.

Pirmoji FRP realizacija ne Haskell kalboje buvo Frappe [Cou01], realizuota naudojantis Java Beans karkasu. Frappe yra sukurta remiantis įvykių supratimu bei Beans karkaso susietomis sąvybėmis (bound properties) teikiant abstrakčias sąsajas FRP įvykiams ir elgesiui bei kombinatorius kaip konkrečias klases, realizuojančias šias sąsajas.

2.1.10 Dabartinės klasikinės FRP sistemos

FrTime [ICK06] kalba išplečia Scheme transliatorių nepastoviu priklausomybių grafu, kuris yra sukonstruojamas įvertinus programą. Signalų pasikeitimai atnaujina šį grafą. FrTime nesuteikia atskirų įvykių sąvokų ir pasirenka priklausomybių grafo šakas naudojant sąlyginį elgesio įvertinimą, o ne elgesio pakeitimą, naudojamą FRP sistemų.

Reactive [Ell09] sistema yra dvitaktė (angl. push-pull) funkcinio reaktyvaus programavimo sistema su pirmos klasės elgesiu bei įvykiais. Pirminė Reactive įžvalga yra reaktyvumo (arba kitaip atsako pasikeitimai į įvykius, kurių atsitikimo laikas negalėjo būti žinomas prieš tai) ir laiko priklausomybės atskyrimas. Tai duoda kelią reaktyviąjai normalinėj formai, kuri atvaizduoja elgesį kaip konstantą arba paprastai priklausančią nuo laiko reikšmę, kartu su įvykių srauto nešamomis reikšmėmis, kurios irgi yra reaktyvios normalinės formos elgesys. Stūmimu (angl. push) paremtas įvertinimas yra pasiekiamas šakojant Haskell gijas, jog būtų įvertintas galvos elgesys kol yra laukiama įvykių srauto įvertinimo. Įvykus įvykiui, dabartinio elgesio gija yra nužudoma ir sukuriama nauja gija įvertinti naują elgesį. Deja Reactive realizacija naudoja nepatvarią techniką, kuri priklauso nuo gijų šakojimosi įvertinant dvi Haskell reikšmes lygiagrečiai, kad būtų galima realizuoti įvykių sujungimą. Tai priklauso nuo bibliotekos autoriaus, kad būtų galima užtikrinti darną kai naudojama ši technika ir priveda prie gijų nuotėkio kai vienas iš sujungtų įvykių yra įvykių sujungimo rezultatas.

Evan Czaplicki [CC13] aprašo Elm - autonominę kalbą reaktyvumui. Elm suteikia kombinatorius manipuliuoti diskrečiais įvykiais ir kompiliuojasi į Javascript kalbą, kas padaro tai naudinga kliento pusės web programavimui. Tačiau Elm nesuteikia perjungimo arba besitęsiančio laiko elgesio, nors suderinimas yra pateikiamas naudojant diskretaus laiko įvykius, kurie yra sužadinami pasikartojančiais intervalais, specifikuotais apibrėžiant įvykį. Tezė teigia, jog Arrowized FRP (signalų-funkcijų FRP) gali būti integruota į Elm, bet suteikia mažai paramos šiam teiginiui.

Reactive-banana[1] biblioteka yra dvitaktė (angl. push-pull) FRP sistema sukurta naudoti su Haskell GUI karkasu. Visų pirma, ji charakterizuoja monadą elgesio ir įvykių kūrimui, kuri gali būti komponuojama bei įvertinama. Ši monada apima

[1] http://hackage.haskell.org/package/reactive-banana

konstrukcijas GUI bibliotekos konstrukcijų pririšimui prie primityvių įvykių. Ji privalo būti įkomponuojama į Haskell IO veiksmą įvertinimui įvykti. Reactive-banana realizacija yra panaši į FrTime - naudojant priklausomybių grafą tinklo atnaujinimui įvykus įvykiui. Reactive-banana taip pat kaip Frtime vengia apibendrinto perjungimo vietoje elgesio reikšmių šakojimosi funkcijų, bet išlaiko elgesio ir įvykių atskyrimo. Užuot apibendrinto perjungimo kombinatoriaus, kuris leidžia pakeisti sutartinį elgesį, reactive-banana suteikia žingsninį kombinatorių, kuris pažingsniui sukuria elgesį iš įvykio srauto reikšmių.

2.1.11 Signalo funkcijos FRP

Alternatyvus FRP būdas pirmą kartą pasiūlytas darbe apie Fruit [CE01]. Fruit yra biblioteka, skirta deklaratyviam GUI specifikavimui. Biblioteka naudoja rodyklės sąvoką signalo-funkcijos abstrakcijai. Rodyklės yra abstraktaus tipo konstruktoriai su įvesties ir išvesties tipo parametrais kartu su rinkiniu maršruto parinkimo kombinatorių. Tai demonstruojama 4. Rodyklės idėja Haskell kalboje, įskaitant rodyklių kombinatorių aksiomas, kurias turi tenkinti, yra išvesti iš rodyklių sąvokų iš kategorijų teorijos.

```
(>>>) :: (Arrow a) => a b c -> a c d -> a b d
arr  :: (Arrow a) => (b -> c) -> a b c
first  :: (Arrow a) => a b c -> a (b, d) (c, d)
second :: (Arrow a) => a b c -> a (d, b) (d, c)
(***) :: (Arrow a) => a b c -> a b d -> a b (c, d)
loop :: (Arrow a) => a (b, d) (c, d) => a b c
```

Kodo pavyzdys 4 - rodyklių kombinatoriai

Signalų funkcijos yra nuo laiko priklausančios ir įvykių bei elgesio reaktyvūs transformatoriai. Elgesys ir įvykiai negali būti tiesiogiai manipuliuojami. Šis būdas turi du motyvus: padidina modalumą, kadangi signalo funkcijų įvestis ir išvestis gali būti transformuojama ir tai išvengia didelės laiko klasės ir atminties nuotėkio, kas nutinka kai FRP realizuojamas kaip pirmos klasės elgesys ir įvykiai.

