T A R T U Ü L I K O O L

MATEMAATIKA-INFORMAATIKATEADUSKOND

Arvutiteaduse instituut

Informaatika eriala

Ivo Seeba

Krüptoanalüsaatori täiendamine teisenduste keelega

Magistritöö (30 EAP)

Juhendaja: prof. P. Laud

Autor: ..... “.....” juuni 2011

Juhendaja: ..... “.....” juuni 2011

Lubada kaitsmisele

Professor: ..... “.....” juuni 2011

TARTU 2011

Sisukord

Sissejuhatus

Viimase poole sajandi jooksul on arvutiteadus palju arenenud. On tekkinud lugematul hulgal erinevaid tarkvarasüsteeme. Arendatud on erinevaid mobiil- ja *Interneti*-teenuseid: pangandus, digitaalallkiri, *ID*-kaardi tugi, e-poed jpm. Paraku kõigi nende teenustega kaasnevad ka ohud: ebaausad teenuste kasutajad, kes tegelevad näiteks pangakaardi pettustega, võltsivad erinevaid elektroonilisi dokumente jne. See on tinginud vajaduse suhtluse käigus vahetatavate sõnumite saladuses hoidmise järele. Eesmärk on sõnumeid krüpteerida nõnda, et lahtikrüpteerimine on võimalik vaid selleks volitatud salavõtit teadvatel vastuvõtjatel.

Interneti ohtude probleemi lahendamiseks on mõeldud erinevaid krüptograafilisi protokolle (st osapooltevahelisi suhtlemisviise), mille (mitte)turvalisust üritatakse tõestada. Protokollide turvalisuse tõestusteks on erinevaid mooduseid: matemaatilised teadmised (nt tõenäosus- keerukus- ja arvuteooria, algebra ning kombinatoorika), mängude [4, 29] ja nendele vastavad sõltuvusgraafide [19, 30, 18, 20] jadad. On koostatud ka spetsiaalseid arvutiprogramme [19, 6, 5, 18].

Käesolev töö on raamistiku *Krüptoanalüsaator* [18] täiendamine spetsiifilise programmeerimiskeelega, mis võimaldab kirja panna sõltuvusgraafi fragmentide (s.o alamgraafide) teisendusi. Seda keelt nimetame siin töös teisenduste keeleks. Analüsaatorirakendus ja sellele lisatud teisenduste keel on kirjutatud *OCaml*'is [17]. Analüsaator ise töötab tarkvara *uDraw(Graph)* [1] abil.

Keel võimaldab vältida iga uut liiki teisenduse jaoks eraldi *OCaml*-mooduli kirjutamist ning teisenduste kirjelduste koostamist tihti alla kümne koodirea abil. Enne käesoleva töö kirjutamist on mitmesajast reast koosnevat teisendusmooduleid programmeeritud üle viiekümne. Raamistikku on algusest peale teinud P. Laud, I. Tšahhirov ja E. Jaaniso.

Esimeses peatükis antakse sõltuvusgraafide, krüptoloogias nende kasutamise ja varasemate teemaga seotud tööde kohta üldine ülevaade. Teises peatükis on ära toodud teisenduste keele formaalne grammatika ja sellele vastav semantika. Kolmas peatükk kirjeldab lahenduskäiku, kus on keelega seotud failide struktuur ja tegevused transformatsiooni käigus. Enne implementimist on uuritud keelt *OCaml* [17, 9, 15, 16]. Neljandas peatükis on juttu *Krüptoanalüsaatori* käivitamisest ja teisenduste keelsete koodide kasutamisest sõltuvusgraafide transformatsioonides ning antakse selle kohta lühike näide. Lisades 1 ja 2 on ära toodud nimekirjad vastavalt *DGFR*-tippude operatsioonidest ja teisenduste keele veateadetest. Lisas 3 on *CD*-plaat täiendatud *Krüptoanalüsaatori* ning vajalike muude tarkvarade ja materjalidega.

1. peatükk

Sõltuvusgraafide kasutamine krüptoanalüüsis

Osapooltevaheline suhtlus (nt server ja klient) on tarvis korraldada krüpteeritud sõnumitega ning peab kindel olema et protokoll täidab talle etteantuid turvaomadusi. Turvalisuse all mõistame kolme komponenti: konfidentsiaalsust, terviklust ja käideldavust [30, 20]. Konfidentsiaalsuse all mõistame varade kättesaadavust vaid valitud osapooltele (nt teatud vastuvõtjale krüpteeritud sõnumi lugemine), terviklikkus on varade modifitseerimisvõimalus vaid volitatud osapooltele ja käideldavus on varade kasutatavus. Ehkki sõltuvusgraafidest ja nende abil krüptoanalüüsist on pikalt juttu allikates [30], [19], [18] ja [20], on ka siin sõltuvusgraafide teema kohta ülevaade.

1.1. Mängud

Turvatõestuste üks levinud meetod on mängutehnika [4, 29], kus krüptograafilist protokolli näidatakse mänguna. Mänguks nimetatakse suhtlust ründaja[1] ja keskkonna vahel, mis kujutab endast pseudokoodi krüptosüsteemi turvalisuse formaliseerimiseks. Mängude kasutus on levinud ning on krüptoprotokollide turvalisuse uurimiseks võimas ja keskne vahend. Nii ründajat kui ka teisi osapooli keskkonnas näidatakse alamfunktsioonidena pseudokoodis. Mänge võib jagada mõnikord alamkomponentideks ja nende kirjutamisel võib kasutada vastavaid formaalseid protokollikeeli [4, 30]. Mängu teisendus on mängu muutmine nõnda, et säiliks samade sisendite korral senise ja muudetud mängu väljundite eristamatus ründaja jaoks. Mängu sisend ja väljund defineeritakse vastavalt ründajalt saadav ja talle edastatav info. Vastase edukust defineeritakse suutelisust talle jõudva info põhjal seda välja lugeda (piiratud katsete arvu korral). Kahte mängu loetakse arvutuslikult eristamatuks (samade sisendite korral), kui nende väljundi(te)lt tulevast infost vastase poolt arusaamise tõenäosuste erinevused liginevad nullile.

Mängu teisendatakse alati lihtsuse suunas. Mängu enne sellist muutmist ja mängu pärast seda nimetame käesolevas töös naabermängudeks. Naabermängud on eristamatud, kui mängude väljundeid jälgiv ja mõjutav arvutusliku mudeli aktiivne vastane ei suuda mittetühiste tõenäosustega kindlaks teha, kumba ta mängib. Seda kaduvväiksust võib

[1] Kasutatakse ka nimetust "vastane" (inglise k. *adversary*)

säilitada matemaatiliste funktsioonide ja krüptoprimitiivide (nt *Diffie-Hellman*'i otsi- ja otsustusülesande, [29]) eelduste abil. Teisendused võivad jaguneda tüüpideks: eristamatus, ebaõnnestumise sündmus ja ületussammud [29]. Mis tahes krüptograafilise protokolli mängude ümberkirjutamise teel (mitte)turvalisuse tõestuse käigus järjest tekkinud mänge, millest esimene mäng kujutab endast reaalset ja viimane (turvalise protokolli korral) ideaalset protokolli, nimetatakse mängude jadaks. See tekitatakse mängu teisendamise teel, seejuures säilitades naabermängude omavahelist eristamatust. Kui kõik mängud on järjest oma naabriga eristamatud, siis ka mängujada esimene ja viimane mäng on omavahel arvutuslikult eristamatud, mis tuleneb eristamatuste transitiivsest seosest. Kui selline mängujada on õnnestunud tekitada, siis on selge, kas krüptosüsteem on turvaline.

Artiklis [21] ja selle pikemas versioonis [22] käsitletakse staatilise analüüsi tehnikaid krüptoprotokollide keerukusteoreetilise turvalisuse kindlakstegemiseks. Defineeritud on mustreid, mille turvalisus on samaväärne vastavate protokollidega. Ründajat ja teisi osapooli on kujutatud efektiivsete algoritmidega ja teateid osapoolte vahel bitijadadena. Protokollide süntaksit kirjeldatakse vastava formaalse keelega. Selle keele abil näidatakse mängude ümberkirjutamise tehnikat, kus tõestused on mängujadad. Näitena kasutatakse artiklis [21] *Needham-Schroeder* võtmevahetuse protokolli.

Mängujadade abil tõestusteks on välja mõeldud ka arvutiprogramme, mis genereerivad mängujadasid automaatselt. Näiteks on B. Blanchet koostanud formaalsel (kergesti kasutatava *Dolev-Yao*) mudelil põhineva tarkvara *ProVerif* [6]. Blanchet on arendanud ka teise rakenduse – arvutuslikul mudelil (keerukus- ja tõenäosusteoorial) põhineva *CryptoVerif*'i [5], mis kasutab sisemiselt süntaksipuud. Mõlemad programmid põhinevad π-arvutusel ja kirjutatud keeles *OCaml*.

1.2. Sõltuvusgraafid

Mängutehnikast uuem moodus on sõltuvusgraafide kasutamine. Mänge tõlgitakse sisemiselt sõltuvusgraafideks, mis välimuselt sarnanevad visuaalse programmeerimiskeskkonna *LabVIEW* koodidega [7, 8], kus tipud kujutavad samuti erinevaid operatsioone. Kui *LabVIEW* koodi servad kannavad edasi (olenevalt operatsioontipu väljundist) sõnesid, täisarve, ujuvkomaarve, jadasid, eri tüüpi väärtuste ühendeid, siis krüptoanalüüsis kasutatavate sõltuvusgraafide koodiservad võivad edastada tõeväärtust või bitijada. Sõltuvusgraafid kujutavad krüptograafilistele mängudele vastavat visuaalset koodi, mille vähehaaval muutmise teel saadakse mängujadaga samaväärne sõltuvusgraafide jada, millest esimene kujutab reaalset ja võib olla ideaalne protokoll. Mänge on võimalik teisendada sõltuvusgraafideks ja vastupidi. Imperatiivse programmi miinuseks on see, et programm võiks tahta muutujaid üle kirjutada, kuid peale teisenduste tekkimist võib tulla olukord, kus samast muutujast on korraga vaja koopiat nii enne kui peale tema modifitseerimist.

Sõltuvusgraafide eelisteks on mugavus (süntaksipuudega võrreldes) neid teisendada ideaali suunas, lokaalse teisendamise (sõltuvusgraafi fragmendi asendamine teise arvutuslikult eristamatu fragmendiga) jaoks paremad tingimused, graafi suutelisus endas hoida kontrollivoo sõltuvusi kõikide protokollis toimuvate arvutuste vahel, võimaldamine keskenduda andme- ja kontrollivoo olulistele detailidele, visuaalse koodi lugemise ja kasutamise lihtsus võrreldes tekstilisega, kogu vajaliku info kättesaadavus protokollile vastavale

graafile ning loomulikud teisendused, mida süntaksipuud ei võimalda.

ProVerif'ist ja *CryptoVerif*'ist parema sisemise struktuuri ja graafilise liidesega tarkvara on sõltuvusgraafidel põhinev *Krüptoanalüsaator*, mille kohta on pikemalt juttu artiklis [18] ja doktoritöös [30]. P. Laud ja I. Tšahhirov on teinud *Krüptoanalüsaatorile* ka varasema versiooni ([19], Lisas 3 *grb090210.zip*), mis tekitab etteantud protokollide automaatse tõestuse. Täisautomaatse analüsaatoriga võib juhtuda, et see liigub tõestusega vales suunas, st. teisendused, mis ta teeb, pole turvalise protokollini jõudmisel alati kõige õigemad. Analüsaatori sobivas suunas liikuma panemiseks on välja pakutud palju heuristikaid. Töödes [18] ja [20] käsitletakse ka enne teisenduste keele lisamist tehtud analüsaatorist ja selle kasutamisest.

Esimene teadaolevalt eestikeelne materjal sõltuvusgraafide teemal on bakalaureusetöö [20], kus kirjeldatakse senist krüptoanalüsaatori kasutamist, sisendit võtva teisenduse lisamist analüsaatorile ja seda, mida peaks analüsaatorile veel lisama. Üheks ideeks oli selles töös oli teisenduste keelega genereerida uusi transformatsiooni mooduleid, kuid ülesande õnnestus ka ilma selleta lahendada.

1.3. Lõplikud sõltuvusgraafid

Sõltuvusgraaf (*Dependency Graph – DG*) on suunatud graaf, mille tipud on krüptosüsteemi poolt teostatavad operatsioonid ja servad (ühesuunalisi servi nimetatakse ka kaarteks), tippude vahel on sõltuvused, mis kannavad edasi serva algtippude operatsiooni väljundeid lõpptippudesse. Tippudel võib olla üks või mitu vahetut eellast ja järglast. Tipu vahetuks eellaseks nimetatakse sellesse tippu saabuva serva algtippu. Tipu vahetuks järglaseks nimetatakse sellest tipust väljuva serva lõpptippu. Lõpliku sõltuvusgraafina (lõpliku arvu tippude ja servadega) näidatakse piiratud sessioonide arvuga krüptoprotokolli.

