TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Alex Viil

Interaktiivne Abstraktne Interpretaator IntelliJ IDEA jaoks

Bakalaureusetöö (9 EAP)

Juhendaja: Vesal Vojdani

Tartu 2021

Interaktiivne Abstraktne Interpretaator IntelliJ IDEA jaoks

Lühikokkuvõte:

Abstraktne interpreteerimine on arvutiprogrammi osati täitmine eesmärgiga aru saada, kuidas programm käitub, ilma et peaks kõiki arvutusi tegema. Selle abil saab tuvastada, kas rakendus käitub nagu oodatud. Vastasel juhul saab arendaja teada vea olemasolust ja selle parandamisega tööle asuda.

IntelliJ IDEA on arenduskeskkond, kus arendaja saab Java programmeerimiskeele koodi näol programme kirja panna ja samas neid kompileerida ja jooksutada. Peaaegu kõik programmid kasutavad oma töös muutujaid, millele omistatakse vähemalt ühe korra mingi väärtus ning mida hiljem loetakse. Tihti hargneb programmi töö olenevalt nende muutujate väärtustest ning olenevalt arendaja tööst võib tekkida olukordi, kus kindlatel väärtustel tekib programmi töös erind ehk tõrge. Java hea tava puhul sellised olukorrad püütakse tavaliselt kinni ja lastakse programmil vastavalt reageerida, kuid keerulisemates süsteemides on tihti keeruline kõiki juhte ette näha.

Eelnimetatud arenduskeskkond toetab pistikprogramme, mis on eraldi tööd soodustavad programmid. Interaktiivse abstraktse interpretaatori pistikprogramm tagab võimaluse arendajal programmeerimise ajal teostada programmi kohta analüüs staatilise analüüsi raamistiku Põder abil. Selle bakalaureusetöö uuritakse olemasolevaid raamistikke ja võimalikku teostust, mis võimaldaks lihtsamaid omaduse kontrolle teostada, näiteks kas täisarvulise muutuja väärtus jääb mingisse vahemikku piiritletud või mitte.

Võtmesõnad:

abstraktne interpretaator, interaktiivsus, IntelliJ, Java

CERCS: P175 Informaatika, süsteemiteooria

An Interactive Abstract Interpreter for IntelliJ IDEA

Abstract:

Abstract interpretation is the partial execution of a computer program to better understand how it behaves without having to perform all the calculations otherwise done in a regular run. It's useful in determining if the program functions as expected. If not, the developer then becomes aware of an issue and can focus on resolving it.

IntelliJ IDEA is an integrated development environment in which developers can write code in the form of the programming language Java, compile it, and execute the resulting instructions. Almost all programs use variables that are assigned values and have their values read. Often the program branches into different parts, and which parts get executed depends on variables. Depending on the quality of the developer's work, certain situations might arise from an incorrect combination of variable values that can cause an exception or error to occur. Best practice in Java foresees proper handling of such exceptions, but in more complicated systems it's a lot more difficult to think of every situation.

The aforementioned development environment supports plugins, which are separate development assisting programs. An interactive abstract interpretation program provides the developer with the ability to analyse a program with the help of the static analysis framework Põder. In the scope of this bachelor's thesis, development of such a plugin is investigated, that can conduct simpler property checks. For example, whether an integer variable is bound by certain limits or not.

Keywords:

abstract interpreter, interactivity, IntelliJ, Java

CERCS: P175 Informatics, systems theory

Sisukord

1. Sissejuhatus

Programmide kirjutamisel IntelliJ IDEA integreeritud arenduskeskkonnas on arendajale abiks palju juba tavasätetega olemasolevaid abifunktsioone. Peamiselt piirduvad need aga lihtsamate soovituste ja kindlate vigade tuvastuse ning paranduste pakkumisega. Arendajal endal on vähe võimalusi veatuvastusele lisainfot anda. Hetkel on olemas programmeerimiskeele Java enda sisseehitatud annotatsioonid, kuid neid leidub vähe ning nende kasutus on piiratud. Selle bakalaureusetöö eesmärk on uurida Javas veatuvastuse võimaluste laiendamist uute annotatsioonide lisamisega pistikprogrammi kaudu Dafny programmeerimiskeele [1] näitel ning anda hinnang sellise implementatsiooni võimalikkusele ja mahule.

Dafnys on olemas annotatsioonid, mille abil saab programmeerija täpsustada rakenduse soovitud käitumist. Abstraktne interpretatsioon kontrollib seejärel baitkoodi analüüsimisega, kas annotatsioonide nõutud käitumine on tagatud. Nii saab koodi tavapärase käitamise kohta esitada garantiisid, mis peavad alati paika. See aitab täpsustavate kontrollide kaudu vältida ootamatuid tulemusi, mille käsitsi otsimine võib osutuda ajakulukaks.

Selliseid annotatsioone on erinevaid. On olemas lihtsamaid kinnituslauseid, mis koodi keskel kontrollivad mingi tingimuse kehtimist, kui ka tsükli päise kontrollid, mis igal tsükli läbimisel kontrolle teostavad [2]. Java programmeerimiskeeles saaks sarnaseid kontrolle käsitsi koodi sisse kirjutada vähese vaevaga, kuid need nõuaksid terve koodi kompileerimist, käitamist ning see mõjutaks lõpptulemust. Sellised käsitsi kirjutatud kontrollid toimuksid vaid etteantud sisendite puhul. Seega motivatsioon abstraktseks interpretatsiooniks on ilmne.

Eelnevaid funktsioone saab arenduskeskkonda juurde lisada pistikprogrammi abil. JetBrains, IntelliJ IDEA toote omanik, on selleks loonud võimalusterohke tarkvaraarenduskomplekti koos dokumentatsiooniga [3]. Selle abiga on võimalik luua liidestuv rakendus, mis tunneb arendaja lisatud annotatsioonid ära ning teostab nende põhjal baitkoodi analüüsi. Abstraktseks interpretatsiooniks kasutab pistikprogramm olemasolevat analüüsi raamastikku. Bakalaureusetöö raames uuritakse kahte võimalust selleks: levinud Checker raamistik [4] ning Tartu Ülikoolis loodud staatilise analüüsi raamistik Põder [5].

