Relatsiooniline programmianalüüs binaarsete otsustusdiagrammidega

Raivo Laanemets
rlaanemt@ut.ee

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

Kokkuvõte

Programmianalüüs on tänapäeva programmeerimises tähtsal kohal. Järjest enam arendatakse suure mahuga programme ja täiendatakse olemasolevaid, pidevalt suureneb programmide keerukus ja koos sellega potentsiaalsete programmivigade hulk. Programmivead võivad põhjustada privaatsete andmete lekkimist, suurt majanduslikku kahju või halvimal juhul isegi inimese surma. Seetõttu on tarvis avastada võimalikult suur kogus programmi vigu enne selle kasutuselevõttu. Vigade otsimise algoritmide põhikomponendiks on viidaanalüüs *(pointer analysis)*, mis üritab välja selgitada, millistele kuhjaobjektidele võivad programmi lähtekoodis esinevad muutujad viidata.

Järgnev referatiivne kirjatöö vaatleb lähemalt ühte efektiivset meetodit viidaanalüüsi realiseerimiseks binaarsete otsustusdiagrammidega *(BDD - Binary Decision Diagram)* ning annab ülevaate tarkvarast *bddbddb (BDD-Based Deductive DataBase)*, mis seda meetodit ka kasutab. Teisi sarnaseid lähenemisi töö ei käsitle.

1. Sissejuhatus

Käesolevas referaadis vaatleme programmide staatilist analüüsimist, kus binaarseid otsustusdiagramme kasutatakse kui efektiivset viisi programmiandmete hoidmiseks. Staatiline analüüs üritab välja selgitada programmi omadusi ilma koodi ennast käivitamata. Üheks tähtsaimaks omaduseks, mis meid huvitab, on see, millistele programmi poolt loodud kuhjaobjektidele *(heap object)* võivad koodis esinevad muutujad viidata. Sellist viitamistele keskenduvad analüüsi nimetatakse viidaanalüüsiks *(pointer analysis)*.

Üldisel juhul on programmide staatiline analüüs mittelahenduv [15]. Seda juba sellepärast, et mingi muutuja väärtustamine võib olla tingimuslik (näiteks kasutades *if-else*-konstruktsiooni), kus tingimuse osas võib esineda kuitahes keeruline aritmeetiline avaldis. Seetõttu tuleb lihtsustada analüüsiprotsessi ja rahulduda ebatäpsete tulemustega [3]. Samas on praktilistes rakendustes siiski ka sellised tulemused kasutatavad. Lihtsustada on võimalik järgmiste võtete abil:

1. *if-else*-konstruktsioonide ignoreerimine; vaadeldavas koodiosas toimub näivalt mõlema haru korraga täitmine.

2. Programmisammude järjekorra mittearvestamine.

3. Sama meetodi kõigi väljakutsete üheks liitmine.

Analüüsimeetodit, mis rakendavad esimest ja teist lihtsustamist, nimetatakse *voost mittesõltuvaks (flow-insensitive)*, seda mis kasutab kolmandat, nimetatakse *kontekstist mittesõltuvaks (context-insensitive)*. Mõlema lihtsustuse täpsema kirjelduse koos illustreerivate jooniste ning koodinäidetega on võimalik leida allikast [3].

Selles kirjatöös vaatleme universaalset meetodit, mida on seni rakendatud põhiliselt kontekstist sõltuva, kuid voost mittesõltuva analüüsi tarbeks. Meetodi põhiidee seisneb programmiandmete, s.o esinevate muutujate, väärtustuste, loodavate objektide, meetodi kutsete, parameetrite jms. kirjeldamisel ja hoidmisel relatsioonilisel kujul. Kui andmed on niisugusel esitusviisil antud, tuleb analüüsi spetsifitseerimiseks koostada vaid andmebaasipäring üle meid huvitavate relatsioonide. Paraku väljenduvad mahukate programmide (suuremad kui 100000 rida) andmed äärmiselt suurte relatsioonide kujul, lisaks kasvatab täpsema analüüsi (kontekstist sõltuva) vajadus nende mahtu omakorda eksponentsiaalselt [13, 8].

On selgunud, et programmianalüüsis esinevate ekstreemselt mahukate relatsioonilise andmekoguste efektiivseks ja kompaktseks hoidmiseks sobivad eriti hästi binaarsed otsustusdiagrammid [8]. Siin töös lähtume just selles artiklis kirjeldatud vaatepunktist otsustusdiagrammide kasutamisel. Muid lähenemisviise (mis ka mingil viisil kasutavad sarnaseid diagramme ära) ajapuudusel ja referaadi autori ebapiisavate teadmiste tõttu programmianalüüsi osas siin kirjeldatud ei ole.

Töö baasosa põhineb ainult artiklitel [11, 13, 8, 12, 9] ja kasutab teema käsitlemisel ülevalt-alla süvitsi lähenemist, võttes järjest läbi kõik nendes artiklites esitatud põhiideed ning vaatleb põhiliselt nende matemaatilist ja tehnoloogilist (sh. implementeerimise probleemid) tausta. Käesoleva kirjatöö eesmärk ei ole valdkonnast ülevaate andmine, seega üldistavat ja võrdlevat analüüsi teiste sarnaste lähenemiste suhtes siit ei leia.

Referaat koosneb neljast osast:

1. Keele Datalog kui loogilise andmebaasikeele tutvustus.

2. Datalog programmide teisendamine relatsioonalgebra valemiteks ja lihtsustatud lähenemine rekursiivsete programmide (päringute) realiseerimisele.

3. Binaarsed otsustusdiagrammid, nendega teostatavad operatsioonid, relatsioonalgebra väljendamine nende operatsioonide abil ning relatsioonide kodeerimine diagrammidena.

4. Pikem praktiline näide.

Referaadi autor loodab, et töö pakub lugejale kasulikke teadmisi programmianalüüsist. Materjal on esitatud küll kokkuvõtlikult, kuid juurde on lisatud viited põhimaterjalile, mida uurides on võimalik juba valdkonnaga põhjalikumalt tutvuda.

2. Datalog

Datalog on loogiline andmebaasikeel, mis on süntaktiliselt keele Prolog alamosa. Tema kasutamist andmebaaside päringukeelena alustati 1970ndate aastate lõpus. Keele süntaks sarnaneb Prologi omaga, kuid funktsionaalsümbolite kasutamine pole lubatud [14].

Datalogi programm koosneb reeglitest (predikaadid). Igal predikaadil on unikaalne nimi ja fikseeritud arv argumente. Argumentidena võivad esineda muutujad või konstandid. Kui kõik argumendid on konstandid, siis tähistab predikaat mingit fakti. Predikaadi definitsioon koosneb peast *(head)* ja kehast *(body)*. Keha omakorda koosneb konjuktsiooni abil seotud teistest predikaatidest. Üks predikaat võib olla defineeritud mitme reegli abil, sellele vastab disjunktsioon. Näide Datalogi reeglist on järgmine:

$$D(w, z) \leftarrow A(w, \text{“a“}, x), B(x, y), C(y, z)$$

Siin on kahe argumendiga predikaat `D` defineeritud kui predikaatide `A`, `B` ja `C` konjuktsioon. Väiketähti `x,y,z,...` kasutatakse muutujate tähistamiseks. Konstandid (siin `a`) kirjutatakse jutumärkide vahele (v.a arvulised konstandid). See ei lähe päris kokku levinud süntaksiga keele Prolog (ja teiste sellesarnaste loogiliste keelte) jaoks, kus funktsionaalsümbolid ja konstandid peavad algama väikese tähega ja muutujad suure tähega, aga just sellist varianti kasutatakse allikates [11, 13, 8, 12, 9]. Miks standardset süntaksit ei eelistatud, pole teada.

Niisugusel viisil üles kirjutatud reegleid võib interpreteerida kui andmebaasipäringuid. Siis vastab kahe argumendiga predikaat `D` kahe atribuudiga relatsioonile, mis on saadud sobivalt kombineerides relatsioonid `A`, `B` ja `C`. Kuidas see täpsemalt käib, on seletatud järgnevalt.

2.1. Datalogi transleerimine

Erinevalt keelest Prolog ei oma Datalog eeldefineeritud lahendusstrateegiat. See tähendab, et Datalogi implementeerimiseks on oluliselt suurem vabadus. Mitterekursiivsed

Datalogi programmid saab esitada relatsioonalgebra valemitega. Sissejuhatuse relatsioonalgebrasse leiab allikatest [1, 14]. Relatsioonalgebra põhioperatsioonideks on kuus relatsioonide hulga suhtes kinnist operatsiooni:

1. Relatsioonide ühend *(set union)* $R_1 \cup R_2$ sisaldab kõiki korteeže mõlemast relatsioonist R_1 ja R_2. R_1 ja R_2 peavad olema ühesuguste atribuutidega.

2. Filter *(selection)* $\sigma_\varphi R$ annab kõik R korteežid, mis rahuldavad predikaati φ.

3. Projektsioon *(projection)* $\pi_{a_1,\ldots,a_n} R$ annab R korteežid suurusega n, mis koosnevad ainult R atribuutidest $a_1, \ldots, a_n$.

4. Atribuudi ümbernimetamine *(rename)* $\rho_{a/b} R$ nimetab R atribuudi a ümber atribuudiks b.

5. Relatsioonide otsekorrutis *(Cartesian product)* $R_1 \times R_2 = \{r_1 \cup r_2 | r_1 \in R_1, r_2 \in R_2\}$ annab korteežid suurusega $m+n$, kus m on relatsiooni R_1 suurus ja n on relatsiooni R_2 atribuutide arv. Relatsioonidel R_1 ja R_2 ei tohi olla sama nimega atribuute.

6. Relatsioonide vahe *(set difference)* $R_1 - R_2 = \{r | r \in R_1, r \notin R_2\}$ annab kõik korteežid, mis esinevad relatsioonis R_1, aga mitte relatsioonis R_2. Mõlemal relatsioonil peavad olema ühesugused atribuudid.

Relatsioonalgebra tarvitamise vahekeelena tingib optimiseerimise lihtsus, kasutades ära teadmisi algrelatsioonide suuruse kohta ning ülemise kuue operatsiooni algebralisi omadusi (distributiivsus, assotsiatiivsus, kommutatiivsus). Tegemist on läbiuuritud valdkonnaga ja selline lähenemine on andmebaasipäringute optimeerimisel levinud [14].

Datalogi predikaadite väljendamisest relatsioonalgebra valemitega võib lugeda pikemalt kirjandusest [6, 14]. Näitena toome siin järgmise predikaadi transleerimise (kus TP ja PJ väljendavad algrelatsioone):

$$TJ(t, j) \leftarrow TP(t, p), PJ(p, j)$$

Esitatud predikaate võib interpreteerida järgmiselt: $TJ(t, j)$: töötaja t töötab juhi j alluvuses; $TP(t, p)$: töötaja t töötab projektis p; $PJ(p, j)$: projekti p juhib inimene j.

Kõigepealt nimetame atribuudi p ümber atribuudiks p' relatsioonis TP. Seejärel leiame saadud relatsiooni ja PJ otsekorrutise, millest filtriga võtame need korteežid, kus $p = p'$. Viimase relatsiooni projekteerime atribuutidele t ja j. Seega

$$TJ = \pi_{t,j}(\sigma_{p=p'}(\rho_{p/p'}(TP) \times PJ)).$$

2.2. Rekursiivsed programmid

Datalog võimaldab kirja panna rekursiivseid predikaate, paraku puudub relatsioonalgebras sobiv operatsioon, mis aitaks otseselt väljendada selliseid reegleid. Vajadus rekursiivsete predikaatide järele on siiski olemas. Näiteks programmianalüüsis soovime leida muutuja-kuhjaobjekti suhteid läbi meetodite kutsete tekkivate formaalsete-tegelike parameetrite ahelate. Üldistatult väljendavad sellised relatsioonid mingi(te) teis(t)e relatsiooni(de) transitiivset sulundit.

Relatsiooni nimetame *ekstensionaalseks (Extensional Database - EDB)*, kui ta pole defineeritud teiste reeglite abil, st. tegemist on salvestatud andmetega, ja *intensionaalseks (Intensional Database - IDB)*, kui ta on defineeritud reeglite abil. Arvestame, et ükski predikaat ei väljenda korraga EDB ja IDB relatsiooni (sellest saab üle, kui defineerida kolmas predikaat, mis võib olla EDB ja IDB predikaadi disjuktsioon) [14].

Järgnevalt on kirjeldatud lihtsat kuid ebaefektiivset *(naive fixpoint evaluation)* lähenemist (mille optimeeritumat varianti kasutab ka *bddbddb* tarkvara [11]). Näitena vaatleme klassikalist transitiivse sulundi arvutamise programmi:

$$\begin{aligned} &\mathit{järglane}(y,x) \leftarrow \mathit{vanem}(x,y).\\ &\mathit{järglane}(y,x) \leftarrow \mathit{järglane}(z,x), \mathit{vanem}(z,y). \end{aligned}$$

Predikaadi *järglane* mõlemad definitsioonid saab panna kirja otse relatsioonalgebra valemitega:

$$\begin{aligned} \mathit{järglane}_1 &= \rho_{z/y}\rho_{y/x}\rho_{x/z}(\mathit{vanem})\\ \mathit{järglane}_2 &= \pi_{y,x}(\sigma_{z_1=z}(\rho_{z/z_1}(\mathit{järglane}) \times \mathit{vanem})) \end{aligned}$$

Terve relatsiooni arvutamine käib aga püsipunkti leidmise teel:

$\mathit{järglane} := \mathit{järglane}_1$
repeat
  $\mathit{järglane} := \mathit{järglane} \cup \mathit{järglane}_2$
until ükski intensionaalne relatsioon ei muutunud viimases iteratsioonis

Kui kõik ektensionaalsed relatsioonid on lõpliku suurusega, siis on see triviaalne, et püsipunkt leidub (ülaloleva algoritmi täitmine lõpeb mingil ajahetkel) [14].

Keele Datalog programm võib sisaldada ka eitust predikaadi definitsiooni kehas. Sellisel juhul ülemine algoritm otseselt kasutamiseks ei kõlba. Tarkvarapakett *bddbddb* võimaldab eitust kasutada tänu sellele, et kõik relatsioonid on lõplikud ja relatsiooni täiendi (vastava korteežide hulga täiendi) arvutamine on madala hinnaga operatsioon [11]. Operatsioonide madal maksumus saavutatakse otsustusdiagrammide kasutamise teel andmete hoidmiseks, millest annab ülevaate referaadi järgmine osa.

3. Binaarsed otsustusdiagrammid

Binaarne otsustusdiagramm *(BDD - binary decision diagram)* on tsüklivaba suunatud graaf, mille abil saab arvutada temale vastava lausearvutusvalemi väärtuse valemi muutujate etteantud väärtustusel. Graafi tippe on kahte liiki:

- *Terminaaltipud*, mis tähistavad tõeväärtusi;
- *Otsustustipud*, mis vastavad muutujatele.

Otsustustippudest väljuvad suunatud kaared terminaaltippude või teiste otsustustippude juurde. Kaared on märgendatud muutuja väärtusega 1 või 0 (tõene või väär). Tippu, millesse ei sisene ühtegi kaart nimetatakse diagrammi juurtipuks.

BDD saab koostada nii valemi kui ka tõeväärtustabeli põhjal. Järgnevas näites (tabel 1) on kasutatud tabelit. Diagrammi koostamiseks tuleb kõigepealt valida üks muutujatest x_1, x_2 või x_3 ja selle järel hakata (rekursiivselt) koostama alamgraafe. Iga muutuja jaoks saame kaks alamgraafi: esimeses on muutuja valitud tõeseks, teises vääraks. Tabelile 1 vastab otsustusdiagramm joonisel 1. Joonisel tähistab tõest kaart pidevjoon ning väära kriipsjoon. Otsustustipud on ümmargused ja terminaaltipud kandilised. Juurtipuks on muutujat x_1 esitav tipp.

Ilma lisakitsendusi tegemata saame üldjuhul erinevad graafid, valides esimeseks tipuks erinevad muutujad (tabelit ei saa esitada unikaalselt). Sõltuvalt rakendusvaldkonnast võib unikaalsuse omadus tähtis olla.

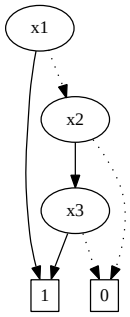
x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Tabel 1: Näidistabel

Kui otsustusdiagramm on valmis, siis selle kasutamiseks tuleb lihtsalt läbida tee juurtipust ühte terminaaltippu mööda kaari, mis vastavad muutujate väärtustusele. Valemi väärtuse saame terminaaltipust, milleni lõpuks jõudsime.

3.1. BDD erikujud

Üheks otsustusdiagrammi erikujuks on OBDD *(Ordered BDD)*, kus kõik teed juurtipust terminaaltippudeni järgivad mingit kindlat lineaarset järjestust $x_1 < x_2 < \cdots < x_n$.



Joonis 1: Otsustusdiagramm

Taandatud *(reduced)* OBDD otsustusdiagrammiks (ROBDD) nimetatakse diagrammi, milles kehtivad lisakitsendsed:

1. Tippude unikaalsus: diagrammi iga kahe otsustustipu u ja v jaoks kehtib $var(u) = var(v) \wedge low(u) = low(v) \wedge high(u) = low(v) \Rightarrow u = v$, kus: $var(t)$ tähistab tipu t poolt esitatavat muutujat; $low(t)$ tipu t negatiivse kaarest lähtuvad alamdigrammi ja $high(t)$ tipu t positiivsest kaarest lähtuvat alamdiagrammi.

2. Liiasuse puudumine: ühestki tipust u ei lähtu nii positiivne kui negatiivne kaar sarnastesse alamdiagrammidesse, st. $low(u) \neq high(u)$.

ROBDD tähtsaimaks omaduseks on suvalise Boole' funktsiooni $f : \mathbb{B}^n \to \mathbb{B}$ esitamine unikaalselt, mis tähendab, et igale Boole' funktsioonile vastab täpselt üks ROBDD diagramm (kanoonilisuse lemma). Samuti on sellised diagrammid mõnevõrra kompaktsemad (täpsemalt minimaalsed (vähima tippude arvuga) fikseeritud muutujate järjestuse jaoks). Lisaks on nende koostamiseks olemas lihtne algoritm [4].

3.2. Relatsioonide kodeerimine BDD kujule

Relatsioonide esitamiseks asendatakse kõigepealt atribuutide väärtused mittenegatiivsete täisarvudega. Olgu R n-aarne relatsioon, $R = (A_1, \ldots, A_n)$, kus $A_1, \ldots, A_n$ on atribuudid (mille väärtushulgad on lõplikud). Atribuutide $A_1, \ldots, A_n$ väärtuste asendamisel vastavate väärtustega täisarvude hulkadest $D_1, \ldots, D_n$ saab relatsiooni R esitada n-aarse funktsioonina (predikaat):

$$f : D_1 \times \cdots \times D_n \to \mathbb{B}$$

Kusjuures $(d_1, \ldots, d_n) \in R \Leftrightarrow f(d_1, \ldots, d_n) = 1$ ja $(d_1, \ldots, d_n) \notin R \Leftrightarrow f(d_1, \ldots, d_n) = 0$, $\forall i = 1, \ldots, n$ korral $d_i \in D_i$.

Väärtused hulkades $D_1, \ldots, D_n$ saab omakorda kodeerida binaararvudena. Väärtuse $a \in D_i$ kodeerimiseks läheb siis tarvis $\lceil \log_2 |D_i| \rceil$ bitti, terve relatsiooni tarbeks aga

$m = \sum_{i=1}^{n} \lceil \log_2 |D_i| \rceil$ bitti. Asendades d_i $(i = 1, \ldots n)$ vastavate binaarmuutujatega $b_1^i, \ldots, b_{k_i}^i$ saame m-kohalise Boole' funktsiooni $f'(b_1^1, \ldots, b_{k_1}^1, b_1^2, \ldots, b_{k_2}^2, \ldots, b_1^n, \ldots b_{k_n}^n)$.

Kodeerimise illustreerimiseks on koostatud järgnev näide.

Näide $D_1 = \{0, 1, 2, 3\}$, $D_2 = \{0, 1\}$ ja $R = \{(1, 1), (2, 0), (2, 1), (3, 0), (3, 1)\}$. Siin piisab D_1 väärtuste jaoks kahest bitist ja D_2 jaoks ühest. Tähistame vastavaid lausearvutusmuutujaid b_1^1, b_2^1 ja b_1^2. Täisarvuliste väärtuste teisendamiseks on mugav kasutada nende standardset esitust kahendsüsteemis, st. näiteks $d_1 = 1 \in D_1$ korral $b_1^1 = 1$ ja $b_2^1 = 0$ $(d_1 = b_1^1 2^0 + b_2^1 2^1 = 1)$. Relatsiooni R esitab funktsioon $f_R(b_1^1, b_2^1, b_1^2)$.

D_1			D_2		R	
$d_1 = b_1^1 2^0 + b_2^1 2^1$	b_1^1	b_2^1	$d_2 = b_1^2 2^0$	b_1^2	$r = (d_1, d_2)$	$f_R(b_1^1, b_2^1, b_1^2) \equiv r \in R$
0	0	0	0	0	(0,0)	0
0	0	0	1	1	(0,1)	0
1	1	0	0	0	(1,0)	0
1	1	0	1	1	(1,1)	1
2	0	1	0	0	(2,0)	1
2	0	1	1	1	(2,1)	1
3	1	1	0	0	(3,0)	1
3	1	1	1	1	(3,1)	1

Joonis 2: Näidisrelatsiooni kodeerimine

Tabel 2 on sama, mis oli esitatud eespool valemi otsustusdiagrammiks kodeerimise juures (tabel 1), kui võtta $x_1 = b_1^1$, $x_2 = b_2^1$ ning $x_3 = b_1^2$. Loomulikult tuleb sarnane ka otsustusdiagramm (joonis 1). Üldiselt saadava otsustusdiagrammi suurus relatsiooni enda suurusest ei sõltu. Mõned BDD tarkvarapaketid piiravad siiski kodeerimiseks kasutada olevate bittide arvu. Selleks on tavaliselt masina sisemiselt täisarvu esitamiseks kasutatavate bittide arv (näiteks 64). 64 bitti võimaldab kodeerida relatsioone, mille suurus (elementide arv) peab jääma väiksemaks kui $2^{64} \sim 10^{19}$. Allikas [11] raporteerib ka praktikas esinenud suuremate (10^{23}) relatsioonide tarvidusest.

3.3. BDD operatsioonid

Kui on antud kaks (RO)BDD otsustusdiagrammi, mis esitavad Boole' funktsioone, siis saab nendega teostada operatsioone, mis vastavad funktsioonide vahel teostatavatele loogilistele tehetele, millele vastavad omakorda relatsioonalgebra operatsioonid. Järgnevates lõikudes kasutatud algoritmid pärinevad allikast [4].

Suvalise loogilise operaatori $\circ$ (nende hulgas $\wedge$, $\vee$ jt. standardsed operaatorid) rakendamiseks kahe BDD u_1 ning u_2 jaoks on olemas universaalne algoritm APPLY$(\circ, u_1, u_2)$, mis ei sõltu operaatorist endast.

Kui otsustusdiagrammid u_1 ja u_2 väljendavad mingeid relatsioone R_1 ja R_2 , siis nende ühendit $R_1 \cup R_2$ väljendab diagramm $v = \text{APPLY}(\vee, u_1, u_2)$.

Lihtne on ka relatsiooni R täiendi R' leidmine, tuleb vaid terminaaltipud 1 ja 0 ümber vahetada. Kasutades ära täiendit ja APPLY algoritmi, kus loogiliseks operaatoriks valida $\wedge$, saame ka relatsioonide vahe $R_1 - R_2 = R_1 \cap R_2'$ kirja panna.

Filtrile σ_φ (predikaadina φ esineb võrdus konstandiga - sellest väljendusvõimsusest piisab siin) vastab algoritm RESTRICT(u, j, b), mis asendab diagrammis u muutuja j tõeväärtusega b. Algoritmis toimub konstandi kodeerimiseks kasutatud muutujate asendamine konstandile vastavate väärtustega.

Projektsiooni $\pi_{a_1,\ldots a_n}R$ leidmiseks tuleb ära jätta relatsiooni diagrammi tipud, mis ei vasta ülejäänud R atribuutide väärtuste kodeerimiseks kasutatud muutujatele. Kui tegemist on taandatud ja järjestatud BDD diagrammiga, tuleb mõnevõrra lisatööd teha, et puuduks liiasus ning kehtiks tippude unikaalsus (järjestuse kitsendus jääb püsima).

Otsekorrutisele $R_1 \times R_2$ vastab algoritmi APPLY kasutamine operaatoriga $\wedge$. Kuna töö kontekstis kasutatavas relatsioonalgebra rakenduses esineb otsekorrutis vahetult koos atribuutide ümbernimetamise, filtri ja projektsiooniga, siis on need kõik kokku võetud ühe operatsioonina - *relprod*. *Relprod* leiab naturaalse ühendi (ühend sama nimega atribuutide väärtuste võrduste järgi) ja eemaldab üleliigsed atribuudid töö käigus [11].

Lisaks nendele operatsioonidele on vajalikud ka diagrammide koostamine ja andmete väljastamine. Nende olemasolu sõltub juba konkreetsest BDD tarkvarapaketist.

4. *bddbddb* tarkvara

Tarkvarapakett *bddbddb (BDD-Based Deductive-DataBase)* [2] on Datalogi realisatsiooni. Andmeid hoitakse sisemiselt BDD kujul. Kasutada saab erinevaid BDD realisatsioone, nende hulgas *JavaBDD* [5] (vaikimisi kasutusel). Tarkvara on programmeeritud Java keeles. Tegemist on üldise Datalogi realisatsiooniga, mis sobib ka muuks otstarbeks kui programmianalüüs. Java-keelsest programmist relatsioonide saamiseks on vajalik *Joeq* Java virtuaalmasin [10], mis sisaldab koodi *bddbddb* andmefailide väljastamiseks (nii *bddbddb, JavaBDD* ja *Joeq* autoriks on John Whaley). Relatsioonid loetakse välja otse baitkoodist, mitte programmi tekstilisest kujust. Baitkood sisaldab piisavalt informatsiooni, et *bddbddb* väljundtulemusi programmi tekstilise kuju peal kasutada (reanumbrid). Tähtsamad relatsioonid, mille *Joeq* väljastab, on järgmised:

- $vP_o(V, H)$: $(v, h) \in vP_o$, kui programmis omistatakse muutujale $v \in V$ objekt $h \in H$. Siia alla kuuluvad programmilaused kujul `s=new String()`.
- $S(V, F, V)$: $(v_1, f, v_2) \in S$, kui programmis omistatakse muutuja $v_1 \in V$ (objekti tüüpi) väljale $f \in F$ muutuja $v_2 \in V$ väärtus. Näide: `x.f=y`.
- $L(V, F, V)$: $(v_1, f, v_2) \in L$, kui programmis omistatakse muutujale $v_2 \in V$ muutuja $v_1 \in V$ (objekti tüüpi) välja $f \in F$ väärtus. Näide: `x = f.y`.

- $mI(M, I, N)$: $(m, i, n) \in mI$, kui programmis toimub meetodis $m \in M$ teise meetodi $n \in N$ väljakutse $i \in I$.
- $Iret(I, V)$: $(i, v) \in Iret$, kui meetodi väljakutse $i \in I$ tulemusena tagastatakse väärtus muutujasse $v \in V$.
- $actual(I, Z, V)$: $(i, z, v) \in actual$, kui meetodi väljakutse $i \in I$ korral antakse meetodi parameetriks järjekorranumbriga z muutuja $v \in V$ väärtus. Esimese parameetri jaoks $z = 1$, teise jaoks $z = 2$ jne. Virtuaalse meetodi jaoks parameeter $z = 0$ tähistab objekti, mille meetodit kutsutakse. Staatilise meetodi korral on selleks konstant `null`.
- $formal(M, Z, V)$: $(m, z, v) \in formal$, kui meetodi $m \in M$ parameetrit järjekorranumbriga $z \in Z$ tähistab muutuja $v \in V$.
- $hP_o(H, F, H)$: $(h_1, f, h_2) \in hP_o$, kui objekti $h_1 \in H$ väli $f \in F$ viitab objektile $h_2 \in H$.

Lisaks nendele kirjutatakse välja ka relatsioonid objektide tüübiinfo kohta. Täpsemalt tuleb seda uurida *Joeq* lähtekoodist [7], sest dokumentatsiooni, mis relatsioone üldiselt kirjeldaks, ei ole. Artiklites [11, 13] kasutatud relatsioonid esitab *Joeq* teiste nimedega, sest tarkvara on mõnevõrra edasi arendatud artiklite ilmumise ajast alates.

4.1. Praktiline näide

Referaadis esitatud praktiline näide kuulub turvalise andmevoo valdkonda kuuluva programmianalüüsi. See on referaadi autori arvates palju lihtsam, selgem, konkreetsem, paremini kommenteeritud ja loetavam, kui originaalallikates [11, 13] esitatud näited, mis sobivad eelkõige valdkonna eksperdile lugemiseks.

Näide Olgu tarvis leida kõik programmi meetodid, kus kirjutatakse välja kasutaja parool. Kasutajat esitab klass `User`, millel on kaks välja: `name` ja `password`. Analüüsi spetsifitseerime kui predikaadi `prints_password(M)`:

```
prints_password(m) :-
  mI(m,i,"java/io/PrintStream/voidprintln(java/lang/String)"),
  actual(i,1,v1),
  vP(v1,h),
  L(_,"ee/pri/rl/test/User/password",v2),
  vP(v2,h).
```

Predikaat `prints_password(M)` on parajasti tõene, kui `M` on selline meetod, mis rahuldab meie predikaadi definitsioonis kasutatavaid predikaate:

1. `mI(m,i,"java/io/PrintStream/voidprintln(java/lang/String)")` - toimub meetodi `java.io.PrintStream.println` väljakutse i.
2. `actual(i,1,v1)` - meetodi väljakutse `i` esimeseks argumendiks on muutuja `v1`.
3. `vP(v1,h)` - muutuja `v1` on väärtustatud objektiga `h` (viidaanalüüs).
4. `L(_,"ee/pri/rl/test/User/password",v2)` - mingi muutuja `v2` väärtustatakse väljal `User.password` oleva väärtusega.
5. `vP(v2,h)` - see muutuja `v2` viitab objektile `h` (sama objekt, millele viitab `v1`).

Predikaat `vP(v,h)` ise on defineeritud järgmiselt:

```
vP(v,h) :- vP0(v,h).
vP(v1,h) :- A(v1,v2), vP(v2,h).
vP(v2,h2) :- L(v1,f,v2), vP(v1,h1), hP(h1,f,h2).

hP(h,f,h2) :- hP0(h,f,h2).
hP(h1,f,h2) :- S(v1,f,v2), vP(v1,h1), vP(v2,h2).

A(v1,v2) :- formal(m,z,v1), IE0(i,m), actual(i,z,v2).
A(v2,v1) :- Mret(m,v1), IE0(i,m), Iret(i,v2).
```

`vP(v,h)` on põhiline viidaanalüüsi predikaat ja seda on põhjalikumalt kirjeldatud allikates [11, 13]. Predikaadi definitsioon praegusel kujul on võetud *bddbddb* tarkvaraga kaasa tulevatest näidetest. Siin on predikaati lihtsustatud objekti tüüpe ja meetodite erindeid käsitleva informatsiooni eemaldamisega.

Loomulikult ei illustreeri see piiratud näide kõiki võimalusi, mida *bddbddb* abil on võimalik kasutada. Analüüsi võib täiendada Java sõnetöötluse arvestamisega (konkateneerimine), et leida potentsiaalsed muutujad, mis sisaldavad parooli.

Lisaks turvalise andmevoo probleemide lahendamisele saab teostada ka järgmist [11]:

- Mälulekete silumine;
- Programmi tüüpimise viimistlemine *(type refinement)*;
- Objektide jagamise probleemid erinevate lõimede vahel *(thread escape)*.

5. Kokkuvõte

Staatilise viidaanalüüsi rakendamine tarkvarast vigade ja andmelekete leidmisel on tehtud mugavaks tänu programmiandmete esitamisele relatsioonilisel kujul. Relatsioonid, millega siin kokku puututakse, võivad paraku mahukaks osutuda, mistõttu omab olulist

tähtsust andmete hoidmise viis. Relatsioonilised andmed saab esitada binaarsete otsustusdiagrammide (BDD) kujul. Teisest küljest muudab programmianalüüside spetsifitseerimine päringukeeles Datalog analüüside koostamise niivõrd lihtsaks ja efektiivseks, et sellega saab hakkama tavaline programmeerija. BDD-põhine keele Datalog implementatsioon *bddbddb* koos *Joeq* tarkvaraga moodustavad tervikliku komplekti vahendeid Java-keelsete programmide staatiliseks analüüsimiseks.

Viited

[1] Anne Villems. Andmebaaside teooria (loengukonspekt). Tartu Ülikool 1997. http://math.ut.ee/kursused/abaasid97/

[2] *bddbddb* projekti koduleht ja lähtekood. http://bddbddb.sourceforge.net/

[3] Derek Rayside. Points-To Analysis (lecture notes). Massachusetts Institute of Technology, 2005

[4] Henrik Reif Andersen. An Introduction to Binary Decision Diagrams (lecture notes). IT University of Copenhagen, 1999. http://www.itu.dk/~hra/notes-index.html

[5] *JavaBDD* projekti koduleht ja lähtekood. http://javabdd.sourceforge.net/

[6] Jeffrey D. Ullman. Implementation of Logical Query Languages for Databases. ACM Transactions on Database Systems, 1985

[7] *Joeq* projekti koduleht ja lähtekood. http://joeq.sourceforge.net/

[8] John Whaley. bddbddb: Using Datalog and BDDs for Program Analysis (tutorial slides). Stanford University, 2006

[9] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog with Binary Decision Diagrams for Program Analysis. In *Proceedings of Programming Languages and Systems: Third Asian Symposium*, 2005

[10] John Whaley. Joeq: A virtual machine and compiler infrastructure. In *Proceedings of the SIGPLAN Workshop on Interpreters, Virtual Machines, and Emulators*, 2003

[11] John Whaley, Monica S. Lam. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, 2004

[12] Kevin Bierhoff. Alias Analysis with bddbddb (class project report). Carnegie Mellon University, 2005

[13] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, Christopher Unkel. Context-sensitive program analysis as database queries. *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2005

[14] Ramez Elmasri, Shamkant B. Navathe. Fundamentals of Database Systems (3rd Ed.). Leheküljed 685-715.

[15] William Landi. Undecidability of Static Analysis. ACM Letters on Programming Languages and Systems, 1992