Servadena kujutatud sõltuvusi jagatakse juht- (milleks sobib vaid tõeväärtuse serv) ja andmesõltuvusteks. Juht- ehk kontrollsõltuvus on mõeldud selleks, et määrata, kas tipp töötab ja väljastab vastava operatsiooni tulemuse. Kui juhtsõltuvus on väärtusega *true* ja sisenditel kõik andmesõltuvused on väärtustatud, siis tipp töötab ja väljundserva(de) olemasolul tulemus väljastatakse. Seejuures kõikide sobivate sisendite (st $\perp$-ist erinevad) korral tulemus on tõeväärtuse puhul *true* ja bitijada puhul mittevõrdne väärtusega $\perp$. Kui juhtsõltuvuse väärtus on $false$, siis tipp on "välja lülitatud" ja väljundiks on $\perp$ või $false$ olenevalt väljundi tüübist. Juhtsõltuvused on *MUX*-tippude normaalselt töötamiseks st et *MUX*-tippu vaid üks sisend tuleks. Saab näidata nõnda, et *Id*-tipule jääks juhtsõltuvus ja teistel ei ole vaja. Aga teisi teisendusi on siis raskem teha, kui soovime öelda, et kui üks asi toimib, siis töötab ka teine asi.

Krüptograafilist protokolli kujutava graafi *semantika* (formaalselt on toodud allikas [30]) on alguses see, et iga tipu väljund on olenevalt selle tüübist (tõeväärtus või bitijada) $false$ või $\perp$ ning fikseeritakse kõik juhuslikud väärtused. Järgmisena korraldatakse tsüklit, kuni vastane otsustab selle katkestada. Iga tsükli samm on selline, et protokolli käivitamise algul seab vastane mõne *Req*-tipu väljundi väärtuseks *true* ja võib lisaks ka mõnda *Receive*-tippu muuta väärtust $\perp$ mingiks muuks väärtuseks, sooritab operatsioone niikaua kui võimalik ning kõik *Send*-tippude väärtused, mis $\perp$ asemel muutusid mingiks bitijadaks, saadetakse talle. Etteantud protokolli käivitab ründaja sel teel korduvalt, ku-

ni tema saab *Send*-tippudest mingi kasuliku info, mille tulemusel otsustab ta selle tsükli katkestada [30, 20].

1.4. Lõpliku sõltuvusgraafi fragmendid

Sõltuvusgraafi teisendamisel ründaja vaatevinklist arvutuslikku eristamatust säilitades ei pea tervet graafi ümber korraldama, vaid seda saab teha teatud alamgraafi teisendamisega. Sellist alamgraafi nimetatakse sõltuvusgraafi fragmendiks (*Dependency Graph Fragment – DGF*). Fragmendi vaatevinklist on lisaks graafi enda tippudele spetsiaalsed sisend- ja väljundtipud, mis sisuliselt on fragmenti kuuluvate tippude vahetud eellased ja järglased fragmenti mitte kuuluvate tippude seas.

Kui terve graaf on lõplik, st tippude ja kaarte arv ei ole lõpmatu, siis on ka fragment lõplik. Sellega saame lõpliku sõltuvusgraafi fragmendi.

1.5. Lõpmatud sõltuvusgraafid

Mis tahes osapool võib sõltuvusgraafina esitatud protokolli käivitada korduvalt ja ründaja võib protokolli "jooksutada" piiramatu arv kordi. Seepärast on välja mõeldud replikatsioonid n-ö ühekordse kasutusega tippude kirjeldamiseks. Kuna graafina esitatava protokolli töö käigus kasutatakse teatuid tippe korduvalt, siis võime tippe replitseerida – teha koopiaid, et igat tippu kasutatakse vaid üks kord. Igal koopial on replikatsiooni indeks, mis näitab, mitmendal sessioonil tipp osaleb. Tipu replitseerimine tekitab aga lõpmatu sõltuvusgraafi (*Infnite Dependency Graph – IDG*). Hoolimata graafi lõpmatusest jääb see siiski piisavalt regulaarseks ja on kujutatav lõpliku graafina. See tähendab, et graafis ei pea mingit lõpmatut tipu koopiate rida ilmutatult kujutama. Selle asemel võime esitleda ühte tippu, millel on dimensioonide arv, mis näitab, mitu korda on tipp replitseeritud. Sellega saame lõpliku sõltuvusgraafi esituse (*Dependency Graph Representation – DGR*). Dimensioonide juures on märgitud ka protokolli jooksutavate osapoolte nimed. Näiteks olgu mingil tipul n dimensioonid $d(n) = [A \to 1, B \to 1, S \to 1]$, siis see näitab, et igale kolmikule osapoole A-sessioonist, osapoole B-sessioonist ja osapoole S-sessioonist vastab tipu n üks võimalik käivitus.

Ka lõpmatust sõltuvusgraafist on võimalik saada fragment, mille tipud, sealhulgas spetsiaalsed sisend- ja väljundtipud, on esitatavad lõplikena, mistõttu terve fragment on esitatav lõpliku fragmendina. Sellega saame sõltuvusgraafi fragmendi esituse (*Dependency Graph Fragment Representation – DGFR*). *Krüptoanalüsaatoris* teisendataksegi *DGFR*'ide abil neid sõltuvusgraafi esitusi, millega on võimalik saada mängudele vastav sõltuvusgraafi esituste jada, millest samuti esimene näitab reaalset protokolli. Kui protokoll täidab turvaomadusi, siis jada viimane *DGR* näitab ideaalset protokolli. *DGFR*'e enne teisendust ja pärast teisendust nimetatakse vasakuks ja paremaks *DGFR*'iks. Ideaalne protokoll tähendab, et *DGR*'i lihtsamaks teha ei ole enam võimalik ja sellele vastava protokolli turvalisus on ilmne.

1.6. Replikatsiooni dimensioonid

Lõpmatu sõltuvusgraafi esituse igal tipul n üks olulisemaid parameetreid on replikatsiooni dimensioonid

$$d(n) = [P_1 \to D_1, \ldots, P_k \to D_k], \tag{1.1}$$

kus k on osapoolte arv, $P_1, \ldots, P_k$ on osapoolte nimed ja $D_1, \ldots, D_k$ osapooltele vastavad tipu n replikatsiooni dimensioonid. Näiteks kui tipu dimensioonid on

$$A \to 1, B \to 2, C \to 4, \tag{1.2}$$

siis see tähendab, et tipp on seitsmemõõtmeline: $a_1, b_1, b_2, c_1, c_2, c_3, c_4$ mis näitab dimensioonide numbreid. Replikatsiooni dimensioone tähendus defineeritakse rekursiivselt. Nulldimensioon ($d = 0$) tähendab üksikut tippu ja dimensioon $d = 1$ lõputu arv tipu n koopiaid $\{n_1, n_2, \ldots\}$. Dimensioon $d = 2$ on see, et jada $\{n_1, n_2, \ldots\}$ saab paljundada lõputult, et tekiks kahemõõtmeline $\{\{n_{1,1}, n_{2,1}, \ldots\}, \{n_{1,2}, n_{2,2}, \ldots\}, \ldots\}$. Dimensioon $d = 3$ näitab sellest kahemõõtmelisest jadast kolmemõõtmelise

$$\{\{\{n_{1,1,1}, n_{2,1,1}, \ldots\}, \{n_{1,2,1}, n_{2,2,1}, \ldots\}, \ldots\}$$
$$\{\{n_{1,1,2}, n_{2,1,2}, \ldots\}, \{n_{1,2,2}, n_{2,2,2}, \ldots\}, \ldots\}, \ldots\}$$

tekkimist jne. Seega d-dimensiooniline tipp on üksüheses vastavuses hulgaga $\mathbb{N}^d$. Hoolimata lõpmatust graafist on ka piirangud. Ründaja tööaeg tähendab seda, et lõpmatust graafist on ainult lõplik alamgraaf väärtustatav.

1.7. Dimensioonide kujutused

Sõltuvusgraafi esituse servade üheks peamiseks parameetriks on kujutused, mis kujutavad algtipu mingi dimensiooni lõpptipu dimensiooniks ning nende kirjapilt on

$$[(P_1, 1) \to D_{1,1}, \ldots, (P_1, D_1) \to D_{1,D_1}, \ldots, (P_k, 1) \to D_{k,1}, \ldots, (P_k, D_k) \to D_{1,D_k}]. \tag{1.3}$$

Tähised $P_1, \ldots, P_k$ näitavad osapooli, $D_1, \ldots, D_k$ on algtipu dimensioonid ja D_{i,j_i} ($1 \leqslant i \leqslant k, 1 \leqslant j_i \leqslant i$) on lõpptipu dimensioon, milleks osapoolele P_i vastav algtipu dimensioon j_i kujutub.

Näiteks $(B, 1) \to 2$ tähendab, et serva algtipu esimene B-dimensioon kujutub lõpptipu teiseks B-dimensiooniks. Selgem oleks seda kujutust esitada $(B, 1) \to (B, 2)$, kuid see on liiaga, sest kujutis peab samuti kuuluma osapoolele B. Serva lõpptipul peab iga osapoole nimega dimensioone olema vähemalt sama palju kui algtipul, mis tähendab, et olgu algtipu dimensioon $P \to a$ ja lõpptipu dimensioon $P \to b$, siis $a > b$ ei saa kehtida[2]. Kui algtipu dimensioon on antud näites (1.2), siis serva kujutused on

$$(A, 1) \to a', (B, 1) \to b_1, (B, 2) \to b_2, (C, 1) \to c_1, (C, 2) \to c_2, (C, 3) \to c_3, (C, 4) \to c_4,$$

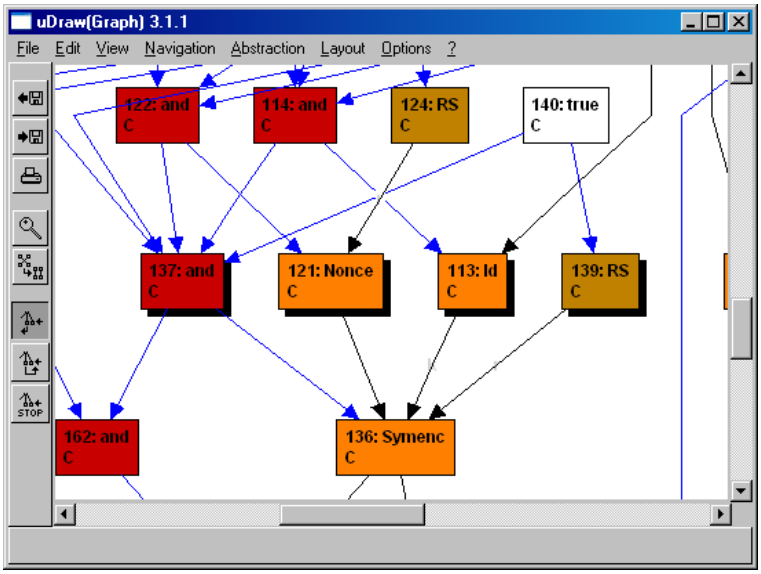
kus $1 \leqslant b_1, b_2 \leqslant b'$ ja $1 \leqslant c_1, c_2, c_3, c_4 \leqslant c'$ ning $b_1 \neq b_2$ ja $\forall i, j \in \{1, .., 4\} : c_i \neq c_j$ ($a' \geqslant 1, b' \geqslant 2, c' \geqslant 4$ ja $i \neq j$).

[2]Erandiks on ainult teatud tipud, mis dimensioone kokku tõmbavad, kuid neid me selles töös ei vaata.

1.8. *DGFR*-transformatsioon

Analoogselt mängudega teisendatakse sõltuvusgraafide esitusi, et naabersõltuvusgraafidel säiliks arvutuslikult eristamatus. Vasak *DGR* (*DGR* enne teisendust) ja parem *DGR* (*DGR* pärast teisendust), millel on samad sisend- (*Req*, *Receive*) ja väljundtipud (*Send*), on väljunditelt arvutuslikult eristamatud, kui ründaja $\mathcal{A}$ ei suuda teha vahet kumba *DGR*'ina esitatud protokolli ta "jooksutab". Eristamatust tähistatakse $\mathcal{G}_1^{\mathcal{A}} \cong \mathcal{G}_2^{\mathcal{A}}$, kus $\mathcal{G}_1$ ja $\mathcal{G}_2$ on vastavalt vasak ja parem *DGR*.

Ka siin on kasutusel naabrite nimetus: naabriteks nimetatakse teisenduse vasakut ja paremat *DGR*'i. Kui vasak *DGR* sisaldab vasakut *DGFR*'i, siis teisenduse käigus asendatakse see paremaga. Vasak ja parem *DGFR* antakse ette teisendustekeelse koodiga parsitud *DGFR*-muutujatega. Teisenduse alguses viiakse vasak *DGFR* ja selle muutuja omavahel kokku. *DGFR*-muutujad koosnevad servamuutujatest, milles omakorda on tipumuutujad. Krüptoanalüsaatori raamistikus koosneb fragment *uDraw(Graph)*-aknas valitud tippudest ning fragmendi vaatenurgast sisend- ja väljundtippudeks (joonis 1.1) sobivatest valimata eellastest ja järglastest. Teisenduse käigus ära võetavat ja asemele pandavat *DGFR*'i, nimetatakse reaalseteks *DGFR*'ideks ning nende koostisosi nimetame reaalseteks tippudeks ja servadeks. Joonisel 1.1 kujutatud igal tipul n on täisarvudena identifikaatorid $\ell_1(n)$, operatsioonid $\lambda_1(n)$ ja C on operatsiooni tegeva osapoole nimi. Programmiga *uDraw(Graph)* avatud graafidel kujutatakse tõeväärtuse servi sinise ja bitijada omi mustaga. Tippude värvus analüsaatoril ei tähenda midagi, kuid oma operatsiooniga tipud on kindlat värvi: *Or*-tipp on roheline, *And*-tipp punane, *Symenc*-tipp oranž jne.



