TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Juhan Oskar Hennoste

Abstraktne silur Goblintile

Bakalaureusetöö (9 EAP)

Juhendaja: Simmo Saan, MSc

Tartu 2023

Abstraktne silur Goblintile

Lühikokkuvõte:

Staatiline analüüs on kasulik meetod programmi omaduste tuvastamiseks ilma programmi jooksutamata. Goblint on Tartu Ülikoolis arendatav staatiline analüsaator C keelele. Staatilise analüüsi käigus tuvastab Goblint palju programmi omadusi, näiteks muutujate võimalike väärtusi programmi eri punktides. Need tulemused on kasulikud nii analüüsitava programmi kui ka Goblinti analüüside käitumise mõistmiseks, kuid hetkel puudub nende kuvamiseks hea viis. Olemasolevad tööriistad esitavad info toorel ja raskesti tõlgendataval kujul. Selles töös luuakse esmane versioon ühest võimalikust lähenemisest selle info kuvamiseks. Loodud lahendus on nn abstraktne silur, mis kasutab tavalise siluri (*debugger*) kasutajaliidest, kuid selle asemel et programmi jooksutada, kasutab abstraktne silur Goblinti analüüsi tulemusi, et simuleerida programmi jooksutamist. Selles töös realiseeritakse siluri esmane versioon ning kogutakse sellele tagasisidet Goblinti arendajatelt. Loodud abstraktne silur on mitmes aspektis edasiminek võrreldes olemasolevate tööriistadega Goblinti analüüsi tulemuste uurimiseks.

Võtmesõnad:

Staatiline analüüs, abstraktne interpretatsioon, silur, Goblint

CERCS:

P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Abstract debugger for Goblint

Abstract:

Static analysis is a useful method for determining properties of programs without executing them. Goblint is a static analyser for the C programming language developed at the University of Tartu. During static analysis, Goblint determines many properties of the program, for example possible values of variables at different points in the program. These results are useful to understand the program being analysed, as well as the behavior of the Goblint analyses themselves. However, currently there is no good way to visualize them. Existing tools display the information in a raw and difficult to interpret form. In this thesis the initial version of one approach to visualize this information is implemented. The implemented approach is a special type of debugger, called an abstract debugger, which uses the same interface as a standard debugger, but instead of executing the program, it simulates program execution using the Goblint analysis results. The initial version of the abstract debugger is implemented and feedback for it is gathered from Goblint developers. The implemented abstract debugger is a step forward compared to existing tools for viewing Goblint analysis results.

Keywords:

Static analysis, abstract interpretation, debugger, Goblint

CERCS:

P170 Computer science, numerical analysis, systems, control

Sisukord

Sissejuhatus

Staatiline analüüs on programmi omaduste tuvastamise meetod, kus analüüsitakse programmi koodi ilma seda käivitamata [1]. Staatiline analüüs on kasulik, kuna see võimaldab tuvastada vigu, mis esinevad vaid spetsiifilistel äärejuhtudel ning seetõttu tavakasutuses ja testimise käigus ei avaldu [2]. Goblint[1] on avatud lähtekoodiga[2] staatiline analüsaator programmeerimiskeelele C, mida arendatakse Tartu Ülikooli ja Müncheni Tehnikaülikooli koostöös [3]. Goblinti eesmärgiks on programmist võimalike vigade, ennekõike andmejooksude (*data races*), leidmine analüüsides programmi lähtekoodi [4].

Goblinti analüüsi põhiliseks väljundiks on nimekiri programmist leitud võimalikest vigadest, kuid analüüsi käigus tuvastab Goblint ka palju muid programmi omadusi, näiteks võimalikke programmi seisundeid, muutujate väärtusi ja nendevahelisi seoseid. Need vahetulemused on kasulikud, et mõista, miks Goblint peab mingit viga võimalikuks, kuid hetkel puudub hea viis neid kuvada. Olemasolevad tööriistad esitavad vahetulemused toorel kujul ning nende tõlgendamine vajab seetõttu küllaltki põhjalikke teadmisi Goblinti seesmisest toimimisest ja kasutajapoolset lisatööd.

Selle töö eesmärk on luua Goblinti analüüsi vahetulemuste paremaks kuvamiseks eriline abstraktseks siluriks. Silur (*debugger*) on tööriist, mis võimaldab jooksutada programmi sammhaaval ning vaadelda programmi seisundit. Abstraktne silur seevastu on tööriist, mis võimaldab simuleerida programmi sammhaaval jooksutamist kasutades Goblinti analüüsi vahetulemusi ja vaadelda programmi seisundit kirjeldavaid abstraktseid mudeleid. Abstraktse siluri loomisega on soov kasutada ära kasutajate olemasolevaid siluri kasutamise oskusi, et teha analüüsi vahetulemustes orienteerumine lihtsamaks.

Töö on jaotatud neljaks peatükiks. Esimeses peatükis antakse töö mõistmiseks vajalik taust. Teisese peatükis selgitatakse abstraktse siluri olemust ning antakse ülevaade varasematest töödest selles valdkonnas. Kolmandas peatükis kirjeldatakse abstraktse siluri funktsionaalsusi ja nende tehnilist teostust. Neljandas peatükis antakse ülevaade töö tulemustest ja kasutajatelt saadud tagasisidest ning tuuakse välja teadaolevad probleemid ja võimalikud edasised arendused.

[1] https://goblint.in.tum.de/

[2] https://github.com/goblint/analyzer

1. Taust

Selles peatükis antakse ülevaade teoreetilisest taustast, mis on vajalik töö mõistmiseks.

1.1 Abstraktne interpretatsioon

Goblinti analüüs tugineb **abstraktsel interpretatsioonil** (*abstract interpretation*). Abstraktse interpretatsiooni keskne idee on mudeldada programmi jooksmist kasutades programmi konkreetsete olekute asemel abstraktseid olekuid, mis lähendavad võimalike konkreetsete olekute hulki [5]. Tinglikult võib abstraktsest interpretatsioonist mõelda kui programmi jooksutamisest väärtuste üldistustel, mis mudeldavad võimalike konkreetseid väärtusi.

Abstraktses interpretatsioonis kasutatavaid olekute mudeleid nimetatakse **abstraktseteks domeenideks** (*abstract domains*). Abstraktne domeen on programmi või mingi programmi osa (näiteks muutuja) võimalike seisundite lähenduste hulk [6]. Lihtne näide abstraktsest domeenist on intervalldomeen, mis mudeldab võimalike täisarvulisi väärtusi arvtelje lõiguna [a; b], kus täisarvud a ja b on vastavalt vähim ja suurim võimalik väärtus.

Abstraktse interpretatsiooni paremaks mõistmiseks on siinkohal toodud näide abstraktsest interpretatsioonist intervalldomeeniga lihtsal C-keelsel programmil. Joonisel 1 on toodud analüüsitava programmi C-keelne lähtekood koos abstraktse interpretatsiooni tulemusena leitud muutujate võimalike väärtusi kirjeldavate intervallidega igas programmi punktis.

```
        int main() {
a)              int u = rand() % 100;           u = [0;99]
b)              int r = 0;                      u = [0;99], r = [0;0]
                if (u < 10) {
c)                      r = 2 * u;              u = [0;9], r = [0;18]
                } else {
d)                      r = u - 200;            u = [10;99], r = [-190;-101]
                }
e)              return r;                       u = [0;99], r = [-190;18]
        }
```

Joonis 1. Programmi lähtekood ja abstraktse interpretatsiooni tulemusena leitud muutujate võimalikud väärtused. Kasutatud funktsioon `rand` tagastab suvalise mittenegatiivse täisarvu.

Abstraktse interpretatsioonil on kaks olulist omadust [7, 8]:

- Abstraktne interpretatsioon on korrektne, st kui mingi olek on abstraktses interpretatsioonis võimatu, siis on ka vastav olek programmi jooksutamisel võimatu. Näiteks real c) on abstraktses interpretatsioonis võimatu, et muutuja `u` väärtus on 11 ning see on ka programmi jooksutamisel võimatu.
- Abstraktne interpretatsioon võib olla ebatäpne, st olekud, mis on abstraktses interpretatsioonis võimalikud, ei pruugi olla võimalikud programmi jooksutamisel. Näiteks real e) on võimalike muutuja `r` väärtuste hulka loetud −1, kuigi selline väärtus programmi jooksutamisel võimalik ei ole. Kummaski tingimuslause harus ei saa muutuja `r` väärtus olla −1, seega ei saa ka real e) muutuja väärtus olla −1, kuid täpset võimalike väärtuste hulka real e) pole võimalik ühe arvtelje lõiguna esitada, seega lähendatakse seda lõiguga kuhu kuulub ka −1.

1.2 Andmevooanalüüs ja juhtvoograaf

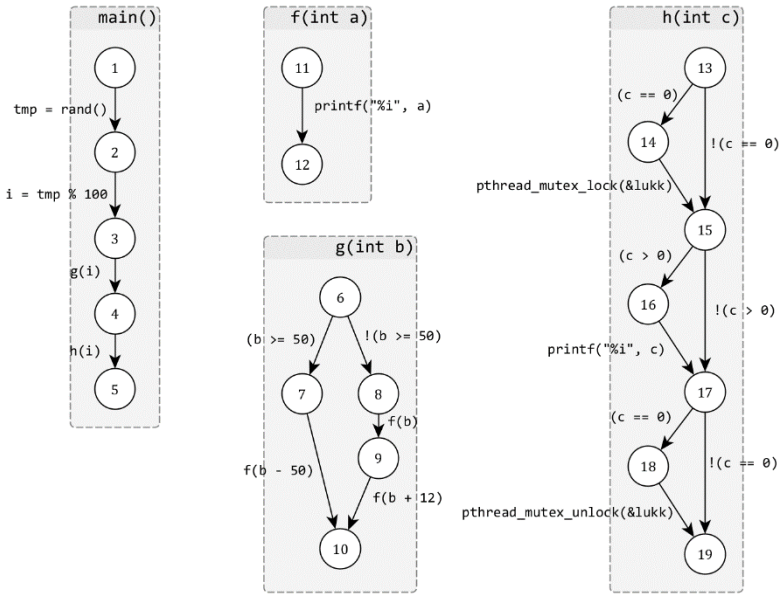
Abstraktse interpretatsiooni teostamiseks kasutab Goblint **andmevooanalüüsi**. Andmevooanalüüsi idee on koostada kõigist programmi poolt täidetavatest käskudest võrrandisüsteem, mis kirjeldab, kuidas need käsud programmi olekut muudavad. Seejärel on võimalik võrrandisüsteemi lahendades leida programmi olek igas programmi punktis. Võrrandisüsteemis olekute vahelisi üleminekuid kirjeldavaid funktsioone nimetatakse **üleminekufunktsioonideks** (*transfer functions*). Abstraktse interpretatsiooni puhul kasutatakse programmi konkreetsete olekute asemel abstraktse domeeni elemente, mis neid lähendavad ning seega on üleminekufunktsioonide lähte- ja sihthulkadeks abstraktsed domeenid [7].