2. Sisendiga veatuvastus

Levinum veatuvastus Java integreeritud arenduskeskkondade poolt on iseseisev. See tähendab, et arendaja ise mingit infot rakenduse kohta ette ei anna, seega saadud tagasiside on väga üldine. Tihti piirdub see duplikaatkoodi, omistuse tüübi mittekattuvuse või ligipääsmatu koodi esile toomisega. Kuigi need vead on triviaalsed, siis nende tähelepanu keskmesse toomine omab suurt kasu. Küll aga on neist vähe kasu põhimõttelisemate loogikavigade puhul, ehk kui kood on kompileeriv ja käitatav, kuid tulemus ei ole oodatav. Selle tõttu pakub huvi võimalus analüsaatorile info ette andmine meetodide sisendite, väljundite ja töö kulgemise kohta. Nii on pikemas koodisektsioonis loogikavea leidmine oluliselt lihtsustatud.

Veaanalüsaatorile lisainformatsiooni ette andmine aga võib olla ajaliselt kulukas ning arendaja sel juhul seda ei kasuta. Olemasolevate võimaluste eelis on just see, et need nõuavad arendaja poolt minimaalset sekkumist, seega ainuke ajakulu võib tekkida valepositiivsetest vigadest. Seega tasub uurida olemasolevaid võimalusi Javas, mida on suhteliselt vähe, ning võrrelda programmeerimiskeelega Dafny, kus on palju viise veaanalüsaatorile info andmiseks.

2.1 Java

Võrreldes teiste programmeerimiskeeltega on Java pigem turvaline – ülesehitusest tulenevalt esineb arendamisel teatud laadi vigu väga vähe. Muutujate loomisel on vältimatu selle tüübi määramine. Kuigi objektide ja pärimuse kaudu võib tekkida olukord, kus kasutaja annab muutujale vale tüübi, mis osas koodis siiski töötab kattuvate isendiväljade ja -meetodide tõttu, siis literaalide puhul on raske eksida. Objektidega toimetamine on siiski võrdlemisi lihtne selles vallas, et tegu on lihtsate viidetega. Javas ei pea tegelema mälupesade osutitega ega eristama neid objektide viidetest, nagu näiteks programmeerimiskeeles C++. Samuti on mälulekete oht oluliselt väiksem, kuna Javas on olemas protsess nimega automaatne prügikorje. See on osa mäluhaldusest, kus kasutuseta objektid automaatselt arvatakse mälust välja. Seega arendaja ei pea väga muretsema, mis objektidest saab peale nende eluiga, ning ei pea looma ja kasutama dekonstruktoreid ehk ei pea objektide kustutamisega tegelema. Vastasel juhul oleks eksimusteks rohkem ruumi.

Javas on aga ka mõned viisid kuidas täpsustada millist käitumist loodavalt programmilt oodatakse.

2.1.1 Lihtne kinnituslause

Võtmesõnaga *assert* saab kontrollida mingi tingimuse kehtimist selle väärtustuse kaudu [6]. Ette antud avaldise keerukusel piiranguid ei ole ning seda saab ehitada loogika operaatoritega nagu mujalgi Javas. Kinnituslause seejärel väärtustab selle tingimuse ning tõese väärtuse puhul jätkab programm tööd. Näide kasutusest, vt joonis 1.

```
1. ...
2. int abs(int x) {
3.   if (x < 0) {
4.     x = -x;
5.   }
6.   assert x >= 0;
7.   return x;
8. }
9. ...
```

Joonis 1. Java näidisprogramm lihtsa kinnituslausega.

Väärase väärtuse puhul tõstatakse viga *AssertionError*. See aga erineb Java tavapärastest erinditest. Erindi, inglise keeles *Exception*, puhul on loomulik selle kinni püüdmine, vastavalt käitumine ja programmi töö jätkamine, kuid vea, inglise keeles *Error*, puhul programmi töö jätkamist ei oodata. Teised vead Javas on seotud näiteks sisend- ja/või väljundseadme töö nurjumise või virtuaalmasina töö katkemisega, mille puhul on programmi töö jätkamine tihti võimatu [7]. Kinnituslause puhul on see valik tehtud seetõttu, et erindite puhul oodatakse põhjustaja leidmist sealt, kus see tekkis. Tingimuse väärtustamisel võib aga viga tekkida väga sügaval koodis, seega sellele üheselt reageerida ei ole alati võimalik.

Kinnituslause kasutamine Javas aga ei ole väga populaarne ning seda raskendab asjaolu, et nende kasutamisest tuleb Java koodi käitamisel teada anda lipuga *-ea*. Lipu vajadus on selletõttu, et neid kinnituslauseid kasutatakse testimisel, kuid mitte sisendi verifitseerimiseks. Lisaks on probleemiks asjaolu, et kinnituslaused on arvutuslikult nõudlikud, seega nende tasuvus on veel rohkemgi küsitav [8].

Olenevalt projektist on võimalik, et kasutatakse mõne levinuma raamistiku kinnituslauseid. Levinud tarkvara rakenduse raamistikul Spring on olemas paljude võimalustega *Assert* klass, kus on olemas nii lihtsad kinnituslaused kui ka spetsiifilisemad kontrollid, vt joonis 2.

```
1. ...
2. int funSpring(A x) {
3.   Assert.isAssignable(A, B, „B must be assignable to variable of type
   A“);
4.   A y = externalFunction(x); // returns variable of type B
5.   Assert.hasText(y.ToString(), „y must have text“);
6.   Assert.doesNotContain(y.ToString(), „invalid“, „y must not be
   invalid“);
7.   return y.calculate();
8. }
9. ...
```

Joonis 2. Java näidisprogramm Spring kinnituslausetega.

Springis on loetavuse mõttes loodud erinevad kinnituslaused, mida oleks võimalik lahti kirjutada ka Java enda sisseehitatud *assert* võtmesõna abil. Need võivad olla päris põhjalikud, näiteks meetod *Assert.isAssignable()*, mis kontrollib pärilust ehk kas teist tüüpi objekt saaks olla ka esimest tüüpi [9]. Javast tavapärastest kinnituslausetest eristab ka reageerimine väärale tingimusele. Springi kinnituslaused tekitavad erindeid, erinevalt vigadest on neid võimalik kinni püüda ja vastavalt käituda, seega need võivad ka koodi täitmise ajal sisse jääda.