Joonis 1.1. Valitud tipud 137, 121, 113 ja 139 ning nende vahetud eellased ja järglased valimata tippude seas moodustavad reaalse *DGFR*'i

1.9. Tippudel kasutatavad pordid

Et tipp saaks ühendada servi teiste tippudega, on tipud varustatud sisendportidega hulgast

IPorts = {	PortRS,	PortPubkey,	PortSigkey,
PortSymkey,	PortText,	PortPubenc,	PortSymenc,
PortSignature,	PortEncKP,	PortSigKP,	PortSingleB,
PortStrictB,	PortUnstrB,	PortLongT,	PortBefLeft,
PortBefRight,	PortCompare,	PortMerge,	PortMUX,
PortToList(n),	PortFromList(n),	PortNand}	

(1.4)

ja ühe väljundpordiga hulgast

OPorts = {	Boolean,	Coins,	Bitstring,	VNonce,	VIntConst,
VSKE,	VPKE,	VSKS,	VPKS,	VSymkey,	VPubenc,
VSymenc,	VSig,	VAny,	VTuple n}		

(1.5)

kus n (port VTuple) on naturaalarv näitamaks sisendite arvu.

Sisendportidest (1.4) tõeväärtuse servade pordid on PortSingleB, PortStrictB, PortUnstrB, PortBefLeft, PortBefRight ja PortNand ning ülejäänud sisendpordid on bitijada servade jaoks. Väljundportidest (1.5) tõeväärtuse servade jaoks on port Boolean, port VAny oli mõeldud bitijada ja tõeväärtuse ühendi jaoks, kuid seda teisenduste keele juures ei kasutata ning ülejäänud pordid on kõik bitijada jaoks.

Igale sisendpordile mahub tavaliselt üks serv, kuid on erandeid. Portidele PortStrictB ja PortUnstrB sobib kui tahes lõplik arv tõeväärtuse serva. Portidele PortMerge ja PortMUX kui tahes naturaalarv bitijada serva. Port PortCompare on täpselt kahe bitijada serva jaoks. Port PortSingleB on kontrollsõltuvuse jaoks, mida leidub peaaegu kõigis tippudes ja see võtab ühe tõeväärtuse serva. Aga väljundportidega on pisut lihtsam. Iga väljundpordiga võib ühendada mis tahes arvu servi, kuid sisendtippude (operatsioonidega *InputB* ja *InputS*) väljundpordi jaoks on analüsaatorile lisatud keeles koht täpselt ühe serva jaoks. See, milline port sellele tuleb, selgub tipumuutujate sobitamise käigus reaalsete tippudega.

1.10. Tippudel kasutatavad operatsioonid

Analüsaatoris *DGR*'idel leidub 49 erinevat operatsiooni, mis on näha failis *GrbGraph.ml* defineeritud tüübis *nodename*. Analüsaatorile lisatud teisenduste keele poolt toetatud operatsioonid on ära toodud Lisas 1.

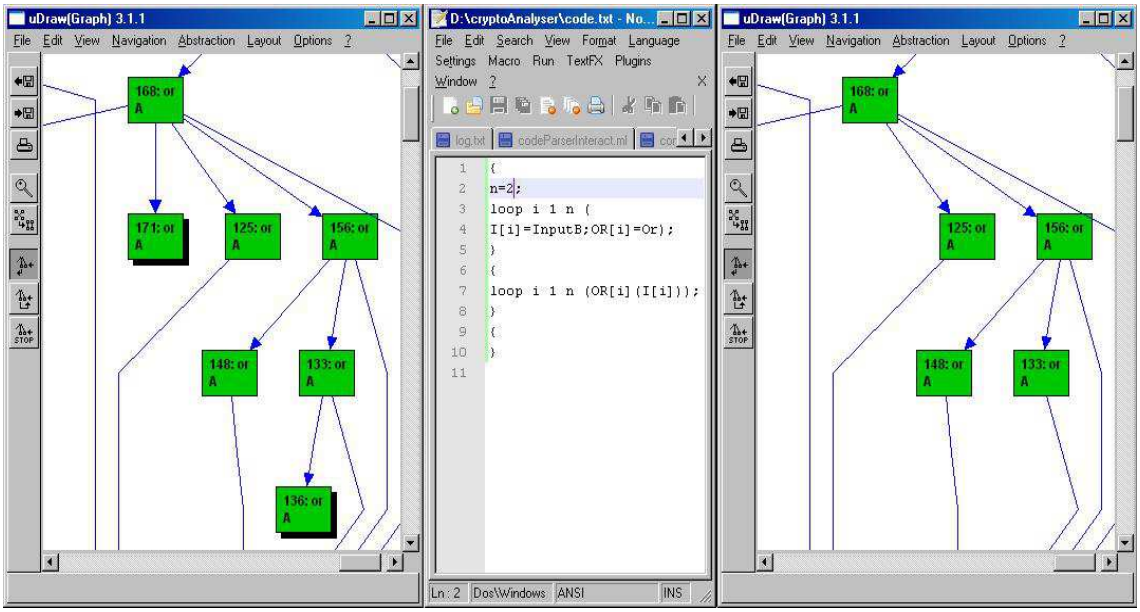
See operatsioonide hulk *Krüptoanalüsaatoris* erineb pisut töös [30] kirjeldatust, kuna seal toodud *And-*, *Or-*, ja *Merge*-tippudel on andmesisendite arve tähistavad parameetrid, mida analüsaatoris ja teisenduste keeles ei kasutata. Ning selles töös ei ole mainitud ka *TTT-*, *Implic-*, *LongMux-*, *ShortMux-*, *LongMux-*, *LongNand-* ja *ShortNand*-tippe.

DGR'ides esineb veel tippe *Send* vastasele teate saatmiseks, *Req* vastase poolt päringu saamiseks, *Receive* vastase poolt teate saamiseks, *Begin* ja *End* on vastavusomaduste kirjapanekuks [13, 12].

Käesolev töö Lisas 1 loetletud tipu operatsioonidest on vaid lõpmatule sõltuvusgraafile omased *DimEq*, *DimNeq*, *LongAnd*, *LongNand*, *LongMux* ja *LongOr* (analüsaator näitab kirjana "ooor"). Lisaks on analüsaatoril ka tipud *CTakeDimEq*, *DTakeDimEq*, *BefP* ja *BefN*, mis samuti esinevad vaid lõpmatutes graafides, kuid teisenduste keel neid tippe ei toeta. Need neli viimast tippu ei kuulunud ka käesoleva töö teemasse.

Vasakusse ja paremasse *DGFR*-muutujasse võivad kuuluda veel eritüüpi tipud: tipp operatsiooniga *InputB*() tõeväärtussisendi jaoks, tipp operatsiooniga *InputS*() bitijada sisendi jaoks, tipp operatsiooniga *OutputB*(*portB*) tõeväärtuse väljundi jaoks ja tipp operatsiooniga *OutputS*(*portS*) bitijada väljundi jaoks. *portB* $\in$ {`Boolean`} ja *portS* $\in$ {`Coins`, `Bitstring`, `VNonce`, `VIntConst`, `VSKE`, `VPKE`, `VSKS`, `VPKS`, `VSymkey`, `VPubenc`, `VSymenc`, `VSig`, `VTuple` *n*} ($n \in \mathbb{N}$).

Lisatud keele jaoks on mõeldud igale servale fragmenti kuuluva ja fragmenti mittekuuluva tipu vahel üks tipp nendest neljast. Analüsaatoris ei ole vasakul ja paremal *DGFR*-muutujatel sisend- ja väljundtippude hulgad üksühesed. On vaid see nõue, et kõik parema *DGFR*'i sisend- ja väljundtipud peavad esindatud olema vasakus *DGFR*'is. Seda selle pärast, et on selliseid teisendusi, kus nt eemaldatakse üleliigseid vaid ühe servaga intsidentseid tippe (joonis 1.2).

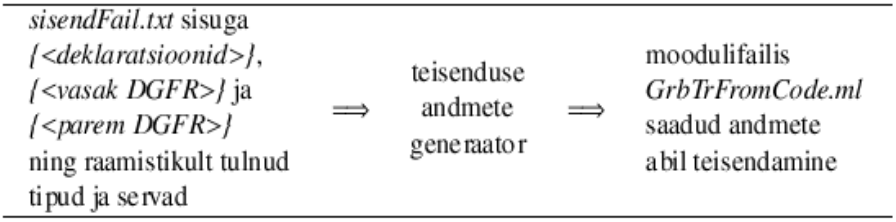


Joonis 1.2. Surnud koodi eemaldamine teisenduste keelega

2. peatükk

Grammatika ja semantika

Käesoleva töö peamiseks ülesandeks on teisenduste keele disainimine ning väljatöötatud programmeerimiskeele parseri lisamine raamistikule *Krüptoanalüsaator*. Selles peatükis vaatleme *Krüptoanalüsaatorile* lisatud keelt lähemalt, et teada saada milline on keele formaalne grammatika ja semantika. Teisendustekeelne sisendfaili sisu koosneb kolmest osast, millest esimene kujutab deklaratsioone, teine transformatsiooni vasakut ja kolmas paremat *DGFR*-muutujat. Neist vasak *DGFR* on eemaldatav graafi osa ja parem see, millega vasak transformatsiooni käigus asendatakse. Koodi kirjutamisel on tarvis jälgida, et säiliks sisendite ja väljundite arvutuslik eristamatus. Käesolevas peatükis vaatleme teisenduse lähtekoodi üldkuju, formaalset grammatikat ja semantikat. Teisenduse skeem on näha joonisel 2.1.



Joonis 2.1. *DGFR'i* teisendamine keele abil

2.1. Tekstiformaadis teisenduse üldkuju

Selles jaotises kirjeldatakse tekstifaili sisu, milles on kolm looksulgudega `'{'` ja `'}'` eraldatud osa, millest rakendus vajaliku info kätte saab. Väljaspool looksulge asuv info võib olla kommentaariks. Ka üherea kommentaare võib kasutada, mis algavad sümboliga `'#'` ja milles asuvad looksulud ei lähe arvesse.

Teisenduste keelse koodifailil laiendil piiranguid ei ole. Koodi võib sisestada tavalisse *.txt*-formaadis faili. Failis peab olema täpselt kolm paari looksulge, kus esimese paari vahel on deklaratsioonid ning teise vahel vasak ja kolmanda vahel parem *DGFR*-muutuja.

2.1.1. Koodi kirjutamine

Deklareerimisel kasutatakse võrdusmärki `'='`. Tipumuutuja deklaratsioon on `V=NE`, kus muutujale `V` omistatakse tipu avaldis `NE`. Avaldiseks võib olla tipp (tähistatakse sellele vastava operatsiooniga) või seda omav muutuja. Arvumuutuja deklaratsioon on `e=Expr`, kus `e` on mingi uus arvumuutuja ja `Expr` on avaldis, mille väärtus arvumuutujale leitakse. Deklaratsioonid eraldatakse semikooloniga `';'`. Deklaratsioonide hulka võib ümbritseda sulgudega `'('` ja `')'`. Arvumuutujad võivad teatud tingimustel väärtusi leida ka automaatselt. Tipumuutujatele peab väärtused kindlasti ette andma. Peale muutujate deklareerimise on võimalik ka muutuja massiive deklareerida stiilis `loop i e1 e2 (A[i]=RS)`, mis on samaväärne deklaratsiooniga `A[e1]=RS;A[e1+1]=RS;..;A[e2]=RS`.

Vasakus ja paremas *DGFR*-muutujas servade kujutamiseks kasutame tippe eellastega, mille kirjapilt on `V(V1;...;VN)`, kus `V` enne algavat sulgu `'('` on lõpptipu muutuja ja `V1`, ..., `VN` on algtippude muutujad. Tipumuutujad sellises kirjutises peavad olema deklareeritud (märgitud vajalikul kujul deklaratsioonide osas). Näiteks eellasi omav tipumuutuja `A(B;C;D)` tähendab, et on kolm servamuutujat $B \to A, C \to A, D \to A$. Sisendid tuleb ka õiges järjekorras paigutada, et tõeväärtussisendid ei suubuks bitijada sisendporti ja vastupidi. Näiteks olgu tipumuutuja eellastega $A(A_1, A_2, A_3, A_4)$, kus A on `Symenc`, A_1 tõeväärtusväljundiga tipumuutuja ning muutujad A_2, A_3 ja A_4 on bitijada väljundiga, siis ei tohi kirjutada $A(A_2, A_1, A_3, A_4)$, mille tulemusena on viga algtipu väljundi ja sisendpordi omavahel sobimatuse tõttu. Sisendeid operatsiooniti saab vaadata käesoleva töö Lisas 1.

Nii deklaratsioonid, sisendid kui ka sisenditega tipumuutujad võib panna tsüklisse. Näiteks `loop i 1 2 (A[i](loop j 1 3 (B[i,j])))` tähendab servamuutujate hulka $\{B_{11} \to A_1, B_{12} \to A_1, B_{13} \to A_1, B_{21} \to A_2, B_{22} \to A_2, B_{23} \to A_2$. Tsüklilausete kirjutamisel peavad avaldised olema tühikute, tabulaatorite või reavahetustega eraldatud (nt `loop i 1 3(A=And)` asemele `loop i 1 3 (A=And)`) ning tsükli alamlaused kindlasti olema sulgudega ümbritsetud (`loop i 1 3 A=And` asemele `loop i 1 3 (A=And)` või `loop i 1 3 A(B)` asemele `loop i 1 3 (A(B))`). Muudel juhtudel antakse selle kohta veateade. Keerulisemate avaldiste (nt üle kümne tipu sisaldava *DGFR*'i teisendamise omad) rakendamine teisenduste keeles ei ole soovitatav, kuna keele parser on algeline ning võib esile kerkida erinevaid vigu.