Panašiai kaip ir FrTime, Netwire[2] biblioteka vengia dinaminio perjungimo, šiuo atveju dėl signalo slopinimo. Netwire yra parašyta kaip rodyklės tranformatorius.

[2]http://hackage.haskell.org/ package/netwire-3.1.0

Signalo slopinimas yra įgyvendintas padarant signalo funkcijų išvestį monoidu ir tada sujungiant signalo funkcijų išvestis. Prislopinta signalo funkcija pagamina monoido nulį (monoid's zero) kaip išvestį. Primityvai apibrėžia elgesio slopinimą ir sukomponuotos signalo funkcijos slopina jeigu jų išvestis dera su monoido nuliu.

Yampa [Nil05] yra rodyklyzuotos FRP sistemos optimizacija, pirmą kartą panaudota Fruit. Yampa realizacija naudoja Generalized Algebraic Datatypes, kad leistų daug didesnę saugaus tipo duomenų tipų klasę signalo funkcijos reprezentaijai. Šis atvaizdavimas kartu su „išmaniais" konstruktoriais ir kombinatoriais suteikia galimybę konstruoti rodyklizuotą FRP sistemą, kuri optimizuoja pati save. Deja pagrindinis neefektyvumas yra nereikalingi įvertinimo žingsniai dėl traukimu paremto (angl. pull-based) įvertinimo. Optimizacija yra speciali ir kieviena nauja optimizacija reikalauja naujų konstruktorių pridėjimo, taip pat kiekvieno primityvaus kombinatoriaus atnaujinimo kiekvienai konstruktorio kombinacijai palaikyti. Tačiau Yampa parodo aiškų efektyvumo privalumą lyginant su prieš tai aprašytomis rodyklizuotomis FRP realizacijomis.

PhD tezė [Scu11] pristatė N-Ary FRP - techniką tipizuojant rodyklizuotas FRP sistemas naudojant priklausomus tipus. Didžioji dalis darbo sudarė priklausomų tipų sistemos korektiškumo įrodymas. Šis darbas pristatė signalų vektorius, tipizuotą konstruktorių, kuris leidžia elgesio bei įvykių atskyrimą FRP sistemos lygyje vietoje įvykių laikymo tik specialiu elgesio tipu.

2.1.12 Neįvykdyti iššūkiai

Yra dvi pagrindinės FRP problemos. Pirma, kol signalo-funkcijos FRP yra iš prigimties saugesnė ir labiau modulinė negu klasikinė FRP, ji dar turi būti efektyviai realizuota. Klasikinės FRP programos yra pažeidžiamos dėl laiko nuotėkio bei priežastingumo pažeidimų dėl galimybės tiesiogiai manipuliuoti reaktyviomis reikšmėmis. Antra, sąsaja tarp FRP programų ir daugybės atskirų įvesties ir išvesties šaltinių išlieka specialūs ir daugeliu atveju realizacijos limituotu variantu.

Viena pagrindinė išimtis yra Reactive-banana sistema, kuri suteikia monadą primityvių įvykių konstravimui ir elgesiui iš kuriuos FRP programa gali būti sukonstruota. Tačiau šis būdas yra nelankstus, nes jis reikalauja bibliotekos palaikymo sistemai. Negana to, būnant klasikine FRP sistema, Reactive-banana pritrūksta galimybės transformuoti elgesio bei įvykių įvestį, kadangi visa įvestis yra neišreikštinė.

2.1.13 Formali semantika

Šiame skyriuje bus apibrėžta formali elgesio ir įvykių domeno sritys bei jų kombinatoriai [EH97].

2.1.13.1 Semantinės domeno sritys

Abstrakti laiko domeno sritis yra vadinama *Time*. Abstrakti polimorfinio elgesio(α-*behaviors*) domeno sritis yra žymima $Behavior_\alpha$, o polimorfinių įvykių(α-*events*) yra žymima $Event_\alpha$.

Daugiausia šių sričių (sveikieji skaičiai, loginės reikšmės) yra standartinės ir nereikalauja paaiškinimo. *Time* domeno sritis reikalauja specialaus traktavimo kadangi laiko reikšmės įtraukia dalinius elementus. Ypač yra žinoma, jog laikas bent jau kažkokia reikšmė netgi nežinant galutinės reikšmės. Tiksliau laiko domeno sritį galima apibrėžti taip: tarkime R yra rinkinys realių skaičių ir jame yra elementai ∞ ir -∞. Šis rinkinys turi standartinį aritmenį rikiavimą $\leq$ įskaitant faktą, jog -∞ $\leq \infty$ kiekvienam x $\in R$.