Levinud on ka automatiseeritud ühiktestide raamistiku JUnit abil kinnituslausete kasutamine [10]. Selle raamistiku klass *Assert* on põhimõtte ja võimaluste poolest sarnane Springi samanimelisele klassile, kuid on pigem mõeldud testide sees kasutamiseks. Seda võikski lugeda levinuimaks lähenemiseks Javas. Põhjalike testide puhul ei olegi kinnituslaused koodis endas vajalikud, kuid piisavalt palju olukordi katvaid teste ei ole alati ajaliselt odav luua.

2.1.2 Annotatsioonid @Nullable ja @NotNull

Javas vaieldavalt enim esinev loogikaviga on viitamine erilisele literaalile *null*. See literaal, erinevalt objektidest, ei pärine klassist *Object* ega oma ühtegi isendimeetodi ega -välja. Küll aga saab teda omistada kõikidele muutujatele, mis peaksid objektile viitama. Kui aga sellise viite kaudu proovida mingile väljale või meetodile ligi pääseda, siis visatakse erind *NullPointerException* [11].

Selle erindi vältimiseks on olemas annotatsioonid *@Nullable* ja *@NotNull*, mis ühtlasi on ühed vähestest võimalustest arendajal veaanalüsaatorile info andmiseks. Neil aga puudub ühene standard ning implementatsioone on Oracle'i, Findbugsi, JetBrainsi, Lomboki, Androidi ja Eclipse'i poolt. Osadel on *@NotNull* teise nimega, *@NonNull*. Bakalaureusetöö fookusena vaatame JetBrainsi enda lahendust.

Annotatsioon *@Nullable* käib muutuja, parameetri või tagastatava väärtuse ette ning märgib, et talle järgnev objekt võib olla *null*. Eelnevalt juba sai mainitud, et iga viide võib viidata *null*ile. Käesolev annotatsioon märgib veaanalüsaatorile, et nulli omistus on oodatud. Seega, kui sellise annotatsiooniga muutujat kasutame kontekstis, kus võib tekkida erind *NullPointerException*, siis arenduskeskkond IDEA annab sellest teada. Samal põhimõttel, kuid vastupidi töötab annotatsioon *@NotNull*, mis näeb ette, et talle järgnev objekti viide ei tohi kunagi viidata *null*ile [12]. Demonstratsiooniks vt joonis 3.

```
1. ...
2. int IDEANulls(@NotNull Object o) {
3.   @Nullable Object b = externalFunction();
4.   int x = b.concatenate(o); // b can be null, IDEA will highlight this
5.   return x;
6. }
7. ...
8. int result = IDEANulls(null); // parameter expected to never be null,
   IDEA will highlight this
9. ...
```

Joonis 3. Java näidisprogramm annotatsioonidega @Nullable ja @NotNull.

Javas on nähtavasti olemas mingid viisid, kuidas arendaja saab teatada, millist käitumist ta programmilt ootab, kuid need on vähelevinud ja neid on vähe. Kontrastiks on võimalik tuua programmeerimiskeeli, mis on maast madalast sellise põhimõttega disainitud.

2.2 Dafny

Microsoft Researchi abil valminud avatud lähtekoodiga programmeerimiskeel Dafny on näide keelest, kus saab väga kergelt programmi töö käigus tingimuste kehtimist kontrollida [13]. Dafny ongi seda silmas pidades loodud ning tänapäeval näeb kasutust peamiselt pedagoogikas ja matemaatiliste teoreemide tõestamisel. Olenemata sellest on koodi kirjutamine piisavalt sarnane Javale, et neid saaks omavahel võrrelda. Dafny enda põhimõte seisneb selles, et veatuid annotatsioone on kergem kirjutada, kui veatut koodi. Lisaks omab Dafny veel rohkem võimalusi nagu predikaadid ja kvantorid.

2.2.1 Süntaks

Dafnyl on Javaga võrreldes mõningad erinevused ning lisavõimalused. Muutujale väärtuse omistamine on teistsuguse kirjapildiga, meetodid võimaldavad mitut väärtust korraga

tagastada ning üksikud pisiasjad veel. Kui Java on tuttav, siis piisava ülevaate annab näidisprogrammi uurimine, vt joonis 4.

```
1. ...
2. method Sums(a: int, b: int, c: int) returns (ab: int, bc: int, ac: int)
3. {
4.   ab := a + b;
5.   bc := b + c;
6.   ac := a + c;
7. }
8. ...
```

Joonis 4. Dafny näidisprogramm mitu väärtust tagastava meetodiga.

2.2.2 Lihtne kinnituslause

Dafnys on olemas Javaga analoogne võtmesõna *assert*, mis töötab samal kombel. See kontrollib talle ette antud tingimuse tõeväärtust, andes märku, kui see võib olla väära tõeväärtusega. Dafny aga reageerib teisiti, kontrollides tingimuse võimalikke väärtustusi kompileerimise ajal, seega veateate saamiseks ei ole rakenduse enda käitamine vajalik.

2.2.3 Eel- ja järeltingimused

Annotatsioonidega *requires* ja *ensures* saab Dafny programmides meetodidele seada eelning järeltingimusi. Vastavalt nimele peavad nad kehtima kas enne meetodi töö alustamist või sellele järgnevalt, ehk täpselt enne väärtuste tagastamist. Need annotatsioonid kirjutatakse enne meetodi sisu, vt joonis 5. Dafny on tark selle kohalt, et ta arvestab neid meetodist väljaspool ka. Kuna aga Dafny tervet programmi alati ei analüüsi, vaid teeb seda loogeliste sulgudega eraldatud koodiplokkide kaupa, siis need annotatsioonid on konteksti andmiseks vajalikud. Vastasel juhul ei pruugi annotatsioon *assert* piisavalt teada, et garantiid anda.

```
1. ...
2. method Difference(a: int, b: int) returns (diff: int)
3.   requires a >= b;
4.   ensures diff >= 0;
5. {
6.   diff := a - b;
7. }
8. var value := Difference(5, 2);
9. assert value >= 0; // will raise error without „ensures“ in Difference
10....
```

Joonis 5. Dafny näidisprogramm eel- ja järeltingimustega.