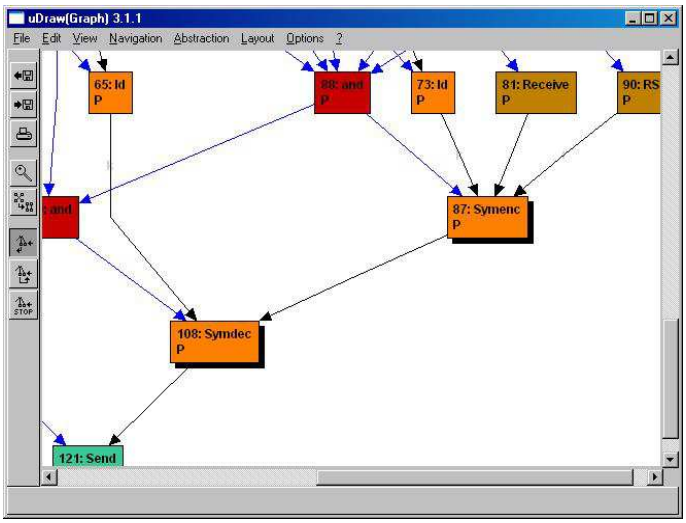
2.1.2. Lihtne näide teisendusest

Et sooritada teisendust $dec(k, enc(r, k, x)) \to x$ sümmeetrilise võtme abil, tuleb genereerida vastav graaf (joonis 2.2), millelt need tipud valida. Sisend- ja väljundtipud on alati täpselt ühe serva abil ühendatud. Antud näite juures oleks teisenduste keelne kood:

```
Deklaratsioonid
{
   ENC=Symenc;     #krüptimine
   DEC=Symdec;     #lahtikrüptimine
   CE=InputB;      #kontrollsõltuvus
   CD=InputB;      #kontrollsõltuvus
   R=InputS;       #juhumündid
```

```
   KE=InputS;      #võti
   KD=InputS;      #võti
   X=InputS;       #sõnum
   OD=OutputS;     #bitijada väljund
}
Vasak DGFR-muutuja
{
   ENC(CE;R;KE;X); #servad CE -> ENC, R -> ENC, KE -> ENC ja X -> ENC
   DEC(CD;ENC;KD); #servad CD -> DEC, ENC -> DEC ja KD -> DEC
   OD(DEC);        #serv   DEC -> OD
}
Parem DGFR-muutuja
{
   OD(X);          #serv X -> OD
}
```

Peale teisenduse võimalikkuse tuleks tähelepanu pöörata, et defineeritud kaks *DGFR*'i oleks arvutuslikult eristamatud. Teisendamise tulemusena tekib pilt joonisel 2.3, kus pärast teisendust liigub serv otse *Receive*-tipust *Send*-tippu.



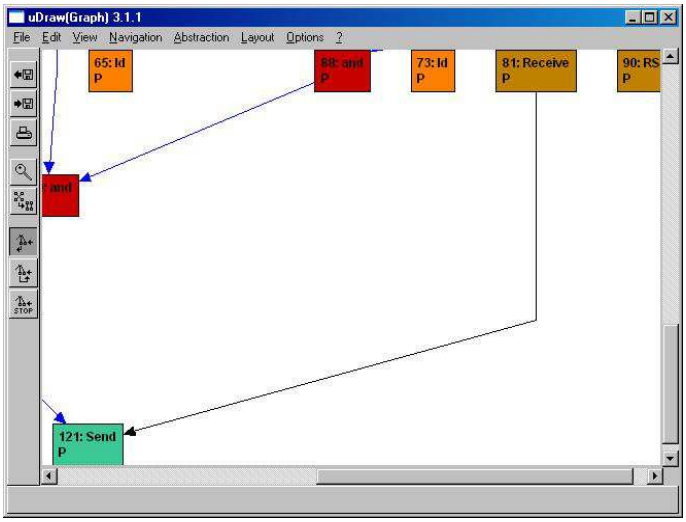
Joonis 2.2. Sümmeetriline krüpteerimine

2.1.3. Massiivide kasutamise näide

Teine lühike näide on massiividega transformatsioon

$$and_1(x_1,\ldots,x_m), and_2(x_1,\ldots,x_m),\ldots,and_n(x_1,\ldots,x_m) \rightarrow \\ and(x_1,\ldots,x_m), and(x_1,\ldots,x_m),\ldots,and(x_1,\ldots,x_m)$$

mida pannakse kirja koodina



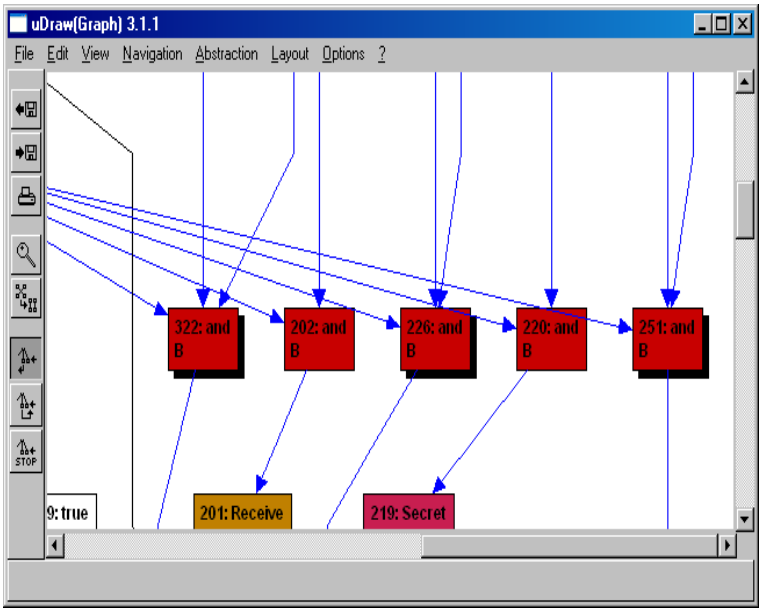
Joonis 2.3. *DGR* 2.2 ilma sümmeetrilise krüpteerimiseta

```
Deklaratsioonid: {
   n=3;m=3;
   loop i 1 n (
      A[i]=And;
      loop j 1 m ( #alamtsükkel
        B[i,j]=InputB
      );
      OA[i]=OutputB
   );
}
Vasak DGFR: {
   loop i 1 n (
      A[i](loop j 1 m (B[i,j]));
      OA[i](A[i])
   )
}
Parem DGFR: {
A[1](loop j 1 m (B[1,j]));
loop i 1 n (OA[i](A[1]));
}
```

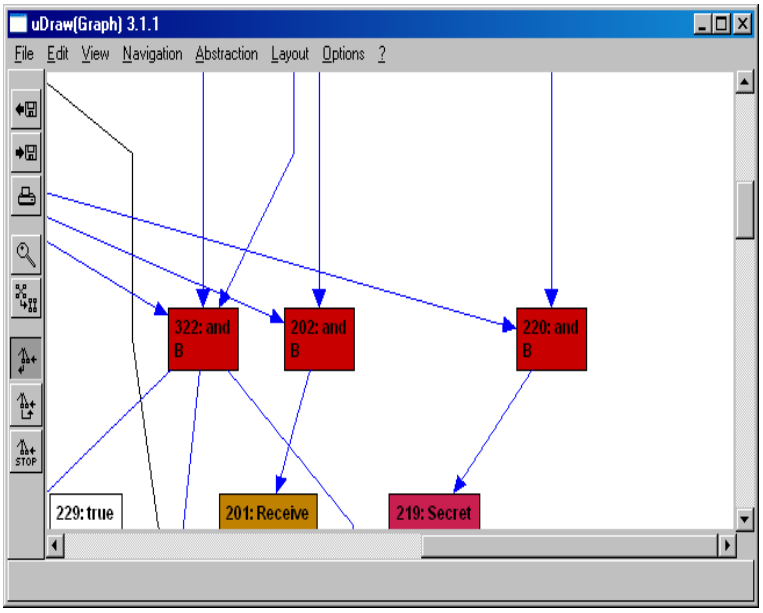
ja saab rakendada joonisel 2.4 näidatud graafi peal. Teisendatud graaf on joonisel 2.5.

2.1.4. Näiteid tippude ja servade kirjutamisest

Selles alamjaotises on kirjeldatud massiivide deklareerimist ja tsüklite kirjutamist koodifaili. Deklareerimise mõningad näited:



Joonis 2.4. Samade sisenditega *And*-tipud



Joonis 2.5. Üks *And*-tipp

- `loop i 1 n (V[i]=Op)` – Tähistab deklaratsioone $V_1 = Op, \ldots, V_n = Op$ eeldusel, et arvumuutuja `n` omab naturaalarvulist väärtust.
- `loop i 1 n (V[i]=Constant i)` – Tähistab deklaratsioone $\{V_i = Constant(i) | 1 \leqslant i \leqslant n\}$ eeldusel, et arvumuutuja n omab naturaalarvulist väärtust.
- `loop i 1 n (loop j 1 m (V[i,j]))` ja `loop i 1 n (loop j 1 n (loop k 1 o (W[i,j,k])))` – tähistavad vastavalt kahe- ja kolmemõõtmelist massiivi:

$$\begin{matrix} V_{11}, V_{12}, \ldots, V_{1m} \\ V_{21}, V_{22}, \ldots, V_{2m} \\ \ldots \\ V_{n1}, V_{n2}, \ldots, V_{nm} \end{matrix} \quad \text{ja} \quad \begin{matrix} W_{111}, W_{112}, \ldots, W_{11o} \\ W_{121}, W_{122}, \ldots, W_{12o} \\ \ldots \\ W_{1m1}, W_{1m2}, \ldots, W_{1mo} \end{matrix} \quad \ldots \quad \begin{matrix} W_{n11}, W_{n12}, \ldots, W_{n1o} \\ W_{n21}, W_{n22}, \ldots, W_{n2o} \\ \ldots \\ W_{nm1}, W_{nm2}, \ldots, W_{nmo} \end{matrix}$$

Mõned näited servade kujutamise kohta *DGFR*'is:

- `A(B)` on lihtsalt üksik serv tipust `B` tippu `A`.
- `A(loop i 1 n (B[i]))` on servade massiiv $B_1 \to A, \ldots B_n \to A$.
- `loop i 1 n (A[i](B[i]))` on servamuutujad $B_1 \to A_1, \ldots B_n \to A_n$.
- `A(loop i 1 n (B[i]);loop j 1 m (C[j]))` on servamuutujad $B_1 \to A, \ldots B_n \to A, C_1 \to A, \ldots C_m \to A$.
- `loop i 1 n (A(B[i];C[i]))` on servamuutujad $B_1 \to A, C_1 \to A, \ldots, B_n \to A, C_n \to A$.

2.2. Teisenduste keele formaalne grammatika

Ka see lihtne teisenduste keel omab formaalset grammatikat. Keel on defineeritud kui nelik $(\mathcal{T}, \mathcal{V}, \texttt{Transform}, \mathcal{P})$ [14], kus $\mathcal{T}$ on terminalide tähestik, $\mathcal{V}$ on mitteterminalide tähestik, $\texttt{Transform} \in \mathcal{V}$ algsümbol ja $\mathcal{P}$ on produktsioonide hulk:

1. `Nat` ::= `( '1' |...| '9' ) ( '0' |...| '9' )`*
2. `Int` ::= `Nat | '0'`
3. `NVar` ::= `( 'a' |...| 'z' ) ( 'a' |...| 'z' | '0' |...| '9' )`*
4. `Var` ::= `( 'A' |...| 'Z' ) ( 'A' |...| 'Z' | '0' |...| '9' )`*
5.
```
Expr ::= Int
       | NElem
       | '(' Expr ')'
       | '-' Expr
       | Expr ( '+' | '-' | '*' | '/' | '%' ) Expr
```

6. NElem ::= NVar ('[' Idxs? ']')?

7. Elem ::= Var ('[' Idxs ? ']')?

8. Idxs ::= Expr | Expr ',' Idxs

9. Node ::= And | Constant Expr | DimEq | DimNeq
| End | Error | False | Id
| Implic | InputB | InputS | IsEq
| IsNeq | IsOK | Keypair | LongAnd
| LongMux | LongNand | LongOr | Merge
| Nonce | Or | OutputB | OutputS
| Pubdec | Proj Expr Expr | Pubenc | PubencZ
| Pubkey | RS | Secret | ShortMux
| ShortNand | SigVer | Signature | SignedMsg
| Symdec | Symenc | SymencZ | Symkey
| TTT | TestSig | True | Tuple Expr
| Verkey

10. Decls ::= NVar ('[' Idxs? ']')? ('=' Expr)?
| Var ('[' Idxs? ']')? '=' (Node | Elem)
| ('loop' NVar Expr Expr)? '(' Decls ')'
| Decls ';' Decls
| ε

11. Sources ::= Elem
| Sources ';' Sources
| ('loop' NVar Expr Expr)? '(' Sources ')'
| ε

12. Edges ::= Elem '(' Sources ')'
| ('loop' NVar Expr Expr)? '(' Edges ')'
| Edges ';' Edges

13. Transform ::= '{' Decls '}' '{' Edges '}' '{' Edges '}'

Mitteterminalide hulga $\mathcal{V}$ moodustavad produktsiooni alguses (enne märki "::=") olevad fraasid Nat, Int, NVar, Var, Expr, NElem, Elem, Idxs, Node, Decls, Sources, Edges ja Transform.

