TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Halliki Mullari

Java baitkoodi sünkroniseerimise analüüs raamistikus Põder

Bakalaureusetöö (9 EAP)

Juhendaja(d): Kalmer Apinis

Tartu 2020

Java baitkoodi sünkroniseerimise analüüs raamistikus Põder

Lühikokkuvõte:

Üha keerukamad rakendused nõuavad arvutitelt üha enam jõudlust. Üks viis jõudlust lisada on arvuti protsessoritele lisada tuumasid. Selleks, et programmid tuumade lisamisest kasu saaksid peab kasutama lõimtöötlust, samas lisab lõimtöötlus programmidele keerukust. Üheks probleemiks, mis lõimtöötlusega kaasneb, on andmejooks. Andmejookse on võimalik avastada staatilise analüüsi vahenditega.

Käesoleva töö eesmärgiks on lisada raamistikku Põder Java baitkoodi sünkroniseerimise analüüsi moodul, mis suudaks programmides välistada andmejooksusid. Analüüsi tulemuseks on raport, milles kas välistatakse andmejooksude esinemine või tuuakse välja muutujad, mille puhul kahtlustatakse andmejooksu ning info, kus ja mis monitorid selle muutuja kirjutamise ja lugemise hetkedel kasutuses olid. Selle infoga on võimalik parandada sünkroniseerimise vigu, et andmejooksusid ei esineks.

Võtmesõnad:

Staatiline analüüs, sünkroniseerimine

CERCS: P175 Informaatika, süsteemiteooria

Static Synchronization Analysis of Java Bytecode in Põder Framework

Abstract:

Complex applications demand more and more performance from computers. One way to add performance is to add more cores to the processor. To utilize those added cores, the use of multithreading in programs is needed. However, multithreading adds additional complexity to programs and one problem that arises with multithreading is data races. Data races can be discovered with static analysis.

The aim of this thesis is to add a module to the Põder framework that analyses Java bytecode synchronization and that could rule out the presence of data races. The result of the analysis produces a report that states if it could rule out data races. If not it brings to attention the variables that could have data races and to points out from where they were accessed and which monitors were guarding them at that time. With this information it is possible to fix synchronization flaws so that data races can be eliminated.

Keywords:

Static analysis, synchronization

CERCS: P175 Informatics, systems theory

Sisukord

Sissejuhatus

Arvutite jõudlus kasvab iga aastaga ning neile loodavad järjest keerukamad rakendused nõuavad omakorda jõudluse edasist kasvu. Moore'i seadus ütleb, et transistorite arv mikroprotsessori kiibil kahekordistub iga kahe aastaga. Protsessorite jõudluse suurendamiseks on püütud väikesele alale mahutada aina rohkem transistoreid ja tõsta protsessorite taktisagedust. Taktisageduse kasv on pidurdunud, aga protsessoritele mitme tuuma lisamisega saab rohkem transistoreid ära kasutada ja jõudlust tõsta. Mitme tuumaga protsessorid on tänapäeval laialdaselt levinud nii serverites, sülearvutites kui ka mobiiltelefonides [1]. Et tuumade lisamisest kasu oleks, peab kasutama lõimtöötlust (*multithreading*). Lõimtöötlus lisab programmidesse keerukust ja seda on raskem teostada ja seetõttu oleks abi analüüsi raamistikest nagu Põder. Käesolevas töös lisatakse raamistikule Põder lõimtöötlusega esinevate kindlat tüüpi vigade analüüsi moodul.

Kui programm täidab käsklusi ainult ühel lõimel (*thread*), täidab programm ühe käsu teise järel. Üks lõim saab samal ajahetkel kasutada ühte tuuma. Kasutame näitena serverit, mis võtab klientidelt päringuid vastu ja töötleb neid. Ühe lõimega saab korraga võtta vastu ühe päringu, seda töödelda ja alles siis järgmise päringu vastu võtta. Kui päringu töötlemine võtab kaua, lükkub teiste päringute töötlemine selle võrra edasi. Lõimtöötlusega on võimalik seda optimeerida nii, et üks lõim võtab vastu päringuid ja suunab edasi teistele lõimedele töötluseks. Samal ajal kui teised lõimed tegelevad päringute töötlusega, saab esimene lõim järgmise päringu vastu võtta. Mitu lõime saab korraga ära kasutada mitut tuuma. Kui mitu lõime täidavad sama programmi osa samaaegselt või vaheldumisi, siis nimetatakse seda **lõimtöötluseks**. Lõimtöötluse puhul jagatakse programmi töö mitme lõime vahel [2].

Lõimede vahel programmi töö laiali jagamisel võivad tekkida omad probleemid. Joonisel 1 on toodud klass *Konto*. *Konto*-l on saldo, millele saab raha lisada meetodiga *lisa* ja raha välja võtta meetodiga *eemalda*. Meetodis *main* luuakse uus konto *minuKonto*, ja lisatakse sinna 99 eurot. Järgmisena püütakse kontolt kaks korda raha välja võtta. Raha välja võtmisel kontrollitakse meetodis kas konto saldo on väiksem kui väljavõetav summa. Esimesel korral raha välja võtmine õnnestub, kui kutsutakse välja *eemalda(20)*, mis võrdleb seda summat saldoga. Kuna 99 < 20, siis lahutatakse saldost 20 ja tagastatakse 20. Uus saldo väärtus on 79. Teisel korral raha välja võtmine ebaõnnestub, sest küsitav summa on väiksem kui konto saldo ja tagastatakse 0. Kui nende meetodite täitmine jagada mitme lõime vahel võib tekkida olukord, kus *minuKonto.eemalda(20)* ja *minuKonto.eemalda(80)* kutsutakse kahe erineva lõime poolt välja samaaegselt. Mõlemal juhul võrreldakse summat saldoga ja mõlemal juhul on saldo suurem kui summa. Seega mõlemal korral lahutatakse summa saldost ja raha välja võtmine õnnestub. Kokku eemaldatakse kontolt rohkem raha kui seal oli, kuigi oli olemas kontroll, et kontol oleks piisavalt raha. Selliseid olukordasid nimetatakse andmejooksudeks. **Andmejooks** (*data race*) on olukord kus kaks või enam lõime üritavad samaaegselt kasutada sama asukohta mälus ja nendest vähemalt üks on kirjutamine [3].

```
public class Konto {
    private int saldo = 0;

    public static void main(String[] args) {
        Konto minuKonto = new Konto();
        minuKonto.lisa(99);
        minuKonto.eemalda(20);
        minuKonto.eemalda(80);
    }

    public int eemalda(int summa) {
        if (saldo < summa) {
            return 0;
        }
        saldo = saldo - summa;
        return summa;
    }

    public void lisa(int summa) {
        saldo = saldo + summa;
    }
}
```

Joonis 1. Klass *Test* meetodiga *liida*.

Staatilise analüüsiga on võimalik andmejooksusid leida ja sünkroniseerimisega parandada. Antud töö eesmärgiks on lisada raamistikule Põder sünkroniseerimise analüüsi moodul, millega leida programmist andmejooksud. Mida suurem programm on, seda keerulisem on selliseid vigu leida ja automaatsetest analüüsi vahenditest on palju abi.

Esimeses peatükis antakse ülevaade probleemist, mis vajab lahendamist ja lahenduse jaoks olulisest taustainfost. Peatükk 1.1 on lähemalt juttu andmejooksudest ja nende vältimise võimalusest õige sünkroniseerimisega. Peatükid 1.2 ja 1.3 tutvustavad Java virtuaalmasina ülesehitust ja kuidas seal Java baitkood töötab. Peatükk 1.4 räägib lähemalt sellest kuidas toimib sünkroniseerimine baitkoodi tasandil. Peatükkides 1.5, 1.6, 1.7 tuleb juttu juhtvoograafidest ja võredest ning näitab, et nende abil saab kokku panna staatiliseks analüüsiks vajaliku võrrandisüsteemi. Viimases alapeatükis on lühike tutvustus Põder raamistikust ning mida on vaja implementeerida, et sinna uut moodulit lisada.

Teine peatükk tutvustab probleemi lahendust. Esimeses alampeatükis tutvustatakse näite põhjal, mida loodud analüüsi mooduliga teha saab. Edasi on juttu lahenduse detailidest. Peatükis 2.2 kirjeldatakse kuidas sai kokku pandud programmi analüüsi jaoks olulist infot sisaldav struktuur. Peatükis 2.3 on välja toodud mõned üleminekufunktsioonide implementatsiooni näited ja kirjeldused. Viimases alapeatükis võetakse kokku lahenduse probleemid ja võimalikud edasiarendused.

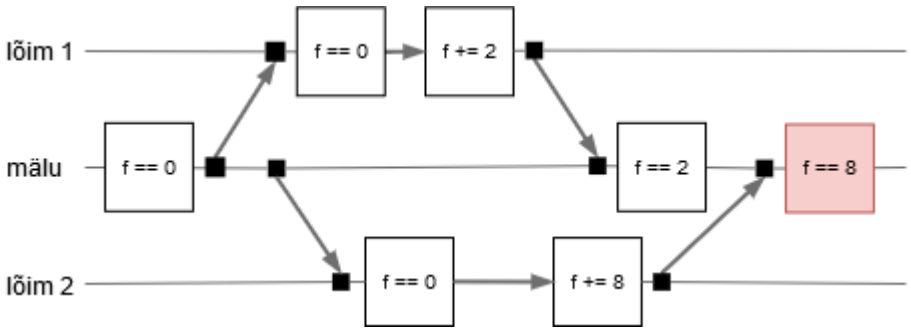
1. Ülesande püstitus ja taust

1.1 Andmejooks ja sünkroniseerimine

Paljusid lõimtöötlusega seotud vigu on raske avastada, sest lõimede käitumine ei ole determineeritud – need võivad esineda mitmete sündmuste harva esinevate järjestuste tulemusena [4]. Eriti keeruline on avastada andmejookse, kus viga muudetud andmetes võib välja tulla alles hiljem [4]. Tavaliste testidega on andmejookse raske leida, sest andmejooksust tulenevad probleemid tulevad välja enamasti vaid harvade juhuste kokkulangemise tõttu ning on raskesti korratavad [5]. Sellises olukorras on kasu programmi analüüsi vahenditest.