2.2.4 Tsükli invariandid ja kahanevuse kontroll

Tulenevalt Dafny analüüsi konteksti piirangust, ehk korraga vaadatakse vaid ühte alamosa rakendusest, on tsüklite jaoks vajalik eraldi lahendus. Selleks on eraldi annotatsioonid *invariant* ja *decreases*. Itereeritav muutuja, millega jälgime mitmendat tsüklit täidame, peab iga tsükli täitumisel alluma tingimusele, mis on annotatsiooni *invariant* järele kirjutatud. Lõputu tsükli vältimiseks kontrollib annotatsioon *decreases*, et talle ette antud avaldis kahaneks iga tsükli keha täitumise järel. Nende kahe koostööna saab püstitada garantii, et tsükkel lõpetab töö. Annotatsioonile *decreases* vastandit *increases* ei eksisteeri Dafnys, seega kasvava funktsiooni puhul on vajalik rohkem ette anda, kui ainult itereeritav muutuja, vt joonis 6.

```
1. ...
2. while 0 < i
3.   invariant 0 <= i
4.   decreases i
5. {
6.   // descending loop body
7.   i := i - 1;
8. }
9. ...
10.while i < n
11.  invariant 0 <= i <= n
12.  decreases n - i
13.{
14.  // ascending loop body
15.  i := i + 1;
16.}
17....
```

Joonis 6. Dafny näidisprogramm kahaneva ja kasvava tsükliga.

2.3 Võrdlus Java ja Dafny vahel

On kerge näha, et Java võimalused veaanalüsaatorile lisainfo andmiseks on tagasihoidlikud ja vähe arendatud. Kui neid väheseidki võimalusi uurida, siis näib, et need on tihti ebaefektiivsed, ehk nõuavad ebaproportsionaalselt palju arvutusvõimsust. See võibki suuresti tulla Java enda disainist. Java on loodud põhimõttega, et koodi peaks vaid ühe korra kompileerima ning tulemust võiks saada käivitada igal platvormil, kus juba töötab Java virtuaalmasin. Lisaks on sellel palju varem mainitud lihtsustusi, mille tõttu arendaja peab vähem pead vaevama, kuid sellised üldistused tavaliselt omavad suuremat arvuti tööjõukulu [14].

Siinkohal omab suurt erinevust Dafny, mis on maast madalast loodud garantiidele orienteeruvalt. Võimalusi on rohkem ning need on hästi teostatud – nad ei ole arvutuslikult nii kulukad. Samas on vähem keskendatud standardiseerimisele, mida läheks vaja suuremas projektis ja mida teostatakse Javas näiteks liideste abil.

Need programmeerimiskeeled on suurte põhimõtteliste erinevustega, nende disainimisel silmas peetud printsiibid pigem ei kattu. Sellest tulenevalt võiks vaielda, et Java võimaluste laiendamise asemel on mõistlik erinevates projektides kasutada erinevaid tehnoloogiaid vajadustele vastavalt. Vastuargumendina saab samas kohe tõstatada kontekstivahetuse kulukust arendaja jaoks. On võimalik, et kontekstivahetuse kulu on suurem kui Javas implementeeritud abstraktse interpretaatori arvutuste ajaline kulu. Olemasolevate lahenduste uurimine annab rohkem tausta.

2.4 Koodilepingud Javas

Nagu ka Dafny, on Microsoft Researchi abil loodud ka koodilepingute projekt eesmärgiga sarnast kontseptsiooni laiendada programmeerimiskeeltele C# ja VisualBasic [15]. Koodilepingu nimetus tuleneb disainiprintsiibist Design by Contract™, mis omakorda pandi esmalt kirja Eiffeli programmeerimiskeelega seoses [16]. Dafnyt uurides ei ole sealseid võimalusi sama nimega kirjeldatud, sest Dafny tegeleb tõestustega, mitte vea asukoha tuvastusega. Olemasolevate Java sisendiga veatuvastuse implementatsioonide peamisteks inspiratsiooniallikateks ongi just koodilepingut ja Design by Contract™.

2.4.1 C4J

C4J on lepingute raamistik Javale, mis tagab võimaluse klassidele ja liidestele luua lepinguid [17]. Need lepingud omakorda kohanevad kõikidele klassidele, mis neid kasutavad, kas siis liidese või ülemklassi kaudu. Lepingute endi sisse saab siis kirjutada eel- ja

järeltingimusi sarnaselt Dafnyle. Asjaolu teeb keeruliseks eraldi lepingu defineerimise nõue, kuid nende kasutamine on lihtne. Klassile lepingu määramisel öeldakse, et leping hakkab seda klassi kaitsma. C4J täiendab Java süntaksit annotatsioonidega, vt joonis 7 [18].

```
1. ---  Class  ---
2. @ContractReference(TestClassContract.class)
3. public class TestClass {
4.   public void addition(int variable) {
5.     // method implementation
6.   }
7. }
1. --- Contract ---
2. import static de.vksi.c4j.Condition.*;
3.
4. public class TestClassContract extends TestClass {
5.   @Target
6.   private TestClass target;
7.
8.   @Override
9.   public void addition(int variable) {
10.    if (preCondition()) {
11.      assert variable > 0;
12.    }
13.    if (postCondition()) {
14.      // old() remembers the previous result of whatever is passed
15.      assert target.method() > old(target.method());
16.    }
17.  }
18.}
```

Joonis 7. Java näidisprogramm C4J lepinguga.

C4J raamistiku puhul on kirjapilt üsnagi mahukas, peamiselt eraldi lepingufaili koostamise pärast. Kuigi see annab võimaluse samade tingimuste kasutamiseks mitmes kohas, on see pigem harva esinev vajadus. Muret tekitab veel see, et korrektsuses veendumine käib läbi Java enda *assert* võtmesõna, mis on arvutuslikult nõudlik vahend tingimuste kontrollimiseks. Kuigi C4J on üks esimestest suurematest sellistest projektidest, siis kahjuks ei ole ta viimase seitsme aasta jooksul täiendusi saanud.