Need, mis pole mitteterminalid, moodustavad terminaalsete sümbolite hulga $\mathcal{T}$: '0',...,'9', 'a',...,'z', 'A',...,'Z', '(', ')', '[', ']', '{', '}', '=', '+', '-', '*', '/', '%' ja ε ning operatsioonid And,..., Verkey.

Grammatika kirjeldusel on kasutatud *Backus-Naur*'i kuju. Produktsiooni defineerimiseks kasutatakse tähistust ::= (⟨*mitteterminal*⟩ ::= *avaldised*). püstkriips | on "välistav või", mis annab valida (mitte)terminalist täpselt ühe. Tärn '*' tähendab kordade arvu $n \in \{0, 1, 2, \ldots\}$, (nt 'a'* on ε |'a'| 'a' 'a'| 'a' 'a' 'a'|...) ja küsimärk '?' näitab nulli või ühte korda. Nt Var ('[' Idxs? ']')? võib kirjutada Var | Var '[' Idxs? ']' ja Var | Var '[' ']' | Var '[' Idxs ']'. Sulud (...) on grupeerimiseks.

2.3. Keele semantika

Vaatleme lähemalt, kuidas iga produktioon tegelikult töötab. Produktsioonide jaoks on oma semantikat defineeriv funktsioon. $\mathbf{AST}_{\mathtt{p}}$ tähendab mitteterminalile $\mathtt{p} \in \mathcal{V}$ vastavate abstraktsete süntaksipuude hulka. Iga mitteterminali jaoks on olemas (ja järgnevas kirjeldatud) tema denotatsioonide hulk ning semantiline funktsioon süntaksipuudest sellesse hulka. Implementatsioonis on semantika realiseeritud failis *parser.ml*.

2.3.1. Arvud

Naturaalarvude hulgaks loeme $\mathbb{N} = \{1, 2, \ldots\}$ ja massiivi indeksiteks sobivad vaid naturaalarvulised väärtused. Selle hulga formaalne grammatika on produktsioon `Nat` ::= (`'1'` |...| `'9'`) (`'0'` |...| `'9'`)*. Mitteterminali `Nat` semantikat defineeriv funktsioon on $\mathcal{NAT} : \mathbf{AST}_{\mathtt{Nat}} \to \mathbb{N}$ ehk $\mathcal{NAT}[\![\mathtt{Nat}]\!] \in \mathbb{N}$:

$$\begin{aligned}
\mathcal{NAT}[\![\mathtt{Nat\ '0'}]\!] &= 10 * \mathcal{NAT}[\![\mathtt{Nat}]\!] + 0,\\
&\ldots,\\
\mathcal{NAT}[\![\mathtt{Nat\ '9'}]\!] &= 10 * \mathcal{NAT}[\![\mathtt{Nat}]\!] + 9,\\
\mathcal{NAT}[\![\mathtt{'1'}]\!] &= 1,\\
&\ldots,\\
\mathcal{NAT}[\![\mathtt{'9'}]\!] &= 9.
\end{aligned}$$

Lisaks selle hulgale on ka positiivsete täisarvude hulk $\mathbb{Z}^+$, mille formaalne grammatika on `Int` ::= `Nat` | `'0'` ja `Int` semantikat defineeriv funktsioon on $\mathcal{INT} : \mathbf{AST}_{\mathbf{Int}} \to \mathbb{Z}^+$:

$$\mathcal{INT}[\![\mathtt{'0'}]\!] = 0,\ \mathcal{INT}[\![\mathtt{Nat}]\!] = \mathcal{NAT}[\![\mathtt{Nat}]\!].$$

Implementatsioonis $\mathcal{INT}$ ja $\mathcal{NAT}$ on funktsioonid vastavalt `z` ja `nat`.

2.3.2. Muutujanimed

Keeles on kahte tüüpi muutujaid – arvumuutujad, mille väärtuste hulk on täisarvud $\mathbb{Z}$ ja tipumuutujad, mille väärtused kuuluvad hulka **Node** (2.2). Arvu- ja tipumuutujate nimede kujude eristamine (esimesel väikesed- ja teisel suured tähed) on mõeldud kirjutatava koodi loetavuse huvides ja nende omavahel segiajamise vältimiseks.

Arvumuutuja nimed koosnevad väiketähtedest ja numbritest, kusjuures esimene sümbol on väiketäht. Seda hulka tähistame $\mathbf{NVar} = \{(b_1, b_2, \ldots, b_n) | b_1 \in \{\mathtt{'a'}, \ldots, \mathtt{'z'}\}, b_2, \ldots, b_n \in \{\mathtt{'a'}, \ldots, \mathtt{'z'}, \mathtt{'0'}, \ldots, \mathtt{'9'}\}, n \in \mathbb{N}\}$. Arvumuutuja nime formaalne grammatika on produktsioon `NVar` ::= (`'a'` |...| `'z'`) (`'a'` |...| `'z'` | `'0'` |...| `'9'`)*. Mitteterminali `NVar` semantikat defineeriv funktsioon on $\mathcal{NVAR} : \mathbf{AST}_{\mathtt{NVar}} \to \mathbf{NVar}$ ehk $\mathcal{NVAR}[\![\mathtt{NVar}]\!] \in \mathbf{NVar}$:

$$
\begin{aligned}
\mathcal{NVAR}[\![\texttt{NVar 'a'}]\!] &= \mathcal{NVAR}[\![\texttt{NVar}]\!]\texttt{++"a"},\\
&\ldots,\\
\mathcal{NVAR}[\![\texttt{NVar 'z'}]\!] &= \mathcal{NVAR}[\![\texttt{NVar}]\!]\texttt{++"z"},\\
\mathcal{NVAR}[\![\texttt{NVar '0'}]\!] &= \mathcal{NVAR}[\![\texttt{NVar}]\!]\texttt{++"0"},\\
&\ldots,\\
\mathcal{NVAR}[\![\texttt{NVar '9'}]\!] &= \mathcal{NVAR}[\![\texttt{NVar}]\!]\texttt{++"9"},\\
\mathcal{NVAR}[\![\texttt{'a'}]\!] &= \texttt{"a"},\\
&\ldots,\\
\mathcal{NVAR}[\![\texttt{'z'}]\!] &= \texttt{"z"}.
\end{aligned}
$$

Märk ++ tähendab sõnede konkateneerimist ja $\texttt{NVar} \in \mathbf{AST}_{\texttt{NVar}}$.

Tipumuutuja nimed koosnevad suurtähtedest ja numbritest, kusjuures esimene sümbol on suurtäht. Seda hulka tähistame $\mathbf{Var} = \{(b_1, b_2, \ldots, b_n) | b_1 \in \{\texttt{'A'}, \ldots, \texttt{'Z'}\}, b_2, \ldots, b_n \in \{\texttt{'A'}, \ldots, \texttt{'Z'}, \texttt{'0'}, \ldots, \texttt{'9'}, n \in \mathbb{N}\}\}$. Tipumuutuja nime mitteterminal on `Var` ja vastav formaalne grammatika on `Var ::= ( 'A' |...| 'Z' ) ( 'A' |...| 'Z' | '0' |...| '9' )`*. Mitteterminali semantikat defineeriv funktsioon on $\mathcal{VAR} : \mathbf{AST}_{\texttt{Nar}} \rightarrow \mathbf{Var}$ ehk $\mathcal{VAR}[\![\texttt{Var}]\!] \in \mathbf{Var}$:

$$
\begin{aligned}
\mathcal{VAR}[\![\texttt{Var 'A'}]\!] &= \mathcal{VAR}[\![\texttt{Var}]\!]\texttt{++"A"},\\
&\ldots,\\
\mathcal{VAR}[\![\texttt{Var 'Z'}]\!] &= \mathcal{VAR}[\![\texttt{Var}]\!]\texttt{++"Z"},\\
\mathcal{VAR}[\![\texttt{Var '0'}]\!] &= \mathcal{VAR}[\![\texttt{Var}]\!]\texttt{++"0"},\\
&\ldots,\\
\mathcal{VAR}[\![\texttt{Var '9'}]\!] &= \mathcal{VAR}[\![\texttt{Var}]\!]\texttt{++"9"},\\
\mathcal{VAR}[\![\texttt{'A'}]\!] &= \texttt{"A"},\\
&\ldots,\\
\mathcal{VAR}[\![\texttt{'Z'}]\!] &= \texttt{"Z"}.
\end{aligned}
$$

Implementatsioonis $\mathcal{NVAR}$ ja $\mathcal{VAR}$ on vastavalt funktsioonid `nvar` ja `var`. Selline teisendus on dokumenteeritud, kuna $\mathcal{NVAR}$ ja $\mathcal{VAR}$ argumentideks on sõned ning väljunditeks vastavalt hulkade **NVar** ja **Var** elemendid. Sisuliselt on need eritüüpi objektid, kuigi näevad samasugused välja.

2.3.3. Olekud

Nii arvu kui ka tipumuutujatel on olekud ehk väärtused, mida muutujad omavad antud ajahetkel. Arvumuutujate väärtused defineerib funktsioon $ns \in \mathbf{NState}$, kus $\mathbf{NState} = \mathbf{NVar} \times \mathbb{N}^* \hookrightarrow \mathbb{Z}$ ning tipumuutujate oma $s \in \mathbf{State}$, kus $\mathbf{State} = \mathbf{Var} \times \mathbb{N}^* \hookrightarrow \mathbf{Node}$ (2.2). $\mathbb{N}^*$ on naturaalarvude listide hulk $\mathbb{N}^* = \mathbb{N}^0 \bigcup \mathbb{N}^1 \bigcup \mathbb{N}^2 \bigcup \ldots$, kus $\mathbb{N}^0 = \{[]\}$ on tühilistist koosnev hulk, $\mathbb{N}^1 = \{[n_1] | n_1 \in \mathbb{N}\}$, $\mathbb{N}^2 = \{[n_1, n_2] | n_1, n_2 \in \mathbb{N}\}$, jne. $\hookrightarrow$ tähistab osalist funktsiooni. Osaline funktsioon hulgast X hulka Y on sama, mis täielik funktsioon hulgast X hulka $Y_\perp$ ($Y_\perp = Y \bigcup \{\perp\}$). Hulk $\mathbb{N}^*$ on tarvis kuitahes mõõtmeliste tipu- ja arvu muutujate massiivide jaoks. Massiivid aitavad koodi kirjutamist lühendada, et mitmekümne tipu korraga valimisel ei pea iga tipu, kus on üks ja sama operatsioon, jaoks muutujat deklareerima.

2.3.4. Arvavaldised

Arvavaldis võib olla täisarv, arvumuutuja, unaarne või binaarne operatsioon või avaldis sulgude vahel. Avaldiste denotatsioonide hulk on **Expr** = **NState** $\hookrightarrow \mathbb{Z}$ = **NState** $\rightarrow \mathbb{Z}_\perp$ st et kui avaldise väärtust üritatakse leida mingis punktis, kus ta on määramata, siis $\mathcal{EXPR}[\![\texttt{Expr}]\!]ns = \perp$ ($\texttt{Expr} \in \mathbf{AST}_{\texttt{Expr}}$). Märk '$\perp$' tähendab määramata väärtust. Implementatsioonis tähendab see tavalist veateadet käesoleva töö käigus lisatud programmeerimiskeeles. Avaldiste formaalne grammatika on

```
Expr ::= Int
       | NElem
       | '(' Expr ')'
       | '-' Expr
       | Expr ( '+' | '-' | '*' | '/' | '%' ) Expr
```

ning mitteterminali `Expr` semantikat defineeriv funktsioon on $\mathcal{EXPR} : \mathbf{AST}_{\texttt{Expr}} \rightarrow \mathbf{Expr}$ ehk $\mathcal{EXPR}[\![\texttt{Expr}]\!]ns \in \mathbf{Expr}$. Täisarvu avaldise äratundmiseks kasutatakse funktsiooni

$$\mathcal{EXPR}[\![\texttt{Int}]\!]ns = \mathcal{INT}[\![\texttt{Int}]\!].$$

Arvumassiivi elemendi lugemiseks kasutatakse funktsiooni, mille definitsioon on (2.1):

$$\mathcal{EXPR}[\![\texttt{NElem}]\!]ns = \mathcal{NELEM}[\![\texttt{NElem}]\!]ns.$$

Suluavaldise lugemiseks kasutatakse funktsiooni rekursiivselt sulgude vahel oleva avaldise peal:

$$\mathcal{EXPR}[\![\texttt{'(' Expr ')'}]\!]ns = \mathcal{EXPR}[\![\texttt{Expr}]\!]ns.$$

Vastandarvu võtmiseks võetakse avaldise semantikat defineeriv funktsiooni väärtuse vastandarv. Veateate korral tagastatakse see:

$$\mathcal{EXPR}[\![\texttt{'-' Expr}]\!]ns = \begin{cases} \perp, \mathcal{EXPR}[\![\texttt{Expr}]\!]ns = \perp \\ -\mathcal{EXPR}[\![\texttt{Expr}]\!]ns, \mathcal{EXPR}[\![\texttt{Expr}]\!]ns \neq \perp \end{cases}.$$

Liitmise, lahutamise ja korrutamise korral võetakse nende poolte semantiliste funktsioonide väärtused. Mitteveateate ilmumisel tagastatakse operatsiooni tulemus:

$$\mathcal{EXPR}[\![\texttt{E1 '+' E2}]\!]ns = \begin{cases} \perp, \text{kui } e_1 = \perp \vee e_2 = \perp \\ e_1 + e_2, \text{kui } e_1 \neq \perp \wedge e_2 \neq \perp \end{cases},$$

$$\mathcal{EXPR}[\![\texttt{E1 '-' E2}]\!]ns = \begin{cases} \perp, \text{kui } e_1 = \perp \vee e_2 = \perp \\ e_1 - e_2, \text{kui } e_1 \neq \perp \wedge e_2 \neq \perp \end{cases} \text{ ja}$$

$$\mathcal{EXPR}[\![\texttt{E1 '*' E2}]\!]ns = \begin{cases} \perp, \text{kui } e_1 = \perp \vee e_2 = \perp \\ e_1 * e_2, \text{kui } e_1 \neq \perp \wedge e_2 \neq \perp \end{cases},$$

kus $e_1 = \mathcal{EXPR}[\![\texttt{E1}]\!]ns$, $e_2 = \mathcal{EXPR}[\![\texttt{E2}]\!]ns$ ja $\texttt{E1}, \texttt{E2} \in \mathbf{AST}_{\texttt{Expr}}$.