Laias plaanis jaotuvad programmide analüüsi meetodid kaheks – staatiline koodi analüüs ja dünaamiline programmi analüüs. Staatilise koodi analüüsi vahenditega analüüsitakse programmi koodi. Seda kasutatakse näiteks programmide optimeerimiseks, korrektsuse kontrolliks ja andmejooksude leidmiseks [6]. Mitmetes koodiredaktorites kasutatakse staatilise koodi analüüsi vahendeid programmeerimise lihtsustamiseks, näiteks süntaksi ja tüübivigade välja toomiseks ning koodistiili parandamiseks. Dünaamiline programmi analüüs uurib konkreetset kompileeritud programmi. Sellega saab leida vigu, mis tekivad programmi töötamise ajal. Näiteks saab dünaamilise analüüsi abil jälgida mälukasutust või teha jõudlusteste. Dünaamilise analüüsiga ei saa kinnitada, et mingit viga ei leidu, sest vigade esile tulemuseks on vaja, et jälgimisel olevas programmis analüüsi käigus juhtuks see olukord, milles viga esile tuleb.

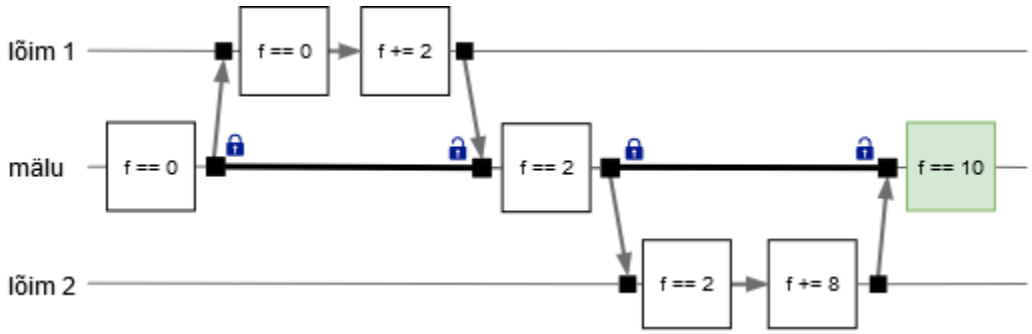
Käesolevas töös leitakse staatilist analüüsi kasutades andmejooksusid. Joonisel 2 on illustreeriv näide, kus kaks lõime muudavad muutujat *f*. Mõlemal lõimel on lihtne ülesanne – võtta mälust *f*-i väärtus, liita sellele oma number ja tulemus muutujasse tagasi salvestada. Joonisel on kujutatud lõimede võimalikku ajastust. Muutuja *f* väärtus on mõlema lõime pärimise ajahetkel 0. Lõim 1 liidab *f*-i algsele väärtusele 2 ja salvestab tulemuse mällu tagasi. Samal ajal liidab teine lõim algsele väärtusele 8. Kuna teine lõim ei teadnud midagi sellest, et esimene lõim on vahepeal *f*-i muutnud siis kirjutab see esimese lõime väärtuse



Joonis 2. Andmejooks

üle. Programmi autori eesmärgiks oli, et esialgsele väärtusele liidetaks kaks ja kaheksa, mille tulemuseks oleks kümme. Antud näite puhul oli lõpptulemuseks kaheks aga lõimede erinevate ajastuste puhul võib *f*-i lõppväärtus olla ka 2 või 10.

Selleks, et vältida selliseid andmejooksudest tulenevaid vigu tuleks sünkroniseerida koodi osa, mis küsib mälust muutuja väärtuse ja salvestab selle tagasi. Joonis 3 on lihtsustatud näide sünkroniseerimisest. Selleks ajaks kui lõim 1 küsib *f*-i väärtust, seda muudab ja see tagasi salvestatakse, on teine lõim blokeeritud seda muutujat kasutama. Kui lõim 1 on oma tegevuse lõpetanud, saab lõim 2 *f*-i väärtust pärida, mis selleks hetkeks on 2, liidab sellele 8 salvestab tulemuse. Nii ei kirjuta teine lõim esimese lõime muudatust üle, vaid võtab esimese lõime muudatuse arvesse. Selle näite puhul, kus on vaja ainult neid kahte ülesannet täita, pole lõimtöötluse sisse toomisest kasu. Suuremate programmide puhul saavad lõimed üheaegselt täita koodi osasid, mis sünkroniseerimist ei vaja ja seega võimaldaks lõimetöötlus programmil kiiremini töötada.



Joonis 3. Sünkroniseerimine

Staatilise koodi analüüsiga on võimalik leida neid kohti, kus kirjutatakse või loetakse mälust mingit muutujat. Selle info kaudu on võimalik tuvastada andmejooksu kohad, mis vajavad sünkroniseerimist. Sünkroniseerimine toimub koodi plokkide kaupa. Sünkroniseeritud koodi plokile määratakse monitor, kui lõim hakkab seda koodi täitma, siis seatakse monitorile eelnevalt lukk. Kui sellele monitorile on mõne teise lõime poolt juba lukk määratud, siis selle koodiosa täitmiseks peab ootama kuni monitor lukust vabastatakse. Luku vabastamine toimub peale sünkroniseeritud koodi ploki täitmist [2] [7].

Java koodis võiks see välja näha selliselt, et objekti välja *f* väärtusele liitmiseks on meetod, mida kõik lõimed kasutavad ja mis on sünkroniseeritud (Joonis 4). Siin peaks tagatud olema, et kuskilt mujalt *f*-i lugeda ega kirjutada ei saa. Kui *f*-i saab lugeda ja kirjutada ka

```
public void liitmine(int a) {
    synchronized (l) {
        f = f + a;
    }
}
```

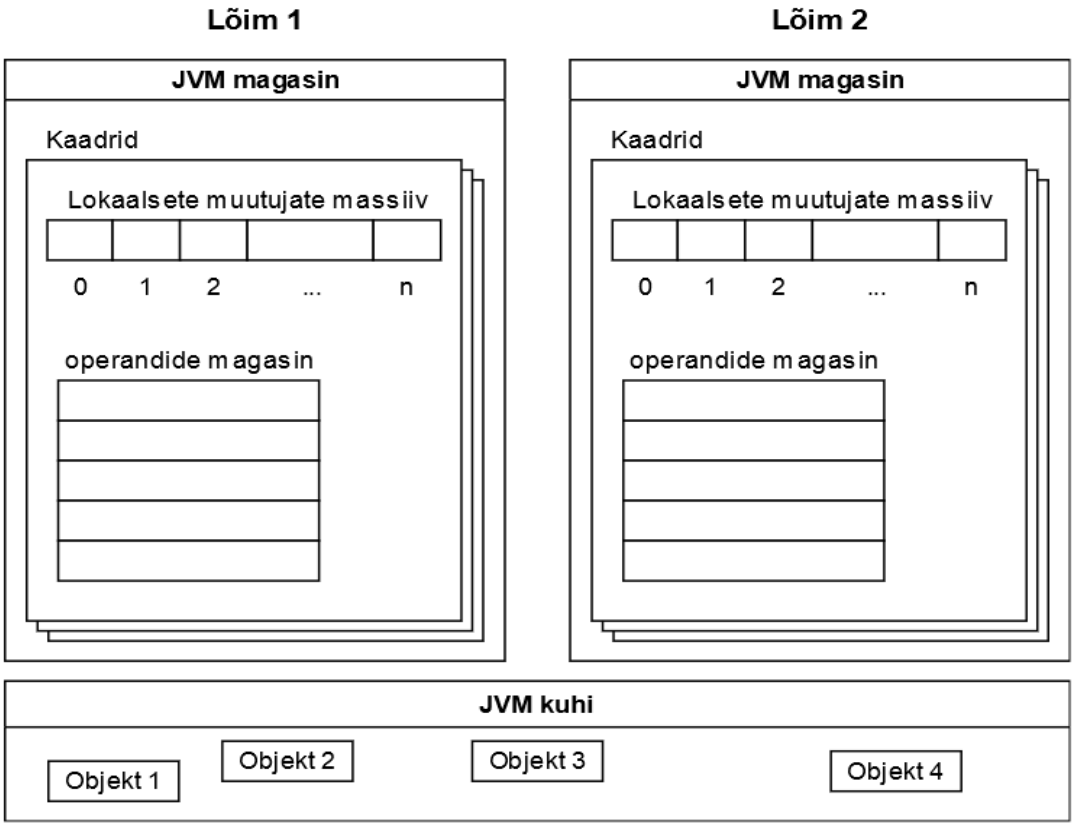
Joonis 4. Meetod *liitmine* sünkroniseeritud koodiplokiga

mujalt peavad kõik need kohad olema sünkroniseeritud sama monitoriga tagamaks, et samal ajal ei saa muutujale mitu lõime ligi ja andmejooksu ei tekiks.

Võimalike andmejooksude leidmiseks on vaja tuvastada koodis kohad, kus erinevatest lõimedest kasutatakse sama mälu aadressi. Andmejooksu definitsiooni järgi peab üks neist kasutuskohtadest olema kirjutamine. Seejärel tuleb kontrollida, kas neid kohti koodis saab samal ajal käivitada mitu lõime korraga. Selleks on vaja kontrollida, kas need koodiplokid on õigesti sünkroniseeritud või mitte. Käesolevas töös tehakse lihtsustav eeldus, et koodi saab kutsuda erinevatest lõimedest ja mitu korda.

1.2 Java virtuaalmasin

Java ja ka mitmed teised keeled kompileeritakse Java baitkoodiks. Java virtuaalmasin (JVM) interpreteerib selle omakorda platvormi jaoks arusaadavasse masinkeelde. Java virtuaalmasina teostusi on erinevaid, aga need peavad vastama kindlatele reeglitele, mis on kirjas Java virtuaalmasina spetsifikatsioonis [7]. Java baitkoodiks kompileerimisel tekivad *.class-failid, mis sisaldavad baitkoodi ja mida Java virtuaalmasin oskab lugeda. JVM alustab programmi käivitamist kindla klassi *main* meetodist [8]. Millist baitkoodi käsku meetodis parajasti täidetakse, jälgitakse käsuaadresside registris (*PC register*). Peale iga käsu täitmist uuendatakse käsuaadresside registrit, kuni kogu programm on täidetud. Iga lõime jaoks on eraldi käsuaadresside register [7].