2.4.2 Cofoja

Cofoja ehk Java lepingud on lepingutepõhine programmeerimise raamistik Java jaoks, mis töötab annotatsioonide abil ja analüüsib käituses olevat koodi [19]. Põhjalikku baitkoodi analüüsi ei sooritada, seega abstraktset interpretatsiooni ei rakendata. Uuritavatest tööriistadest on Cofoja kõige sarnasem Dafnyle. Kasutusel olevat annotatsioonid on samade nimetuste ja põhimõtetega, vt joonis 8.

```
1. ...
2. @Requires(„x <= 127 && x >= -128“)
3. @Ensures(„result >= 0“)
4. int abs8bit(int x) {
5.   int result;
6.   if (x < 0) {
7.     result = -x;
8.   } else {
9.     result = x;
10.  }
11.  return result;
12.}
13....
```

Joonis 8. Java näidisprogramm Cofoja annotatsioonidega.

Cofoja kasutus on seega üsnagi lihtne, nõudes arendajalt sama vähe tööd kui Dafnyga arendades. Cofoja loojad märgivad eelisteks veel, et annotatsioonide kaudu loodud lepingute loogika on eraldi rakenduse enda loogikast, tagades iseseisvuse ning annotatsioonid kaasatakse Javadoci. Sarnaselt C4J-le kanduvad nõuded edasi ka päriluse teel. Paraku eristab Dafnyst asjaolu, et kontrollid teostatakse koodi käitamise ajal, mitte kompileerimisel, seega võimalikud vead saavad esineda alles töö käigus. Lisaks on ka see projekt jäänud viieks aastaks täiendusteta.

2.4.3 jqwik

JUnit 5 automatiseeritud ühiktestide raamistik võimaldab enda testide sooritamise mootorit asendada alternatiiviga. Omaduste põhise testimise mootor jqwik sellel laienduse kaudu töötabki [20]. Implementatsioon aga väljendub testide kaudu sarnaselt JUniti enda testidele. Laiendusena on olemas palju erinevaid annotatsioone, näiteks *@Property*, mille abil saab sarnaselt Dafnyle koostada teste, millel on eel- ja järeltingimused. Huvi omab jqwik selles vallas, et ta on aktiivses arenduses ja praktiseerib uuemaid võtteid, kuid Dafnyga võrreldes

on tema puudujääk selles osas, et ta on sisuliselt ühiktestide laiendus. See tähendab, et arendaja jaoks ei ole enam programmi kohta info andmine väike ettevõtmine, ta peab selleks siiski eraldi teste looma, mille tõttu ta ei pruugigi seda üldse teha.

2.4.4 Java Modeling Language

Samuti võttes inspiratsiooni Eiffelilt, on loodud Java modelleerimise keel, mis kasutab Dafnys nähtud annotatsioone [21]. Nende kasutuse põhimõtted on samad, kuid JML-i annotatsioonid lähevad kirja kommentaaridena, vt joonis 9. Tavapärasel koodi kompileerimisel ja käitamisel nad kuidagi tulemust ei mõjuta ning mingit analüüsi ei tehta. Spetsifikatsiooni paika pidamist uuritakse eraldi rakenduse abil. JML ise on vabavaraline, seega teda implementeerivad tööriistad on tihti isegi avatud lähtekoodiga ning levinum neist on OpenJML [22].

```
1. ...
2. //@ requires x >= 0;
3. //@ ensures result >= 0 && (x <= result && result <= 1 || x > result &&
   result > 1);
4. float sqrtWrapper(int x) {
5.   // ...
6.   float result = sqrt(x);
7.   // ...
8.   return result;
9. }
10. ...
```

Joonis 9. Java näidisprogramm JML-i annotatsioonidega.

OpenJML on loodud ka pistikprogrammina Eclipse'i jaoks, mis oluliselt lihtsustab selle kasutamist. Paraku on aga see rohkem kontseptsiooni tõestus ja selle arendamine jäänud poolikuks. Viimane toetatud Microsofti platvorm sellel laiendusel on Windows XP. Käsurea ja graafilise liidesega rakendustena on OpenJML kaasajalisem, kuid eraldi rakenduse kasutamine on ajakulukas ja keeruline, palju eelistatum on pistikprogrammi variant.

2.4.5 The KeY Project

Karlsruhe tehnoloogiainstituudis loodud KeY Project on Java loogika teoreemide tõestaja, mis toetub JML-ile programmide käitumise spetsifitseerimise osas [23]. See on graafilise liidesega rakendus, mis sarnaneb OpenJML-ile, kuid on kohati ambitsioonikam ja seab fookuse tõestamisele. KeY Projectis on kasutusel eraldi tõestuste keel, mis lähtub JML

annotatsioonidest. Seda on kasutatud ühe magistritöö raames tõestamaks, et Norra valimisteks kasutatav süsteem EVA vastab spetsifikatsioonile [24].

Kuigi KeY Projectit on edukalt praktikas rakendatud, siis sellelgi esineb veel puudujääke. Sarnaselt OpenJML-ile, saab seda kasutada hetkel vaid eraldi rakendusena ja mitte pistikprogrammina. Kodulehe ülesehitus viitab sellele, et neil on plaanis Eclipse'i jaoks pistikprogramm luua, kuid selle kohta avalikku infot ei ole. Puudu on ka ujukomaarvude, lõimede, *try-with-resources* ja lambda funktsioonide tugi.

2.4.6 Järeldused

Javas on olnud palju katseid luua lepingupõhist disaini implenteerivaid rakendusi. Kuigi need on kõik mingil määral edu saavutanud, on neil paraku puudujääke ja omadusi, mille tõttu on arendajal neid raske praktikasse kaasata. Suurematekss murekohtadeks on eraldatus arenduskeskkonnast ja ajaline kulu. Sellest tulenevalt võiks üks sobilik lahendus olla pistikprogrammi loomine, mis kasutab abstraktset interpretatsiooni koodi analüüsimiseks. Nii oleks kontekstivahetus minimaalne ning veaanalüüs võimalikult väheste arvutustega.