Jagatise ja jäägi leidmisel kontrollitakse parempoolse avaldise erinevust nulliga (nulliga ei saa jagada):

$$\mathcal{EXPR}[\![\texttt{E1 '/' E2}]\!]ns = \begin{cases} \bot, \text{ kui } e_1 = \bot \vee e_2 = \bot \vee e_2 = 0 \\ floor(e_1/e_2), \text{ kui } e_1 \neq \bot \wedge e_2 \neq \bot \wedge e_2 \neq 0 \end{cases} \quad \text{ja}$$

$$\mathcal{EXPR}[\![\texttt{E1 '\%' E2}]\!]ns = \begin{cases} \bot, \text{ kui } e_1 = \bot \vee e_2 = \bot \vee e_2 = 0 \\ floor(e_1/e_2), \text{ kui } e_1 \neq \bot \wedge e_2 \neq \bot \wedge e_2 \neq 0 \end{cases},$$

kus $e_1 = \mathcal{EXPR}[\![\texttt{E1}]\!]ns$, $e_2 = \mathcal{EXPR}[\![\texttt{E2}]\!]ns$ ja $\texttt{E1}, \texttt{E2} \in \mathbf{AST}_{\texttt{Expr}}$. Funktsioon $floor$ väljastab etteantud arvu täisosa ning `%` on jagatise jäägi leidmine. Implementatsioonis on avaldiste jaoks funktsioon `sem_expr`.

2.3.5. Arvumuutujad

Peale üksikmuutujate on arvu- ja tipumassiivid, mille elementide indeksite jaoks on produktsioon `Idxs ::= Expr | Expr ',' Idxs`. mitteterminali `Idxs` semantikat defineeriv funktsioon on $\mathcal{IDXS} : \mathbf{AST}_{\texttt{Idxs}} \to \mathbf{Idxs}$, kus $\mathbf{Idxs} = \mathbf{NState} \hookrightarrow \mathbb{N}^* \setminus \mathbb{N}^0$. $\mathcal{IDXS}[\![\texttt{Idxs}]\!]ns \in \mathbf{Idxs}$ ($ns \in \mathbf{NState}$) on defineeritud kui

$$\mathcal{IDXS}[\![\texttt{Expr}]\!]ns = \begin{cases} \bot, \mathcal{EXPR}[\![\texttt{Expr}]\!]ns = \bot \vee \mathcal{EXPR}[\![\texttt{Expr}]\!]ns < 1 \\ [\mathcal{EXPR}[\![\texttt{Expr}]\!]ns], \\ \quad \mathcal{EXPR}[\![\texttt{Expr}]\!]ns \neq \bot \wedge \mathcal{EXPR}[\![\texttt{Expr}]\!]ns \geqslant 1 \end{cases} \quad \text{ja}$$

$$\mathcal{IDXS}[\![\texttt{Expr ',' Idxs}]\!]ns = \begin{cases} \bot, \mathcal{IDXS}[\![\texttt{Expr}]\!]ns = \bot \vee \mathcal{IDXS}[\![\texttt{Idxs}]\!]ns = \bot \\ \mathcal{IDXS}[\![\texttt{Expr}]\!]ns@\mathcal{IDXS}[\![\texttt{Idxs}]\!]ns, \\ \quad \mathcal{IDXS}[\![\texttt{Expr}]\!]ns \neq \bot \wedge \mathcal{IDXS}[\![\texttt{Idxs}]\!]ns \neq \bot \end{cases},$$

kus @ on listi konkatenatsioon. Indeksitena sobivad vaid naturaalarvud. Implementatsioonis toimib sellena funktsioon `idxs`.

Arvumuutujate massiivi elementide jaoks on produktsioon `NElem ::= NVar ( '[' Idxs? ']' )?` ja mitteterminali `NElem` semantikat defineeriv funktsioon

$$\mathcal{NELEM} : \mathbf{AST}_{\texttt{NElem}} \to (\mathbf{NState} \hookrightarrow \mathbb{Z}). \tag{2.1}$$

Üksikmuutujale omistatud väärtuse kättesaamiseks on

$$\mathcal{NELEM}[\![\texttt{NVar}]\!]ns = ns(nv, []),$$

kus $nv = \mathcal{NVAR}[\![\texttt{NVar}]\!]$ ja $\texttt{NVar} \in \mathbf{AST}_{\mathbf{NVar}}$.

Ühe- või mitmemõõtmelise massiivi elemendi väärtuse kättesaamiseks:

$$\mathcal{NELEM}[\![\texttt{NVar '[' Idxs ']'}]\!]ns = \begin{cases} \bot, ns(nv, []) = \bot \vee I = \bot \\ ns(nv, []), ns(nv, []) \neq \bot \wedge I \neq \bot \end{cases},$$

kus $nv = \mathcal{NVAR}[\![\texttt{NVar}]\!]$, $\texttt{NVar} \in \mathbf{AST}_{\mathbf{NVar}}$ ja $I = \mathcal{IDXS}[\![\texttt{Idxs}]\!]ns$. Implementatsioonis on funktsioon `sem_nae`.

Funktsioon $ns(\cdot, \cdot)$ annab lihtsalt nv_I väärtuse (kui see on olemas):

$$ns(nv, I) = \begin{cases} x, \text{ kui } nv_I \text{ on deklareeritud ja väärtus on } x \in \mathbb{Z} \\ \bot, \text{ muidu} \end{cases}, \text{ kus } nv \in \mathbf{NVar} \text{ ja } I \in \mathbb{N}^*.$$

2.3.6. Tipumuutujad

Ka tippudel on üksikmuutujad ja massiivi elemendid. Nendeks kasutame massiivi elementide mitteterminali `Elem` ja vastavat produktsiooni `Elem ::= Var ( '[' Idxs ? ']' )?`. Selle mitteterminali semantikat defineeriv funktsioon on $\mathcal{ELEM} : \mathbf{AST}_{\texttt{Elem}} \to \mathbf{Elem}$ ehk $\mathcal{ELEM}[\![\texttt{Elem}]\!](ns, s) \in \mathbf{Node}$ ehk $\mathcal{ELEM}[\![\texttt{Elem}]\!] \in \mathbf{Elem}$, kus $\mathbf{Elem} = \mathbf{NState} \times \mathbf{State} \hookrightarrow \mathbf{Node}$ ($ns \in \mathbf{NState}$ ja $s \in \mathbf{State}$):

$$\begin{aligned} \mathcal{ELEM}[\![\texttt{Var}]\!](ns, s) &= s(v, []), \\ \mathcal{ELEM}[\![\texttt{Var '[' ']'}]\!](ns, s) &= s(v, []), \end{aligned}$$

kus $v = \mathcal{VAR}[\![\texttt{Var}]\!]$ ja $\texttt{Var} \in \mathbf{AST}_{\mathbf{NVar}}$.

Massiivi elemendi korral kontrollitakse ka indeksite sobivust:

$$\mathcal{ELEM}[\![\texttt{Var '[' Idxs ']'}]\!](ns, s) = \begin{cases} \bot, s(v, I) = \bot \vee I = \bot \\ s(v, I), s(v, I) \neq \bot \wedge I \neq \bot \end{cases},$$

kus $v = \mathcal{VAR}[\![\texttt{Var}]\!]$, $\texttt{Var} \in \mathbf{AST}_{\mathbf{Var}}$ ja $I = \mathcal{IDXS}[\![\texttt{Idxs}]\!]ns$. Implementatsioonis on selleks `sem_ae`.

Funktsioon $s(\cdot, \cdot)$ annab lihtsalt v_I väärtuse (kui see on olemas):

$$s(v, I) = \begin{cases} t, \text{ kui } v_I \text{ on deklareeritud ja väärtus on } t \in \mathbf{Node} \\ \bot, \text{ muidu} \end{cases},$$

kus $v \in \mathbf{Var}$ ja $I \in \mathbb{N}^*$.

2.3.7. Tipud

Implementatsioonis on tipud kujutatud sõnedena. Nendeks on operatsioonid (nt *PubencZ*, *Pubdec*, jt) ja mõnedel on ka naturaalarvulised parameetrid (nt *Proj 4 5*), mida eraldatakse tühikutega ja mis on lisatud operatsioonile.

Tippude formaalne grammatika on

```
Node ::= And        | Constant Expr  | DimEq      | DimNeq
       | End        | Error          | False      | Id
       | Implic     | InputB         | InputS     | IsEq
       | IsNeq      | IsOK           | Keypair    | LongAnd
       | LongMux    | LongNand       | LongOr     | Merge
       | Nonce      | Or             | OutputB    | OutputS
       | Pubdec     | Proj Expr Expr | Pubenc     | PubencZ
       | Pubkey     | RS             | Secret     | ShortMux
       | ShortNand  | SigVer         | Signature  | SignedMsg
       | Symdec     | Symenc         | SymencZ    | Symkey
       | TTT        | TestSig        | True       | Tuple Expr
       | Verkey
```

ja mitteterminali `Node` semantikat defineerib funktsioon $\mathcal{NODE} : \mathbf{AST}_{\texttt{Node}} \to \mathbf{NState} \hookrightarrow$

Node, kus

$$\textbf{Node} = \{\underline{\texttt{And}}, \underline{\texttt{Constant 1}}, \underline{\texttt{Constant 2}}, \underline{\texttt{Constant 3}}, \ldots, \underline{\texttt{DimEq}}, \underline{\texttt{DimNeq}}, \underline{\texttt{Error}}, \underline{\texttt{False}}, \underline{\texttt{Id}}, \underline{\texttt{Implic}}, \underline{\texttt{InputB}}, \underline{\texttt{InputS}}, \underline{\texttt{IsEq}}, \underline{\texttt{IsNeq}}, \underline{\texttt{IsOK}}, \underline{\texttt{Keypair}}, \underline{\texttt{LongAnd}}, \underline{\texttt{LongMux}}, \underline{\texttt{LongNand}}, \underline{\texttt{LongOr}}, \underline{\texttt{Merge}}, \underline{\texttt{Nonce}}, \underline{\texttt{Or}}, \underline{\texttt{OutputB}}, \underline{\texttt{OutputS}}, \underline{\texttt{Proj 1 1}}, \underline{\texttt{Proj 1 2}}, \underline{\texttt{Proj 2 2}}, \underline{\texttt{Proj 1 3}}, \underline{\texttt{Proj 2 3}}, \underline{\texttt{Proj 3 3}}, \ldots, \underline{\texttt{Pubdec}}, \underline{\texttt{Pubenc}}, \underline{\texttt{PubencZ}}, \underline{\texttt{Pubkey}}, \underline{\texttt{RS}}, \underline{\texttt{Secret}}, \underline{\texttt{ShortMux}}, \underline{\texttt{ShortNand}}, \underline{\texttt{SigVer}}, \underline{\texttt{Signature}}, \underline{\texttt{SignedMsg}}, \underline{\texttt{Symdec}}, \underline{\texttt{Symenc}}, \underline{\texttt{SymencZ}}, \underline{\texttt{Symkey}}, \underline{\texttt{TTT}}, \underline{\texttt{TestSig}}, \underline{\texttt{True}}, \underline{\texttt{Tuple 1}}, \underline{\texttt{Tuple 2}}, \ldots, \underline{\texttt{Verkey}}.\} \tag{2.2}$$