Dabar apibrėžkime laiką kaip $Time = \mathrm{R} + R$, kur elementai antrame rinkinyje R yra atskiriami rikiaviu $\geq$ (pavyzdžiui, $\geq$ 42 skaitytume kaip "daugiau arba lygu 42". Tada galima apibrėžti $\perp$ *Time* =$\geq$(-∞) ir domeno srities (pavyzdžiui, informacijos) rikiavimą pagal laiką:

$$x \sqsubseteq \mathrm{x},\ \forall \mathrm{x} \in R$$

$$\geq \mathrm{x} \sqsubseteq \mathrm{y} \text{ if } \mathrm{x} \leq \mathrm{y},\ \forall \mathrm{x,y} \in R$$

$$\geq \mathrm{x} \sqsubseteq \geq \mathrm{y} \text{ if } \mathrm{x} \leq \mathrm{y},\ \forall \mathrm{x,y} \in R$$

Lengva pastebėti, jog $\perp$*Time* yra apatinis elementas. Svarbu paminėti, jog y yra tik bent viršutinė dalinių elementų rinkinio riba (angl. leat upper bound), kuri apytiksliai lygi:

$$y = \sqcup \{\geq \ \| \ \mathrm{x} \leq \ y\}$$

Kadangi laiko domeno rikiavimas yra grandinės tipo ir kiekviena tokia grandinė turi bent viršutinę ribą (prisiminkime R turi viršutinį elementą ∞), laiko domenas

yra pilnos dalinės tvarkos (angl. complete partial order). Šis faktas yra būtinas, jog būtų galima užtikrinti, kad rekursyvūs apibrėžimai yra gerai apibrėžti.

Time rinkinio elementai naudingiausi įvertinant laiką kada atsitinka įvykis. Įvykis kurio laikas apytiksliai $\geq$t yra tas, kurio konkretus įvykimo laikas yra didesnis nei *t*. Svarbu, jog įvykio, kuris niekada neįvyksta, laikas yra ∞, bent viršutinė *R* riba.

Galiausiai apibrėžimą galima išplėsti aritmetiniui operatoriui $\leq$ visam *Time* apibrėžiant jo elgesį visuose subdomenuose:

$$x \leq_{\geq} \text{y if x} \leq y$$

Tai gali būti skaitoma kaip: „Laikas x yra mažesnis arba lygus laikui, kuris yra bent y jeigu x mažiau arba lygu y". Lengva parodyti, jog ši išplėsta tipo $Time \rightarrow Time \rightarrow Bool$ funkcija yra tolydi atsižvelgiant į $\sqsubseteq$.

2.1.13.2 Semantinės funkcijos

Polimornio elgesio interpretaciją galima apibrėžti kaip funkciją, kuri priima polimorfinę reikšmę bei pagamina elgesio b reikšmę laiku t.

$$at : Behavior_{\alpha} \rightarrow \text{Time} \rightarrow \alpha$$

Tada galima apibrėžti polimorfinių įvykių interpretaciją kaip paprastą ir negriežtą $Time \times \alpha$ porą, aprašančią laiką ir informaciją, susijusią su įvykio atsitikimu:

$$occ : Event_{\alpha} \rightarrow \text{Time} \times \alpha$$

Žinant semantinę domeno sritį, galima pateikti formalias elgesio bei įvykių kombinatorių interpretacijas.

2.1.13.3 Elgesio semantika

Elgesys yra kuriamas iš kito elgesio, statinių (nesikeičiančių laike) reikšmių ir įvykių naudojantis kolekcija konstruktorių (kombinatorių).

- **Laikas**. Paprasčiausias primityvus elgesys yra laikas - time, kurio semantika yra:

$$time : Behavior_{Time}$$
$$\mathbf{at}[[time]]t = t$$

Šiuo atveju **at**[[time]]t = t yra tik *Time* tapatumo funkcija.

- **Pakėlimas**[3] (angl. lifting) - įprastas būdas funkcijoms, apibrėžiančioms nekintančias reikšmes, „pakelti" į analogines funkcijas, apibrėžtoms elgesiu. Pakėlimas yra įvykdomas naudojant (konceptualiai begalinę) šeimą operatorių - vieną keikvienam funkcijos valentingumui (angl. arity).

$$lift_n : (\alpha_1 \rightarrow \ ... \rightarrow \ \alpha_n \rightarrow \ \beta) \rightarrow \ Behavior_{\alpha_1} \rightarrow \ ... \rightarrow \ Behavior_{\alpha_n} \rightarrow \ Behavior_\beta$$
$$\mathbf{at}[[lift_n \ f \ b_1...b_n]]t = f(\mathbf{at}[[b_1t]])...(\mathbf{at}[[b_n]]t)$$

Svarbu paminėti, jog nesikeičiančios reikšmės pakėlimas yra $lift_0$

- **Laiko transformacija**. Laiko transformacija leidžia vartotojui pakeisti lokalų laikotarpį. Toks būdas palaiko bet kokio elgesio laikiną modalumą. (Panašiai 2D ir 3D transformacijos palaiko erdvinį modalumą geometrijoje)

$$timeTransform : Behavior_\alpha \rightarrow \ Behavior_{Time} \rightarrow \ Behavior_\alpha$$
$$\mathbf{at}[[timeTransform \ b \ tb]] = \mathbf{at}[[b]] \ o \ \mathbf{at}[[tb]]$$