3. IntelliJ IDEA pistikprogramm

IntelliJ IDEA on eraettevõtte JetBrains s.r.o. poolt loodud ja hallatud integreeritud arenduskeskkond, mis toetab pistikprogramme. Sama ettevõtte poolt on ka tarkvaraarenduspakett nende loomiseks. Nende arendamisel on võimalusi palju, väga suur osa põhirakendusest on tehtud ligipääsetavaks. Samas on SDK loojate poolt olemas ka dokumentatsioon, kuigi mitte põhjalik. Nimetatud oludest tulenevalt on pistikprogrammi loomine keeruline, suures osas käib see esialgu läbi katse-eksituse meetodi. Kui IntelliJ IDEA sisemise dünaamikaga vähe kursis olla, siis on kerge kogemata mälulekkeid ja kasutajaliidese ülekoormust tekitada.

3.1 JetBrainsi pistikprogrammi tarkvaraarenduspakett

Tarkvaraarenduspaketi võimekus on suur ning sellega loodud pistrikprogrammide ampluaa on suur. Project Lombok näiteks on Java teek, mis annotatsioonide abil lihtsustab objektidega toimetamist [25]. Lomboki annotatsioonid lihtsustavad konstruktorite, *getter*ite ja *setter*ite juurde panekut ja pakub konstruktorile alternatiivset meetodi *build*, kus saab isendivälju ühekaupa defineerida.

Teistlaadi näide, SpotBugs, võimaldab olemasolevat koodi põhjalikumalt analüüsida ja murekohti esile tuua [26]. Võimalikele vigadele määratakse tõsiduse aste ja kindluse hinnang. Seejärel saab arendaja need üle vaadata ja ise veenduda, kas tegu on õige- või valepositiiviga ja vastavalt edasi käituda.

Pistikprogramme on erinevaid – on olemas koodi funktsionaalsust muutvaid, kuid ka lihtsalt analüüsivaid. Selles vallas sarnanevad mõlemad eelnevalt mainitud programmid selle töö uuringuga. Soovitud eesmärk on Lomboki laadselt annotatsioone implementeerida, kuid need ei tohiks funktsionaalsust lisada ja peaksid pigem piirduma analüüsiga, aga sarnaselt SpotBugsile.

3.1.1 Projekti alused

SDK poolt on projekti loomisel mitu valikut, mida teha. Esimene valik on programmeerimiskeelte Java ja Kotlin vahel. Kotlin on samuti JetBrainsi loodud, kuid kompileerub siiski Java baitkoodiks. Julgustatakse Kotlini valimist, kuid esimese projektina on kindlasti Java eelistatum, kuna enamik näidetest on Javas kirjutatud, sh JetBrainsi enda näidisprogrammid. Projekti põhjaks saab seejärel valida Gradle'i, mis on populaarsem ja ka JetBrainsi enda poolt soovitatud, või pistikprogrammi DevKit, kuid seda ei plaanita edasi arendada. Seega esimese pistikprogrammina on kindlaim teha Gradle'i projekt Javaga.

3.1.2 Komponendid

Pistikprogrammi arendamisel on arv klasse, mida tuleb päriluse teel laiendada, et erinevaid funktsionaalsuseid luua [3]. Tegevuse klassist *AnAction* päritakse sellised klassid, mille eesmärk on mingit kindlat tegevust ette kutsuda, kui menüüs või tööriistaribal mingile nupule vajutatakse. Selle tegevuse keerukus võib olla nii lihtne kui lingi avamine veebibrauseris kui ka mingi keerulisema teenuse töö käitamine. Teenuseid luuakse klassi *ServiceManager* abil. Nendel on oma skoop, milleks võib olla mooduli, projekti või terve rakenduse tase, ning need tavaliselt teevadki pistikprogrammides suuremat ja keerulisemat tööd taustal.

Abstraktne süntaksipuu on graafiteooria mõistes puu, mis esindab mingi süntaksi struktuuri, tavaliselt programmikoodi [27]. Abstraktsus tuleneb sellest, et ainult struktuur ja sisu säilitatakse. Olenevalt parserist, mis loobki abstraktse süntaksipuu, on erinevad detailid, mille kaasamine puusse ei oma väärtust, näiteks liigsed tühikud või kommentaarid, sest need ei muuda programmi kompileerimisel midagi ja on arendaja enda aru saamise huvides lisatud. Abstraktsete süntaksipuude abil saab koodi üheselt mõistetaval viisil esindada, seda uurida ning ka seda muuta.

Programmi struktuuri liides PSI on IntelliJ enda poolt loodud laiendustega abstraktne süntaksipuu. PSI puhul on juureks fail, kust mööda struktuuri allapoole liikudes saab ligi individuaalsete elementideni. Nende ükshaaval läbi vaatamiseks on tarkvaraarenduspaketi poolt olemas erinevat laadi *Visitor* klasse. Nende abil saab uurida olemasolevat koodi ja vajadusel ka muuta.

Interaktiivse abstraktse interpretaatori loomisel oleks seega tähtis keskenduda teenuse ja tegevuse klassidele ning ka PSI-le. JetBrainsi poolt on lisaks dokumentatsioonile olemas ka hulk erinevaid näiteid [28]. Kahjuks aga keskenduvad nad väga kitsatele olukordadele üksikute tarkvaraarenduspaketi komponentidega. Sarnaselt dokumentatsioonile on väga vähe infot komponentide ühildamise kohta. See tuleb ise välja nuputada katsetamise, koodi uurimise ja olemasolevate pistikprogrammide analüüsimise kaudu. Paraku on keeruline SDK kätte võtta ja midagi funktsionaalset kiiresti luua. Bakalaureusetöö väikese mahu tõttu ei ole seega selle teostus võimalik. Pigem on see heaks teemaks suurema mahuga magistritöö jaoks.