Proj-tipu parameetritest esimene on alati mitte suurem teisest parameetrist ning *Constant*- ja *Tuple*-tipu parameetrid on naturaalarvud. Semantikafunktsiooni definitsiooniks on

$$\mathcal{NODE}[\![\texttt{'And'}]\!]ns = \underline{\texttt{And}};$$

$$\mathcal{NODE}[\![\texttt{'Constant' Expr}]\!]ns = \begin{cases} \bot, \text{ kui } e = \bot \vee e < 1 \\ \underline{\texttt{Constant E}}, \text{ kui } e \neq \bot \wedge e \geqslant 1 \end{cases},$$

kus $e = \mathcal{EXPR}[\![\texttt{Expr}]\!]ns$ ja $E = toStr(e)$ (avaldis sõnekujul);

$$\mathcal{NODE}[\![\texttt{'DimEq'}]\!]ns = \underline{\texttt{DimEq}};$$
$$\mathcal{NODE}[\![\texttt{'DimNeq'}]\!]ns = \underline{\texttt{DimNeq}};$$
$$\mathcal{NODE}[\![\texttt{'Error'}]\!]ns = \underline{\texttt{Error}};$$
$$\mathcal{NODE}[\![\texttt{'False'}]\!]ns = \underline{\texttt{False}};$$
$$\mathcal{NODE}[\![\texttt{'Id'}]\!]ns = \underline{\texttt{Id}};$$
$$\mathcal{NODE}[\![\texttt{'Implic'}]\!]ns = \underline{\texttt{Implic}};$$
$$\mathcal{NODE}[\![\texttt{'InputB'}]\!]ns = \underline{\texttt{InputB}};$$
$$\mathcal{NODE}[\![\texttt{'InputS'}]\!]ns = \underline{\texttt{InputS}};$$
$$\mathcal{NODE}[\![\texttt{'IsEq'}]\!]ns = \underline{\texttt{IsEq}};$$
$$\mathcal{NODE}[\![\texttt{'IsNeq'}]\!]ns = \underline{\texttt{IsNeq}};$$
$$\mathcal{NODE}[\![\texttt{'IsOK'}]\!]ns = \underline{\texttt{IsOK}};$$
$$\mathcal{NODE}[\![\texttt{'Keypair'}]\!]ns = \underline{\texttt{Keypair}};$$
$$\mathcal{NODE}[\![\texttt{'LongAnd'}]\!]ns = \underline{\texttt{LongAnd}};$$
$$\mathcal{NODE}[\![\texttt{'LongMux'}]\!]ns = \underline{\texttt{LongMux}};$$
$$\mathcal{NODE}[\![\texttt{'LongNand'}]\!]ns = \underline{\texttt{LongNand}};$$
$$\mathcal{NODE}[\![\texttt{'LongOr'}]\!]ns = \underline{\texttt{LongOr}};$$
$$\mathcal{NODE}[\![\texttt{'Merge'}]\!]ns = \underline{\texttt{Merge}};$$
$$\mathcal{NODE}[\![\texttt{'Nonce'}]\!]ns = \underline{\texttt{Nonce}};$$
$$\mathcal{NODE}[\![\texttt{'Or'}]\!]ns = \underline{\texttt{Or}};$$
$$\mathcal{NODE}[\![\texttt{'OutputB'}]\!]ns = \underline{\texttt{OutputB}};$$
$$\mathcal{NODE}[\![\texttt{'OutputS'}]\!]ns = \underline{\texttt{OutputS}};$$

$$\mathcal{NODE}[\![\texttt{'Proj' E1 E2}]\!]ns = \begin{cases} \bot, \text{ kui } e_1 = \bot \vee e_1 < 1 \vee e_2 = \bot \vee e_2 < 1 \vee e_1 > e_2 \\ \underline{\texttt{Proj } E_1\ E_2}, \text{ muudel juhtudel} \end{cases},$$

kus $e_1 = \mathcal{EXPR}[\![\texttt{E1}]\!]ns$, $E_1 = toStr(e_1)$, $e_2 = \mathcal{EXPR}[\![\texttt{E2}]\!]ns$,
$E_2 = toStr(e_2)$ ja $\texttt{E1}, \texttt{E2} \in \mathbf{AST}_{\texttt{Expr}}$;

$$\mathcal{NODE}[\![\texttt{'Pubdec'}]\!]ns = \underline{\texttt{Pubdec}};$$
$$\mathcal{NODE}[\![\texttt{'Pubenc'}]\!]ns = \underline{\texttt{Pubenc}};$$

$$\mathcal{NODE}[\![\texttt{'PubencZ'}]\!]ns = \underline{\texttt{PubencZ}};$$
$$\mathcal{NODE}[\![\texttt{'Pubkey'}]\!]ns = \underline{\texttt{Pubkey}};$$
$$\mathcal{NODE}[\![\texttt{'RS'}]\!]ns = \underline{\texttt{RS}};$$
$$\mathcal{NODE}[\![\texttt{'Secret'}]\!]ns = \underline{\texttt{Secret}};$$
$$\mathcal{NODE}[\![\texttt{'ShortMux'}]\!]ns = \underline{\texttt{ShortMux}};$$
$$\mathcal{NODE}[\![\texttt{'ShortNand'}]\!]ns = \underline{\texttt{ShortNand}};$$
$$\mathcal{NODE}[\![\texttt{'SigVer'}]\!]ns = \underline{\texttt{SigVer}};$$
$$\mathcal{NODE}[\![\texttt{'Signature'}]\!]ns = \underline{\texttt{Signature}};$$
$$\mathcal{NODE}[\![\texttt{'SignedMsg'}]\!]ns = \underline{\texttt{SignedMsg}};$$
$$\mathcal{NODE}[\![\texttt{'Symdec'}]\!]ns = \underline{\texttt{Symdec}};$$
$$\mathcal{NODE}[\![\texttt{'Symenc'}]\!]ns = \underline{\texttt{Symenc}};$$
$$\mathcal{NODE}[\![\texttt{'SymencZ'}]\!]ns = \underline{\texttt{SymencZ}};$$
$$\mathcal{NODE}[\![\texttt{'Symkey'}]\!]ns = \underline{\texttt{Symkey}};$$
$$\mathcal{NODE}[\![\texttt{'TTT'}]\!]ns = \underline{\texttt{TTT}};$$
$$\mathcal{NODE}[\![\texttt{'TestSig'}]\!]ns = \underline{\texttt{TestSig}};$$
$$\mathcal{NODE}[\![\texttt{'True'}]\!]ns = \underline{\texttt{True}};$$
$$\mathcal{NODE}[\![\texttt{'Tuple' Expr}]\!]ns = \begin{cases} \bot, \text{ kui } e = \bot \vee e < 1 \\ \underline{\texttt{Tuple E}}, \text{ kui } e \neq \bot \wedge e \geqslant 1 \end{cases},$$
kus $e = \mathcal{EXPR}[\![\texttt{Expr}]\!]ns$ ja $E = toStr(e)$ (avaldis sõnekujul) ning
$$\mathcal{NODE}[\![\texttt{'Verkey'}]\!]ns = \underline{\texttt{Verkey}}.$$

Selle ülesannet implementatsioonis täidab funktsioon `sem_node`.

2.3.8. Deklareerimine

Enne teisendust ja selles avaldiste ja muutujate kasutamist on vaja teada, mis väärtusi muutujad omavad. Selleks on muutujad tarvis deklareerida. Deklareerimisel kasutame produktsiooni

```
Decls ::= NVar ( '[' Idxs? ']')? ( '=' Expr )?
        | Var ( '[' Idxs? ']')? '=' ( Node | Elem )
        | ( 'loop' NVar Expr Expr)? '(' Decls ')'
        | Decls ';' Decls
        | ε.
```

Mitteterminali `Decls` semantikat defineeriv funktsioon on

$$\mathcal{DECLS} : \mathbf{AST}_{\texttt{Decls}} \to (\mathbf{NState} \times \mathbf{State}) \hookrightarrow (\mathcal{P}((\mathbf{Var} \times \mathbb{N}^*) \times \mathbf{Node}) \uplus \mathcal{P}((\mathbf{NVar} \times \mathbb{N}^*) \times \mathbb{Z}) \uplus \mathcal{P}(\mathbf{NVar} \times \mathbb{N}^*)) \text{ st}$$

$$\mathcal{DECLS}[\![\texttt{Decls}]\!] \in \mathcal{P}((\mathbf{Var} \times \mathbb{N}^*) \times \mathbf{Node}) \uplus \mathcal{P}((\mathbf{NVar} \times \mathbb{N}^*) \times \mathbb{Z}) \uplus \mathcal{P}(\mathbf{NVar} \times \mathbb{N}^*),$$

kus $\uplus$ on lõikumata ühend ja $\mathcal{P}(X)$ on hulga X kõigi alamhulkade hulk. Komponent $\mathcal{P}((\mathbf{Var} \times \mathbb{N}^*) \times \mathbf{Node})$ on tipu muutujate deklareerimiseks, kus **Var**, $\mathbb{N}^*$ ja **Node** on vastavalt tipumuutuja tähis, indeksid ja tipud (2.2). Komponent $\mathcal{P}((\mathbf{NVar} \times \mathbb{N}^*) \times \mathbb{Z})$ on arvumuutuja

deklareerimiseks käsitsi väärtustamise korral, kus **NVar**, $\mathbb{N}$ ja $\mathbb{Z}$ on vastavalt arvu muutuja tähis, indeksid ja arvuline väärtus. Komponent $\mathcal{P}(\mathbf{NVar} \times \mathbb{N}^*)$ on arvumuutuja deklareerimine automaatse väärtustamise korral. $\texttt{Decls} \in \mathbf{AST}_{\texttt{Decls}}$ semantika defineerib seega, milliste massiivide millised elemendid olemas on ja mis on nende väärtused. Implementatsioonis seda semantikat defineerivat funktsiooni täidab `sem_decl`.

Arvmuutuja automaatsel väärtustamise korral: $nv = \mathcal{NVAR}[\![\texttt{NVar}]\!]$ ja e on uuele muutujale automaatselt leitud väärtus:

$$\begin{aligned}\mathcal{DECLS}[\![\texttt{NVar '[' ']'}]\!](ns, s) &= \begin{cases}\bot, \text{ kui avaldisele } e \text{ ei leita väärtust}\\ \{((nv, []), e)\}, \text{ muidu}\end{cases} \quad \text{ja}\\ \mathcal{DECLS}[\![\texttt{NVar}]\!](ns, s) &= \mathcal{DECLS}[\![\texttt{NVar '[' ']'}]\!](ns, s).\end{aligned}$$

Deklareerimisel üksiku arvumuutuja väärtustamine käsitsi, kus $nv = \mathcal{NVAR}[\![\texttt{NVar}]\!]$ ja $e = \mathcal{EXPR}[\![\texttt{Expr}]\!]ns$:

$$\mathcal{DECLS}[\![\texttt{NVar '[' ']' '=' Expr}]\!](ns, s) = \begin{cases}\bot, \text{ kui } e = \bot\\ \{((nv, []), e)\}, \text{ kui } e \neq \bot\end{cases} \quad \text{ja}$$

$$\mathcal{DECLS}[\![\texttt{NVar '=' Expr}]\!](ns, s) = \mathcal{DECLS}[\![\texttt{NVar '[' ']' '=' Expr}]\!](ns, s)\ .$$

Mingi konkreetse massiivi elemendi väärtustamine käsitsi, kus $nv = \mathcal{NVAR}[\![\texttt{NVar}]\!]$, $I = \mathcal{IDX}[\![\texttt{Idxs}]\!]ns$ ja $e = \mathcal{EXPR}[\![\texttt{Expr}]\!]ns$:

$$\mathcal{DECLS}[\![\texttt{NVar '[' Idxs ']' '=' Expr}]\!](ns, s) = \begin{cases}\bot, \text{ kui } I = \bot \vee e = \bot\\ \{((nv, I), e)\}, \text{ kui } I \neq \bot \wedge e \neq \bot\end{cases}\ .$$

Ükiku tipumuutuja deklareerimisel väärtustatakse käsitsi, kus $v = \mathcal{VAR}[\![\texttt{Var}]\!]$ ja $n = \mathcal{NODE}[\![\texttt{Node}]\!]ns$:

$$\mathcal{DECLS}[\![\texttt{Var '[' ']' '=' Node}]\!](ns, s) = \begin{cases}\bot, \text{ kui } n = \bot\\ \{((v, []), n)\}, \text{ kui } n \neq \bot\end{cases} \quad \text{ja}$$

$$\mathcal{DECLS}[\![\texttt{Var '=' Node}]\!](ns, s) = \mathcal{DECLS}[\![\texttt{Var '[' ']' '=' Node}]\!](ns, s).$$

Tipumassiivi elemendi korral, kus $v = \mathcal{VAR}[\![\texttt{Var}]\!]$, $I = \mathcal{IDXS}[\![\texttt{Idxs}]\!]ns$ ja $n = \mathcal{NODE}[\![\texttt{Node}]\!]ns$:

$$\mathcal{DECLS}[\![\texttt{Var '[' Idxs ']' '=' Node}]\!](ns, s) = \begin{cases}\bot, \text{ kui } I = \bot \vee n = \bot\\ \{((v, I), n)\}, \text{ kui } I \neq \bot \wedge n \neq \bot\end{cases}\ .$$

Sulgudes asuvatel definitsioonidel deklareeritakse sulgudes olev:

$$\mathcal{DECLS}[\![\texttt{'(' Decls ')'}]\!](ns, s) = \mathcal{DECLS}[\![\texttt{Decls}]\!](ns, s).$$

Semikooloniga eraldatud definitsioonidel deklareeritakse pooled eraldi. Algul deklareeritakse vasak pool, siis võivad täieneda olekud ns ja s uute muutujatega. d_1 =

$\mathcal{DECLS}[\![\texttt{D1}]\!](ns, s)$, $d_2 = \mathcal{DECLS}[\![\texttt{D2}]\!](ns', s')$, $ns, ns' \in$ **NState**, $s, s' \in$ **State** ning $\texttt{D1}, \texttt{D2} \in \mathbf{AST}_{\texttt{Decls}}$:

$$\mathcal{DECLS}[\![\texttt{D1 ';' D2}]\!](ns, s) = \begin{cases} \bot, \text{ kui } d_1 = \bot \vee d_2 = \bot \\ (d_1 \bigcup d_2), \text{ kui } d_1 \neq \bot \wedge d_2 \neq \bot \end{cases},$$

kus $ns' = nupdate(ns, d_1) \in$ **NState** ja $s' = vupdate(s, d_1) \in$ **State**. Funktsioonid $nupdate(\cdot, \cdot)$ ja $vupdate(\cdot, \cdot)$ defineeritakse vastavalt:

$$nupdate(ns, d)(nv, I) = \begin{cases} e, ((nv, I), e) \in d \\ ns(nv, I), \nexists e : ((nv, I), e) \in d \end{cases} \quad \text{ja}$$

$$vupdate(s, d)(v, I) = \begin{cases} n, ((v, I), n) \in d \\ s(v, I), \nexists n : ((v, I), n) \in d \end{cases}.$$