Programmi juhtvoo mudeldamiseks andmevooanalüüsis tarvis kasutab Goblint juhtvoograafi. **Juhtvoograaf** (*control-flow graph*) on suunatud graaf mille tipud tähistavad programmi erinevaid seisundeid ning servad tähistavad üleminekuid nende vahel [6].

Andmevooanalüüsiga seonduvate ideede paremaks mõistmiseks vaadatakse edaspidi illustreerivaid näiteid C-keelsel näidisprogrammil. Joonisel 2 on toodud näidisprogrammi lähtekood. Joonisel 3 on toodud Goblinti poolt loodud näidisprogrammi funktsioonide juhtvoograafid lihtsustatud kujul. Goblint loob igale funktsioonile eraldi juhtvoograafi. Funktsioonidevahelisi servi Goblinti juhtvoograaf ei sisalda.

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t lukk =
    PTHREAD_MUTEX_INITIALIZER;

void f(int a) {
    printf("%i", a);
}

void g(int b) {
    if (b >= 50) {
        f(b - 50);
    } else {
        f(b);
        f(b + 12);
    }
}

void h(c) {
    if (c == 0) {
        pthread_mutex_lock(&lukk);
    }
    if (c > 0) {
        printf("%i", c);
    }
    if (c == 0) {
        pthread_mutex_unlock(&lukk);
    }
}

int main() {
    int i = rand() % 100;
    g(i);
    h(i);
}
```

Joonis 2. Näidisprogrammi C-keelne lähtekood.



Joonis 3. Näidisprogrammi funktsioonide juhtvoograafid.

1.3 Goblinti analüüsid

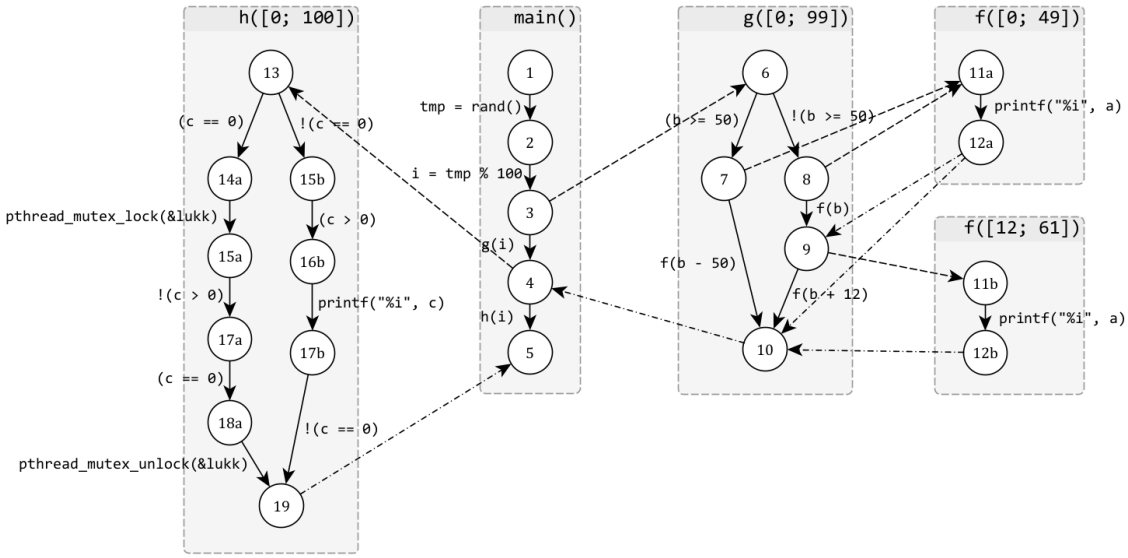
Goblint koosneb paljudest erinevatest analüüsidest, mis suhtlevad omavahel, esitades analüsaatori sees üksteisele üldise liidese kaudu päringuid. Iga analüüs määratleb oma abstraktse domeeni, mis mudeldab analüüsile huvipakkuvaid programmi omadusi, ning vastavad üleminekufunktsioonid, mis kirjeldavad programmi oleku muutumist selles abstraktses domeenis. Erinevaid analüüse on võimalik vastavalt vajadusele sisse ja välja lülitada [9].

Üheks põhiliseks Goblinti poolt kasutatavaks analüüsiks on nn baasanalüüs, mis analüüsib programmis esinevate muutujate võimalike väärtusi. Baasanalüüsi abstraktne domeen kombineerib eri tüüpi muutujate lähendamiseks mitmeid erinevaid abstraktseid domeene sh ka intervalldomeeni [1]. Lisaks baasanalüüsile on Goblintis muid analüüse, mis mudeldavad programmi olekut, näiteks võetud lukuhulkade analüüs, mis jälgib lukke, mis on mingis programmi punktis kindlasti võetud [4]. Samuti on Goblintis palju analüüse, mis keskenduvad konkreetsete vigade leidmisele, näiteks nullviitade analüüs, mis tuvastab olukordi, kus võib toimuda nullviida väärtuse lugemine (*null pointer dereference*), ja algväärtustamata muutujate analüüs, mis tuvastab olukordi, kus loetakse potentsiaalselt algväärtustamata muutujate väärtusi [1].

1.4 Abstraktne saavutatavusgraaf

Andmevooanalüüsi tulemuste esitamiseks on mitmeid viise. Üks võimalus, mida Goblint toetab, on siduda analüüsi tulemused abstraktse saavutatavusgraafi tippudega.

Abstraktne saavutatavusgraaf (*abstract reachability graph*, edaspidi lühendatult saavutatavusgraaf**)**, on suunatud graaf, mille tipud tähistavad programmi poolt saavutatavaid olekuid sobival abstraktsioonitasemel ning servad võimalike üleminekuid nende vahel [10]. Erinevalt juhtvoograafist sisaldab saavutatavusgraaf ka funktsioonidevahelisi üleminekuid. Joonisel 4 on näidatud Goblinti poolt koostatud näidisprogrammi saavutatavusgraaf lihtsustatud kujul. Goblint koostab saavutatavusgraafi läbides analüüsi käigus juhtvoograafi tippe, seega vastab igale saavutatavusgraafi tipule mingi juhtvoograafi tipp. Erinevalt juhtvoograafist on saavutatavusgraafist eemaldatud olekud, mille kohta analüüs tuvastab, et need ei ole saavutatavad. Näiteks näidisprogrammi meetodis `h`, kui kehtib `c == 0`, siis on kindel, et `c > 0` ei saa kehtida, ning seega vastava tingimuslause tõest haru saavutatavusgraafis ei ole.



Joonis 4. Näidisprogrammi abstraktne saavutatavusgraaf.

Saavutatavusgraafi koostamisel mängivad samuti olulist rolli kaks Goblinti andmevooanalüüsi omadust:

- Analüüs on kontekstitundlik (*context sensitive*), st funktsioonide väljakutseid eristatakse parameetrite väärtuste põhjal vaadeldavas abstraktses domeenis [7]. Näiteks näidisprogrammis analüüsitakse funktsiooni `f` kaks korda, kuna seda kutsutakse välja kahe erineva parameetri `a` väärtusega. Seetõttu esineb funktsioon `f` ka saavutatavusgraafis kaks korda.
- Analüüs on rajatundlik (*path sensitive*), st teatud tingimustel eristatakse analüüsis erinevaid programmiradu [1]. Näiteks kui programmis võetakse tinglikult mingi lukk, siis eristatakse programmirada, kus lukk on võetud, programmirajast, kus lukk ei ole võetud. Rajatundlikust on näha näidisprogrammi saavutatavusgraafis funktsiooni `h` puhul, kus analüüsitakse olukorda, kus `c == 0` ning seega lukk võetakse, eraldi olukorrast, kus `c != 0` ning seega lukku ei võeta.

2. Kontseptsioon

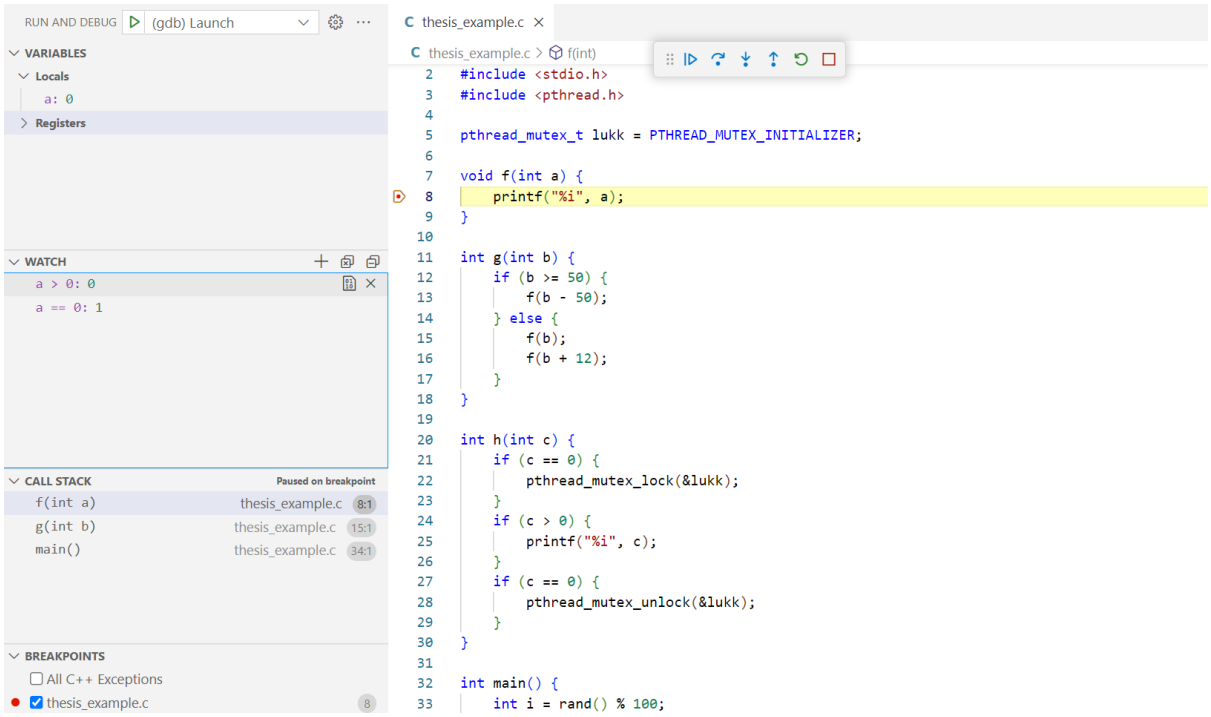
Selles peatükis selgitatakse abstraktse siluri olemust ning antakse ülevaade asjakohastest varasematest töödest.

2.1 Abstraktse siluri olemus

Abstraktse siluri olemuse mõistmiseks on oluline kõigepealt määratleda mis on silur. Silur (*debugger*) on tööriist, mis võimaldab jooksutada programmi samm-sammult ning uurida programmi seisu iga sammu järel. Peaaegu kõik programmeerimiskeskkonnad toetavad mingil kujul silureid ning peaaegu kõigi populaarsete programmeerimiskeelte jaoks on olemas silur. Joonisel 5 on näidatud C keele siluri GDB kasutajaliides programmeerimis-keskkonnas Visual Studio Code.

Siluris mängivad keskset rolli katkepunktid (*breakpoints*) ja sammumine (*stepping*). Katkepunktid võimaldavad märkida koha programmi lähtekoodis, mille juures silur peatub ning võimaldab kasutajal koodi sammhaaval jooksutada. Koodi sammhaaval jooksutamist nimetataksegi sammumiseks.

Sammumise ajal võimaldab silur uurida programmi olekut. Selleks on siluris võimalik näha muutujate hetkeväärtusi (joonisel vasak ülemine paneel), väärtustada avaldisi (joonisel vasak keskmine paneel) ning uurida väljakutsete pinu (*call stack*) (joonisel vasak alumine paneel).



Joonis 5. Siluri GDB kasutajaliides Visual Studio Code programmeerimiskeskkonnas.

Abstraktne silur toimib analoogselt tavalisele silurile ja kasutab sama kasutajaliidest, kuid selle asemel, et vaadelda jooksva programmi olekuid, kasutab abstraktne silur Goblinti abstraktsel interpretatsioonil põhineva analüüsi tulemusi, et kuvada abstraktsed olekud, mis mudeldavad tegeliku programmi võimalike olekuid ilma programmi jooksutamata.

Tavaline silur võimaldab kasutajal jooksutada samm-sammult programmi poolt täidetavaid käske ning vaadelda samal ajal programmi olekut. Abstraktne silur võimaldab analoogselt läbida programmi sammhaaval, kuid programmi jooksutamise asemel liigub abstraktne silur mööda Goblinti poolt loodud abstraktset saavutatavusgraafi, jäljendades seeläbi programmi jooksutamisel käskude täitmist. Sammumise ajal võimaldab abstraktne silur analoogselt tavalisele silurile vaadelda muutujate ning avaldiste väärtusi, kuid erinevalt tavalisest silurist kuvatakse konkreetsete väärtuste asemel võimalike väärtuste lähendused Goblinti baasanalüüsi abstraktses domeenis.

2.2 Varasemad tööd

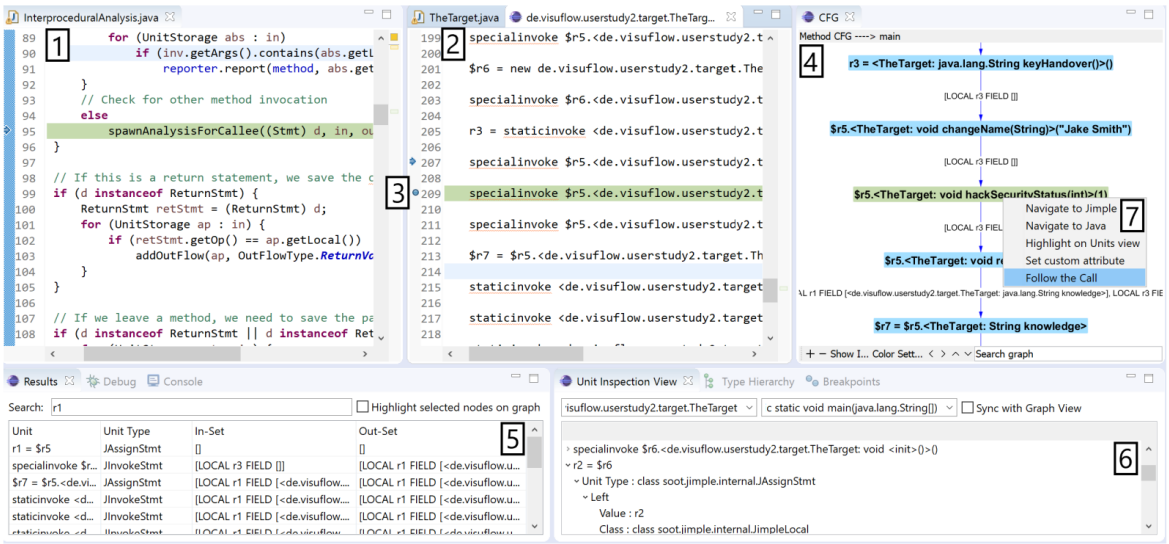
Autorile teadaolevalt on abstraktsel interpretatsioonil põhineva analüüsi kasutamine, et jäljendada tavalise siluri toimimist uudne idee, mida pole varem sellisel kujul realiseeritud. Selles töös kasutatavad terminid abstraktne silur (*abstract debugger*) ja abstraktne silumine (*abstract debugging*) on mõeldud välja selle töö jaoks. Programmianalüüsiga seonduvas kirjanduses neid termineid ei kasutata või kasutatakse mingis teises tähenduses [11, 12].

Kuna sellise lähenemise kirjeldamiseks puudub üldlevinud terminoloogia, siis on varasemalt loodud abstraktsete silurite olemasolu raske välistada.

Selles peatükis tutvustatakse varasemaid töid, mis on oma olemuselt loodava abstraktse siluriga sarnased (VisuFlow) või täidavad Goblinti jaoks sarnast eesmärki (g2html ja GobView).

2.2.1 VisuFlow

Üks tööriist, mis kasutab abstraktsele silurile sarnast ideed on VisuFlow. VisuFlow[3] [13] on pistikprogramm programmeerimiskeskkonnale Eclipse, mis võimaldab siluda Java keele staatilise analüsaatori Soot analüüse. Joonisel 6 on näidatud VisuFlow kasutajaliides. VisuFlow võimaldab sammuda samaaegselt nii analüüsi koodis (joonisel märge 1) kui ka analüüsitava programmi koodis (joonisel märge 2), ning uurida samal ajal nii analüüsi tulemusi vaadeldavas programmi punktis (joonisel märge 5) kui ka analüüsi koodi enda seesmist olekut. Erinevalt loodavast abstraktsest silurist ei ürita VisuFlow jäljendada tavalist silurit. Silumisel esitatakse analüüsitav programm mitte lähtekoodina, vaid kasutades vahepealset esitust programmi koodist nimega Jimple, mis on analüsaatori sisendiks (joonisel märked 2 ja 6), ning programmi meetodite juhtvoograafe (joonisel märge 4).



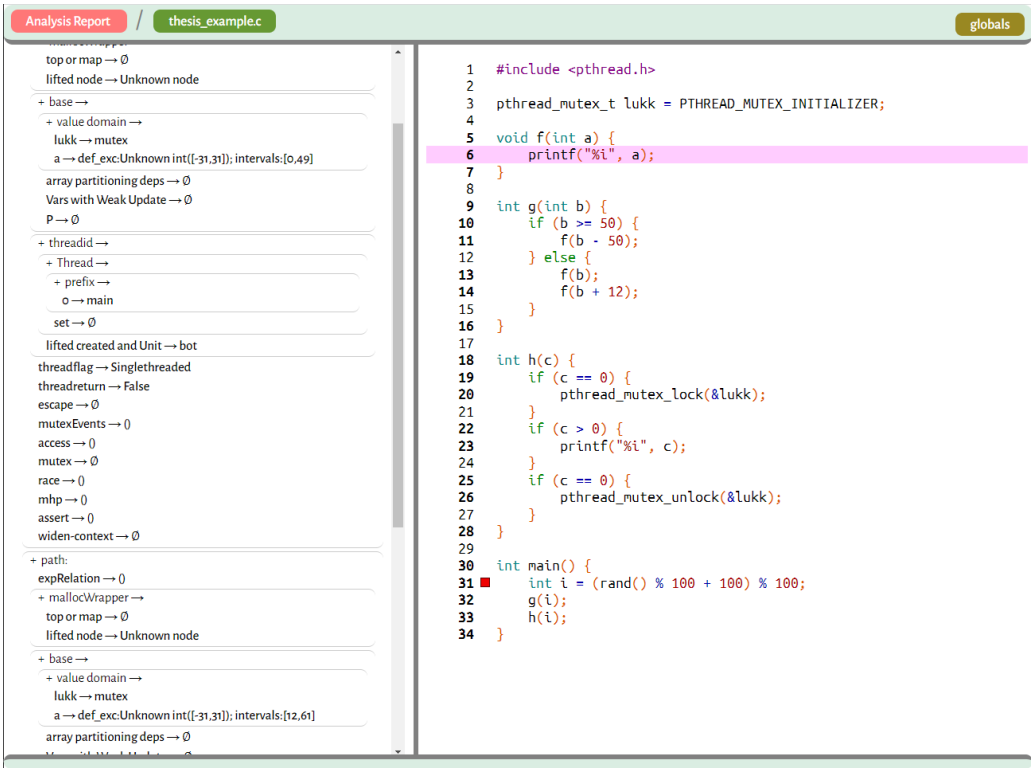
Joonis 6. VisuFlow kasutajaliides [13].

[3] https://github.com/VisuFlow

2.2.2 g2html ja GobView

Goblinti analüüsi tulemusena leitud programmi olekute kuvamiseks on varasemalt loodud kaks tööriista: g2html[4] ja GobView[5]. See peatükk keskendub g2html-ile, kuna see on kahest tööriistast rohkem kasutatud, kuid GobView tööpõhimõte ja ülesehitus on analoogsed. Põhiliseks erinevuseks on, et GobView'l on modernsem kasutajaliides ja võimekas otsingu süsteem, mis võimaldab leida erinevaid analüüsi tulemuste elemente, näiteks kindlatele tingimustele vastavaid muutujate väärtustamisi, funktsioonide väljakutseid jms.

G2html võimaldab vaadelda Goblinti analüüsi tulemusi teisendades Goblinti poolt XML kujul väljastatud tulemused interaktiivseks HTML dokumendiks, mida saab avada veebibrauseris. Joonisel 7 on näidatud g2html-i kasutajaliidese põhivaade näidisprogrammi analüüsitulemustega. G2html seob analüüside tulemusena leitud programmi abstraktsed olekud juhtvoograafi tippudega. Joonisel parempoolsel paneelil on programmi lähtekood, kus saab vastavale reale vajutades valida soovitud juhtvoograafi tipu. Vasakpoolsel paneelil on erinevate kontekstide ja programmiradade abstraktsed olekud antud juhtvoograafi tipus.



Joonis 7. g2html-i kasutajaliidese põhivaade.

[4] https://github.com/goblint/g2html

[5] https://github.com/goblint/gobview

G2html on tööriist mis on ennekõike mõeldud Goblinti arendajatele. Kasutajal, kes Goblinti analüüse ning nende poolt kasutatavaid abstraktseid domeene ei tunne, on kuvatud tulemusi küllaltki keeruline lahti mõtestada. Kuna analüüsi tulemused on seotud juhtvoograafi tippudega, siis on konteksti- ja rajatundliku analüüsi puhul keeruline leida, milline teekond läbi programmi viis mingi konkreetse konteksti või programmirajani.

3. Tehniline teostus

See peatükk annab ülevaate valminud abstraktse siluri tehnilisest teostusest ning olulisematest arenduse käigus langetatud otsustest.

3.1 Ülevaade

Selle töö raames loodi abstraktse siluri esialgne versioon. Abstraktne silur on realiseeritud osana Visual Studio Code'i laiendusest GobPie. GobPie[6] on kasutajaliides Goblintile, mis võimaldab analüüsida Goblintiga C koodi ning kuvada analüüsi käigus leitud potentsiaalsed programmivead koodiredaktoris. GobPie tugineb raamistikul MagpieBridge[7], mis on üldine lahendus staatiliste analüsaatorite tulemuste kuvamiseks programmeerimiskeskkondades. Hetkel toetab GobPie ja seega ka loodud abstraktne silur ainult VS Code'i, kuid tulevikus on võimalik tuge laiendada ka teistele MagpieBridge'i poolt toetatud programmeerimis-keskkondadele.

Abstraktse silur loodi osana GobPie'st, kuna see võimaldab taaskasutada GobPie's olemasolevat loogikat Goblintiga suhtlemiseks ning analüüside käitamiseks. Loodud abstraktne silur on kirjutatud programmeerimiskeeles Java, kuna see on GobPie arendamisel kasutatav programmeerimiskeel.

Kuna GobPie ja Goblint on rahvusvahelised projektid, siis on nende kasutajaliidesed ingliskeelsed. Sellest tulenevalt on ka loodud abstraktse siluri kasutajaliides inglise keeles.

Loodud siluri lähtekood on avalikult kättesaadav ning juhised sellele ligi pääsemiseks on lisas I. Selle töö raames toimus arendus autori enda koopial GobPie koodist, kuid tulevikus on plaanis viia arendus üle GobPie ametliku koodivaramusse, kus abstraktne silur oleks kättesaadav kõigile GobPie kasutajatele.

3.2 Kasutatud tehnoloogiad

Järgnevalt tutvustatakse abstraktse siluri loomisel kasutatud tehnoloogiaid ning antakse ülevaade nendega seotud tehnoloogilistest valikutest.

[6] https://github.com/goblint/GobPie

[7] https://github.com/MagpieBridge/MagpieBridge

3.2.1 JSON-RPC

JSON-RPC[8] (*JavaScript Object Notation Remote Procedure Call*) on kaugprotseduuri-väljakutsete protokoll, st protokoll välises protsessis protseduuride välja kutsumiseks. Väljakutsete vahendamiseks saab kasutada ükskõik millist suhtluskanalit, mis võimaldab samaaegselt saata ning vastu võtta tekstivoogu. Päringute ning vastuste kodeerimiseks kasutatakse laialt levinud andmevahetusvormingut JSON.

JSON-RPC-d kasutavad suhtlusprotokollina DAP (vt peatükk 3.2.3) ja Goblinti serverirežiim (vt. peatükk 3.2.4).

3.2.2 IPC soklid

IPC sokkel (*inter-process communication socket*, otsetõlkes protsesside vahelise suhtluse sokkel, osades allikates ka *UNIX domain socket*) on üks võimalik suhtluskanal JSON-RPC sõnumite vahendamiseks. IPC soklid kasutavad sama rakendusliidest, mis võrgusuhtluseks kasutatavad soklid, kuid erinevalt võrgusuhtluseks kasutatavatest soklitest kasutavad IPC soklid adresseerimiseks eriotstarbelisi faile [14]. See tähendab, et IPC sokleid saab üldjuhul kasutada ainult suhtluseks kahe samas arvutis jooksva protsessi vahel. See piirab küll võimalike kasutusvaldkondi, kuid sellel on kaks olulist eelist:

- See teeb suhtluskanali turvalisemaks, kuna on tagatud, et ainult samas füüsilises seadmes jooksvad protsessid saavad IPC sokliga ühenduda.
- IPC sokleid saab kasutada ka seadmetes, kus võrguliides puudub või on ligipääs sellele tulemüüri poolt piiratud.

3.2.3 DAP

DAP (*Debug Adapter Protocol*, otsetõlkes siluriadapteri protokoll) on protokoll mis võimaldab programmeerimiskeskkondadel suhelda erinevate siluritega kasutades ühtset, standardiseeritud liidest [15]. DAP-i toetavad muuhulgas Visual Studio Code, Eclipse, Vim ja Emacs [16]. DAP-i spetsifikatsioon on võrgus avalikult saadaval[9] ja on põhiline allikas, millel DAP-i kohta esitatud väited selles töös tuginevad.

---

[8] https://www.jsonrpc.org/specification

[9] https://microsoft.github.io/debug-adapter-protocol/overview, https://microsoft.github.io/debug-adapter-protocol/specification

DAP silur toimib serverina, mille külge klient (enamasti programmeerimiskeskkond) ühendub. Serverile erinevaid päringuid saates saab klient anda silurile käske ning hankida infot silutava programmi oleku kohta. Suhtlus siluri ning arenduskeskkonna vahel toimub JSON-RPC päringute kaudu.

Selles töös valminud abstraktne silur kasutab DAP-i arenduskeskkonnaga suhtlemiseks. Sellisel lähenemisel on võrreldes ise silurile graafilise kasutajaliidese loomisega kaks olulist eelist:

- Silurit on võimalik väga vähese vaevaga liidestada suvalise DAP-i toetava programmeerimiskeskkonnaga.
- Silur saab kasutada programmeerimiskeskkonda sisse ehitatud siluri kasutajaliidest. See säästab siluri loomisel arendusvaeva. Samuti, kui kasutaja on sisseehitatud kasutajaliidesega juba tuttav, siis ei pea ta siluri kasutamiseks uut kasutajaliidest selgeks õppima.

Samas kaasneb sellega ka üks oluline miinus – kuna silur kasutab arenduskeskkonna sisseehitatud kasutajaliidest, siis tähendab see, et võimalused kasutajaliidest vastavalt arendusvajadustele ümber kohandada on piiratud.

Protokollis kasutatavad sõnumitüübid ja üldine suhtlusloogika on Javas realiseeritud teegis **lsp4j**[10], mida selles töös kasutatakse.

3.2.4 Goblinti serverirežiim

Goblintil on sisseehitatud serverirežiim[11], mis võimaldab JSON-RPC liidese kaudu Goblintile interaktiivselt päringuid esitada. Serverirežiimi liidese kaudu on võimalik jooksutada analüüse ja pärida erinevaid analüüside tulemusi. Muuhulgas on serverirežiimis võimalik väärtustada C-keelseid avaldisi baasanalüüsi abstraktses domeenis, st on võimalik leida C-keelse avaldise võimalike väärtuste lähendus mingis kindlas abstraktse saavutatavusgraafi tipus.

Selles töös kasutatakse Goblinti serverirežiimi Goblintiga liidestumiseks. Serverirežiim valiti selle töö jaoks osalt selle tõttu, et GobPie juba kasutas serverirežiimi ning osalt selle

[10] https://github.com/eclipse-lsp4j/lsp4j

[11] https://github.com/goblint/analyzer/pull/522

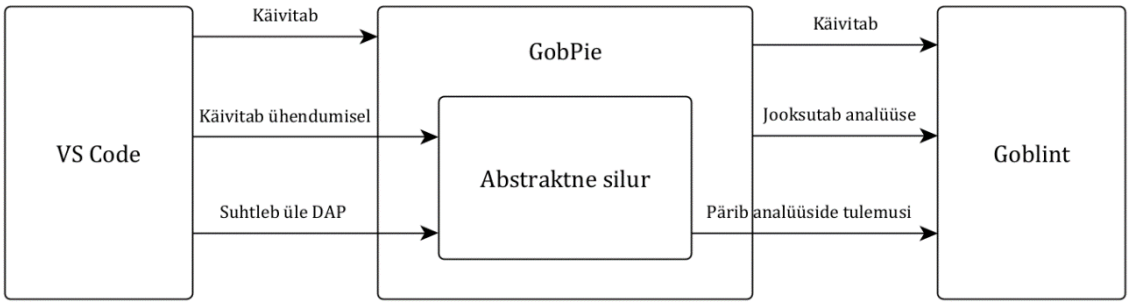
tõttu, et see on hetkel ainus Goblinti rakendusliides, mis võimaldab analüüsi tulemuste põhjal avaldisi väärtustada.

3.3 Arhitektuur

Järgnevalt antakse ülevaade abstraktse siluri süsteemi ülesehitusest ning komponentide omavahelisest suhtlusest.

Abstraktne silur on realiseeritud DAP serverina, mis käivitatakse GobPie käivitumisel eraldi lõimes. Siluri DAP serveriga ühendumisel käivitatakse silumissessioon.

Abstraktse siluri toimimiseks on vajalik GobPie'st ning sellega suhtlevatest komponentidest koosnev süsteem. Joonisel 8 on kujutatud süsteemi eri osade vahelist suhtlust. Süsteem koosneb kolmest erinevast protsessist: VS Code, GobPie ja serverirežiimis Goblint. VS Code käivitab GobPie laienduse, mis jookseb eraldi Java protsessis ning suhtleb VS Codeiga. GobPie käivitab omakorda serverirežiimis Goblinti ning jooksutab vastavalt hetkel avatud kataloogis olevatele seadistusfailidele analüüsi. Samaaegselt käivitatakse GobPie sees eraldi lõimes abstraktse siluri server, mis loob IPC sokli, millega silurit kasutav programm saab ühenduda. Kui käivitada VS Code'is C keelse lähtekoodi silumine ning valida võimalike silurite hulgast abstraktne silur, siis ühendub VS Code abstraktse siluri serveriga ning käivitab siluri. Selle peale loob abstraktse siluri server uue siluri objekti, mis säilitab endas kogu abstraktse siluri olekut ning millele edastatakse DAP päringud, mis VS Code silumise käigus esitab. Abstraktne silur ja GobPie suhtlevad sama Goblinti serveri protsessiga. See võimaldab siluril kasutada GobPie poolt eelnevalt sooritatud analüüside tulemusi.



Joonis 8. Süsteemi osade omavaheline suhtlus.

3.4 Realiseeritud funktsionaalsused

Selles alampeatükis antakse ülevaade loodud abstraktse siluris realiseeritud funktsionaalsustest. Kirjeldatakse, kuidas on funktsionaalsused kättesaadavad siluri kasutajaliideses ning selgitatakse funktsionaalsuste tehnilist teostust.

Selles peatükis kasutatud näidete lähtekood ning Goblinti seadistus on leitavad abstraktse siluri lähtekoodis kaustast *adb_examples*. Juhised lähtekoodi hankimiseks on lisas I.

3.4.1 Katkepunktid ja silumislõimed

Silumiseks huvipakkuva koha valimiseks on võimalik määrata programmi lähtekoodis katkepunktid. Silumise käivitamisel peatub abstraktne silur iga katkepunkti juures ning võimaldab kasutajal läbida juhtvoogu samm-sammult ja vaadelda programmi olekut.

Katkepunkti juures peatumiseks leiab Goblint saavutatavusgraafist tipud, mis vastavad selle katkepunkti asukohale lähtekoodis. Kuna Goblinti analüüs on teekonna- ja kontekstitundlik, siis võib ühele katkepunktile vastata mitu erinevat saavutatavusgraafi tippu, mis tavalise siluri terminoloogias tähendab, et sama katkepunkti läbitakse mitu korda. Tavalises siluris lahendatakse olukorrad, kus sama katkepunkti läbitakse jooksutamisel mitu korda sellega, et silur peatub iga kord kui katkepunkti läbitakse. Sellega kaasneb oluline miinus – nimelt kui silur tabab ühte katkepunkti palju kordi, siis on kasutajal raske leida endale huvipakkuv juhtum, kuna iga katkepunkti juures peatumist saab vaadelda ainult ühe korra ning seejärel peab tegema otsuse, kas see konkreetne juht on huvitav ilma võimaluseta järgnevaid juhte näha. Huvipakkuvate juhtude leidmise lihtsustamiseks kuvab abstraktne silur kõik samale katkestuspunktile vastavates saavutatavusgraafi tippudes korraga. Siluri kasutajaliideses kuvatakse erinevad saavutatavusgraafi tipud lõimedena, kust kasutaja saab kiiresti leida endale huvipakkuva(d) olekud. Sama katkepunkti eri olekute kuvamiseks on kasutusele võetud lõimed, kuna DAP ei võimalda muud viisi esitada rohkem kui ühte programmi olekut korraga. Kuna Goblint käsitleb analüüsitava programmi lõimi kasutades abstraktseid domeene, mis mudeldavad teiste lõimede olemasolu, mitte simuleerides samaaegselt mitme lõime jooksmist [7], siis ei ole lõimede vaade analüüsitava programmi lõimede kuvamiseks vajalik ning seega on võimalik seda nö väärkasutada katkepunkti eri olekute kuvamiseks. Edaspidi nimetatakse katkepunkti eri olekute esitamiseks kasutatavaid lõimi segaduse vältimiseks silumislõimedeks.

Lisaks tavalistele katkepunktidele on abstraktses siluris toetatud ka tingimuslikud katkepunktid (*conditional breakpoints*), mis peatuvad ainult nende saavutatavusgraafi tippude juures, mis vastavad seatud tingimusele. Tingimuseks on C-keelne avaldis, mis väärtustatakse vastavas saavutatavusgraafi tipus. Avaldises saab kasutada analüüsitavas programmis esinevaid muutujaid, et kontrollida soovitud tingimuse kehtimist vastavas programmipunktis. Lisaks sellele on võimalik valida kahe tingimuse kontrollimise režiimi vahel:

- Tingimus kehtib, kui tõene tõeväärtus kuulub vaadeldavas olekus avaldise võimalike väärtuste hulka (režiimitähis `\may`).
- Tingimus kehtib, kui tõene tõeväärtus on vaadeldavas olekus avaldise ainus võimalik väärtus (režiimitähis `\must`).

Režiimi saab valida pannes avaldise ette vastava režiimitähise. Näiteks tingimus `\must a > 0` kehtib, kui muutuja `a` väärtus on vaadeldavas saavutatavusgraafi tipus alati suurem kui 0. Langkriips režiimitähise alguses on lisatud selleks, et režiimitähis oleks selgelt eristuv C-keelsest avaldisest. Kui režiimitähis puudub kasutatakse vaikimisi režiimi `\may`.

Tingimuste kontroll on realiseeritud kasutades Goblinti serverirežiimi võimet väärtustada C-keelseid avaldisi. Igas katkepunktile vastavas saavutatavusgraafi tipus väärtustatakse antud avaldis baasanalüüsi abstraktses domeenis ning kontrollitakse avaldise võimalike väärtusi vastavalt valitud režiimile.

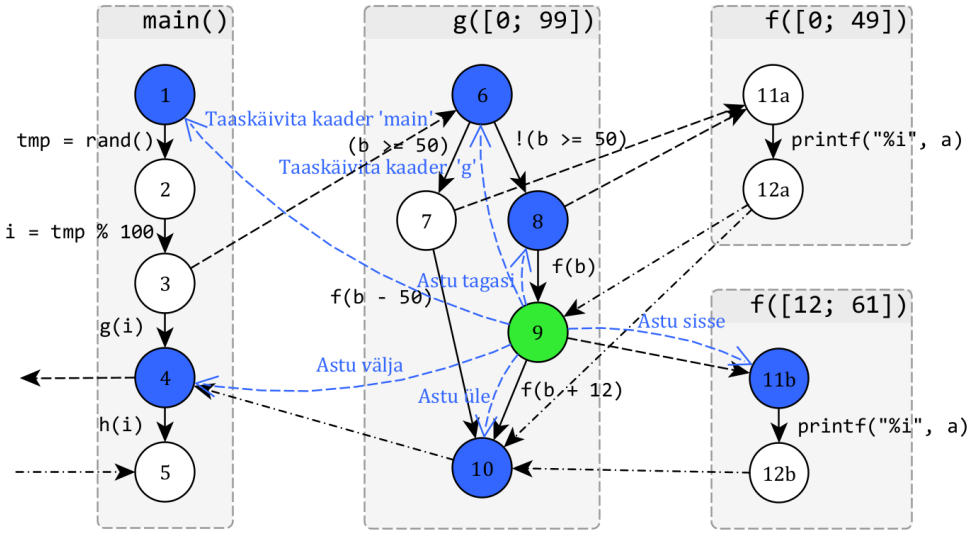
3.4.2 Sammumine

Silurite üks põhilisi funktsioone on sammumine ehk programmi käskude sammhaaval jooksutamine. Abstraktses siluris on toetatud järgnevad sammumisoperatsioonid:

- Astu üle (*step over*). Jooksutab järgmise käsu. Kui järgmiseks käsuks on funktsiooni väljakutse, siis jooksutatakse kogu väljakutse ühe sammuna.
- Astu tagasi (*step back*). Toimib analoogselt üle astumisele, kuid järgmise käsu jooksutamise asemel võetakse tagasi eelmise käsu jooksutamine;
- Astu sisse (*step into*). Kui järgmiseks käsuks on funktsiooni väljakutse, siis liigub välja kutsutud funktsiooni esimese käsuni. Vastasel juhul toimib samamoodi nagu astu üle.

- Astu välja (*step out*). Jooksutab funktsiooni väljakutse lõpuni ning peatub välimises funktsioonis, kohas, kuhu juhtvoog jõuab funktsioonist tagastamisel.
- Taaskäivita kaader (*restart frame*). Liigub valitud pinukaadrile (*stack frame*) vastava väljakutse algusesse.

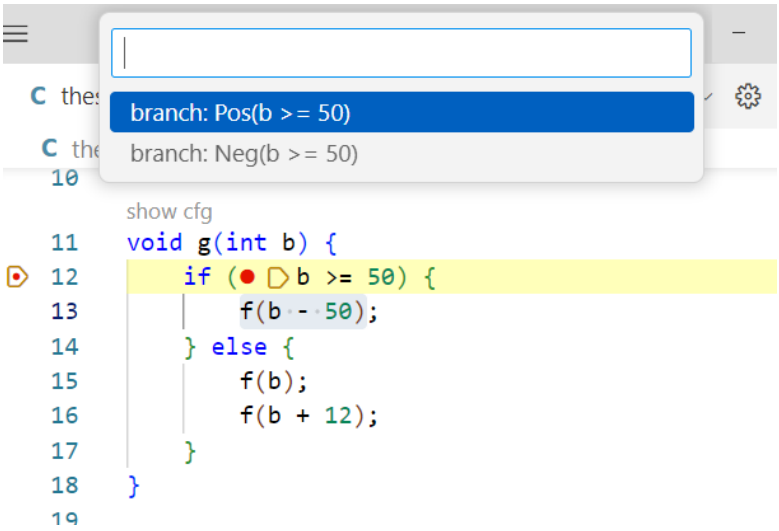
Abstraktses siluris sooritatakse need operatsioonid liikudes programmi käskude jooksutamise asemel abstraktse saavutatavusgraafi tippude vahel. Joonisel 9 on näidatud sammumisoperatsioonide sihtpunktid näidisprogrammi saavutatavusgraafil.



Joonis 9. Sammumisoperatsioonid saavutatavusgraafil. Praegune tipp on tähistatud rohelisega ja sammumisoperatsioonide sihttipud sinisega.

Tavalise programmi jooksutamise käigus läbitakse programm ainult ühte teed pidi, st näiteks tingimuslause korral läbitakse alati kindel tingimuslause haru ning funktsiooni väljakutsete korral kutsutakse alati välja üks kindel funktsioon. Abstraktses interpretatsioonis aga tuleb ette olukordi, kus ei ole üheselt määratud, milline tingimuslause haru läbitakse või mis funktsioon välja kutsutakse. Sellised olukorrad on probleemiks abstraktses siluris sammumise realiseerimisel, kuna siluri kasutajaliides ei võimalda läbida mitut erinevat tingimuslause haru või funktsiooni väljakutset paralleelselt.

Selles töös lahendatakse probleem sellega, et kasutaja peab käsitsi valima, millisesse tingimuslause harusse või funktsiooni väljakutsesse ta soovib siseneda. VS Code'is on see realiseeritud kasutades *Step into targets* käsku. Joonisel 10 on näidatud kasutajaliidest mis võimaldab kasutajal valida kahe tingimuslause haru vahel.



Joonis 10. Siluri kasutajaliides sammumisel eri harude vahel valimiseks.

Kui samal katkepunktil on vaatluse all mitu erinevat silumislõime, siis peab kasutaja sammumiseks valima neist ühe. Eri olekute samaaegse haldamise lihtsustamiseks on abstraktses siluris sammumine sünkroniseeritud, st kui ühes silumislõimes tehakse samm, siis tehakse kõigis teistes vaatluse all olevates silumislõimedes ekvivalentne samm. Ekvivalentseks sammuks loetakse samm, mille sihtpunktiks on sama juhtvoograafi tipp. Kui ekvivalentne samm pole võimalik, kuna vajalik juhtvoograafi tipp pole saavutatav, siis muutub see olek kättesaadamatuks. See tagab, et kõik vaatluse all olevad olekud on alati samas juhtvoograafi tipus ning on läbinud sammumisel ekvivalentse teekonna, mis võimaldab kasutajal vahetada suvalisel hetkel vaatluse all olevate olekute vahel, ilma, et ta peaks eraldi meeles pidama, mis teekonda pidi iga olek on saavutatud. Tagurpidi sammudes on võimalik kättesaadamatuid silumislõimi taastada – kui tagurpidi sammudes läbitakse tipp, millele vastava juhtvoograafi tipu juures silumislõim kättesaadamatuks muutus, siis muutub silumislõim uuesti kättesaadavaks.

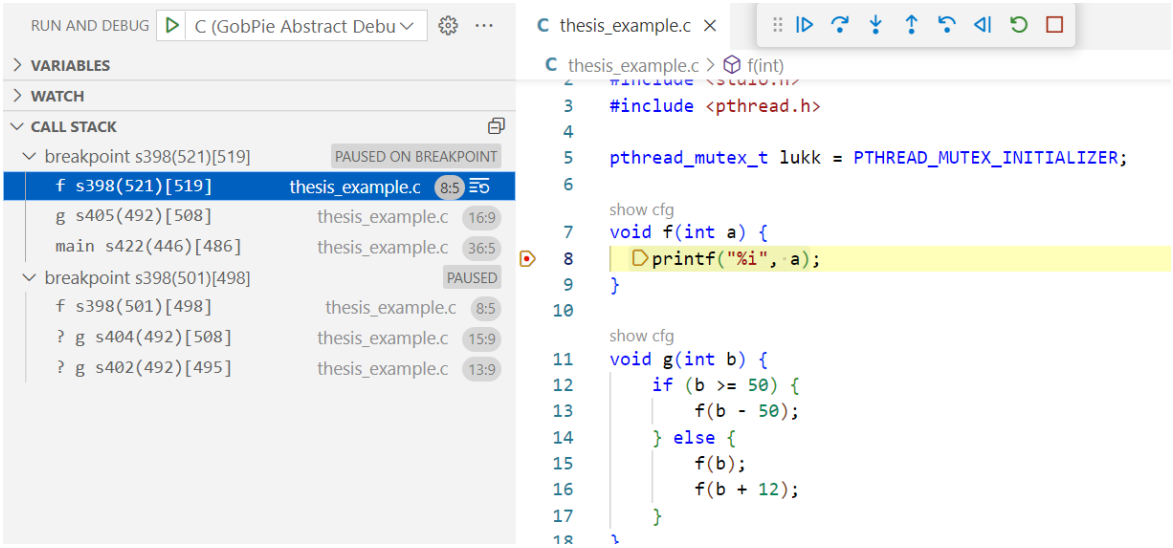
3.4.3 Väljakutsete pinu

Tavalises siluris kuvatakse kasutajale iga lõime juures väljakutsete pinu, mis näitab, millised väljakutsed viisid hetkel vaadeldava funktsiooni välja kutsumiseni. Abstraktses siluris konstrueeritakse väljakutsete pinu leides abstraktsest saavutatavusgraafist ahela, mis viib programmi lähtepunktist hetkel vaadeldava tipuni – leitud ahelas olevad väljakutsed moodustavadki väljakutsete pinu. Väljakutsete pinu koostamisel loetakse väljakutseks ka funktsiooni käitamine uues lõimes. See võimaldab kasutajal väljakutsete pinu vaadates näha, kus antud funktsiooni jooksutav lõim loodi ning tagab, et ka mitmelõimelise koodi

puhul on väljakutsete pinus näha kogu teekond programmi lähtepunktist funktsiooni välja kutsumiseni.

Juhul kui võimalikke ahelaid on mitu, siis esitatakse kasutajale pikim võimalik ühine osa ahelast ning seejärel kõik võimalikud vahetult eelnevad väljakutsed, mis viivad selle ahelani. See on töö autori hinnangul mõistlik kompromiss. Kõigi väljakutsete pinude täielik kuvamine ei ole mõistlik kahel põhjusel. Esiteks, kuna tavalises siluri kasutajaliideses ei ole selleks head võimalust ning teiseks, kuna teatud juhtudel on võimalike väljakutsete pinude hulk väga suur või koguni lõpmatu.

Joonisel 11 on mõned näited abstraktse siluri väljakutsete pinudest Visual Studio Code'is.



Joonis 11. Abstraktse siluri väljakutsete pinude vaade (vasakul) ning silutav programm (paremal).

3.4.4 Muutujate ja avaldiste väärtuste kuvamine

Silur võimaldab kasutajal näha hetkel vaadeldavas programmipunktis muutujate väärtusi. Abstraktses siluris täidavad väärtuste rolli abstraktsete domeenide väärtused. Kuna Goblint on ülesehituselt modulaarne, siis on võimalike analüüse, mis erinevaid väärtusi eri domeenidega modelleerivad väga palju. Abstraktses siluris kasutatakse avaldiste väärtuste leidmiseks kahte lähenemist. Kui võimalik, siis võetakse muutujate väärtused otse baasanalüüsi abstraktsest domeenist, milleks on lihtsalt tabel, kus võtmeks on muutuja nimi ning väärtuseks muutuja väärtus abstraktse domeenina. Globaalsete muutujate puhul ei ole see tihti võimalik, kuna globaalseid muutujaid saab muuta mitu lõime läbisegi ning sellisel juhul kasutab Goblint nende väärtuste jälgimiseks keerulisemaid analüüse. Sellistel juhtudel

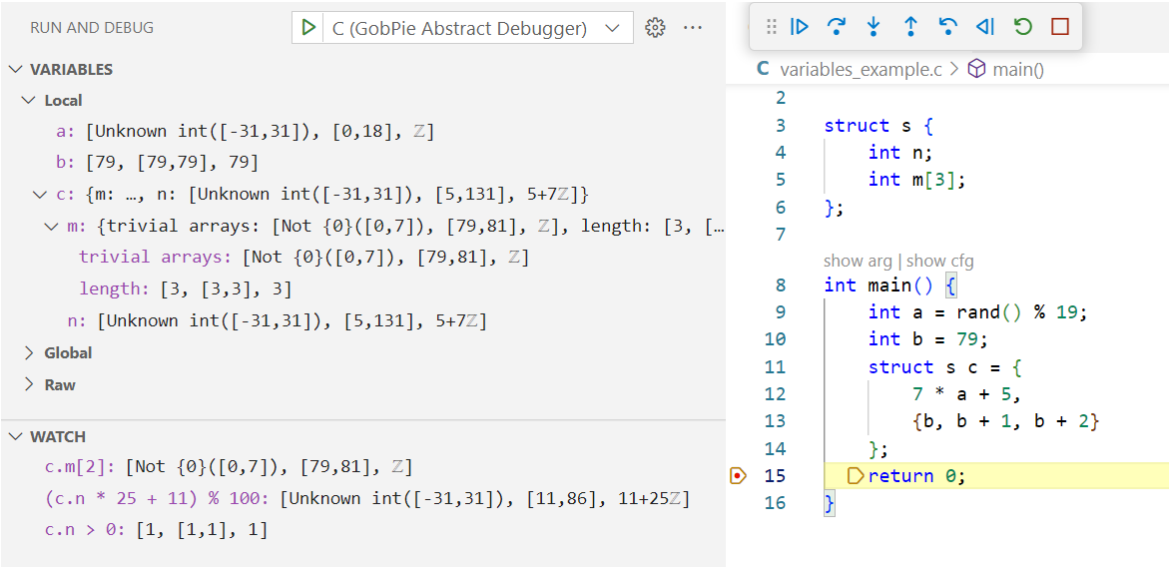
leitakse muutuja väärtus esitades Goblintile päring väärtustada avaldis, milleks on lihtsalt vaadeldava muutuja nimi. Avaldiste väärtustamiseks esitab Goblint seesmiselt päringuid nendele erinevatele analüüsidele, et leida võimalikult täpne avaldise väärtust kirjeldav abstraktse domeeni väärtus. See tagab, et abstraktne silur suudab ära kasutada avaldise väärtustamiseks kõiki analüüse, mida Goblint ise kasutab.

Abstraktsete domeenide väärtused on suhtluses Goblinti ja siluri vahel esitatud JSON objektidena. Kuvamiseks teisendab silur selle kuju mõningate ilustustega ümber kujuks mida DAP muutujate esitamiseks kasutab. Teisendamisloogika on võimeline teisendama suvalisi JSON kujul objekte ning ei tee eeldusi väärtuste tähenduse ega struktuuri kohta. See tähendab, et silur ei saa Goblinti poolt määratud abstraktsete domeenide esitusi loetavuse huvides ega muudel eesmärkidel kohandada. Sellise lähenemise oluliseks eeliseks on see, et see võimaldab siluril toetada kõiki Goblintis kasutatavaid abstraktseid domeene sh ka selliseid mis Goblintisse kunagi tulevikus lisatakse.

Lisaks lokaalsete ja globaalsete muutujate väärtustele kuvatakse muutujate vaate alamjaotises Goblinti analüüside abstraktsete domeenide väärtused toorel kujul. See on kasulik Goblinti arendamisel, et mõista analüüse, mis ei käsitle muutujate väärtusi (nt lukuhulkade analüüs).

Lisaks muutujate väärtustele on abstraktses siluris võimalik jälgida kõrvalefektideta C-keelsete avaldiste väärtusi. Kasutaja saab lisada huvipakkuva avaldise jälgitavate avaldiste (*watch expressions*) nimekirja, kus avaldis väärtustatakse automaatselt vaadeldavas programmipunktis iga kord kui vaadeldav programmipunkt vahetub. Samuti on võimalik avaldisi hetkel vaatluse all olevas programmi punktis väärtustada kirjutades avaldis siluri konsooli. Sarnaselt globaalsetele muutujatele, kasutatakse avaldiste väärtustamiseks päringut Goblinti baasanalüüsile, mis väärtustab avaldise baasanalüüsi abstraktses domeenis.

Joonisel 12 on näidatud muutujate väärtustamislaused silutava programmi lähtekoodis ning abstraktse siluri poolt kuvatud muutujate ja jälgitavate avaldiste väärtused.



Joonis 12. Abstraktse siluri poolt kuvatud muutujate (vasakul üleval) ning avaldiste (vasakul all) väärtused koos silutava programmi lähtekoodiga (paremal).

4. Tulemused

Selles peatükis antakse ülevaade töö tulemustest. Esitatakse autori hinnang tehtud tööle ja kasutajatelt saadud tagasiside ning tuuakse välja probleemid ning võimalikud edasised arendused.

4.1 Ülevaade

Töö põhiliseks väljundiks on abstraktse siluri esimese versiooni loomine.

See tulemus on saavutatud – abstraktne silur toimib, kõik eelmises peatükis kirjeldatud funktsionaalsused on realiseeritud ning nende funktsionaalsust ja töökindlust on testitud kasutades erinevaid programme Goblinti testkomplektidest ning erinevaid Goblinti seadistusi.

Lisaks abstraktse siluri lisamisele tegi töö autor muudatusi GobPie keskses loogikas, et parandada GobPie töökindlust ja kasutajamugavust. Need muudatused on praeguseks osa GobPie avalikult kättesaadavast versioonist. Ammendav nimekiri tehtud muudatustest ning nendega seotud tõmbetaotlustest (*pull requests*) on leitav lisas II.

Samaaegselt abstraktse siluri arendamisega arendati Goblinti põhiarendajate poolt välja Goblinti serverirežiimi rakendusliidese funktsioonid, mis võimaldavad pärida analüüsitud programmi juhtvoo- ja saavutatavusgraafe ning pärida programmi olekut ja väärtustada avaldisi saavutatavusgraafi tippudes. Töö autor panustas sellesse testides serverirežiimi uusi funktsionaalsusi ning teavitades arendajaid leitud vigadest ja puudujääkidest.

4.2 Tagasiside kasutajatelt

Abstraktse siluri väärtuse ja puudujääkide paremaks mõistmiseks koguti tagasisidet kahelt Goblinti arendajalt: Vesal Vojdani ja Simmo Saan. Arendajatel paluti kasutada abstraktset silurit mingi nende poolt vabalt valitud C-keelse programmi või programmide analüüsitulemuste lahtimõtestamiseks ning vastata küsimustele kasutajakogemuse kohta vabas vormis. Esitatud küsimused on toodud lisas III. Järgnevalt on toodud kokkuvõte saadud tagasisidest.

Tagasisides toodi välja, et üldiselt kasutati abstraktset silurit kahel viisil:

- et mõista, miks Goblinti analüüs leiab mingi vea. Selleks seati katkepunkt vea tekkimiskohta ning uuriti teekondi läbi programmi, mis viisid selle programmipunktini. Selleks olid kasulikud operatsioonid, mis võimaldasid uurida vea tekkimisele eelnevaid olekuid sh tagurpidi sammumine ja väljakutsete pinu kuvamine.
- et mõista, miks Goblinti analüüs peab mingit osa programmist saavutamatuks (*unreachable*). Selleks seati katkepunkt programmi algusesse või mingisse saavutatavasse programmipunkti huvipakkuva koha lähedal ning uuriti, kuidas tekkib olukord, mis viib selleni, et mingi osa programmist ei ole saavutatav. Selleks olid kasulikud operatsioonid, mis võimaldasid uurida saavutatavale kohale järgnevaid olekuid sh edasi sammumine, väljakutsetesse sisse astumine ja juhtvoo harude vahel valimine.

Võrreldes g2html-ga peeti abstraktset silurit edasiminekuks. Eriti kasulikuna toodi välja funktsionaalsusi, mida g2html-is ei ole sh võimalust avaldisi väärtustada ning väljakutsete pinu kuvamist. Samuti toodi kasulikuna välja asjaolu, et sammumisel suudab silur automaatselt järgida seotud programmiradu ja kontekste, st näiteks funktsiooni sisse astumisel leitakse õige kontekst, mis vastab just selles konkreetses programmiraja ja kontekstis tehtud väljakutsele, samas kui g2html-is peab selliseis seoseid looma käsitsi võimalike kontekste ja programmiradu kõrvutades.

Samuti osutus kasulikuks siluri võime kuvada sama katkepunkti erinevaid võimalike kontekste ja programmiradasid silumislõimedena, kuna see võimaldas neid omavahel võrrelda. Puudusena toodi sealjuures välja, et võrdlemiseks tuli käsitsi silumislõimede olekust erinevusi otsida, mis tegi selle protsessi pisut kohmakaks. Samuti toodi välja, et teatud juhtudel oleks kasulik võimalus seada katkepunktidele tingimusi kasutades muude analüüside domeene, mitte ainult baasanalüüsist pärinevaid muutujate väärtusi. Näiteks andmejooksude uurimisel oleks kasulik võimalus määrata katkepunkt, mis peatub ainult juhul, kui mingi kindel lukk on võtmata.

Sammumise osas toodi välja puudusena, et juhtvoo hargnemiste vahel valimiseks kasutatav *Step into targets* toimib ainult edaspidi sammumisel ning seetõttu pole tagurpidi sammudes võimalik liikuda tagasi kohtadest, kus juhtvoo mitu haru jõuavad samasse olekusse. See teeb vaadeldavale programmipunktile eelnevate olekute uurimise ebamugavamaks.

4.3 Hinnang tehtud tööle ja edasised arendused

Loodud abstraktne silur on töökindel ja täidab oma eesmärki hästi. Tagasiside põhjal on selge, et võrreldes g2html-ga on tegemist edasiminekuga. Sellegipoolest on siluril mõned teadaolevad probleemid ja palju võimalusi edasisteks arendusteks.

Abstraktse siluri esmase versiooni arendamisel oli üheks arendusel järgitud põhimõtteks, et siluri tehniline teostus ei tohiks olla ülemäära keeruline ning siluri käitumine peaks olema võimalikult ettearvatav. Sellest tulenevalt on siluril mõningaid piiranguid, millest saaks edasise arenduse käigus vabaneda.

Näiteks on silumislõimede sammumise sünkroniseerimisel olemuslik piirang – kuna ekvivalentsete sammude valimiseks kasutatakse juhtvoograafi tippe, siis on sünkroniseerimine võimatu juhul kui ühest saavutatavusgraafi tipust väljub kaks serva, millel sihttipud on erinevad, kuid vastavad samale juhtvoograafi tipule. Selline olukord on võimalik näiteks juhul kui Goblint on seadistatud analüüsima olukorda, kus luku võtmine õnnestus eraldi olukorrast, kus üritati lukku võtta ja see ei õnnestunud. Sellisel juhul tekkib pärast luku võtmise katset saavutatavusgraafis kaks erinevat programmirada ning mõlema esimesele tipule vastab sama juhtvoograafi tipp, kuid ühes on lukk võetud ning teises mitte. Sellistel juhtudel annab silur hetkel veateate.

Selle lahendamiseks on laias laastus kaks võimaliku lähenemist. Esimene võimalus on leida täpsem kriteerium ekvivalentsete sammude valimiseks, mis tuleks toime ka juhul kui vastavad juhtvoograafi tipud on kahel võimalikul sammul samad. Tõenäoliselt on selleks vaja võrrelda sihttippude olekut, mis on küllaltki keeruline, kuna olekus koosneb paljudest abstraktsetest domeenidest ning nende vahel sobiva võrdlemisfunktsiooni määratlemiseks tuleb nende domeenide tähenduse kohta teha palju eeldusi. Teine võimalus on tekitada iga võimaliku ekvivalentse sammu jaoks uus silumislõim. See aga tekitab lisakeerukusi tagurpidi sammumisel, kuna sellisel juhul tuleb silumislõimed mingil viisil uuesti kokku liita.

Samuti oleks võimalik vähendada sammumisoperatsioonidele kehtivaid piiranguid. Nagu ka tagasisides mainiti, puudub abstraktsel siluril hetkel tagurpidi sammudes võimalus valida soovitud haru, kui saavutatavusgraafis on hargnemine. Siinkohal saaks ära kasutada *Step in targets* operatsiooni, mida juba kasutatakse, et edaspidi sammudes harude vahel valida. Sarnaseid piiranguid esineb ka muudel sammumisoperatsioonidel. Näiteks juhul kui eelmine väljakutsete pinu kaader pole üheselt määratud, siis puudub hetkel võimalus välja

astumisel soovitud sihtkohta valida. Seda probleemi saab kaudselt lahendada kasutades kaadri taaskäivitamise operatsiooni, kuid ka siin saaks edasises arenduses võimaldada soovitud sihtkoha valimist kasutades *Step into targets* operatsiooni.

Praegune abstraktse siluri versioon kasutab DAP kaudu tavalise siluri kasutajaliidest. See tagab, et abstraktset silur toetab paljusid erinevaid programmeerimiskeskkondi, kuid sellel on ka mõned puudujäägid. Abstraktsele silurile eraldiseisva kasutajaliidese arendamine oleks kasulik edasiarendus, mis võimaldaks lisada abstraktsele silurile operatsioone, mida tavaline silur ei toeta.

Üheks selliseks operatsiooniks on silumislõimede omavaheline võrdlemine. Nagu tagasisides välja toodi, vajab silumislõimede võrdlemine hetkel käsitsi erinevuste otsimist. Tavalisel siluril ei ole sellise tegevuse automatiseerimiseks sobivat kasutajaliidest ning seega puuduvad DAP-is selleks operatsioonid, mille kaudu automaatset võrdlemist teostada. Eraldiseisva kasutajaliidese puhul oleks võimalik luua silumislõimede võrdlemiseks eraldi kasutajaliides, kus erinevusi võimalikult mugava kujul esile tõsta.

Lisaks sellele on võimalike edasiarendusi, mis teeksid lihtsamaks tagasisides välja toodud levinumad kasutusvood.

Üheks selliseks kasutusvooks on vea päritolu otsimisel vea tekkimiskohale eelnevate olekute uurimine. Selle lihtsustamiseks saaks abstraktsele silurile lisada eraldi režiimi, kus silur leiab automaatselt ühe võimaliku teekonna programmi lähtepunktist vaatluse all oleva programmi punktini ning võimaldab sammuda ainult mööda seda konkreetset teekonda. See lihtsustaks oluliselt veani viiva teekonna uurimist, kuna sellisel juhul oleksid kõik valikud saavutatavusgraafi hargnevuskohtades kasutaja eest ära tehtud ning eelnevate olekute uurimiseks piisaks lihtsalt korduvalt tagasi astumisest.

Teiseks selliseks kasutusvooks on Goblinti poolt leitud vigade analüüs, kus uuritav viga on seotud mingi muu abstraktse domeeniga, mitte muutujate väärtustega. Üheks selliseks näiteks on tagasisides välja toodud andmejooksude analüüs. Selle lihtsustamise saaks lisada võimaluse seada katkepunktidele tingimusi, mis kontrollivad, kas mingi kindel lukk on võetud või võtmata. Selliste tingimusavaldiste realiseerimiseks saaks kasutada Goblinti seesmist analüüsidevaheliste päringute liidest, et pärida lukuhulki. See vajaks lisaks siluripoolsetele täiendustele Goblinti serverirežiimi rakendusliidese täiendamist vastavate päringute toega. Alternatiivseks lähenemiseks oleks pakkuda üldisemat võimalust seada tingimusi, mis otsivad programmipunkti oleku toorest kujust mingeid kindlaid väärtusi või

struktuure. Kuna olekuid esitatakse JSON kujul siis saaks selleks kasutada olemasolevaid JSON-ist väärtuste eraldamiseks mõeldud keeli nagu näiteks JSONPath[12] või JSONata[13].

Tehnilise poole pealt oleks üheks pikemas perspektiivis kasulikuks edasiarenduseks põhjaliku automaatsete testide komplekti loomine. Selle töö raames viidi testimine läbi käsitsi, kuna arvestades, kui keskset rolli mängib siluri arenduses kasutajaliides, peeti vajalikuks valideerida funktsionaalsusi ennekõike kasutajakogemuse osas. Samuti mõjutas otsust automaattestimise kahjuks asjaolu, et kuna abstraktne silur suhtleb paljude väliste komponentidega, sh programmeerimiskeskkonna ja Goblinti serveriga, siis oleks tema eraldiseisev testimine küllaltki keerukas ja madala kasuteguriga tegevus. Siiski oleks automaatne testimine kasulik, ennekõike selleks, et võimaldada koodibaasi suuremahulisemat ümberstruktureerimist ilma, et oleks vajalik käsitsi kõikide äärejuhtude toimimist pärast iga muudatust kontrollida.

[12] https://datatracker.ietf.org/doc/draft-ietf-jsonpath-base/

[13] https://jsonata.org/

Kokkuvõte

Töö eesmärgiks oli luua abstraktse siluri esmane versioon, mis võimaldaks kuvada Goblinti poolt teostatavate analüüside vahetulemusi kasutades tavalise siluri kasutajaliidest. Abstraktse siluri esmane versioon realiseeriti ja selle funktsionaalsust ja töökindlust testiti kasutades seda olemasolevate Goblinti testprogrammide analüüsimiseks.

Töö teoreetilises osas anti ülevaade andmevooanalüüsi ja abstraktse interpretatsiooniga seotud teadmistest, mis on vajalikud töö mõistmiseks ning kirjeldati abstraktse siluri olemust ja varasemaid töid selles vallas.

Töö praktilises osas realiseeriti abstraktne silur osana Visual Studio Code'i pistikprogrammist GobPie. Kasutades DAP-i suhtluseks abstraktse siluri ja VS Code'i vahel realiseeriti kõik siluri üldlevinud funktsionaalsused, sh katkepunktide tugi, sammumine, avaldiste väärtustamine ja muutujate väärtuste ning väljakutsete pinu kuvamine.

Töö tulemuste hindamiseks koguti kvalitatiivset tagasisidet Goblinti arendajatelt, kes on abstraktse siluri esmast versiooni kasutanud. Tagasiside näitas, et loodud abstraktne silur on edasiminek võrreldes varasemalt Goblinti analüüsi vahetulemuste kuvamiseks kasutatud tööriistaga g2html ning valdav enamus abstraktses siluris realiseeritud funktsionaalsusi on ka praktikas analüüsi tulemuste tõlgendamisel kasulikud.

Sellegipoolest on antud tööl on mitmeid võimalusi edasisteks arendusteks. Näiteks on abstraktse siluri esmasel versioonil mõningaid piiranguid sammumise osas, mida saaks leevendada. Samuti on võimalik edasises arenduses abstraktsele silurile lisada erinevaid funktsionaalsusi mis teevad mugavamaks tüüpilised abstraktse siluri kasutusvood.

Viidatud kirjandus

[1] K. Apinis, „Programmianalüüs Goblint raamistikus,“ Tartu Ülikool, Tartu, 2009.

[2] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security & Privacy*, vol. 2, no. 6, pp. 76-79, 2004.

[3] Goblint maintainers, "Current Contributors," [Online]. Available: https://goblint.in.tum.de/people. [Accessed 22 04 2023].

[4] V. Vojdani, K. Apinis, V. Rõtov, H. Seidl, V. Vene and R. Vogler, "Static race detection for device drivers: the Goblint approach," in *Proceedings of the 31st IEEE/ACM International Conference*, 2016.

[5] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977.

[6] S. Saan, „Abstraktsete domeenide omaduspõhine testimine,“ Tartu Ülikool, Tartu, 2018.

[7] V. Vojdani, „Mitmelõimeliste C-programmide kraasimine analüsaatoriga Goblin,“ Tartu Ülikool, Tartu, 2006.

[8] S. Saan, „Abstraktne interpretatsioon,“ [Võrgumaterjal]. Saadaval: https://courses.cs.ut.ee/2023/AKTSP/spring/Main/HomePage?action=download&upname=abstraktne-interpretatsioon.pdf. [Kasutatud 29 04 2023].

[9] S. Saan, M. Schwarz, K. Apinis, J. Erhard, H. Seidl, R. Vogler and V. Vojdani, "Goblint: Thread-Modular Abstract Interpretation Using Side-Effecting Constraints," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2021.

[10] S. Saan, "Witness Generation for Data-flow Analysis," University of Tartu, Tartu, 2020.

[11] M. Comini, G. Levi and G. Vitiello, "Abstract debugging of logic programs," in *Logic Program Synthesis and Transformation - Meta-Programming in Logic*, 1994.

[12] F. Bourdoncle, "Abstract debugging of higher-order imperative languages," in *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, 1993.

[13] L. Nguyen Quang Do, S. Krüger, P. Hill, K. Ali and E. Bodden, "VISUFLOW: a debugging environment for static analyses," in *Proceedings of the 40th International Conference on Software Engineering*, 2018.

[14] M. Kalin, "Inter-process communication in Linux: Sockets and signals," 17 04 2019. [Online]. Available: https://opensource.com/article/19/4/interprocess-communication-linux-networking. [Accessed 28 04 2023].

[15] Microsoft Corporation, "DAP Overview," [Online]. Available: https://microsoft.github.io/debug-adapter-protocol/overview. [Accessed 28 04 2023].

[16] Microsoft Corporation, "Tools supporting the DAP," [Online]. Available: https://microsoft.github.io/debug-adapter-protocol/implementors/tools/. [Accessed 06 04 2023].

Lisad

I. Abstraktse siluri lähtekood

GobPie lähtekood koos selle töö raames loodud abstraktse siluri esmase versiooniga on leitav aadressil https://github.com/FeldrinH/GobPie/tree/abstract-debugging.

Selle töö raames autori poolt tehtud muudatuste ajalugu on leitav aadressil https://github.com/FeldrinH/GobPie/commits/abstract-debugging?author=FeldrinH.

Lähtekoodiga on kaasas lühike siluri kasutusjuhend, mis asub failis *Readme.md*.

Peatükis 3 kasutatud näited asuvad kataloogis *adb_examples*.

II. GobPie muudatused ja vastavad tõmbetaotlused

Muudatus	**Tõmbetaotlus**
Abiskriptid GobPie mugavamaks kompileerimiseks ja pakendamiseks.	https://github.com/goblint/GobPie/pull/47, https://github.com/goblint/GobPie/pull/50
Töökindluse parandamine.	https://github.com/goblint/GobPie/pull/48
Valik mis võimaldab lülitada välja inkrementaalse analüüsi. (Inkrementaalse analüüsi väljalülitamine teeb Goblinti serverirežiimi töökindlamaks.)	https://github.com/goblint/GobPie/pull/51, https://github.com/goblint/GobPie/pull/54
Valik mis võimaldab kasutajal määrata Goblinti asukoha failisüsteemis.	https://github.com/goblint/GobPie/pull/52, https://github.com/goblint/GobPie/pull/54
Kasutaja probleemidest teavitamine käivitumisel ja analüüside ebaõnnestumisel.	https://github.com/goblint/GobPie/pull/53, https://github.com/goblint/GobPie/pull/56
Analüüsi automaatne jooksutamine .h ning .i failide salvestamisel. (Eelnevalt jooksutati analüüs ainult .c failide salvestamisel.)	https://github.com/goblint/GobPie/pull/58

III. Tagasiside küsimustik

Kasutajatele esitati tagasiside kogumiseks järgnevad küsimused:

1. Kuidas silurit Goblinti analüüside/analüüsitulemuste mõistmiseks kasutatakse (üldine tegevuste loetelu vms)?
2. Millised siluri funktsioonid on kasulikuud analüüsi mõistmisel ja millised mitte? Miks?
3. Kas on mingeid kiikse või vigasid mis takistavad siluri kasutamist?

Lisaks nendele küsimustele vastamisele paluti kasutajatel jagada muid ettepanekuid ja mõtteid vabas vormis.

IV. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, **Juhan Oskar Hennoste**,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose „**Abstraktne silur Goblintile**“, mille juhendaja on **Simmo Saan**, 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.

Juhan Oskar Hennoste

09.05.2023