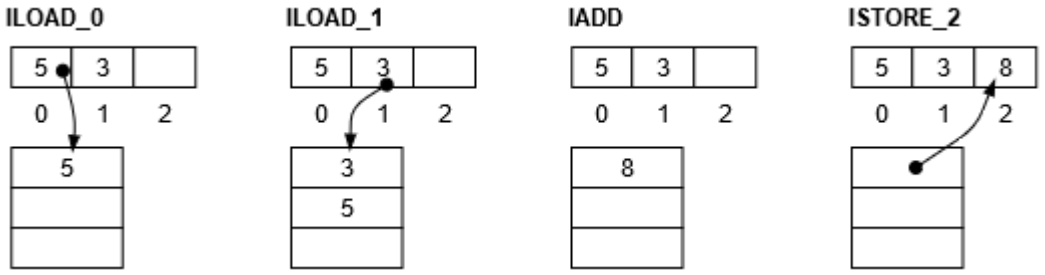
Joonis 5. JVM magasin ja kuhi

Igal lõimel on oma JVM magasin (*stack*), kus lisatakse iga meetodi väljakutsega uus kaader (joonis 5), mis hoiab endas meetodi käskude täitmiseks vajaminevat informatsiooni. Selleks, et järge pidada, milliseid muutujaid meetodi sees kasutatakse hoitakse neid selle kaadri lokaalsete muutujate massiivis (*local variable array*). Meetodi väljakutsel salvestatakse meetodi parameetrid lokaalsete muutujate massiivi alates indeksist 0, kui tegu on klassi meetodiga. Muudel juhtudel salvestatakse viide objektile endale (*this*) ja selle järel meetodi parameetrid. Baitkoodi käsud saavad neile viidata magasini indeksiga. *Long* ja *double* tüüpi väärtused võtavad lokaalsete muutujate massiivis kaks kohta (i ja i+1, kätte saab indeksiga i) [7].

Selleks, et Java baitkoodi käsud saaks lokaalsete muutujatega või muude väärtustega midagi teha kasutatakse operandide magasini (*operand stack*). Lokaalsete muutujate massiivis ja operandide magasinis saab hoida kas lihttüüpe või viiteid objektidele. Objekte hoitakse JVM kuhjas (*heap*). Kuna kuhi on kõikide lõimede jaoks ühine, siis objektid on samuti kõikide lõimede jaoks ühised [7]. See annab võimaluse andmejooksude tekkeks, sest ühte objekti saab muuta mitu lõime, kui neil on viide objektile s.

1.3 Java baitkood

Paljud baitkoodi instruktsioonid opereerivad operandide magasinis olevate andmetega. Näiteks kahe arvu liitmiseks eemaldatakse magasini kaks pealmist numbrit, liidetakse kokku ja tulemus jääb magasini peale [7]. Joonisel 6 on illustreeritud näide kahe arvu liitmisest baitkoodiga. Näidatud on operandide magasini ja lokaalsete muutujate massiivi seis peale iga baitkoodi käsklust. Käsk *iload_0* lisab lokaalsete muutujate massiivist kohalt 0, väärtuse operandide magasini peale. Sama teeb käsk *iload_1* massiivis kohal 1 oleva väärtusega. *iadd* liidab kaks pealmist väärtust kokku ja *istore_2* salvestab magasini peal oleva tulemuse lokaalsete muutujate massiivi.



Joonis 6. Kahe arvu liitmine baitkoodiga. Lokaalsete muutujate massiivi (üleval) ja operandide magasini (all) muutumine peale igat baitkoodi käsku.

Enamus Java baitkoodi käske töötavad kindlat tüüpi väärtustega. Sellele tüübile vihjavad baitkoodi instruktsioonide mnemoonilised nimetused, mille alguses on tüübile vastav täht. *i* tähistab *int* tüüpi, *l - long, s - short, b - byte, c - char, f - float, d - double* ja *a - reference* tüüpi. Seega käsud *iload, lload, fload, dload* kui ka *aload* võtavad kõik lokaalsete muutujate massiivist kindlalt kohalt väärtuse, mis peavad vastama kindlale tüübile, ja laevad selle operandide magasini. Number, mis on peale käsu koodi, tähistab milliselt kohalt lokaalsete muutujate massiivis väärtus võetakse. Mõnele käsu koodile on see

number juba sisse kirjutatud. Näiteks *iload 1*(käsu kood koos operandiga) ja *iload_1* (käsu kood) teevad täpselt sama asja. [7].

JVM spetsifikatsioonis on mitut eri tüüpi instruktsioone:

1. laadimise (*load*) ja salvestamise (*store*) käsud,
2. aritmeetikatehted,
3. tüübiteisendused (*type conversion*),
4. objektide loomine ja nendega manipuleerimine,
5. operandide magasini muutmine,
6. suunamise (*control transfer*) käsud,
7. meetodi väljakutse ja naasmise käsud,
8. erandi viskamise käsud,
9. sünkroniseerimise käsud [7].

Sünkroniseerimise analüüsi kontekstis on oluline, millised neist käskudest mõjutavad lokaalsete muutujate massiivi, operandide magasini ja kasutuses olevate lukkude hulka. Nendeks on (1) laadimise ja salvestamise käsud, mis laevad lokaalsete muutujate massiivist väärtuse operandide magasini või salvestavad operandide magasinist operandi lokaalsete muutujate massiivi. (2) Aritmeetika käsud, mis teevad tehteid operandide magasinil. (4) Objektide loomise ja muutmise käskudest need, mis muudavad lokaalsete muutujate massiivi ja operandide magasini. (5) Operandide magasini muutmise instruktsioonid, mis muudavad magasini seisu. (7) Meetodi väljakutsed, mis eemaldavad operandide magasinist meetodi argumendid ja kui tegu ei ole klassimeetodiga, siis ka objektiviite ning lisavad need uue kaadri massiivi. Naasmise käsud, mis võivad muuta operandide magasini. Meetodi väljakutse ja naasmise käsud on ka otseselt seotud sünkroniseerimisega. Lisaks on veel eraldi (9) sünkroniseerimise käsud [7].

1.4 Sünkroniseerimine Java virtuaalmasinas

Sünkroniseerimisega on seotud peamiselt kaks baitkoodi instruktsiooni *monitorenter* ja *monitorexit* [9]. Iga objekt JVM-is on seotud talle vastava monitoriga. Kui lõim täidab *monitorenter* käsku, siis võetakse operandide magasini pealmine element, mis peab olema viide objektile, määratakse lõim selle objektiga seotud monitori omanikuks ja sellega lukustatakse monitor. Kui mõni teine lõim on juba selle monitori lukustanud, siis see lõim blokeeritakse. Seega ei saa teised lõimed käivitada koodi, mis on kaitstud lukustatud monitoriga enne kui monitor lukust vabaneb ja mõni teine lõim saab ise monitori omanikuks. Sama lõim võib ühte monitori lukustada mitu korda: iga *monitorenter* käsuga suurendatakse luku loendurit. Iga *monitorexit* käskluse täitmisel vähendatakse seda sama loendurit ühe võrra. Kui luku loenduri väärtus jõuab tagasi nullini, vabastatakse monitor [7].

Kui konkreetne monitor on lukustatud, siis ei tohiks ükski teine lõim välja kutsuda koodi lõikusid, mis nõuaks sama monitori lukustamist [9]. Kui sünkroniseeritud on terve meetod, siis ei kasutata *monitorenter* ja *monitorexit* käske, aga üldine põhimõte jääb samaks. Monitor, mida sel juhul kasutatakse on seotud objektiga, millel meetodit välja kutsutakse. Kui meetod kutsutakse välja, siis toimub lukustus ja meetodi naasmislausega lukk vabastatakse. Sellisel juhul on meetodil lipp *acc_synchronized* ja lukustamisel osalevad

käsklused on *invokeinterface*, *invokespecial*, *invokestatic*, *invokevirtual* lukkude vabastamisel osalevad *athrow* ja kõik *return* käsud [7].

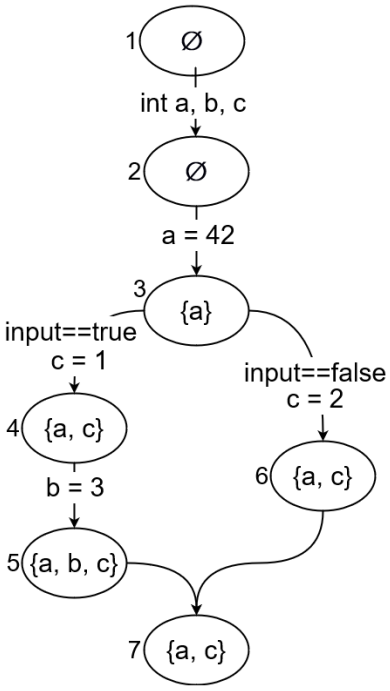
Sünkroniseerimine toimub ka klassi objekti loomise ajal, sellel ajal ei saa ükski teine lõim objekti muuta. See peab olema tagatud Java virtuaalmasina tasandil [8]. Selle tõttu ei ole vaja andmejooksude leidmiseks jälgida milliseid kirjutamise ja lugemise operatsioone objekti loomisel tehakse.

1.5 Juhtvoograaf

Programmi koodi analüüsimiseks on vaja kirjeldada uuritavate omaduste seisu igas vaadeldavas programmi punktis. Selleks kasutatakse juhtvoograafe (*control-flow graph*). Juhtvoograaf on suunatud graaf, mille servadeks on programmi käsud ja tipud tähistavad programmi seisu peale käsku. Joonisel 7 on toodud näidisprogramm ja selle programmi juhtvoograaf, mille külgedeks on programmi käsklused.