Tsüklis olevates definitsioonides deklareeritakse igal sammul eraldi, ka siin muutuvad olekud nii uute muutujate tekkimise kui ka tsükli indeksi *nv* muutumise teel, kus igal tsüklisammul on täienenud olekud $s_{e_1}, s_{e_1+1}, \ldots, s_{e_2} \in$ **State** ja $ns_{e_1}, ns_{e_1+1}, \ldots, ns_{e_2} \in$ **NState**, kus $e_1 = \mathcal{EXPR}[\![\texttt{E1}]\!]ns$, $e_2 = \mathcal{EXPR}[\![\texttt{E2}]\!]ns$, $nv = \mathcal{NVAR}[\![\texttt{NVar}]\!]$. Olekud defineeritakse

$$\begin{aligned} d_{e_1} &= \mathcal{DECLS}[\![\texttt{Decls}]\!](ns[v \mapsto e_1], s) \\ d_{e_1+1} &= \mathcal{DECLS}[\![\texttt{Decls}]\!](ns_{e_1}[v \mapsto e_1 + 1], s_{e_1}) \\ d_{e_1+2} &= \mathcal{DECLS}[\![\texttt{Decls}]\!](ns_{e_1+1}[v \mapsto e_1 + 2], s_{e_1+1}) \ , \\ &\ldots \ , \\ d_{e_2} &= \mathcal{DECLS}[\![\texttt{Decls}]\!](ns_{e_2-1}[v \mapsto e_2], s_{e_2-1}) \end{aligned}$$

kus $ns_{e_1+i} = nupdate(ns_{e_1+i-1}, d_{e_1+i})$.

$$\mathcal{DECLS}[\![\texttt{'loop' NVar E1 E2 '(' Decls ')'}]\!](ns, s)$$
$$\begin{cases} \bot, \text{ kui } e_1 = \bot \vee e_2 = \bot \\ \begin{cases} \bot, \text{ kui } \bot \in \{d_{e_1}, \ldots, d_{e_2}\} \\ (d_{e_1} \bigcup \ldots \bigcup d_{e_2}), \text{ kui } \bot \notin \{d_{e_1}, \ldots, d_{e_2}\} \end{cases} \end{cases},$$

Tühja sõne korral deklareeritakse kui ühikfunktsioon: $\mathcal{DECLS}[\![\varepsilon]\!](ns, s) = ()$.

2.3.9. Arvumuutujate automaatne väärtustamine

Automaatne väärtustamine toimub mõnedel erijuhtudel ning ainult üksikmuutujate peal. Kui tegelikus fragmendis esineb *Proj*-tippe, siis see saadakse teise parameetri kaudu vasakus *DGFR*'is. Arvumuutujate väärtustamine toimub ka *Tuple*- ja *Constant*-tipu parameetri kaudu juhul, kui valitud tippude seas eksisteerib ühesugune parameeter.

2.3.10. Algtipud

Algtippude jaoks on produktsioon

```
Sources ::= Elem
          | Sources ';' Sources
          | ('loop' NVar Expr Expr)? '(' Sources ')'
          | ε
```

ja mitteterminali `Sources` semantikat defineeriv funktsioon on $\mathcal{SOURCES}$: $\mathbf{AST}_{\texttt{Sources}} \to$ **Sources** ehk $\mathcal{SOURCES}[\![\texttt{Sources}]\!] \in$ **Sources** = (**NState** $\times$ **State**) $\hookrightarrow$ (**Var** $\times \mathbb{N}^*)^*$. Vastavaks funktsiooniks implementatsioonis on `sem_sources`.

Serva konkreetne algtipu skaneerimine, kus $e = \mathcal{ELEM}[\![\texttt{Elem}]\!](ns, s)$:

$$\mathcal{SOURCES}[\![\texttt{Elem}]\!](ns, s) = \begin{cases} \bot, \text{ kui } e = \bot \\ [e], \text{ kui } e \neq \bot \end{cases}.$$

Kõrvuti asetsevate servade algtippude korral töödeldakse need eraldi. $S_1 = \mathcal{SOURCES}[\![\texttt{S1}]\!](ns, s)$, $S_2 = \mathcal{SOURCES}[\![\texttt{S2}]\!](ns, s)$, $\texttt{S1}, \texttt{S2} \in \mathbf{AST}_{\texttt{Sources}}$ ja '@' on listide ühendamise märk:

$$\mathcal{SOURCES}[\![\texttt{S1 ';' S2}]\!](ns, s) = \begin{cases} \bot, \text{ kui } S_1 = \bot \vee S_2 = \bot \\ S_1 @ S_2, \text{ kui } S_1 \neq \bot \wedge S_2 \neq \bot \end{cases}.$$

Sulgude vahel olevate algtippude korral kasutatakse sama funktsiooni rekursiivselt:

$$\mathcal{SOURCES}[\![\texttt{'(' Sources ')'}]\!](ns, s) = \mathcal{SOURCES}[\![\texttt{Sources}]\!](ns, s).$$

Tsüklis kasutatakse sama funktsiooni rekursiivselt igal tsüklisammul eraldi:

$$\mathcal{SOURCES}[\![\texttt{'loop' NVar E1 E2 '(' Sources ')'}]\!](ns, s) =$$
$$\begin{cases} \bot, \text{ kui } e_1 = \bot \vee e_2 = \bot \\ \begin{cases} \bot, \text{ kui } \bot \in \{S_{e_1}, \ldots, S_{e_2}\} \\ S_{e_1} @ \ldots @ S_{e_2}, \text{ kui } \bot \notin \{S_{e_1}, \ldots, S_{e_2}\} \end{cases} & \text{, kui } e_1 \neq \bot \wedge e_2 \neq \bot \end{cases} \text{, kus}$$
$$\forall i \in \{e_1, \ldots, e_2\} \text{ korral } S_i = \mathcal{SOURCES}[\![\texttt{Sources}]\!](ns[nv \mapsto i], s)$$

Tühja sõne ε korral tagastatakse tühi list: $\mathcal{SOURCES}[\![\varepsilon]\!](ns, s) = []$.

2.3.11. Servad

Sõltuvusgraafi fragment koosneb teatud hulk servadest, millel on alg- ja lõpptipud. Siin kirjeldame servi lõpptippude abil, millel on vahetud eellased. Sõltuvuste formaalse grammatikana kasutame produktsiooni

```
Edges ::= Elem '(' Sources ')'
        | ('loop' NVar Expr Expr)? '(' Edges ')'
        | Edges ';' Edges
        | ε.
```

Mitteterminali `Edges` semantikat defineeriv funktsioon on $\mathcal{EDGES}$: $\mathbf{AST}_{\texttt{Edges}} \rightarrow \mathbf{NState} \times \mathbf{State} \hookrightarrow \mathbf{Edges}$, kus $\mathbf{Edges} = \mathcal{P}((\mathbf{Var} \times \mathbb{N}^*)^* \times \mathbf{Var} \times \mathbb{N}^*)$. Hulga $(\mathbf{Var} \times \mathbb{N}^*)^* \times \mathbf{Var} \times \mathbb{N}^*$ elemendi vasak pool $(\mathbf{Var} \times \mathbb{N}^*)^*$ kirjeldab eellaste hulka ja parem pool $\mathbf{Var} \times \mathbb{N}^*$ tippu millel on need eellased. Implementatsioonis on selleks funktsiooniks `sem_edges`.

Üksiku lõpptipu korral saame kätte selle vahetud eellased, mis tulevad funktsioonilt $\mathcal{SOURCES}$:

$$\mathcal{EDGES}[\![\texttt{Elem '(' Sources ')'}]\!](ns, s) = \begin{cases} \bot, \text{ kui } e = \bot \vee S = \bot \\ \{(S, e)\}, \text{kui } e \neq \bot \vee S \neq \bot \end{cases},$$

kus $e = \mathcal{ELEM}[\![\texttt{Elem}]\!](ns, s)$ ja $S = [e_1; \ldots; e_n] = \mathcal{SOURCES}[\![\texttt{Sources}]\!](ns, s)$.

Sulgude vahel olevate algtippude korral kasutame funktsiooni rekursiivselt ilma sulgudeta:

$$\mathcal{EDGES}[\![\texttt{'(' Edges ')'}]\!](ns, s) = \mathcal{EDGES}[\![\texttt{Edges}]\!](ns, s).$$

Semikooloniga eraldamise korral rakendame funktsiooni rekursiivselt eraldi kummagi poole peal. $E_1 = \mathcal{EDGES}[\![\texttt{E1}]\!]ns$, $E_2 = \mathcal{EDGES}[\![\texttt{E2}]\!]ns$ ja $\texttt{E1}, \texttt{E2} \in \mathbf{AST}_{\texttt{Edges}}$:

$$\mathcal{EDGES}[\![\texttt{E1 ';' E2}]\!](ns, s) = \begin{cases} \bot, \text{ kui } E_1 = \bot \vee E_2 = \bot \\ E_1 \bigcup E_2, \text{ kui } E_1 \neq \bot \wedge E_2 \neq \bot \end{cases}.$$

Tsüskli korral rakendame funktsiooni rekursiivselt igal tsükli sammul eraldi. $\forall i \in \{e_1, \ldots, e_2\} : E_i = \mathcal{EDGES}[\![\texttt{'(' Sources')'}]\!](ns[nv \mapsto i], s)$:

$$\mathcal{EDGES}[\![\texttt{'loop' NVar E1 E2 '(' Sources ')'}]\!](ns, s) =$$
$$\begin{cases} \bot, \text{ kui } nv = \bot \vee e_1 = \bot \vee e_2 = \bot \\ \begin{cases} \bot, \text{ kui } \bot \in \{E_{e_1}, \ldots, E_{e_2}\} \\ E_{e_1} \bigcup \ldots \bigcup E_{e_2}, \text{ kui } \bot \notin \{E_{e_1}, \ldots, E_{e_2}\} \end{cases}, \text{ kui } nv \neq \bot \wedge e_1 \neq \bot \wedge e_2 \neq \bot \end{cases}.$$

Tühja sõne korral tekib tühi servade hulk: $\mathcal{EDGES}[\![\varepsilon]\!](ns, s) = []$.

2.3.12. Transformatsioon

Viimasena selles jaotises on teisenduse semantika kirjeldus. Semantika formaalne grammatika on produktsioon `Transform ::= '{' Decls '}' '{' Edges '}' '{' Edges '}'`. Mitteterminali `Transform` semantikat defineeriv funktsioon on $\mathcal{TRANS}$: $\mathbf{AST}_{\texttt{Trans}} \rightarrow \mathbf{Transform}$ ehk $\mathcal{TRANS}[\![\texttt{Trans}]\!] \in \mathbf{Transform}$, kus $\mathbf{Transform} = \mathbf{Edges}^2$, mis implementatsioonis on `parseTransformation`. Olekud $ns \in \mathbf{NState}$ ja $s \in \mathbf{State}$ on semantika defineerimise alguses tühjad st kõikjal määramata ning pärast deklareerimist on teised olekud $ns' = nupdate(ns, D) \in \mathbf{NState}$ ja $s' = vupdate(s, D) \in \mathbf{State}$, kus kõik vajalikud muutujad on deklareeritud. $D = \mathcal{DECLS}[\![\texttt{D}]\!](ns, s)$, $L = \mathcal{EDGES}[\![\texttt{L}]\!](ns', s')$ ja $R = \mathcal{EDGES}[\![\texttt{R}]\!](ns', s')$ ning $\texttt{D} \in \mathbf{AST}_{\texttt{Decls}}$ ja $\texttt{L}, \texttt{R} \in \mathbf{AST}_{\texttt{Edges}}$:

$$\mathcal{TRANS}[\![\texttt{'{' D '}' '{' L '}' '{' R '}'}]\!](ns, s) =$$
$$\begin{cases} \bot, \text{ kui } D = \bot \\ \begin{cases} \bot, \text{ kui } L = \bot \vee R = \bot \\ (L, R), \text{ kui } L \neq \bot \wedge R \neq \bot \end{cases}, \text{ kui } \neq \bot \end{cases}.$$

2.3.13. Portide määramine servadele

Parseri juures servamuutujate genereerimisel lisatakse märge pordi $p \in$ **IPorts** (1.4) kohta igale mitte väljundtippu suunduvale servale. See näitab, kuhu porti lõpptipus serv suundub. Teisenduse koodi kirjutamisel peab jälgima, et tipu sisenditest koolonite arv seal vahel oleks õige ehk ühe võrra väiksem sisendite arvust, mis suuremal osal tippudest on konstantne. Porte määrab rakendus pärast koodi parsimist üksikuteks servadeks: näiteks

```
   loop i a b (A[i](B;loop j 1 2 (C[j])))
→  A[a](B;loop j 1 2 (C[j]));...;A[b](B;loop j 1 2 (C[j]))     ,
→  A[a](B;C[1];C[2]);A[a+1](B;C[1];C[2]);...;A[b](B;C[1];C[2])
```

leiab muutujatega $A_a, \ldots, A_b$ seotud tipud ja lõpuks õige sisendite arvu korral paneb funktsiooniga `setPorts` servadele $B \to A_a, C_1 \to A_a, C_2 \to A_a, \ldots, B \to A_b, C_1 \to A_b, C_2 \to A_b$ sobivad pordid.

3. peatükk

Lahenduskäik

Selles peatükis kirjeldatakse töös koostatud ülesande lahenduse käiku. *Krüptoanalüsaator*il on igale teisendusmoodulile eraldi pandud laiendusfunktsioon `expand`, mis oma teisenduste jaoks kasutatavaid abifunktsioone välja kutsub. