KONKUREERIVA RESSURSITAOTLUSE STAATILINE OTSING JAVA KOODIS

Maiga Kolk
maiga@ut.ee

Programmeerimiskeelte semantika uurimisseminar MTAT.03.204
Arvutiteaduse instituut, Tartu Ülikool
Detsember 2008

Kokkuvõte

Multilõimeliste programmide koodi analüüsil on palju kõlapinda leidnud üheaegse ressursitaotluse avastamise meetodid. Üheaegsest sama ressursi järele haaramisest tingitud anomaaliad programmi töös on raskesti avastatavad ja taasesitatavad, mis muudab automaatsete koodi kontrollerite olemasolu oluliseks. Käesolev referaat kirjeldab üht võimalikest staatilistest analüüsimeetoditest, mis on realiseeritud rakendusena Chord ja mis on praktiliste testide käigus osutunud arvestatavalt efektiivseks.

1. Sissejuhatus

Multilõimeline programm sisaldab konkureerivat ressursitaotlust, kui kahel lõimel on ilma täiendavate järjestuspiiranguteta ligipääs ühele ja samale asukohale mälus, kusjuures vähemalt üks ligipääsudest on kirjutamine. Konkureeriva ressursitaotluse avastamine ja taasesitamine on keeruline, mistõttu jäävad paljud seda tüüpi vead programmides üldiselt kõrvaldamata.

Lahendusena on läbi arvutiteaduse ajaloo loodud mitmeid tarkvaralisi vahendeid konkureeriva ressursitaotluse avastamiseks ja seeläbi tarkvara kvaliteedi tõstmiseks.

Erinevad lähenemisviisid võib suures plaanis jagada kaheks grupiks – dünaamilised ja staatilised. Dünaamilised vahendid põhinevad juhtub-enne seosel (*happens-before relation*) [1], *Lockset* algoritmil [2] või nende kahe meetodi kombinatsioonil [6]. Miinusena on dünaamilised vahendid mittetäieliku, ei suuda analüüsida nö avatud programme ja vajavad küllaltki palju sisendinformatsiooni analüüsitava programmi kohta. Staatilised vahendid on oma iseloomult enamasti käsuvoost sõltumatud (*flow insensitive*) tüübipõhised süsteemid [3], käsuvoost sõltuvad *Lockset* algoritmi staatilised versioonid [4] või teest sõltuvad (*path sensitive*) mudelikontrollerid [7]. Staatilised vahendid leiavad reeglina üles vaid väikese osa kõigist vigadest, kuna liiga detailsed analüüsimeetodid ei oleks mahukatele programmidele rakendatavad, jämedakoelised algoritmid seevastu ei ole võimelised kõiki vigu märkama ja raporteerivad vaid paar viga miljoni koodirea kohta [9]. Enamlevinud dünaamilised ja staatilised lähenemisviisid on nimetatud põgusa ülevaate andmiseks, kuid lähemalt me nendega käesolevas referaadis ei tutvu.

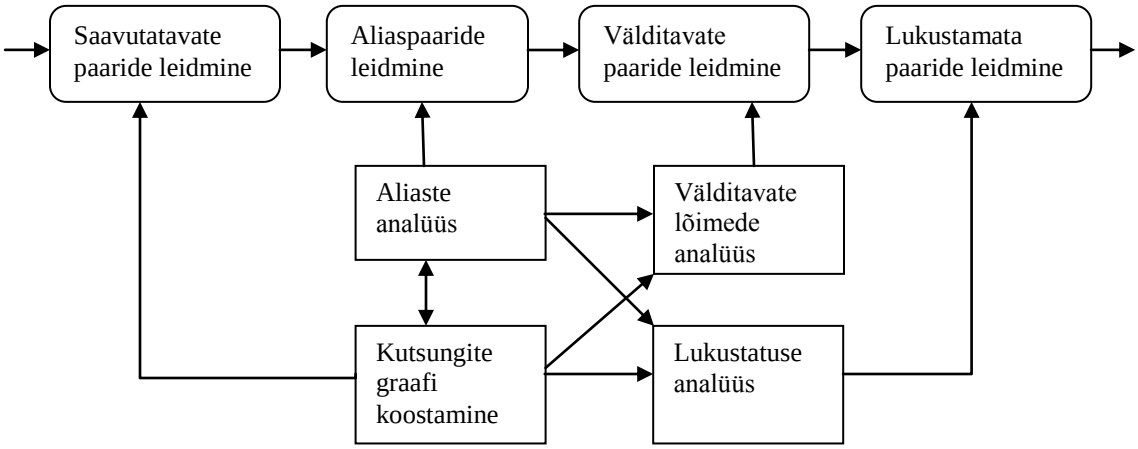
Järgnevad peatükid kirjeldavad uudset eelloetletud süsteemidest erinevat staatilist tehnikat konkureeriva ressursitaotluse avastamiseks Java koodis. Tehnika autoriteks on Mayuk Naik, Alex Aiken ja John Whaley Standfordi Ülikoolist ning käesoleva referaadi peamiseks aluseks on autoritelt 2006. aastal ilmunud artikkel „*Effective Static Race Detection for Java*“ [10].

2. Üheaegse ressursitaotluse avastamise meetod

2.1 Ülevaade üheaegse ressursitaotluse avastamise meetodist

Meetodi idee on järgmine. Esimese sammuna koostatakse komplekt ressursitaotluste paare (nn originaalsed paarid), mis võivad teoreetiliselt olla osalised üheaegses ressursitaotluses. Seejärel eemaldatakse esialgsest komplektist nelja sammuga paarid, mille puhul võib üheaegse sama ressursi järele haaramise välistada. Läbitavad etapid on järgmised: saavutatavate paaride leidmine (*reachable pairs*), aliaspaaride leidmine (*aliasing pairs*), välditavate paaride leidmine (*escaping pairs*) ja lukustamata paaride leidmine (*unlocked pairs*). Viimase elimineerimise tulemusena saadav paaride komplekt sisaldab endas vaid selliseid ressursitaotluste paare, mille puhul üheaegne sama ressursi järele haaramine on realistlik. Tuleb kindlasti mainida, et lõplik paaride hulk sisaldab lisaks programmi vigadele ka nn valepositiivseid juhtumeid ja vajab seetõttu manuaalset ülevaatust. Analüüsi käigus kasutatakse nelja erinevat staatilise analüüsi meetodit: kutsungite graafi koostamine (*call-graph*), aliaste analüüs (*alias analysis*), välditavate lõimede analüüs (*thread-escape analysis*) ja lukustatuse analüüs (*lock analysis*). Algoritmi tööetappide ja kasutatavate analüüsimeetodite seoseid illustreerib järgmine joonis (Joonis 1).

Aliaste analüüsi ja kutsungite graafi koostamise vahele tõmmatud kahesuunaline nool märgib nende omavahelist sõltuvust, lisaks toimuvad kutsungite graafi genereerimine ja aliaste analüüsi läbiviimine üheaegselt, kasutades k-objektitundlikku lähenemist. Kahes viimases etapis kasutatavad välditavate lõimede analüüs ja lukustatuse analüüs sõltuvad omakorda nii aliaste analüüsi tulemustest kui ka kutsungite graafist.



Joonis 1. Üheaegse ressursitaotluse algoritmi ülesehitus

Järgnevalt kirjeldatava tehnika ja seda tehnikat kasutava näidisrakenduse Chord tugevusteks võib lugeda:

- täpsus (rakenduse töö tulemusena väljastatakse mõistlikult väike hulk nn valepositiivseid juhte);
- skaleeritavus (rakendus võimaldab analüüsida oluliselt suuremaid programme kui paljud teised samal eesmärgil loodud analüsaatorid);
- sünkroniseerimise idioomidega arvestamine (rakendus oskab analüüsi käigus arvestada enim kasutatavate sünkroniseerimise idioomidega: leksikoloogiliselt piiritletud lukustamisel põhinev sünkroniseerimine (*lexically-scoped lock-based synchronisation*), fork/join kasutamisel põhinev sünkroniseerimine (*fork/join synchronisation*) ja wait/notify kasutamisel põhinev sünkroniseerimine (*wait/notify synchronisation*));
- avatud programmide analüüsimise võime;
- raporteeritavate vigade kohta näidete toomine (näited sisaldavad piisaval hulgal informatsiooni, mis võimaldab mõista vea olemust, vigast kohta koodist kergemini üles leida ja kõrvaldada).

Siinkohal tuleb kindlasti välja tuua ka mõned selle tehnika nõrgemad küljed. Kasutatav algoritm on kontekstitundlik, kuid käsuvoost sõltumatu. Sõltumatus käsuvoost toob positiivse küljena kaasa väiksema ressursinõudluse arvutuste teostamisel ja lihtsustab nn avatud programmide analüüsimist, kuid vähendab arvutuste täpsust. Eelmises lõigus oli tehnika tugevusena välja toodud võime analüüsida enim kasutatavaid sünkroniseerimise idioome, mida on kokku kolm. Põhjus, miks algoritm suudab arvestada vaid kolme ja mitte kõigi Java programmeerimisel kasutatavate sünkroniseerimise idioomidega, seisnebki algoritmi sõltumatuses käsuvoost [9].

Vaadeldav tehnika ei ole terviklik. Näidetena mitteterviklikkusest võib tuua avatud programmide analüüsimise, mille käigus koodis puuduvad kutsungite vastuvõtjad

asendatakse nn tupikutega ja nendega seotud situatsioone analüüsitakse vaid osaliselt. Lukustatud paaride leidmisel neljandal analüüsisammul kasutatakse lukustatuse mõiste defineerimisel üheelemendilist viidete lähendit, mis ei ole terviklik. Testimisel jäetakse (sarnaselt paljudele teistele sama eesmärki täitvatele tehnikatele) vahele juurdepääsud klassi initsiaatorites, konstruktorites ja *finalizer*'ites, kuna need osalevad üliharva konkureerivas ressursitaotluses, kuid genereerivad analüüsi käigus hulgaliselt valealarme. Viimase punktina ignoreeritakse analüüsi käigus klasside dünaamilise laadimisega seotud efekte.

2.2 k-objektitundlik analüüs (*k-object-sensitive analysis*)

Üheaegse ressursitaotluse avastamise tehnikas on väga olulisel kohal k-objektitundlikkus [8] – üks viimastel aastatel defineeritud kontekstitundlikkuse vorme. K-objektitundlik analüüs on objekti- ja kontekstitundlik, kuid käsuvoost sõltumatu programmi koodi analüüsimise meetod, mille käigus koostatakse programmi kutsete omavahelisi seoseid kirjeldav kutsungite graaf (meie poolt vaadeldavas algoritmis saavutatavate paaride arvutamise aluseks) ja viidete lähend (*points-to approximation*, aluseks aliaste analüüsil).

Meetodi objektitundlikkus väljendub analüüsimeetodi võimes eristada objekte nende allokatsioonijadade (*sequence of allocation sites*) põhjal. Java koodis luuakse objekt võtmesõna `new` kasutades. Meetodit, kus objekt luuakse, nimetatakse objekti allokatsioonikohaks. Eksisteerigu kaks erinevat allokatsioonikohtade jada $[h_1 :: h_2' \ldots ::h_n']$ ja $[h_1 :: h_2'' :: \ldots :: h_n'']$. Jada esimene element märgib objekti allokatsioonikohta. Toodud jadade alamjadad $[h_2' \ldots ::h_n']$ ja $[h_2'' :: \ldots :: h_n'']$ esindavad objekte, millele viitavad vastavate allokatsioonikohtade meetodite `this` muutujad. `This` objekt on allokeeritud vastavalt asukohas h_2' või h_2'', meetodis, mille `this` objekti esindab $[h_3' \ldots ::h_n']$ või $[h_3'' :: \ldots :: h_n'']$. Ja analoogselt kuni viimaste jada elementideni välja. Olgu jadade esimesed elemendid võrdsed (objektidel on ühine allokatsioonikoht), kuid ülejäänud elementide hulgas esinegu vähemalt üks erinevus. K-objektitundlik analüüs võimaldab erinevate allokatsioonijadadega objekte üksteisest eristada ka juhul, kui jadad tähistavad üht ja sama objekti. Meetod on lisaks kontekstitundlik, mis väljendub analüüsimeetodi võimes analüüsida programmi koodi arvestades erinevate abstraktsete kontekstidega. Meetodi käsuvoost sõltumatus väljendub asjaolus, et analüüsi käigus arvutatakse viidete nimekirjad globaalsetena ja ei arvestata sellega, millises programmi punktis parasjagu asutakse [11].

Kutsungite graaf sisaldab endas nii lõimest sõltuvaid kui ka sõltumatuid teid. Lõimest sõltumatuid teid saab kirjeldada kujul $(i, c') \rightarrow^* (m, c)$, mis ütleb, et meetod m kontekstis c on otseselt või kaudselt saavutatav kutsungikohast i kontekstis c'. Lõimest sõltuvaid teid saab kirjeldada kujul $(i, c') =>^* (m, c)$, mis ütleb, et meetod m kontekstis c on saavutatav kutsungikohast i kontekstis c' ilma lõimi vahetamata. See tähendab, ilma sellist kutsungikohta rakendamata, mis võib olla enam kui ühe lõime algatajaks.

Muutuja k väärtust kasutatakse viidete lähendi leidmisel. Viidete lähend koosneb kolmikutest (c, v, o). Kolmikut tõlgendatakse järgnevalt: lokaalne muutuja v võib viidata abstraktsele objektile o abstraktses kontekstis c. Kolmiku viimane element o on lõplik mittetühi objekti allokatsioonikohtade jada $[h_1:: \ldots ::h_n]$, mille nn saba $[h_2:: \ldots ::h_n]$ koosneb maksimaalselt k-1 allokatsioonikohast.

Mitmete praktiliste testide käigus on jõutud tulemuseni, et k-objektitundliku analüüsi meetodit üheaegsete ressursinõuete avastamise tehnikas kasutades piisab, kui k väärtuseks omistada maksimaalselt 3.

3. Algoritm

3.1 Rakendi genereerimine (*Harness Synthesis*)

Üheks kirjeldatava üheaegse ressursitaotluse avastamise algoritmi tugevuseks on võime analüüsida lisaks suletud programmidele ka avatud programme. Avatud programmide hulka kuuluvad sellised rakendused, millel puudub `main` meetod ja mis on mõeldud kasutamiseks teiste rakenduste koosseisus (näiteks draiverid jm).

Avatud programmi analüüsimisega on seotud kaks probleemi – puudu on osa nii kutsungite esitajatest kui ka kutsungi vastuvõtjatest. Kutsungi vastuvõtjad asendatakse nn tupikutega ja analüüsi käigus neid ei arvestata. Seevastu puuduvate kutsungi esindajate tarbeks genereeritakse nn rakend, mis võimaldab simuleerida mitmesuguseid programmi väljakutsumise stsenaariume.

Rakend, mis sisaldab endas programmi testimiseks vajalikku tehislikult genereeritud `main` meetodit, luuakse järgmisel põhimõttel. Genereerimisel võetakse sisendiks avatud programmi liideste (*interface*) hulk, mille toimimist programmi testimise käigus vaadeldakse.

Iga liidese kohta:

a. deklareeritakse üks lokaalne muutuja igast tüübist, mis on mõne liideses defineeritud meetodi argumendi või väljundina lubatud;

b. mittedetermineeritult omistatakse igale deklareeritud lokaalsele muutujale konkreetne objekt igast konkreetsest muutuja tüübi klassist;

c. mittedetermineeritult kutsutakse välja iga liideses defineeritud meetod iga lokaalsete muutujate kombinatsiooniga (argumentide tüüpe arvestades) ja omistatakse tagastusväärtus (juhul, kui see on olemas) igale lokaalsele muutujale, mille tüüp kattub välja kutsutud meetodi tagastusväärtuse tüübiga.

Ühtsuse eesmärgil genereeritakse ka suletud programmide puhul tehislik `main` meetod. Suletud programmi tehislik peameetod sisaldab vaid üht kutsungikohta (*call site*), mille poolt algatatud lõim pöördub testitava programmi `main` meetodi poole.

3.2 Originaalsete paaride leidmine (*Original-Pairs computation*)

Järgmise sammuna peale rakendi loomist genereeritakse nn originaalsete paaride hulk, mis on sisendiks analüüsi esimeses etapis – saavutatavate paaride leidmisel. Originaalsete paaride hulk koostatakse tugeva liiaga. See tähendab, et paaride hulga genereerimisel arvestatakse kõikvõimalike kohtadega koodis, mis üritavad saavutada ligipääsu mõnele staatilisele või isendiväljale või massiivi elemendile, ja luuakse neist kõikvõimalikud kahekaupa kombinatsioonid (e_1, e_2). Ainsa reeglina arvestatakse nõuet, et vähemalt üks ligipääsudest e_1 ja e_2 peab olema kirjutamine.

Ligipääsu nõudmine staatilisele klassi *C* väljale *g* on äratuntav järgmise kuju põhjal (kasutatud lokaalset muutujat *y*): lugemine *y = C.g*, kirjutamine *C.g = y*.

Ligipääsu nõudmine isendiväljale *f* on äratuntav järgmise kuju põhjal (kasutatud lokaalseid muutujaid *x* ja *y*): lugemine *y = x.f*, kirjutamine *x.f = y*.

Ligipääsu nõudmine massiivi elemendile on äratuntav järgmise kuju põhjal (kasutatud lokaalseid muutujaid *x, y* ja *i*): lugemine *y = x[i]*, kirjutamine *x[i] = y*.

3.3 Saavutatavate paaride leidmine (*Reachable-Pairs computation*)

Saavutatavate paaride leidmise käigus eemaldatakse originaalsete paaride hulgast kõik sellised paarid, mille puhul vähemalt üks juurdepääsudest ei ole saavutatav sellisest lõimi algatavast kutsungikohast, mis omakorda oleks saavutatav tehislikult genereeritud `main` meetodist. Lihtsamalt öelduna eemaldatakse kõik paarid, mille puhul vähemalt ühte juurdepääsudest on võimalik saavutada vaid ühe lõime poolt ning üheaegse ressursitaotluse ohtu ei eksisteeri.

Saavutatavate paaride leidmisel kasutatakse punktis 2.2 iseloomustatud kontekstitundlikku kutsungite graafi. Vaatleme juurdepääsude paari (e_1, e_2). Selleks, et eksisteeriks üheaegse ressursitaotluse võimalus, peavad nii e_1 kui ka e_2 olema lõimi vahetamata saavutatavad vastavatest lõimi algatavatest kutsungikohtadest s_1 ja s_2. See tähendab, et saavutatavate paaride leidmisel vaadeldakse kutsungite graafis kõikvõimalikke teid kujul $(i, c') =>^* (m, c)$. Lisaks nii s_1 kui ka s_2 peavad olema saavutatavad `main` meetodist.

3.4 Aliaspaaride leidmine (*Aliasing-Pairs computation*)

Aliaspaaride leidmine põhineb järgmisel asjaolul – üheaegse ressursitaotluse toimumiseks on vajalik, et vaadeldava juurdepääsude paari kumbki pool oleks taotletav ühe ja sama objekti juurest. Analüüsil vaadeldakse eelmises punktis leitud saavutatavate paaride hulka ning eemaldatakse kõik sellised paarid, mis on seotud ühe ja sama isendivälja või massiivi elemendiga, kuid mille puhul on välistatud juurdepääsu taotlemine ühe ja sama objekti juurest.

Käesoleva analüüsisammu käigus ei vaadelda juurdepääse staatilistele väljadele, kuna staatiliste väljade puhul on alati tegemist aliaspaariga.

Iga üksiku juurdepääsu analüüsimisel võetakse sisendiks juurdepääs isendiväljale *x.f* või massiivi elemendile *x[i]* ning juurdepääsu sisaldava meetodi kontekst c_1. Seejärel leitakse kõik objektid, millele muutuja *x* võib antud kontekstis viidata (nn viidete lähend, mainitud punktis 2.2). Aliaspaaride hulk moodustub kõigist sellistest saavutatavatest paaridest, mis viitavad isendiväljadele *x.f* ja *y.f* või *x[i]* ja *y[j]* ja mille puhul eksisteerivad sellised isendivälju sisaldavate meetodite kontekstid c_1 ja c_2 ning objekt *h*, et nii *x* kontekstis c_1 kui ka *y* kontekstis c_2 võivad viidata objektile *h*. Siinjuures on oluline märkida, et nii käesoleval juhul kui ka järgmises peatükis loetakse kontekstid objektiga samaväärseteks ja võivad olla *h* väärtuseks.

3.5 Välditavate paaride leidmine (*Escaping-Pairs computation*)

Välditavate paaride arvutamise aluseks on fakt, et üheaegse ressursitaotluse realiseerumiseks peab mälukoht, millele juurdepääsu nõutakse, olema lõimede vahel ühiskasutuses. Analüüsisammu käigus eemaldatakse eelmises punktis leitud aliaspaaride hulgast kõik sellised ressursitaotluste paarid, mis viitavad lokaalsele mälukohale.

Esimese sammuna leitakse kõigi selliste objektide hulk, mis võivad olla lõimede vahel ühiskasutuses. Objekt *h* loetakse lõimede vahel ühiskasutuses olevaks, kui kehtib mõni järgnevatest punktidest:

a. Lõimi algatavate kutsungikohtade hulgas leidub selline, mille mõni argumentidest võib viidata objektile *h*;

b. Leidub selline lõimede vahel ühiskasutuses olev objekt *h'* ja selle objekti isendiväli *f*, mis võib viidata objektile *h*;

c. Objekt *h* on saavutatav mõnelt staatiliselt väljalt.

Aliaspaar, mis viitab isendiväljadele *x.f* ja *y.f* või massiivi elementidele *x[i]* ja *y[j]*, loetakse välistavaks paariks, kui leiduvad sellised neid juurdepääse sisaldavate meetodite kontekstid c_1 ja c_2 ning objekt *h*, et *x* kontekstis c_1 ja *y* kontekstis c_2 võivad viidata objektile *h* ja *h* kuulub eelmises lõigus leitud objektide hulka.

3.6 Lukustamata paaride leidmine (*Unlocked-Pairs computation*)

Lukustamata paaride leidmisel eemaldatakse eelmises peatükis leitud välistavate paaride hulgast kõik sellised, mille puhul juurdepääsud on nö lukustatud.

Olgu meil lõimi algatav kutsungikoht *i* kontekstis *c'*, juurdepääs *e*, seda juurdepääsu sisaldav meetod *m* kontekstis *c* ja rada *w* kontekstitundlikus kutsungite graafis, mis viib *(i, c') =>* (m, c)*. Kutsungikohas *i* kontekstis *c'* algatatud lõim hoiab juurdepääsu taotlemisel lukku, kui kehtib üks järgnevatest:

a. Juurdepääs *e* on leksikoloogiliselt suletud `synchronized (l) {...}` blokki ja `l` poolt viidatud hulk kontekstis *c* on {*h*};

b. Rada *w* sisaldab kutsungikohta *i''* kontekstis *c''* ja kutsungikoht *i''* on leksikoloogiliselt suletud `synchronized (l) {...}` blokki ja `l` poolt viidatud hulk kontekstis *c''* on {*h*}.

Alles jäävad vaid sellised juurdepääsude paarid, mis ei ole lukustatud.

3.7 Järeltöötlus

Lukustamata paarid raporteeritakse võimalike vigadena. Kuivõrd algoritmi töö tulemused sisaldavad ka nn valepositiivseid juhtumeid ja vajavad manuaalset ülevaatust ning filtreerimist, siis on oluline, et väljastatav info oleks võimalikult põhjalik ja täpne. Sel eesmärgil väljastatakse iga võimaliku üheaegse ressursitaotluse kohta ka näide ning kategoriseeritakse tulemusi. Algoritm tagastab iga paari kohta

vastavad rajad kontekstitundlikus kutsungite puus. Lisaks kategoriseeritakse tulemusi nii välja- kui ka objektipõhiselt.

4. Näide

Järgnev näide illustreerib eespool tutvustatud üheaegse ressursitaotluse avastamise algoritmi tööd.

```
1     public class A {
2           int f;
3           static public void main() {
4                 A a;
5                 if (*) a = new A();
6                 if (*) a.get();
7                 if (*) a.inc();
8           }
9           public A() { this.f = 0; }
10          private int rd() { return this.f; }
11          private int wr(int x) { this.f = x; return x; }
12          public int get() { return this.rd(); }
13          public synchronized int inc() {
14                int t = this.rd() + (new A()).wr(1);
15                return this.wr(t);
16          }
17    }
```

Meetod `main` on rakendi loomise käigus tehislikult genereeritud. Lõimi algatavad kutsungikohad meetodis `main` on `a.get` ja `a.inc`.

Esimese sammuna leiame originaalsete paaride hulga. Kõikvõimalikke ligipääsude taotlusi on näidiskoodis kolm: `this.f` real 9 (tähistame f_w'), `this.f` real 10 (tähistame f_r) ja `this.f` real 11(tähistame f_w). Kõik juurdepääsud on seotud isendiväljaga *f*, indeksid $_r$ ja $_w$ tähistavad vastavalt lugemist ja kirjutamist. Koostame neist kõikvõimalikud juurdepääsude paarid, arvestades reegliga, et vähemalt üks kahest juurdepääsust peab olema kirjutamine. Tulemus on järgmine: (f_r, f_w), (f_w, f_w), (f_r, f_w'), (f_w', f_w'), (f_w, f_w').

Leiame nüüd saavutatavate paaride hulga, võttes aluseks originaalsed paarid. Esimese sammuna eemaldame paarid (f_r, f_w'), (f_w', f_w'), (f_w, f_w'), kuna neis osalev f_w' sisaldub konstruktoris ja käesoleva analüüsi käigus konstruktoreid ei vaadelda. Kasutame 1-objektitundlikkust ja sellele vastavat kutsungite graafi, kuivõrd vaadeldava väikese kooditüki jaoks ei ole suurem k väärtus vajalik. Meetod `rd` on saavutatav ühes kontekstis c_a kahest lõimi algatavast kutsungikohast `a.inc` ja `a.get` kontekstis ε (peameetodil `main` puudub reeglina `this` argument, seetõttu tähistatakse peameetodi konteksti tühisümboliga ε). Meetod `wr` on saavutatav kahes kontekstis c_a ja c_b ühest lõimi algatavast kutsungikohast `a.inc` kontekstis ε. Kontekst c_a tähistab `new A()` allokatsiooni meetodis `main` ja kontekst c_b tähistab `new A()` allokatsiooni meetodis `inc`. Saavutatavad paarid (kombineerituna arusaadavuse ja järgmiste analüüsisammude huvides ligipääsudega seotud kontekstidega) on järgmised: (f_r, c_a, f_w, c_a), (f_r, c_a, f_w, c_b), (f_w, c_a, f_w, c_a), (f_w, c_a, f_w, c_b), (f_w, c_b, f_w, c_a), (f_w, c_b, f_w, c_b).

Saavutatavate paaride hulgast elimineerime järgmise sammuna kõik sellised paarid, mille puhul juurdepääsud ei ole taotletavad ühe ja sama objekti juurest. Kordame siinkohal taas üle, et antud lähenemise puhul loetakse objektid samaväärseks kontekstidega. Juurdepääs f_r kontekstis c_a: `this` argument võib viidata objektidele $\{c_a\}$. Juurdepääs f_w kontekstis c_a: `this` argument võib viidata objektidele $\{c_a\}$. Juurdepääs f_w kontekstis c_b: `this` argument võib viidata objektidele $\{c_b\}$. Aliaspaarid saame, kui eemaldame saavutatavate paaride hulgast kõik sellised, mille puhul ligipääsud antud kontekstides ei ole saavutatavad ühe ja sama objekti juurest ehk eelpool leitud objekte sisaldavate hulkade ühisosa on tühi. Nõnda on see näiteks juhtudel (f_w, c_a, f_w, c_b) ja (f_w, c_b, f_w, c_a), kuna hulkadel $\{c_a\}$ ja $\{c_b\}$ puudub ühisosa. Analüüsi tulemusena saame aliaspaaride hulgaks (f_r, c_a, f_w, c_a), (f_r, c_a, f_w, c_b), (f_w, c_a, f_w, c_a), (f_w, c_b, f_w, c_b).

Aliaspaaride hulgast välditavate paaride hulga saamiseks eemaldame esialgsest hulgast kõik sellised paarid, milles vähemalt üks juurdepääsudest viitab lokaalsele mälukohale. Meie näites on lokaalse mälukohaga tegemist c_b puhul ja eemaldada tuleb (f_r, c_a, f_w, c_b) ja (f_w, c_b, f_w, c_b). Alles jäävad (f_r, c_a, f_w, c_a) ja (f_w, c_a, f_w, c_a), sest lõimi algatavate kutsungikohtade `a.inc` ja `a.get` (meetodis `main`) argument `a` võib viidata objektile c_a.

Viimase sammuna tuleb välistavate paaride hulgast eemaldada lukustatud paarid. Toodud näites eemaldame (f_w, c_a, f_w, c_a), sest ainus rada kutsungite graafis alguspunktiga `a.inc` meetodis `main` kontekstiga ε kuni meetodini `wr` kontekstiga c_a hoiab lukustatuna argumenti `this`, kusjuures argumendi `this` poolt viidatud objektide hulk on $\{c_a\}$. Lõplik paaride hulk (koos kontekstidega) koosneb seega täpselt ühest elemendist (f_r, c_a, f_w, c_a).

Algoritmi töö tulemusena tagastatakse info, et juurdepääsude f_r ja f_w vahel võib esineda konkureeriv ressursitaotlus. Lisaks illustreeritakse tulem kahe rajaga kutsungite graafist, mis annavad kujundlikult edasi info, et üheaegne ressursitaotlus võib tekkida hetkel, mil meetodeid `inc` ja `get` rakendatakse üheaegselt kahes erinevas lõimes ühe ja sama klassi `A` objekti peal [9].

```
field reference A.f(A.java:10) [Rd]
A.get(A.java:6)
Harness.main(Harness.java:4)

field reference A.f(A.java:11) [Wr]
A.inc(A.java:7)
Harness.main(Harness.java:5)
```

5. Meetodi rakendamine praktikas

Kirjeldatud tehnika ei eksisteeri vaid teooria tasemel – meetodi autorid on oma idee realiseerinud staatilise Java koodi analüsaatorina Chord [5] ja viinud läbi rea teste erinevate Java programmide peal, mille maht jääb vahemikku ~2 600 kuni ~650 000 koodirida. Testid on olnud väga produktiivsed hoolimata sellest, et meetod ei arvesta analüüsitava programmi käsuvooga ja kasutab mitmel juhul võimalikust väiksema täpsusega analüüsimeetodeid. Ajaliselt võtab väiksemate programmide testimine aega umbkaudu 1,5 minutit, mahukamate programmide testitulemusi tuleb oodata

keskeltläbi 26 minutit. Kuivõrd algne paaride komplekt – originaalsed paarid – genereeritakse valimatult, on paaride hulk Chord töö alguses väga suur ning selle töötlemine mälunõudlik. Sel põhjusel ei ole Chord'i võimalik kasutada väga mahukate projektide testimisel, kuivõrd programmi tööks tuleb mälust puudu.

Punktis 2.2 mainitud fakt, et piisava arvutustäpsuse saavutamiseks üheaegse ressursitaotluse avastamise algoritmis piisab, kui k väärtuseks omistada maksimaalselt 3, toob meid järgmise punktini – BDD-põhine analüüs ja selle seos Chord implementatsiooniga. k-objektitundliku analüüsi kõrge täpsustase tingib tavalisest suurema mäluressursi kasutuse. Esmaste testide käigus selgus, et avalikult kättesaadavad k-objektitundliku analüüsi vahendid ei suutnud oma tööd ressursipuuduse tõttu lõpetada juba k = 1 puhul, rääkimata k suurendamisest 3-ni. Binary Decision Diagrams (BDD) [12] on ainsana tõestanud oma efektiivsust üleprogrammiliste kontekstitundlike analüüside läbiviimisel. Nõnda on Chord rakenduses kõik neli algoritmi etappi ja etappide raames kasutatavad analüüsimeetodid realiseeritud loogilises programmeerimiskeeles Datalog, rakendades tulemuste arvutamisel BDD-põhist Datalog'i implementatsiooni bddbddb.

6. Kokkuvõte

Kokkuvõttena võib öelda, et tutvustatud tehnika näol on tegemist silmapaistva näitega staatilisest üheaegse ressursitaotluse avastamise algoritmist, mis on erinevalt mitmetest teistest samalaadsetest lahendustest skaleeruv ja piisava täpsusega, tagastades oma töö tulemusena vaid vähesel hulgal valealarme. Eripäradena tuleb kindlasti välja tuua ka võime arvestada analüüsi käigus kolme enim levinud sünkroniseerimise idioomiga, analüüsida avatud programme ja genereerida avastatud vigade illustreerimiseks piisavalt informatiivseid näiteid.

Tehnika miinusteks on mitmetes analüüsi punktides kasutatud ebatäiuslikud lähenemisviisid (näiteks puuduvate kutsungi vastuvõtjate asendamine nn tupikutega ja nendega mitte arvestamine kõikvõimalike situatsioonide simuleerimisel, käsuvoost sõltumatute analüüsimeetodite kasutamine, millest tulenevalt on võimalik analüüsida vaid üht alamhulka kõikvõimalikest sünkroniseerimise idioomidest jne). Samas on algoritmi autorid näiteks esimesena mainitud puuduse kohta lisanud kommentaari, et põhimõtteliselt on täiusliku Rakendi klassi loomine kirjeldatud tehnikale oluliselt lihtsam kui oleks sama asja koostamine sissejuhatuses mainitud mudelikontrolleri tarbeks. Seega tuleb vaid ootama jääda uusi täiendusi ja parendusi, mis aitaksid kaasa kirjeldatud tehnika täielikkusele ja vähendaksid veelgi valepositiivsete kirjete hulka algoritmi töö tulemustes, mõjutamata silmatorkavalt lahenduse skaleeruvust.

Viited

[1] **Adve, S., Hill, M., Miller, B., Netzer, R.** (1991); Detecting data races on weak memory systems. In Proceedings of the 18th Annual International Symposium on Computer Architecture (ISCA'91)

[2] **Agarwal, R., Sasturkar, A., Wang, L., Stoller, S.** (2005); Optimized run-time race detection and atomicity checking using partial discovered types. In Proceedings of

the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)

[3] **Boyapati, C., Lee, R., Rinard, M.** (2002); Ownership types for safe programming: Preventing data races and deadlocks. In Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)

[4] **Choi, J., Loginov, A., Sarkar, V.** (2001); Static datarace analysis for multithreaded object-oriented programs. Technical Report RC22146, IBM Research.

[5] Chord. A Static Analysis Framework for Java [http://chord.stanford.edu/]

[6] **Dinning, A., Schonberg, E.** (1991); Detecting access anomalies in programs with critical sections. In Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging (PADD'91)

[7] **Henzinger, T., Jhala, R., Majumdar, R.** (2004); Race checking by context inference. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04)

[8] **Milanova, A., Rountev, A., Ryder, B.** (2005); Parameterized object sensitivity for points-to analysis for Java. ACMTransactions on Software Engineering and Methodology, 14(1).

[9] **Naik, M., Aiken, A., Whaley, J.**; Effective Static Race Detection for Java [http://research.microsoft.com/CONFERENCES/pldi06/presentations/chord_pldi06.ppt]

[10] **Naik, M., Aiken, A., Whaley, J.** (2006); Effective Static Race Detection for Java [http://portal.acm.org/citation.cfm?id=1133981.1134018]

[11] **Naik, M., Gray, D., Park, C., Sen, K.**; Effective Static Deadlock Detection [http://chord.stanford.edu/pubs/staticdeadlock.pdf]

[12] **Whaley, J.** (2007); Context-Sensitive Pointer Analysis using Binary Decision Diagrams. Ph.D. thesis, Stanford University.