Juhtvoograafide paremaks seletamiseks teeme siinkohal läbi lihtsa näite, kus analüüsime seda, mis muutujad on algväärtustatud. Tippudeks on programmi olek, mis kirjeldab kindlasti väärtustatud muutujate hulka peale programmi käsklusi. Siin võib ära märkida, et antud analüüsis ei ole oluline, mis väärtus muutujatele antud on ja sellepärast ei kajastu see ka seisus.

```
int a, b, c;
a = 42;
if (input) {
    c = 1;
    b = 3;
} else {
    c = 2;
}
```

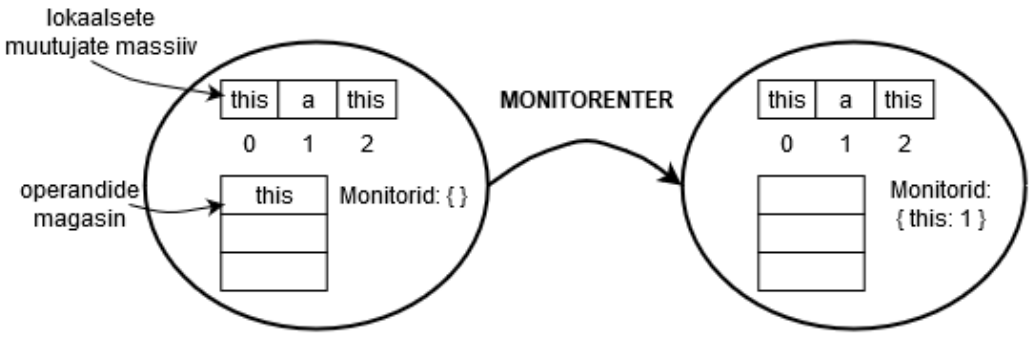


Joonis 7. Näiteprogrammi kood ja selle juhtvoograaf

Joonisel oleval juhtvoograafil on märgitud seitse tippu, milles on välja toodud programmi seis. Programmi analüüsimiseks on ainult kood ja need seisud tuleb kõigepealt leida. Järgnevas loetelus on kirjeldus, kuidas iga tipu väärtus on leitud.

- Programmi algseisuks tipus 1 on tühi hulk.
- Iga järgmise seisu leidmiseks peab vaatama, millised muutujad on vahepealsete käskudega väärtustatud. Kuna muutujate deklareerimise ajal muutujaid ei väärtustatud, siis jääb tipu 2 seis samaks.
- Enne tippu 3 väärtustatakse muutuja a. Seega uus seis on {a}.
- Kuigi tipud 4 ja 6 on tingimuslause eri harudes, siis neile mõlemale eelneb muutuja c väärtustamine ja c lisandub väärtustatud muutujate hulka, mis on mõlemas harus {a, c}.
- Enne tippu 5 väärtustatakse muutuja b, uueks seisuks on {a, b, c}
- Kuna nüüd väljub programm tingimuslausest, siis tipu 7 seisu leidmiseks on vaja ühendada tingimuslausete harude seisud. Kuna ühes harus väärtustati muutuja b ja teises mitte, siis peale harude ühendamist ei ole teada, kas b on väärtustatud või mitte. Kindlalt on teada, et väärtustatud on muutujad a ja c. Järelikult lõppseisuks on {a, c}.

Praktikas võivad analüüsiks vajalikud olekud tihti olla oluliselt keerulisemad kui lihtsad hulgad. Java baitkoodi sünkroniseerimise analüüsi eesmärk käesolevas töös on andmejooksude avastamine. Selleks on vaja leida programmis objekti kirjutamise ja lugemise kohad ning millised monitorid nendes kohtades kasutusel on. Monitoride jälgimiseks on muuhulgas vaja implementeerida *monitorenter* ja *monitorexit* käskude üleminekufunktsioonid. *monitorenter* käsk võtab operandide magasini pealmise elemendi, mis peab olema viide objektile. Selle objektiga seotud monitorile seatakse lukk. Joonisel 8 on kujutatud seis enne ja pärast *monitorenter* käsku. Lukk pannakse objektiga *this* seotud monitorile. *this* tähistab objekti, millel meetod välja kutsuti. Samasuguse muudatuse peab ka üleminekufunktsioon tegema.



Joonis 8. Programmi seis enne ja pärast *monitorenter* käsku

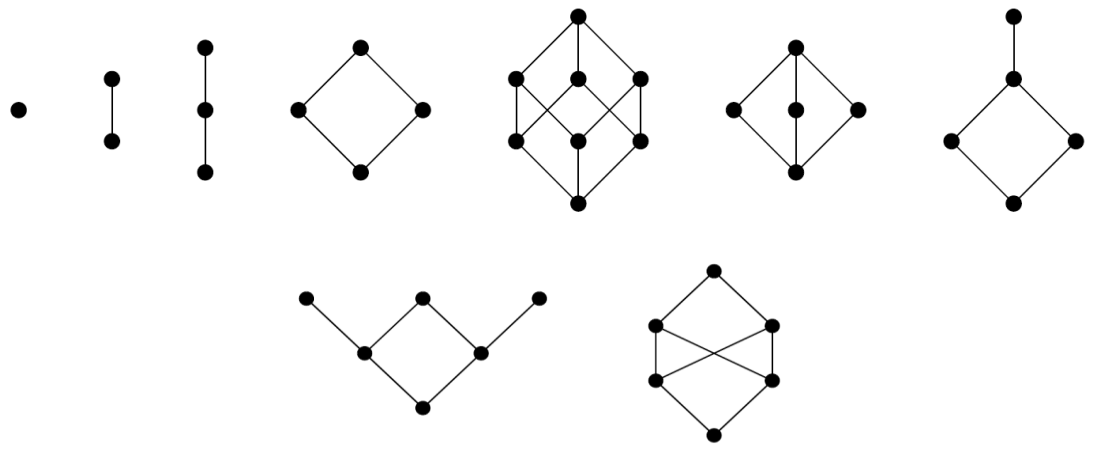
monitorenter üleminekufunktsiooni jaoks oli vaja teada, mis objekt operandide magasini peal on. Seega pidi seisu implementatsioon seda kajastama. Kuna operandide magasini ja lokaalsete muutujate massiivi vahel liigutatakse objekte ringi, pidi seis kajastama ka lokaalsete muutujate massiivi. *monitorenter* käskluse ülemineku funktsioon pidi uue seisu jaoks eemaldama operandide magasinist pealmise objekti (joonisel *this*) ja kontrollima kas see esineb monitoride hulgas. Kui ei esinenud, siis lisatakse see monitoride hulka ja lukkude arvuks määratakse 1. Kui see objekt juba oli monitoride hulgas, siis suurendatakse monitori lukkude arvu ühe võrra.

Üldistades on vaja seisude kirjeldamiseks struktuuri, mille puhul on teada programmi algseisund, millega saab kirjeldada programmi käsule vastavalt muutunud seisundit ja mille puhul saab leida programmi harude ühinemiskohas seisu, mis oleks õige iga haru puhul. Üks hea mudel selliste olukordade kirjeldamiseks on täielik võre (*complete lattice*).

1.6 Võred

Programmi efektiivseks analüüsiks on erinevate võimalike seisundite hulk enamasti liiga suur. Näiteks tsükli puhul nullist 10 000-ni on tsükliloenduri väärtus hulgast {0, 1, ..., 10000}. Nii suuri hulki me analüüsis enamasti käsitleda ei taha. Parem on kasutada väiksemaid hulki – abstraktseid väärtusi, millega saab edasi kanda ainult seda, mis on konkreetse analüüsi jaoks vajalik. Jätkates eelmist näidet, võime hulga asemel kasutada intervalli [0, 10000]. Intervallid on kasutuses analüüsimaks massiivide indekseerimist. Lisaks, kuna peame lubama informatsiooni kadu, on sobilik vaadata programmi seisundeid kui matemaatilist struktuuri nimega täielik võre.

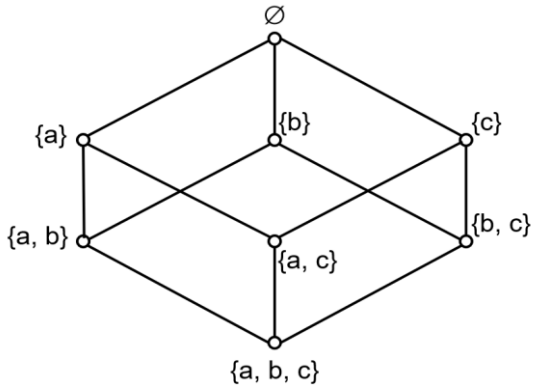
Võre ja teiste osaliste järjestuste kujutamiseks kasutatakse Hasse diagramme, kus kaared tähistavad tippude vahelisi järjestusseoseid järjestatud ülevalt alla. See tähendab, et kaarega ühendatud tippudest kõrgemal olev on suurem ja allpool olev väiksem. Joonisel 9 on toodud näited Hasse diagrammidest, kus ülemises reas on näiteid osalistest järjestustest, mis on võred ja alumises reas on näited, mis ei ole võred. Etteruttavalt öeldes, alumises reas ei ole tegu võredega kuna neis leiduv tippude alamhulk, millel ei ole ülemist raja [6].



Joonis 9. Hasse diagrammid. Ülemises reas on järjestused, mis on ühtlasi ka võred. Alumises reas on järjestused, mis ei ole võred [6]

Kui tahta programmi puhul analüüsida kasutuses olevate monitoride hulka saab lihtsustatud näiteks võtta olukorra kus analüüsitavas programmis on kolm monitori: a, b ja c. Sellisel juhul näitab iga võre element erinevaid võimalusi, millised nendest monitoridest on kasutuses. See on hulga {a, b, c} kõigi alamhulkade hulgal määratud sisalduvusrelatsiooni võre, mille Hasse diagramm on toodud joonisel 10. Programmi alguses oleks kasutuses olevate monitoride hulk tühi hulk. Edasi näiteks võetakse kasutusele monitor a, siis oleks uus seis {a}. Kui järgmisena võetakse kasutusele monitor

b, siis oleks seis {a, b}. Kui ei ole teada kas võeti kasutusele monitor b või c, siis tuleks määrata uueks seisuks {a}, kuna see kehtib nii seisu {a, b} ja {a, c} puhul ja on ühtlasi ka nende kahe seisu ülemine raja.



Joonis 10. Hulga {a, b, c} kõigi alamhulkade hulgal määratud sisalduvusrelatsioon

Diagrammil mööda servi, erinevaid teid pidi alt üles liikudes saab kätte alamhulkade hulga järjestused väikseimast elemendist suurimani. Üks selline järjestus on {a, b, c} ⊑ {a, b} ⊑ {a} ⊑ ∅. Seega antud juhul on suurim element tühi hulk. Kuigi alamhulgad {a} ja {b} on mõlemad suuremad kui alamhulk {a, b}, siis {a} ja {b} omavahel võrreldavad ei ole.

Järgnevalt anname täieliku võre formaalse definitsiooni. Olgu P mittetühi hulk. Seost ⊑ hulgal P nimetatakse **osalise järjestuse seoseks** kui ⊑ on:

1. refleksiivne, $\forall x \in P: x \sqsubseteq x$
2. antisümmeetriline, $\forall x,y \in P: x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$
3. transitiivne. $\forall x,y,z \in P: x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$

Osaliselt järjestatud hulk on hulk, millel on antud osalise järjestuse seos [10].

Olgu $X \subseteq P$, *siis* $y \in P$ on hulga X **ülemine tõke** (*upper bound*) kui *iga* $x \in X$ *puhul kehtib* $x \sqsubseteq y$. Sarnaselt on $y \in P$ hulga X **alumine tõke** (*lower bound*) kui *iga* $x \in X$ *puhul kehtib* $y \sqsubseteq x$ [6].

Hulga ülemiseks rajaks (*join, the least upper bound*) ($\sqcup X$) nimetatakse selle hulga vähimat ülemist tõket ja alumiseks rajaks (*meet, the greatest lower bound*) ($\sqcap X$) selle hulga suurimat alumist tõket [11].

Täielik võre on osaliselt järjestatud hulk, mille igal alamhulgal on olemas ülemine ja alumine raja. Igal täielikul võrel on suurim element $\top = \sqcup P$ ja vähim element $\bot = \sqcup \emptyset$ [11].

Programmianalüüsis vastavad võre elemendid väidetele programmi seisu kohta ning järjestuseks võetakse nende väidete implikatsioon. Kui võre element $x \sqsubseteq y$, siis see tähendab, et x on vähemalt sama täpne, kui y. Ehk kui kehtib x, siis kehtib ka y [6]. Võre suurimat elementi (top, ⊤) kasutatakse kujutamaks seisu, kus me programmi kohta midagi ei tea.

Erinevatel võre omadustel on programmi analüüsis oma kasutus. Suuremad elemendid on üldisemad kui väiksemad elemendid. Eelneva monitoride näite puhul, kui kasutuses monitoride hulk on {a, b}, siis kehtib ka väide, et kasutuses on monitoride hulk {a}. Viimane väide on ebatäpsem. Võre vähimat elementi kasutatakse oleku jaoks, kuhu programm kunagi jõudma ei peaks. Joonisel 10 olevale võrele tuleks sellise elemendi jaoks lisada veel üks vähim väärtus, kuna on võimalik, et võetud on kõik monitorid. Võre suurimat elementi kasutatakse olukorra jaoks, kus meil ei ole programmi seisundi kohta informatsiooni. Kui pole teada, kas ükski monitor on võetud või mitte, siis mõlema olukorra kirjeldamiseke sobib tühi hulk. Võre ülemist raja kasutatakse programmi erinevate seisude ühendamiseks niteks peale *if* või *case* lauseid [6].

1.7 Üleminekufunktsioonid ja võrrandisüsteem

Staatilise analüüsi jaoks on vaja defineerida abstraktne üleminekufunktsioon iga lause liigi kohta $[\![\ . \ ]\!]^{\#} : P \rightarrow P$. Joonisel 7 oli toodud juhtvoograaf. Selle põhjal saab defineerida järgmised üleminekufunktsioonid:

$$p_{start} = \emptyset$$
$$[\![\text{ int } v, ..., v_n;]\!]^{\#} s = s$$
$$[\![v = e;]\!]^{\#} s = s \cup \{ v \}$$
$$[\![e == \text{true}]\!]^{\#} s = [\![e == \text{false}]\!]^{\#} s = s$$

Esimene funktsioon ütleb, et algseis on tühi hulk. Järgmine, et deklareerimine ei muuda seisundit. Muutuja omistamise puhul võetakse uue seisu saamiseks eelmine seis ja lisatakse sinna muutuja. Viimane funktsioon ütleb, et hargnemise valvur ei muuda seisu.

Nüüd saab juhtvograafi ja üleminekufunktsioone kasutades defineerida võrrandisüsteemi, mille järgi lahendus leida.

$$① \sqsupseteq p_{start}$$
$$② \sqsupseteq [\![\text{ int a, b, c; }]\!]^{\#} ①$$
$$③ \sqsupseteq [\![a = 42;]\!]^{\#} ②$$
$$④ \sqsupseteq [\![c = 1;]\!]^{\#} ([\![\text{input} == \text{true};]\!]^{\#} ③)$$
$$⑤ \sqsupseteq [\![b = 3;]\!]^{\#} ④$$
$$⑥ \sqsupseteq [\![c = 2;]\!]^{\#} ([\![\text{input} == \text{false};]\!]^{\#} ③)$$
$$⑦ \sqsupseteq ⑤ \sqcup ⑥$$

Seis 1 on algseis, mis on tühi hulk. Seis 2 saamiseks peab vaatama kuidas üleminekufunktsioon $[\![\text{ int a, b, c; }]\!]^{\#}$ eelmist seisu mõjutab. Kui vaadata üleminekufunktsioonide definitsioone ülevalpool, siis näeme, et antud juhul jääb seis samaks ehk on

tühi hulk. Iga järgneva seisu leidmiseks peab vaatama vahetult eelnevat seisu ja kuidas üleminekufunktsioon seda muudab. Seis 3 puhul peab vaatama omistamise üleminekufunktsiooni. Kuna eelmine seis oli tühi hulk ja väärtustatud muutuja on a, siis uus seis on ∅∪{ a } ehk { a }. Seis kolmel on kaks vahetult järgnevat seost. Kõigepealt seis 4, millele eelneb kaks üleminekufunktsiooni – hargnemise valvur, mis jätab seisu samaks ja siis omistamise funktsioon, millega uude seisu lisandub c ja tulemus on { a, c }. Seis 5-e saamiseks lisandub eelmisele seisule muutuja b, ning. Seis 6 järgneb samuti vahetult seisule 3. Väärtused on natuke erinevad, aga üleminekufunktsioonid ja väärtustatud muutuja on samad. Seega on seis 6 täpselt sama, mis seis 4. Viimase seisu leidmiseks ühendatakse seis 5 ja 6, mis mõlemad on selle seisu vahetud eellased ja tulemuseks saab { a, c } Kokkuvõtvalt on lahendus järgmine:

$$① = ② = ∅; ③ = \{ a \}; ④ = ⑥ = ⑦ = \{ a, c \}; ⑤ = \{ a, b, c \}.$$

1.8 Raamistik Põder

Põder on staatilise analüüsi raamistik, mis analüüsib Java baitkoodi. Iga Põdra analüüsi sisendiks on Java baitkoodi .class fail ja väljundiks on programmi juhtvoograaf, mille kõrval on võimalik näha analüüsi domeeni seisu erinevates programmipunktides. Joonisel 11 on välja toodud meetodid, mis on vajalikud võre struktuuri implementeerimiseks (üleval) ja olulisemad üleminekufunktisioonid (all). Implementeeritav võre sõltub sellest, mida tahetakse analüüsida ja ülemineku funktsioonid peavad lisaks arvestama ka baitkoodi instruktsioonidega [12].

```
trait Lattice[T] {
  def top: T
  def bot: T
  def join(x:T, y:T): T
  def leq(x:T, y:T): Boolean
  def toJson[jv](x: T): jv
  def widen(oldv: T, newv:T): T
}

def sNode
def sStart
def sNormal
def senter
def scombine
def postProcess
```

Joonis 11. Implementeeritavad meetodid

Võre meetodites peab *top* meetod tagastama võre suurima elemendi ja *bot* väikseima. Olukordades, kus analüüs ei tea programmi seisu kohta midagi, kasutatakse *top* väärtust ja kohtades, kuhu programm kunagi jõudma ei peaks, kasutatakse *bot* väärtust. Meetodit *join* kasutatakse olukordades, kus on vaja mitut programmi seisu ühendada, näiteks peale *if*-lauset. Kahe võre elemendi võrdlemiseks on meetod *leq*. Kasutajaliideses võre elemendi info kuvamiseks tuleb implementeerida *toJson* meetod. Kui programmis on 1000

iteratsioon, siis on parem kui analüüs ei pea seda 1000 korda üle käima. Iteratsioonilausete efektiivsemaks analüüsiks on meetod *widen.*

Üleminekufunktsioonid peavad eelmisest seisust saama uue seisu, *sNode* peab tagastama tipu algoleku. Meetodikutsete jaoks on eraldi *sStart, senter ja scombine*. Enne meetodi sisenemist teeb vajaminevad oleku muudatused *senter* ning seejärel leiab *sStart* vastava meetodi algseisu. Meetodist väljumisel kasutatakse *scombine*-i, et ühildada meetodi lõppseis ja seis, mis oli enne meetodi väljakutsumist. See ei ole sama kui *join*. Java virtuaalmasina peatükis oli kirjeldatud kuidas iga meetodi väljakutsega luuakse uus raam uute lokaalsete muutujate massiiviga ja operandide magasiniga, mis kaovad peale meetodit väljumist. Seega neid seisusid ei saa *join*-i abil kokku viia.

Antud töö raames on lisatud sünkroniseerimise analüüsi juhtvoograafi ja seisude juurde lisatud andmejooksude raport, mille järgi oleks kasutajal lihtsam leida nende tekkekohti.

2. Sünkroniseerimise analüüsi teostus

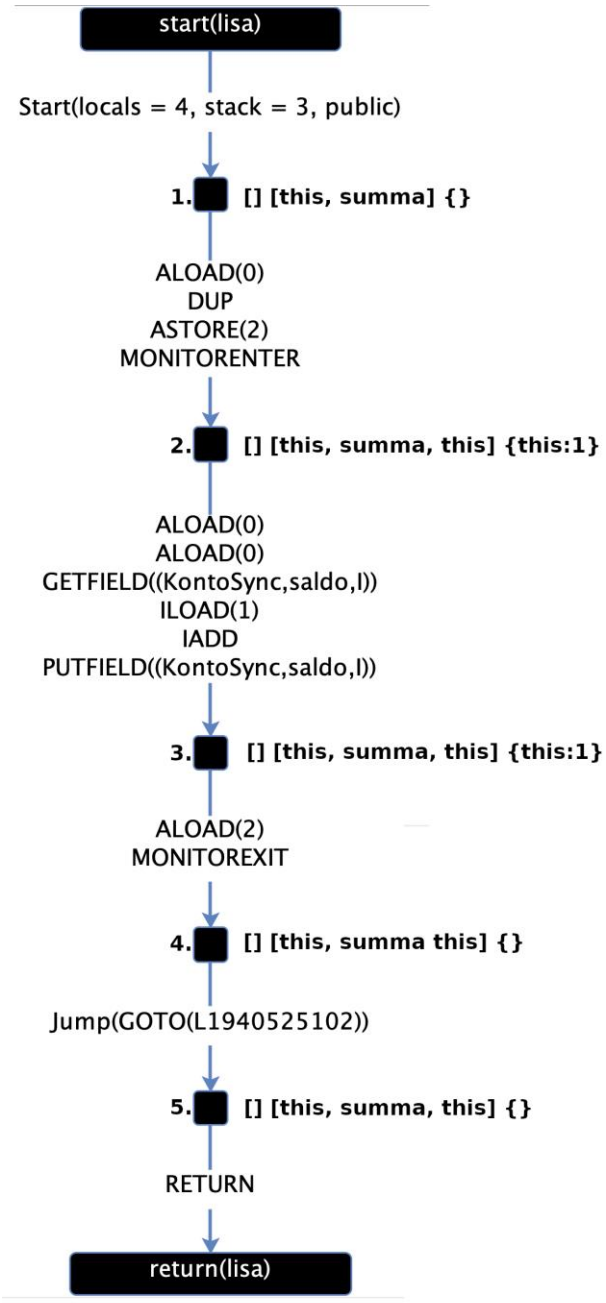
2.1 Programmi analüüs sünkroniseerimise mooduliga

Programmis Põder, saab vaadata analüüsitavate meetodite juhtvoograafe. Joonisel 13 on näha meetod *lisa* ja joonisel 13 selle kohta kuvatav juhtvoograaf. Mustad ruudud on juhtvoograafi tipud ja nende kõrval tipule vastav seis. Esimeste kandiliste sulgude vahel on operandide magasini elemendid, teiste kandiliste sulgude vahel on lokaalsete muutujate massiiv ja loogeliste sulgude vahel on näha lukustatud monitorid. Graafi servadel on toodud baitkoodi käsklused. Ülevaatlikkuse eesmärgil on neid ühe serva peal mitu, aga tegelikkuses on iga baitkoodi käsu jaoks olemas eraldi üleminekufunktsioon. Joonisel on nummerdatud tipud 1-5. Järgnevalt on kirjeldatud märgitud seise ja kuidas üleminekufunktsioonid neid muudavad.

```
public void lisa(int summa) {
    synchronized (this) {
        saldo = saldo + summa;
    }
}
```

Joonis 12. Meetod *lisa*

1. On meetodi algseis. Operandide magasin on algseisus alati tühi. Lokaalsete muutujate massiivis on viide objektile, mille peal meetod välja kutsuti ja meetodi parameetrid, antud juhul [*this, summa*]. Monitoride multihulk on tühi kuna meetod ise ei ole sünkroniseeritud ja ka meetodi väljakutsumise kohas ei olnud ühtegi monitori kasutuses. Enne järgmist seisu on märgitud mitu baitkoodi. Nende üleminekufunktsioonid teevad järgmist:
    a. *aload(0)* lisab operandide magasini peale lokaalsete muutujate massiivist kohalt 0 *this*-i.
    b. *dup* teeb magasini pealmisest elemendist koopia, magasinis on peale seda [*this, this*].
    c. *astore(2)* eemaldab magasinist pealmise elemendi ja salvestab lokaalsete muutujate massiivi kohale 2. Massiivis on nüüd [*this, summa, this*].
    d. *monitorenter* võtab magasinist pealmise elemendi, lisab selle monitoride hulka.
2. Tipus on operandide magasin tühi, lokaalsete muutujate massiivis on [*this, summa, this*] ja kasutusel on monitor *this*, mis on ühekordselt lukustatud. Järgnevad üleminekufunktsioonid:
    a. *aload(0)* on kaks korda järjest. Mõlemal korral laetakse *this* magasini peale.
    b. *getfield((KontoSync, saldo, I))* võtab magasini pealt elemendi viite ja lisab sinna selle objekti välja *saldo.*
    c. *iload(1)* laeb lokaalsete muutujate massiivi kohalt 1 *summa* ja paneb selle magasini peale.



Joonis 13. Meetod *lisa* juhtvoograaf programmis Põder

d. *iadd* liidab magasini kaks pealmist täisarvu ja tulemus jääb magasini peale. Analüüsi vaatenurgast on see tundmatu väärtus ja tähistatakse küsimärgiga.
e. *putfield((KontoSync, saldo, I))* eemaldab magasinist kaks pealmist elementi.

3. Peale eelnevaid üleminekufunktsioonide täitmist on see tipp samas seisu nagu tipp
a. *aload(2)* laeb lokaalsetest muutujatest *this*-i magasini peale.

   b. *monitorexit* magasini pealmine element eemaldatakse ja monitoride hulgast eemaldatakse *this* monitor
4. Siin tipus on magasin ja monitoride hulk tühi ning lokaalsete muutujate massiivis on [*this*, *summa*, *this*].
5. *goto* ei muuda olekus midagi ja seega on tipu 5 seis sama, mis tipu 4 seis.

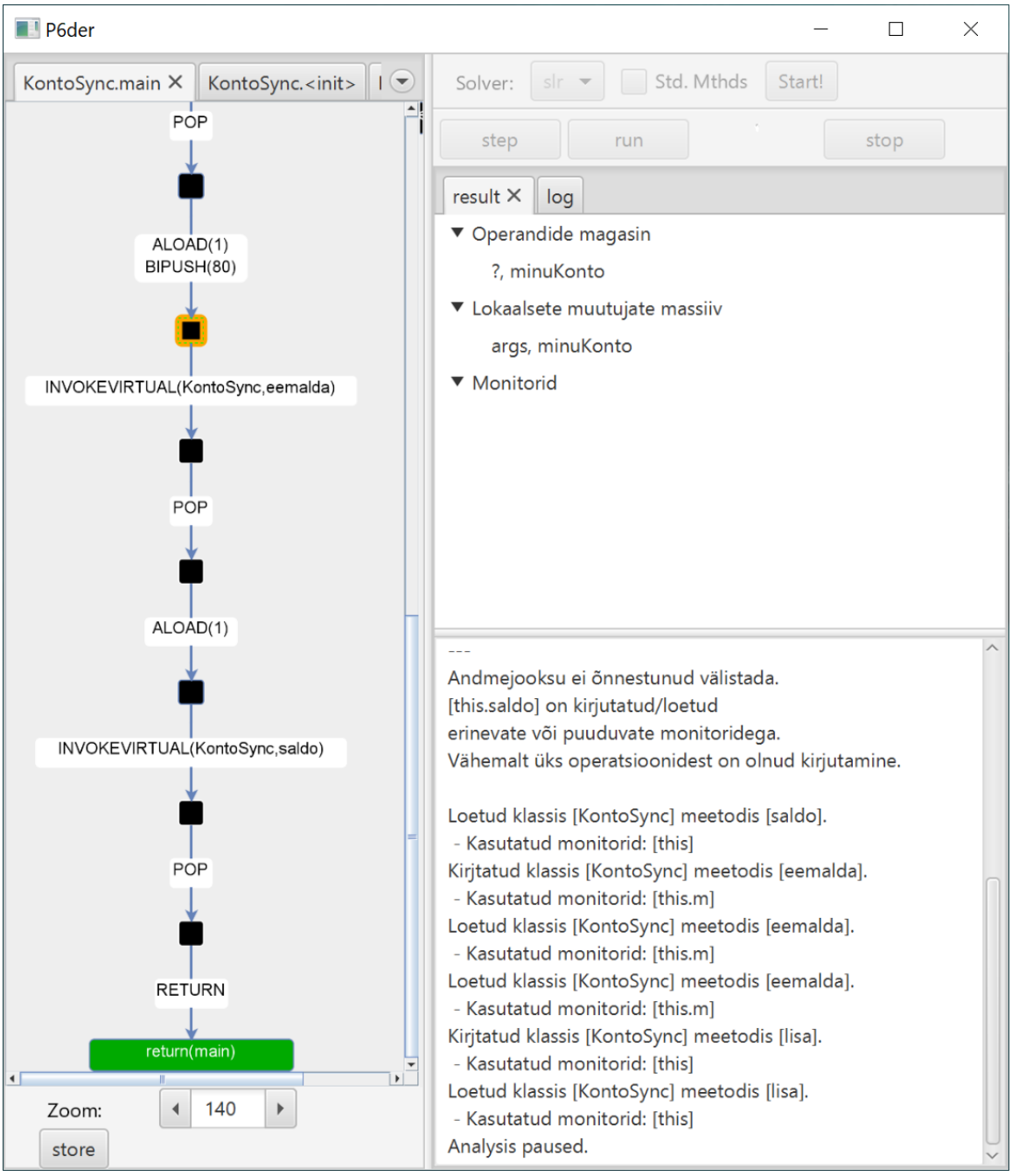
Kõiki neid tippude olekuid saab programmis eraldi vaadata.

Analüüsi teiseks väljundiks on andmejooksude raport. Selle jaoks salvestatakse vajalik info *putfield* ja *getfield* üleminekufunktsioonide käigus. Kui analüüs on oma töö lõpetanud tehakse vaadatakse iga muutuja kõik kirjutamise ja lugemise kohad üles ning kontrollitakse ega nendes kohtades kasutusel olnud monitoride hulkade ühisosa ei ole tühi hulk. Joonisel 14 on toodud klass KontoSync ja joonisel 15 näidatud selle klassi sünkroniseerimisest kuvatõmmis Põdras. Paremal all on välja toodud, et andmejooksu ei õnnestunud välistada muutuja *saldo* puhul. Välja on toodud kõik kohad programmis, kus sellesse muutujasse on kirjutatud või loetud.

Raportis ei tooda välja *init* meetodis toimuvad kirjutamise ja lugemise operatsioone. Initsialiseerimise ajal pole teistel lõimedel viita objektile ja seetõttu ei saa samal ajal teised sinna kirjutada.

```
public class KontoSync {
    private Object m = new Object();
    private int saldo = 0;
    public static void main(String[] args) {
        KontoSync minuKonto = new KontoSync();
        minuKonto.lisa(99);
        minuKonto.eemalda(20);
        minuKonto.eemalda(80);
        minuKonto.saldo();
    }
    public int eemalda(int summa) {
        synchronized (m) {
            if (saldo < summa) {
                return 0;
            }
            saldo = saldo - summa;
        }
        return summa;
    }
    public void lisa(int summa) {
        synchronized (this) {
            saldo = saldo + summa;
        }
    }
    public synchronized int saldo() {
        return saldo;
    }
}
```

Joonis 14. KontoSync klass

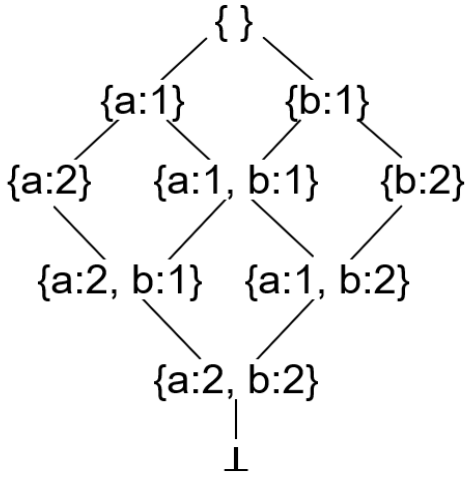


Joonis 15. Põder kuvatõmmis

2.2 Analüüsitava programmi olek

Programmi oleku kirjeldamiseks oli vaja luua programmi analüüsimiseks sobiv seisundi kirjeldus ja võre. Selle probleemi lahendamist saab vaadelda monitoride, lokaalsete muutujate massiivi ja operandide magasini jaoks eraldi ja pärast lahendused ühendada.

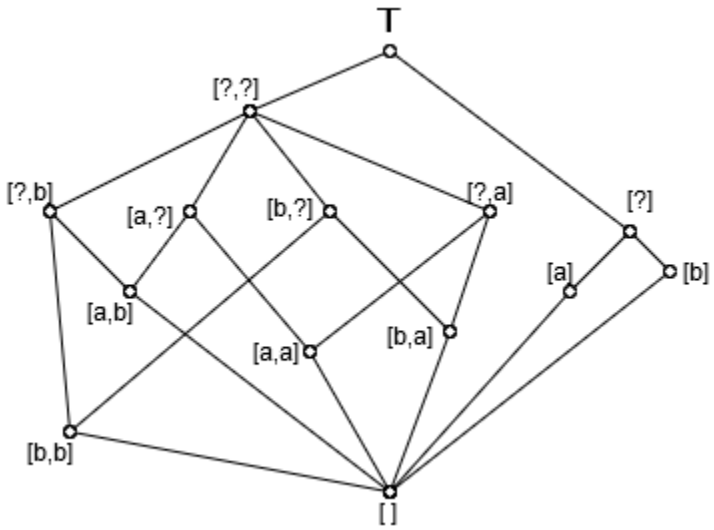
Monitoride puhul on oluline, mitu lukku monitorile on seatud. Kuna nende järjekord ei ole oluline, siis sobib monitoride olekuks multihulk. Võre kirjeldamiseks tuli paika panna järjestusseosed. Kui *{a: 1}* tähistab seda, et monitor a on lukustatud ühe korra, siis *{a: 2}* on väiksem kui *{a: 1}* (*{a: 2}* ⊑ *{a: 1}*) ning kehtib ka *{a: 1, b:1}* ⊑ *{a: 1}*. Seega kui pole teada kas monitor *a* on lukustatud 1 või 2 korda ja pole kindel kas monitor *b* on lukustatud või mitte, siis kehtib väide, et monitor *a* on ühe korra kindlasti lukustatud. Joonisel 16 on näha võre diagramm, kui ainsad monitorid oleks *a* ja *b*. Teoreetiliselt on tegemist lõpmatu võrega, sest programmi objektide hulk on lõpmatu ja ka lukustuste arv saab järjest kasvada. Kui monitoride multihulk *x* sisaldab vähemalt kõiki samu monitore vähemalt sama arvu kordasid, kui multihulk *y*, siis *x*⊑*y*.



Joonis 16. Monitoride alamvõre

Monitoride võre kahe elemendi ülemise raja leiab kui nende elementide multihulkadest võtta ülemine raja. Näiteks elementide {a:2, b:1}, {a:1, c:2} ülemine raja oleks {a:1}. Suurimaks väärtuseks on tühi hulk, see kirjeldab hästi olukorda kus me midagi monitoride kohta ei tea. Väikseimaks väärtuseks sai võetud abstraktne väikseim element ⊥.

Lokaalsete muutujate massiivi ja operandide magasini jaoks loodud võred on ühesugused. Analüüsi jaoks on vaja teada millised muutujad on järjendis millisel kohal ja mis kohtadel on lihtsalt väärtused ning mitu elementi on järjendis. Kuna pole oluline millised väärtused täpselt on, siis tähistati teadaolevad muutujad muutujate nimedega ja lihttüüpi väärtused ning teadmatud muutujad tähistati küsimärkidega. Joonisel 17 on näitlikustatud võre kitsendustega, et on ainult kaks muutujat ja järjendi pikkus saab olla üks või kaks. Implementeeritud võres ei ole järjendi pikkus ega muutujate arv piiratud. Järjendi pikkus on teada kogu aeg välja arvatud suurima ja väikseima elemendi puhul. Üks olek on teisest suurem kui järjend on sama pikk, järjendis on samad elemendid ja vähemalt üks teadaolevatest muutuja nimedest on asendatud küsimärgiga. Jooniselt üks järjestus on alt üles [ ]⊑ [b,b] ⊑ [?,b] ⊑ [?,?] ⊑ ⊤. Erinevate pikkustega elemendid ei ole omavahel võrreldavad. Näiteks elemente [?,a] ning [a] ei saa võrrelda, kuigi sisaldavad samu muutujaid. Nende ülemine raja on suurim element.



Joonis 17. Lokaalsete muutujate massiivi/operandide magasini alamvõre

Võre seisundi implementatsioonis pandi need kolme võre seisud kokku (joonis 18) andmetüüpi *SyncState*, kus *operandStack* tähistab operandide magasini, *localsArray* lokaalsete muutujate massiivi ja *monitors* monitoride hulka.

```
SyncState(operandStack: Option[List[SyncVariable]],
          localsArray:  Option[List[SyncVariable]],
          monitors:     Option[Map[SyncVariable, Int]])
```

Joonis 18. Võre elemendi implementatsioon

2.3 Üleminekufunktsioonid

Andmejooksude leidmiseks on vaja Java baitkoodist leida üles kohad kus objekte loetakse või neisse kirjutatakse. Baitkoodi instruktsioonid, mis sellega tegelevad on *getfield* (objektist lugemine) ja *putfield* (objekti kirjutamine). Joonisel Joonis 19 on näha sünkroniseeritud koodiplokiga meetod ja joonisel 20 sellele meetodile vastav baitkood. Kui vaadata käsklusi *monitorenter*, *getfield*, *putfield* ja *monitorexit*, siis nende järjestusest saab järeldada, et mingi monitor on *getfield* ja *putfield* instruktsioonide ajal lukustatud.

```
public void add(int a) {
    synchronized (this) {
        f = f + a;
    }
}
```

Joonis 19. Meetod *add*

```
public void add(int);
 descriptor: (I)V
 flags: ACC_PUBLIC
 Code:
  stack=3, locals=4, args_size=2
     0: aload_0
     1: dup
     2: astore_2
     3: monitorenter
     4: aload_0
     5: aload_0
     6: getfield      #2    // Field f:I
     9: iload_1
    10: iadd
    11: putfield      #2    // Field f:I
    14: aload_2
    15: monitorexit
    16: goto          24
    19: astore_3
    20: aload_2
    21: monitorexit
    22: aload_3
    23: athrow
    24: return
```

Joonis 20. Meetod *add* baitkood

Baitkoodi instruktsioonide üleminekufunktsioonid kirjeldatakse meetodis *sNormal*. Siin mõned näited implementatsioonidest. *sNormal* meetodis vaadatakse, mis instruktsioon on ja vastavalt sellele tagastatakse uus olek. Joonisel 21 on toodud näide *xadd* kohta. See *x* tähistab seda, et sinna alla lähevad kõik *add* instruktsioonid. Selle instruktsiooni puhul muudetakse ainult operandide magasini – eemaldatakse kaks esimest elementi ja lisatakse algusesse uus tundmatu element. Lokaalsete muutujate massiivi ja monitoride seisud jäävad uues olekus samaks.

```
case xADD(a) =>
  SyncState(Some(unknownVar::st.operandStack.get.drop(2)),
            st.localsArray,
            st.monitors)
```

Joonis 21. *xadd* üleminekufunktsioon

Käsu *monitorenter* üleminekufunktsiooni implementatsioonis (joonis 22) omistatakse muutujale *monitorVar* eelmise seisu operandide magasini pealmise elemendi väärtus. Eelmist seisu tähistab *st*. Järgmisena leitakse *monitorVar* monitoride lukkude arv. Kui *monitorVar* väärtus juba on monitoride hulgas, siis liidetakse lukkude arvule 1 ja kui polnud, siis *monitorVar* lisatakse ja lukkude väärtuseks saab 1. Lõpuks tagastatakse uus olek *SyncState*, kus *operandStack*-is on eemaldatud esimene element, *localsArray* jäi samaks ja monitoridesse lisati uus monitor.

```
case MONITORENTER =>
  val monitorVar = st.operandStack.get.head
  val monitorCount = if (st.monitors.get contains monitorVar) {
    st.monitors.get(monitorVar) + 1
  } else 1
  SyncState(Some(st.operandStack.get.tail),
            st.localsArray,
            Some(st.monitors.get + (monitorVar -> monitorCount)))
```

Joonis 22. *monitorenter* implementatsioon

Käsud *getfiield* (joonis 23) ja *putfield* (joonis 24) üleminekufunktsioonid on üsna sarnased. Mõlema puhul võetakse välja kõigepealt info muutuja, objekti, klassi ja meetodi kohta, mille kirjutamise või lugemise operatsioon see on ja kutsutakse välja *addToReport* meetod, mis säilitab selle info hiljem koostatava raporti jaoks. Ära märgitakse ka kas tegemist on kirjutamise või lugemise operatsiooniga. Kahe üleminekufunktsiooni erinevus seisneb tagastatavas olekus. Mõlema puhul jääb *localsArray* ja monitors samaks. Käsu *getfield* puhul eemaldatakse *operandStacki* esimene element ja selle asemel lisatakse sinna objekti ja välja, mida loeti, nime info. Käsu *putfield* eemaldatakse *operandStack*-ist kaks pealmist elementi – objekt ja väli.

```
case GETFIELD(c) =>
  val obj = st.operandStack.get.head
  val field = c.x.get._2
  val monitors = st.monitors.getOrElse(None)
  val methodName = to.methodInfo.mthd
  val klass = to.methodInfo.cls
  addToReport(obj, field, Read(klass, methodName, st.monitors))
  SyncState(
      Some(SyncVariable(s"$obj.$field")::st.operandStack.get.drop(1)),
      st.localsArray,
      st.monitors)
```

Joonis 23. *getfield* implementatsioon

```
case PUTFIELD(a) =>
  val field = a.x.get._2
  val obj = st.operandStack.get(1)
  val methodName = to.methodInfo.mthd
  val klass = to.methodInfo.cls
  addToReport(obj, field, Write(klass, methodName, st.monitors))
  SyncState(Some(st.operandStack.get.drop(2)),
            st.localsArray,
            st.monitors)
```

Joonis 24 *putfield* implementatsioon

Instruktsiooni *monitorenter* üleminekufunktsioonis väärtustatakse *monitorVar* eelmise seisu *operandStack*-i pealmise elemendiga, vähendatakse sellega seotud monitoride lukkude arvu või kui see on 1, siis eemaldatakse monitoride hulgast. Olek tagastatakse uue

monitoride hulgaga ja ilma eelmise seisu magasini pealmise elemendita. Lokaalsete muutujate massiivi seis ei muutu.

```
case MONITOREXIT =>
  val monitorVar = st.operandStack.get.head
  val monitorCount = st.monitors.get(monitorVar)
  val monitors = if (monitorCount == 1) {
    st.monitors.get - monitorVar
  }  else {
    st.monitors.get + (monitorVar -> monitorCount)
  }
  SyncState(Some(st.operandStack.get.tail),
            st.localsArray,
            Some(monitors))
```

Joonis 25. *monitorexit* implementatsioon

Kogu lisatud kood on nähtav Bitbucket-is Põdra repositooriumis eraldi harus [9].

2.4 Puudused

Andmejooksude leidmiseks tehakse lihtsustus, et koodi saab välja kutsuda mitme lõime pealt ja mitu korda. See tähendab, et teostatud analüüsiga ei saa kindlalt väita, et programmi mitme lõimega kasutamise puhul tekiksid andmejooksud. Võimalik on, et piirangud lõimedele on teisiti seatud. Analüüsi täpsust saaks tõsta lisades mooduleid abistavate analüüsidega. Lõimede ja väärtuste väljavoolu (*escape*) analüüsiga saaks leida olukordasid, kus mingit klassi kasutatakse ainult pealõimes. Juhtub-rangelt-enne analüüsiga saaks leida olukordasid, kus mingeid muutujaid väärtustatakse ainult programmi käivitamise hetkel ja hiljem ainult loetakse. Moodulite lisamine on selgem, kui ühe väga keeruka sünkroniseerimisanalüüsi koostamine.

Kokkuvõte

Käesoleva bakalaureusetöö eesmärgiks oli lisada raamistikku Põder Java baitkoodi sünkroniseerimise analüüsi moodul, mis suudaks programmides välistada andmejooksusid. Loodud moodul analüüsib selleks programmi muutujaid ja monitore ning kirjutamise ja lugemise kohti. Analüüsi tulemusena saab raporti, milles kas välistatakse andmejooksude esinemine või tuuakse välja muutujad, mille puhul kahtlustatakse andmejooksu ning info, kus ja mis monitorid selle muutuja kirjutamise ja lugemise hetkedel kasutuses olid. Selle infoga on võimalik parandada sünkroniseerimise vigu, et andmejooksusid ei esineks.

Töö kirjaliku osa esimeses pooles anti ülevaade teemadest, mis on sellise analüüsi teostuseks olulised. Alustati selgitusest, mis on andmejooks ning kuidas sünkroniseerimisega selle tekkimist vältida. Kuna analüüsi tehakse Java baitkoodiga mindi edasi Java virtuaalmasina ja Java baitkoodi tutvustamisega ning räägiti lähemalt sellest, kuidas selles keskkonnas sünkroniseerimine toimub. See oli oluline analüüsi üleminekufunktsioonide teostuseks. Edasised peatükid andsid edasi info kuidas juhtvoograafi abil programmi kirjeldada, kuidas võred kirjeldavad programmi olekuruumi ning kuidas üleminekufunktsioonidega leitakse programmi abstraktsed seisud. Esimene peatükk lõppes raamistiku Põder kirjeldamisega ning mida sinna uue analüüsi mooduli jaoks lisada on vaja.

Teist peatükki alustati näidetega, kuidas valminud mooduliga programmi analüüsi tehakse, kuidas seal näeb programmi erinevaid seisusid ning milline on andmejooksude raport. Edasi mindi kirjeldusega kuidas praktilise osa jaoks kirjeldati olekuruumi võre. Järgmiseks toodi näiteid mõnest olulisemast baitkoodi üleminekufunktsioonist. Ning kõige lõpus toodi välja mõned lahenduse puudused.

Viidatud kirjandus

[1] M. M. Waldrop, „The chips are down for Moore's law," [Võrgumaterjal]. Kättesaadav aadressil: https://www.nature.com/news/the-chips-are-down-for-moore-s-law-1.19338. [Kasutatud 2. Mai 2020].

[2] J. Jenkov, „Java Concurrency," [Võrgumaterjal]. Kättesaadav aadressil: http://tutorials.jenkov.com/java-concurrency/index.html. [Kasutatud 29. aprill 2020].

[3] V. Vojdani, Static Data Race Analysis of Heap-Manipulating C Programs, Tartu: Tartu Ülikooli Kirjastus, 2010.

[4] D. Engler, K Ashcraft, „RacerX: effective, static detection of race conditions and deadlocks," *nineteenth ACM symposium on Operating systems principles* , New York, 2003.

[5] V. Vojdani, Mitmelõimeliste C-programmide kraasimine analüsaatoriga Goblin, Tartu: Tartu Ülikool, 2006.

[6] A Møller, M. I. Schwartzbach „Static Program Analysis" [Võrgumaterjal]. Kättesaadav aadressil: https://cs.au.dk/~amoeller/spa/. [Kasutatud 20. aprill 2020].

[7] T. Lindholm, F. Jellin, G. Bracha, A. Buckley, D. Smith „The Java® Virtual Machine Specification Java SE 14 Edition," [Võrgumaterjal]. Kättesaadav aadressil: https://docs.oracle.com/javase/specs/. [Kasutatud 5. mai 2020].

[8] J. Gosling, B. Joy, G. Steele, A. Buckley, D. Smith, G. Bierman „The Java® Language Specification Java SE 14 Edition," [Võrgumaterjal]. Kättesaadav aadressil: https://docs.oracle.com/javase/specs/ . [Kasutatud 21. aprill 2020].

[9] T. Weiss, „5 Things You Didn't Know About Synchronization in Java and Scala," OverOps, [Võrgumaterjal]. Kättesaadav aadressil: https://blog.overops.com/5-things-you-didnt-know-about-synchronization-in-java-and-scala/. [Kasutatud 5. aprill 2020].

[10] M. Kilp, „Peatükk 1. Hulgateooria elemendid," *Algebra I*, Tartu, Eesti Matemaatika Selts, 2005, pp. 1-34.

[11] V. Laan, „Võreteooria. Loengukonspekt. Sügis 2017,“ [Võrgumaterjal]. Kättesaadav aadressil: https://courses.ms.ut.ee/MTMM.00.039/2017_fall/uploads/Main/kon.pdf. [Kasutatud 21 aprill 2020].

[12] „Raamistik Põder. Lähtekood“,“ [Võrgumaterjal]. Kättesaadav aadressil: https://bitbucket.org/kalmera/poder. [Kasutatud 18 aprill 2020].

[13] „Raamistik Põder sünkroniserimise analüüsi lähtekood,“ [Võrgumaterjal]. Kättesaadav aadressil: https://bitbucket.org/kalmera/poder/branch/sync. [Kasutatud 2. Mai 2020].

Lisad

I. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, Halliki Mullari,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose **Java baitkoodi sünkroniseerimise analüüs raamistikus Põder**, mille juhendaja on Kalmer Apinis, reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.

2. Annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creative Commonsi litsentsiga CC BY NC ND 3.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni.

3. Olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.

4. Kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Halliki Mullari

08.05.2020