Taigi idėja yra, jog laikas yra *timeTransform* tapatybė:

$$timeTransform \ b \ time = b$$

Pavyzdžiui, laiko transformacijos pavyzdys Fran:

[3]Praktikoje pakėlimas yra reikalingas gana dažnai, todėl būtų nepatogu visur kelti išreikštinai. Vietoje to pageidautina naudoti žinomus vardus, tokius kaip: „sin", „cos", „+", „*" ir netgi literalus, tokius kaip „3" arba „mėlynas" nurodant pakeltas versijas. Pavyzdžiui, literalas „42" turėtų elgtis kaip nekintantis elgesys „$lift_0$ 42", o sudėtis „b_1 + b_2" kaip „$lift_2$ (+) b_1 b_2"

$$timeTransform\ b\ (time/2)$$

sulėtina animaciją dvigubai.

- **Integracija**. Integracija pritaikoma tiek realių skaičių reikšmes turinčiam, tiek 2D ir 3D vektoriaus reikšmes turinčiam elgesiui, arba apskritai vektoriaus erdvei (limituota). Naudojant Haskell notaciją vektoriaus erdvei:

$$integral : VectorSpace\ \alpha \Rightarrow\ Behavior_\alpha \rightarrow\ Time \rightarrow\ Behavior_\alpha$$
$$\mathbf{at}[[integralbt_0]]t = [\int_{t_0}^{t} \mathbf{at}[[b]]]$$

Integracija leidžia specifikuoti elgesio greitį, o naudojama du kartus pagreitį. Pavyzdžiui, judančio kamuoliuko greitis yra duotas kaip elgesys b (gali būti ir pastovus ir nepastovus), tada jo reliatyvi pozicija laiko pradžios atžvilgiu t_0 yra duota kaip *integral b* t_0. Tai leidžia natūraliai išreikšti fizika paremtas animacijas.

- **Reaktyvumas**. Pagrindinė sąveika yra tarp elgesio ir įvykių ir tai padaro elgesį reaktyviu. Konkrečiai elgesys *b untilB e* parodo b elgesį iki tol kol įvyksta e ir tada pasikeičia į elgesį asocijuotą su įvykiu e. Formaliai:

$$untilB : Behavior_\alpha \rightarrow\ Event_{Behavior_{\alpha)}} \rightarrow\ Behavior_\alpha$$
$$\mathbf{at}[[b\ untilB\ e]]t = if\ t \leq t_e\ \mathbf{then}\ \mathbf{at}[[b]]t\ \mathbf{else}\ \mathbf{at}[[b']]t$$
$$\mathbf{where}(t_e, b') = \mathbf{occ}[[e]]$$

2.1.13.4 Įvykių semantika

- **Įvykių apdorojimas**. Norint duoti pavyzdį naudojant specialią rūšį įvykių, pirmiausia reikia apibrėžti įvykių apdorotojus, kurie yra pritaikomi laikui ir su įvykiu susietiem duomenim naudojant šį operatorių:

$$(+\Rightarrow) : \text{Event}_\alpha = (Time \to \alpha \to \beta) \to Event_\beta$$

$$\mathbf{occ}[[e+\Rightarrow \text{f}]] = (\text{t}_e, f\ t_e\ x)$$

$$\mathbf{where}\ (t_e,x) = \mathbf{occ}[[e]]$$

Patogumui galima naudoti šias išvestas operacijas, kurios ignoruoja laiką arba informaciją arba abu kartu:

$$(=\Rightarrow) : \text{Event}_\alpha \to (\alpha \to \beta) \to Event_\beta$$

$$(*\Rightarrow) : \text{Event}_\alpha \to (Time \to \beta) \to Event_\beta$$

$$(-\Rightarrow) : \text{Event}_\alpha \to \beta \to Event_\beta$$

$$ev =\Rightarrow g = ev + \Rightarrow \lambda t\ x.\ g\ x$$

$$ev* \Rightarrow h = ev + \Rightarrow \lambda t\ x.\ h\ t$$

$$ev- \Rightarrow x' = ev + \Rightarrow \lambda x.\ x'$$

(+⇒) gauna visus parametrus, (-⇒) negauna parametrų, (*⇒) gauna tik laiką, o (⟹) gauna tik informaciją.

- **Nekintantys įvykiai**. Paprasčiausia įvykių rūšis yra tiesiogiai specifikuoti pagal laiką ir reikšmę.

$$constEv : Time \to \alpha \to Event_\alpha$$

$$\mathbf{occ}[[constEv\ t_e\ x]] = (t_e,x)$$

Nors, pavyzdžiui, elgesys:

$$b_1\ untilB\ (constEv\ 10\ x) -\Rightarrow \text{b}_2$$

parodo elgesį b_1 iki laiko taško 10, kuriame jis pradeda rodyti elgesį b_2 (x yra ignoruojamas šiame pavyzdyje, bet iš tikro neturėtų).

- **Išoriniai įvykiai**. Išoriniai įvykiai yra pavyzdžiui pelės mygtuko paspaudimas, kuris gali būti kairysis arba dešinysis mygtukas. Reikšmė, asocijuota su kurio nors mygtuko paspaudimu yra interpretuojama kaip atleidimo įvykis, kuris grąžina vienetinę reikšmę (() yra vieneto tipas):

$$lbp, rbp : Time \rightarrow Event_{Event_{()}}$$

Įvykio *lbp* t_0 prasmė, pavyzdžiui, yra pora (t_e, e), tokia kad t_e yra pirmojo kairio mygtuko paspaudimas po laiko t_0 ir e yra įvykis, reiškiantis pirmojo kairiojo mygtuko atleidimą po laiko t_e. Iš to seka, jog:

$$b_1\ untilB\ (lbp, t_0) ==\Rightarrow \lambda e.$$
$$b_2\ untilB\ e(-\Rightarrow)$$
$$b_3$$

parodo elgesį b_1 iki kairiojo mygtuko paspaudimo, kurio metu jis tampa b_2 kol kairysis mygtukas yra atleistas, o tada tampa t_3.

- **Predikatai**. Natūralus būdas specifikuoti tam tikrus įvykius kaip pirmąjį laiką kada loginės reikšmės elgesys tampa tiesa (*true*) po duoto laiko:

$$predicate : Behavior_{Bool} \rightarrow Time \rightarrow Event_{()}$$
$$\mathbf{occ}[[predicate\ b\ t_0]] = (inf\ \{t > t_0 \mid \mathbf{at}[[b]]t\}, ())$$

Predikato įvykio laikas yra begalinis ir sudarytas iš rinkinio laiko taškų didesnių tei t_0, kuriuose elgesys yra teigiamas. Šis laikas gali būti ir t_0.

Tada elgesys:

$$b_1\ untilB\ (predicate\ (sin\ time = 0.5)\ t_0) -\Rightarrow \mathrm{b}_2$$

parodo b_1 iki pirmojo laiko taško t po t_0, kur *sin t* yra 0.5, po kurio jis parodo b_2.

- **Pasirinkimas**. Galima pasirinkti ankstesnįjį iš dviejų įvykių naudojantis operatoriumi .|.:

$$(.|.) : Event_\alpha \rightarrow Event_\alpha \rightarrow Event_\alpha$$
$$\mathbf{occ}[[e\ .|.\ e']] = (t_e,x), \mathbf{if}\ t_e \leq t'_e$$
$$= (t'_e, x'),\ otherwise$$
$$\mathbf{where}\ (t_e,x) = \mathbf{occ}[[e]]$$
$$\mathbf{where}\ (t'_e,x) = \mathbf{occ}[[e']]$$

Pavyzdžiui, elgesys:

$$b_1\ untilB\ (lbp\ t_0\ .|.\ predicate\ (time > 5)\ t_0) {-\!\!\Rightarrow} \mathrm{b}_2$$

laukia kol arba bus nuspaustas kairysis mygukas arba praeis 5 sekundės prieš pakeičiant b_1 elgesį į b_2. Kaip alternatyva, sekantis pavyzdys pakeičia elgesį į b_3 po skirto laiko pabaigos:

$$b_1\ untilB\ (lbp\ t_0 {-\!\!\Rightarrow} \mathrm{b}_2\ .|.\ predicate\ (time > 5)\ t_0) {-\!\!\Rightarrow} \mathrm{b}_3$$

- **Momentinė kopija**. Tuo momentu kai nutinka įvykis yra dažnai patogu padaryti elgesio reikšmės momentinę kopiją tam tikrame laiko taške. Tai formaliai galima užrašyti:

$$snapshot : Event_\alpha \rightarrow Behavior_\beta \rightarrow Event_{\alpha\times\beta}$$
$$\mathbf{occ}[[e\ snapshot\ b]] = (t_e, (x, \mathbf{at}[[b]]t_e))$$
$$\mathbf{where}\ (t_e, x) = \mathbf{occ}[[e]]$$

Pavyzdžiui, elgesys:

$$b_1\ untilB\ (lbp\ t_0\ snapshot\ (sin\ time)) {=\!\!\Rightarrow} \lambda(e,y).\ b_2$$

paima laiko, kai yra nuspaudžiamas kairysis mygtukas, sinusą, priskiria jį y ir seka elgesiu b_2, kuris galimai priklauso nuo y. Nepaisant to, šį pavyzdį taip pat būtų galima realizuoti paimant kairiojo mygtuko paspaudimo įvykio laiką ir skaičiuojant sinusą, bendru atveju elgesio buvimas momentine kopija yra ganėtinai sudėtingas ir gali priklausyti nuo išorinių įvykių.

- **Įvykių sekos** Kartais yra naudinga naudoti vieną įvykį, kad būtų galima sukurti kitą. Įvykis *joinEv e* yra įvykio, kuris nutinka kai nutinka *e'*, kur *e'* yra *e* reikšmės dalis:

$$joinEv : Event_{Event_\alpha} \rightarrow Event_\alpha$$
$$\mathbf{occ}[[joinEv\ e]] = \mathbf{occ}[[snd\ (\mathbf{occ}[[e]])]]$$

Pavyzdžiui, įvykis:

$$joinEv\ (lbp\ t_0 * \Rightarrow predicate\ (b = 0))$$

nutinka pirmą kartą kai elgesys b turi nulinę reikšmę po pirmojo kairiojo mygtuko paspaudimo po laiko tarpo t_0.