4. Interpretaatorid

Oletades, et on olemas pistikprogramm, mis suudab edukalt valikuliselt koodis baitkoodi eraldada, siis on vaja interpretaatorit, mis selle kätte võtaks, seda analüüsiks ning tagasisidet annaks. Tagasiside edastamine ja kuvamine pistikprogrammis jääb pigem disainiküsimuse alla, kuna põhimõttelt midagi keerulist ei ole. Pigem tekitab küsimust interpretaatori valik, kuna neid on erinevaid ja nende tööpõhimõtted alati ei kattu.

4.1 Raamistik Checker

Java tüüpide süsteemi laiendav raamistik Checker pakub erinevat laadi kontrollijaid, mida saab koodile juurde lisada annotatsioonide kujul [4]. Olemas on erinevaid kontrollijaid, näiteks annotatsioon *@Interned*, mis määrab, et etteantud tüübi puhul on mingid objektid või alamosad korduvkasutuses, või annotatsioonid *@Tainted* ja *@Untainted*, mis kontrollivad lekkete püüdmist. Viimase kahe annotatsiooni puhul on loodud pistikprogramm IntelliJ IDEA jaoks ühe Tartu Ülikooli magistritöö raames, mis nende annotatsioonide abil otsis programmis SQL süsti nõrkuse võimalusi, hoiatas nendest ning isegi pakkus lihtsamaid parandusi [29]. Seega selliste kontrollijate ühildamine pistikprogrammiga on kindlasti võimalik.

Checker raamistik võimaldab ka neid kontrollijaid juurde luua, mis sarnaselt olemasolevatele uurivad mingi tingimuse kehtimise kohta, mille kohta on olemas põhjalik juhend ja mingi arv näiteid [30]. Selle alusel võiks olla võimalik luua kontrollijaid, mida saaks sarnaste annotatsioonidega kasutada nagu Dafnys.

Kui need kontrollijad on loodud, siis teoreetiliselt, vaadates Valgma magistritööd, peaks olema võimalik nende ühendamine pistikprogrammiga sarnasel moel. Küll aga on sellise ettevõtmise maht pigem suur kui väike.

4.2 Staatilise analüüsi raamistik Põder

Tartu Ülikoolis loodud staatilise analüüsi raamistikuga Põder saab analüüsida kompileeritud Java koodi ja selle kohta tagasisidet saada, kus iga programmi sammu kohta antakse infot. Selle jaoks saab mooduleid juurde arendada, mis kindlat sorti analüüsi teostavad. Samuti Tartu Ülikoolis on erinevaid selliseid mooduleid loodud, näiteks Mullari bakalaureusetöö käigus loodud Java baitkoodi sünkroniseerimise analüüs [31] või Sinisalu bakalaureusetöö kestel loodud intervallanalüüs [32], mis sarnaneb rohkem käesoleva bakalaureusetöö temaatikaga.

Sarnaselt Checkerile on seega kerge näha, et raamistik Põder võimaldab erinevate kontrollijate kui moodulite arendamist. Paraku aga puudub näide integratsioonist arenduskeskkonnaga. Selle bakalaureusetöö esialgne eesmärk oligi see puudujääk katta vastava pistikprogrammi loomisega, kuid bakalaureusetöö tavapärane maht ei ole piisav nii keerulise ülesande puhul. Valgma tööd vaadates on ilmne, et tegu ei ole triviaalse katsumusega ning ka lihtsamat laadi veaanalüüs nõuab suures mahus süvenemist ja katsetamist [29].

4.3 Interpretaatori valik

Kahe uuritut interpretaatori vahel ei ole ühte õiget valikut. Teoreetiliselt on ülesande lahenduseks sobilikum raamistik Põder, kuna selle loomisel on sarnaseid probleeme silmas peetud. Samas on Checkeri puhul olemas projekt, kus on loodud sarnase põhimõttega pistikprogramm. Seega, väiksema võimaliku ajapanuse puhul on Checker mõistlikum variant, kuna vähem tuleb nullist midagi valmis teha ja on suurem tõenäosus, et ei satuta tööd seiskava probleemi otsa. Põder raamistiku puhul on kompimata maad rohkem ja seetõttu ei saa täie kindlusega öelda, et pistikprogrammiga ühildumine kindlasti võimalik oleks.

Olenemata valikust on siiski vaja kummaski raamistikus valmis luua kontrollijad, mis analüüsimist teostaks. Seepärast on igal juhul terve pistikprogrammi maht suur ning tasub kaaluda tükeldamist analoogselt just uuritud magistri- ja bakalaureusetöödele.

5. Järeldus

Sisendiga veanalüüsi puhul on hea mõte valida lähenemine, mille tulemusena on arendajal kõige kergem seda kaasata enda töövoogu. Sellest tulenevalt on igasugused välised rakendused kontekstivahetuse tõttu keerukad ning oluliselt kasutajasõbralikum lähenemine on pistikprogrammi loomine.

Pistikprogrammi luues on JetBrains enda poolt olemas pinnapealne dokumentatsioon ja arv näiteid. Selline vähene kogus infot teeb arendamise alustamise raskeks, eriti suuremate projektide puhul. Suurem osa ajast kulub uurimisele ja katsetamisele, mida hoiaks ära sisukam ja näiterikkam dokumentatsioon. Sellest tulenevalt on pistikprogrammi alusteks mõistlik valid levinumad tehnoloogiad, ehk programmeerimiskeeleks Kotlini asemel Java ning põhjaks Gradle. Niiviisi on olemasolevaid näiteid uurides vähem eksimist tehnoloogiate erinevuste tõttu.

Interpretaatoriks kui mootoriks on valik kahe vahel ning see suuresti sõltub ajapanusest. Ideaalkorras sobiks raamistik Põder, kuid JetBrainsi pistikprogrammide tarkvaraarenduspaketiga ühildamise näide puudub, seega kindlam võib olla juba läbi käidud tee ehk raamistik Checker.

Interaktiivse abstraktse interpretaatori loomine on kindlasti võimalik eelnevalt mainitud tehnoloogiatega, kuid see on suuremahulisem ja mitme osaga projekt.

6. Kokkuvõte

Programmeerimiskeel Java, toetudes suuresti tüüpide süsteemile, ei anna arendajal palju võimalusi infot anda rakenduse töö kulgemise kohta. Võrreldes keeltega, kus on selleks rohkem võimalusi, nagu Dafny, siis on eelised kergelt tuvastatavad ja omaksid väärtust ka Java keskkondades. Sarnaseid funktsioone on proovitud Javasse tuua, kuid need on tihti kas eraldi rakendustena või aastaid unarusse jäetud. Uudsem lähenemine probleemile oleks luua pistikprogramm hästi teatud tehnoloogiate alusel. Selline sujuv integratsioon arenduskeskkonnaga on arendajate poolt soositud.

Selles bakalaureusetöös sai uuritud ja hinnatud, kuidas sellist pistikprogrammi loomist teostada. Iga sammu juures selgus, et on üsnagi suuremahuline töö, mida igal etapil tuleks teha. Pistikprogrammi ja interpretaatori raamistiku moodulite loomine pole kumbki triviaalsed, kuid need tasub võtta hea kandidaadina magistritööks. Käesolev töö omakorda oluliselt lihtsustab selle alustamist, andes põhjaliku ülevaate ja hinnangu hetke seisukorrast.

7. Viidatud kirjandus

[1] K. Rustan ja M. Leino, "Dafny Programming Language", GitHubi projekt. https://github.com/dafny-lang/dafny (04.04.2021)

[2] Microsoft Research, "Getting started with Dafny". https://rise4fun.com/Dafny/tutorial (04.04.2021)

[3] JetBrains s.r.o., "IntelliJ Platform SDK". https://plugins.jetbrains.com/docs/intellij/welcome.html (04.04.2021)

[4] The Checker Framework. https://checkerframework.org/ (04.04.2021)

[5] K. Apinis, Tarkvarateaduse labor, Tartu Ülikool, "Staatilise Analüüsi Raamistik Põder", Bitbucketi projekt. https://bitbucket.org/kalmera/poder/src/master/ (04.04.2021)

[6] Oracle Coproration, "Programming With Assertions". https://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html (21.04.2021)

[7] Oracle Corporation, "Class Error". https://docs.oracle.com/javase/7/docs/api/java/lang/Error.html (21.04.2021)

[8] eTutorials, "Assertions". http://etutorials.org/Programming/Java+performance+tuning (21.04.2021)

[9] Pivotal Software, "Spring Framework 5.3.6 API, Class Assert". https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/util/Assert.html (22.04.2021)

[10] K. Beck, E. Gamma, D. Saff, K. Vasudevan, "JUnit API, Class Assert". https://junit.org/junit4/javadoc/latest/org/junit/Assert.html (22.04.2021)

[11] Oracle Corporation, "Class NullPointerException". https://docs.oracle.com/javase/7/docs/api/java/lang/NullPointerException.html (22.04.2021)

[12] JetBrains s.r.o., "@Nullable and @NotNull". https://www.jetbrains.com/help/idea/nullable-and-notnull-annotations.html#nullable (22.04.2021)

[13] C. Hawblitzel, J. Lorch, M. Moskal, N Swamy, "Dafny: A Language and Program Verifier for Functional Correctness". https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/ (22.04.2021)

[14] J. P. Lewis, U. Neumann, "Performance of Java versus C++". http://scribblethink.org/Computer/javaCbenchmark.html (24.04.2021)

[15] M. Fähndrich, "Static Verification for Code Contracts", in Static Analysis, Berlin, Heidelberg, 2010, pp. 2–5.

[16] Eiffel Software, "Building bug-free O-O software: An introduction to Design by Contract(TM)". https://archive.eiffel.com/doc/manuals/technology/contract/ (24.04.2021)

[17] J. Bergström, C4J-Team, "The C4J Contracts Framework for Java", GitHubi projekt. https://github.com/C4J-Team/C4J (24.04.2021)

[18] J. Bergström, C4J-Team, "C4J Syntax reference". https://c4j-team.github.io/C4J/syntax.html (24.04.2021)

[19] N. Minh Lê, C. West, L. Miller-Cushon, "Cofoja", GitHubi projekt. https://github.com/nhatminhle/cofoja (24.04.2021)

[20] J. Link, "jqwik: Property-Based Testing in Java". https://jqwik.net/ (25.04.2021)

[21] F. Haag, "JML - Java Modeling Language". https://federicohaag.medium.com/jml-java-modelling-language-55057d03bc51 (26.04.2021)

[22] D. Cok, J. L. Singleton, G. T. Leavens, "OpenJML", GitHubi projekt. https://github.com/OpenJML/OpenJML (26.04.2021)

[23] R. Hähnle, W. Menzel, P. Schmitt, "The KeY Project". https://www.key-project.org/applications/program-verification/ (26.04.2021)

[24] H. T. Kiev, "Verifying EVA: Formal verification of the software for deciding Norwegian governmental elections", in Formal Verification, Oslo, 2020, pp. 1–5.

[25] R. Zwitserloot, R. Spilker, "Project Lombok". https://projectlombok.org/ (27.04.2021)

[26] D. Hovemeyer, "SpotBugs", GitHubi projekt. https://github.com/spotbugs/spotbugs (27.04.2021)

[27] T. Kuhn, Eye Media GmbH, "Abstract Syntax Tree". https://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/index.html (04.05.2021)

[28] JetBrains s.r.o., "IntelliJ SDK code examples". https://github.com/JetBrains/intellij-sdk-code-samples (04.04.2021)

[29] L. Valgma, "Usable and Sound Static Analysis through its Integration into Automated and Interactive Workflows", pp. 16-17.

[30] The Checker Framework, "The Checker Framework Manual: Custom pluggable types for Java". https://checkerframework.org/manual/#creating-a-checker (04.04.2021)

[31] H. Mullari, "Java baitkoodi sünkroniseerimise analüüs raamistikus Põder", p. 2.

[32] A. Sinisalu, "Java programmide staatiline intervallanalüüs raamistikus Põder", p. 2.

Lisad

I. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, Alex Viil,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose **Interaktiivne abstraktne interpretaator IntelliJ IDEA** jaoks, mille juhendaja on Vesal Vojdani, 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.

Alex Viil

07.05.2021