Panaudojimo pavyzdys

Remiantis primityvių kombinatorių pavyzdžiais elgesiui bei įvykiam kartu su jų formalia semantika, galima pateikti konkretų pavyzdį Haskell programavimo kalboje. 5 pateiktame pavyzdyje yra skaitine reikšme išreikštas elgesys, kurio pradinė reikšmė yra 0 ir tampa -1 jeigu yra nuspaustas kairys pelės mygtukas (lbp - left button pressed) arba 1 jeigu yra nuspaustas dešinys pelės mygtukas (rbp - right button pressed).

```
bSign t0 =
        0 'untilB' lbp t0 ==> nonZero (-1) .|.
                              rbp t0 ==> nonZero 1
        where nonZero r stop =
                          r 'untilB' stop *==> bSign
```

Kodo pavyzdys 5 - signalo funkcija nuo pelės mygtuko paspaudimo

Šį elgesį galima panaudoti paveikslėlio dydžio keitimui. Kairio arba dešinio pelės mygtuko paspaudimas priverčia paveikslėlį padidėti arba sumažėti. Pavyzdys pateiktas 6.

```
bSign t0 =
        0 'untilB' lbp t0 ==> nonZero (−1) .|.
                              rbp t0 ==> nonZero 1
        where nonZero r stop =
                          r 'untilB' stop *=> bSign
```

Kodo pavyzdys 6 - paveiksliuko dydžio modifikavimas pelės paspaudimu

2.1.14 Įvykių srautas

Pagal [Bas07], įvykių srautas yra eilė pagal laiką surikiuotų įvykių, pavyzdžiui akcijų rinkos srautas.

Įvykių srautas kaip duomenų srauto tipas formaliai atrodo kaip pora (s, Δ), kur s yra seka surikiuotų sąrašo įvykių, o Δ yra seka laiko intervalų ir kiekvienas $n > 0$.

Tokio duomenų srauto pavyzdžiai gali būti:

- akcijų kursas,
- paspaudimų srautas,
- tinklo srautas,
- GPS duomenys.

Įvykių srauto apdorojimas pagal atsitikimo laiką turi privalumų:

- įvykių apdorojimo algoritmai naudoja mažai atminties, nes jiems nereikia prisiminti daug įvykių;
- algoritmai gali būti labai greiti;
- gavus įvykį, skaičiavimai atliekami iškart, todėl galima perduoti rezultatą kitam skaičiavimui ir pamiršti įvykį.

Įvykių srauto apdorojimas labiau akcentuoja didelio našumo duomenų gavimą ir matematinių algoritmų pritaikymą įvykių duomenims. Taip pat įvykių srautai įprastai pritaikomi konkrečiai sistemai ar organizacijai.

2.2 Įvykių kaupimas

Šiame skyriuje yra aprašomos žinios apie įvykių kaupimą, pliusus ir minusus, įvykių srautus bei įvykių kaupimą funkciniame programavime remiantis daugiausia Vaughn Vernon surinkta ir aprašyta informacija [Ver13].

2.2.1 Įvadas

Kartais verslui svarbu fiksuoti objekto pasikeitimus domeno modelyje. Šiuos pasikeitimus galima stebėti skirtingais būdais. Įprastai yra pasirenkama stebėti kai esybė yra:

- sukurta,
- paskutinį kartą modifikuota
- bei kas atliko modifikaciją.

Tačiau šis būdas nepateikia jokios informacijos apie vienkartinius pasikeitimus.

Atsiradus poreikiui stebėti pasikeitimus detaliau, verslas reikalauja dar daugiau metaduomenenų, ko pasekoje tokie faktai kaip individualios operacijos laiko tekmėje bei jų įvykdymo laikas tampa svarbūs. Šie poreikiai verčia įvesti audito žurnalą fiksuoti labai tikslias panaudojimo atvejų metrikas, tačiau pastarasis būdas turi apribojimų. Jis gali atskleisti dalį informacijos apie tai kas nutiko sistemoje, leisti rasti bei ištaisyti dalį riktų bei klaidų programinėje įrangoje. Bet audito žurnalas neleidžia patikrinti domeno objekto būsenos prieš ir po tam tikrų pasikeitimų. O jeigu būtų galima išgauti daugiau informacijos iš pasikeitimų stebėjimo?

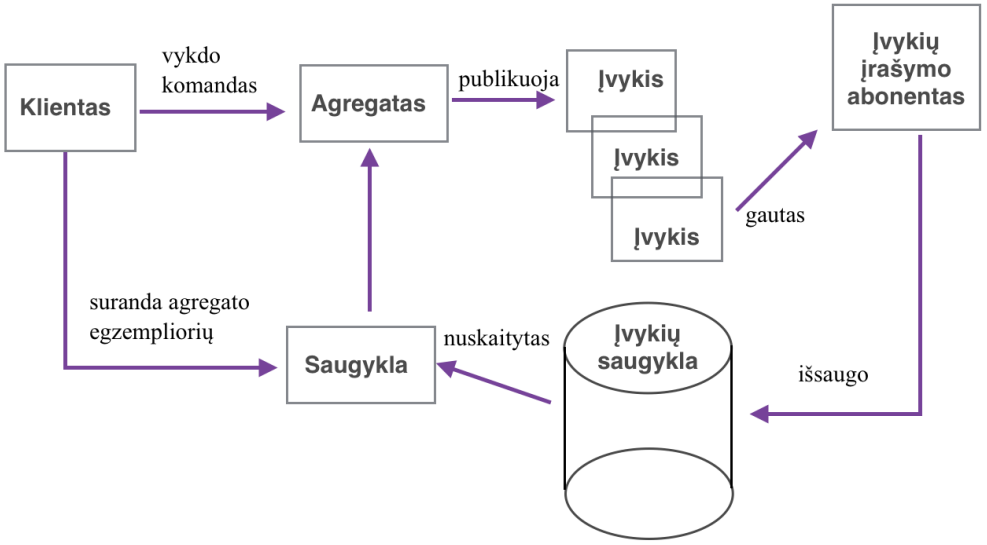
Visi programinės įrangos kūrėjai susiduria su labai tiksliu pasikeitimų stebėjimu. Įprastas ir populiarus pavyzdys yra išeities kodo saugyklos, tokios kaip CVS[4], Subversion[5], Git[6] arba Mercurial[7]. Visos šios pataisų valdymo sistemos leidžia stebėti pirminių failų pasikeitimus. Įrankiai leidžia peržiūrėti išeities kodo artefaktus nuo pačios pirmosios pataisos iki paskutinės. Kai visi išeities failai yra nusiunčiami į pataisų kontrolės sistemą, ši gali stebėti pasikeitimus viso programinės įrangos kūrimo gyvavimo ciklo metu.

[4] http://www.nongnu.org/cvs/
[5] http://subversion.apache.org/
[6] http://git-scm.com/
[7] http://mercurial.selenic.com/

Jeigu šis principas būtų pritaikytas vienai esybei, tada vienam agregatui[8] bei galiausiai kiekvienam modelio agregatui, galima suprasti kokią naudą atneša sistemos objektų pasikeitimų stebėjimas:

- Kas būtent nutiko modelyje, jog agregato egzempliorius buvo sukurtas?
- Kas nutiko agregato egzemplioriui bėgant laikui? (Operacijų požiūriu)

Turint visų atliktų operacijų istoriją, galima palaikyti laikinus modelius. Toks kaitos stebėjimas yra įvykių kaupimo principas. 1 diagramoje pateikta šio principo aukšto lygio reprezentacija. Agregatai publikuoja įvykius, kurie yra išsaugomi įvykių saugykloje ir naudojami sekti modelio būsenos pasikeitimus. Verta paminėti, jog įvykiai reprezentuoja tam tikrą būsenos pasikeitimą bėgant laikui, todėl jie nėra atnaujinami arba ištrinami. Saugykla nuskaito įvykius iš įvykių saugyklos ir pritaiko juos vieną po kito taip atkurdama agregato būseną.



1 pav. Įvykių kaupimo aukšo lygio reprezentacija

2.2.2 Įvykių kaupimo privalumai ir trūkumai

Kaip saugojimo mechanizmas, įvykių kaupimas stipriai skiriasi ir pakeičia ORM[9] įrankį. Kadangi įvykiai dažnai įrašomi kaip dvejetainės reprezentacijos, jie negali būti optimaliai naudojami užklausoms atlikti. Faktiškai įvykių kaupimu pagrįstoms

[8] http://martinfowler.com/bliki/DDD_Aggregate.html
[9] http://www.orm.net/

saugykloms tereikia vienos operacijos - gauti įrašus pagal unikalią agregato tapatybę [You10]. To pasekoje užklausom daryti reikia kito kelio. Dažniausiai tam pasirenkamas CQRS[10] principas [BDM+13].

Įvykių istorija gali padėti surasti bei ištaisyti sistemos defektus bei klaidas [Fit12]. Derinimas naudojant istoriją visų veiksmų, kurie nutiko sistemoje, turi didžiulį pranašumą. Įvykių kaupimas gali vesti prie didelio našumo domeno modelių, tai yra palaikyti ypač didelį skaičių operacijų per sekundę. Pavyzdžiui, įrašymas į vieną duomenų saugyklos lentelę yra ypač greitas. Negana to, tai leidžia CQRS užklausų modelį išplėsti horizontaliai, nes duomenų šaltinio atnaujinimai įvykdomi fone, kai įvykių saugykla yra atnaujinama naujais įvykiais.

2.2.3 Įvykių kaupimas funkciniame programavime

Vaughn Vernon pateikia keletą pastebėjimų apie įvykių kaupimą funkciniame programavime, kurie gali būti naudingi atliekant projektinius sprendimus bei eksperimentinį tyrimą:

- Agregatas projektuojamas kaip nekintantis būsenos įrašas kartu su funkcijomis, kurios keičia būseną. Šios funkcijos paprasčiausiai priima būsenos įrašą ir įvykių argumentus ir gražina naują būsenos įrašą kaip rezultatą. Tokia funkcija pavaizduota 7 kodo pavyzdyje.

```
Funkcija<Busena, Ivykis, Busena>
```

Kodo pavyzdys 7 - agregato būsenos keitimas

- Dabartinė agregato būsena gali būti apibrėžta kaip suskleidimas į kairę visų praeities įvykių, kurie yra perduodami būseną keičiančiai funkcijai.
- Agregato metodai gali būti išreikšti kaip kolekcija funkcijų be būsenos.
- Įvykių saugykla gali būti suvokiama bei naudojama kaip funkcinė duomenų bazė, nes ji perduoda argumentus funkcijoms, kurios keičia agregato būseną.

[10] http://martinfowler.com/bliki/CQRS.html

2.3 Išvados

Literatūros analizės metu remiantis kitų autorių patirtimi:

- išnagrinėtas funkcinis-reaktyvus programavimas,
- išnagrinėtas įvykių kaupimo principas,
- išnagrinėti įvykių srautai bei operacijos su jais,
- susipažinta su įvykių kaupimu funkciniame programavime,
- įvaldyta sąvokų sistema, susijusi su nagrinėjama tematika.

3 Sąvokų apibrėžimai

- **Agregatas** (angl. aggregate) - DDD modelis, rinkinys domeno objektų, kurie gali būti laikomi kaip visuma.
- **Derinimas** (angl. debugging) - riktų bei klaidų paieška programinėje įrangoje bei jų taisymas.
- **Esybė** (angl. entity) - kažkas, kas egzistuoja pats savaime, faktiškai arba hipotetiškai.
- **Horizontalus išplečiamumas** (angl. horizontal scaling) – galimybė sujungti daugybę techninės ar programinės įrangos esybių taip, jog jos dirbtų kaip visuma. Pavyzdžiui, galima pridėti keletą serverių pasinaudojant grupavimu arba apkrovos paskirstymu taip pagerinant sistemos našumą bei prieinamumą.
- **Metaduomenys** (angl. metadata) - duomenys apie kitus duomenis.
- **Tapatybės funkcija** (angl. identity function) - funkcija be jokio poveikio (ji visada grįžta ir jos argumentai tos pačios reikšmės).
- **Valentingumas** (angl. arity) - funkcijos valentingumas yra argumentų kiekis, kurį ji priima.

4 Santrumpos ir paaiškinimai

- **CQRS** (angl. Command Query Responsibility Segregation) – komandų-užklausų atsakomybių atskyrimas.
- **DDD** (angl. Domain-Driven Design) – būdas kurti programinę įrangą, skirtą spręsti sudėtingus uždavinius, bei apjungti realizaciją kartu su augančiu domeno modeliu.
- **NoSQL** – duomenų bazė, skirta architektūriniams modeliams, kuriems nereikia palaikyti stiprios darnos principo, kuris naudojamas reliacinėse duomenų bazėse. Tai įgalina horizontalų išplečiamumą bei aukštesnį prieinamumą.
- **ORM** (angl. Object-Relational Mapping) – programavimo technika duomenų konvertavimui tarp nesuderinamų sistemų tipų, naudojama objektinio programavimo kalbose. Pavyzdys: JAVA programavimo kalba bei Oracle duomenų bazė.

Literatūros sąrašas

[Ams] Edward Amsden. A survey of functional reactive programming. *Rochester Institute of Technology.*

[Bas07] Tim Bass. Mythbusters: Event stream processing versus complex event processing. In *Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems*, DEBS '07, pages 1–1, New York, NY, USA, 2007. ACM.

[BDM+13] Dominic Betts, Julian Dominguez, Grigori Melnik, Fernando Simonazzi, and Mani Subramanian. *Exploring CQRS and Event Sourcing: A Journey into High Scalability, Availability, and Maintainability with Windows Azure.* Microsoft patterns & practices, 1st edition, 2013.

[CC13] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for guis. *SIGPLAN Not.*, 48(6):411–422, June 2013.

[CE01] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *IN PROCEEDINGS OF THE 2001 HASKELL WORKSHOP*, pages 41–69, 2001.

[Cou01] Antony Courtney. Frappé: Functional reactive programming in java. In *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages*, PADL '01, pages 29–44, London, UK, UK, 2001. Springer-Verlag.

[Cza12] Evan Czaplicki. Elm: Concurrent frp for functional guis. Master's thesis, 30 March 2012.

[EH97] Conal Elliott and Paul Hudak. Functional reactive animation. *SIGPLAN Not.*, 32(8):263–273, August 1997.

[Ell09] Conal M. Elliott. Push-pull functional reactive programming. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 25–36, New York, NY, USA, 2009. ACM.

[Fit12] Sean Fitzgerald. A pattern for state machine persistence using event sourcing, cqrs and a visual workbench. Master's thesis, 21 March 2012.

[ICK06] Daniel Ignatoff, Gregory H. Cooper, and Shriram Krishnamurthi. Crossing state lines: Adapting object-oriented frameworks to functional reactive languages. In *Proceedings of the 8th International Conference on Functional and Logic Programming*, FLOPS'06, pages 259–276, Berlin, Heidelberg, 2006. Springer-Verlag.

[KW92] David J. King and Philip Wadler. Combining monads. In John Launchbury and Patrick M. Sansom, editors, *Functional Programming*, Workshops in Computing, pages 134–143. Springer, 1992.

[NCP02] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 51–64, New York, NY, USA, 2002. ACM.

[Nil05] Henrik Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. *SIGPLAN Not.*, 40(9):54–65, September 2005.

[Scu11] Neil Sculthorpe. Towards safe and efficient functional reactive programming, 2011.

[Ver13] Vaughn Vernon. *Implementing Domain-Driven Design*. Addison-Wesley Professional, 1 edition, February 2013.

[You10] Greg Young. Cqrs documents by greg young. Technical whitepaper, 2010.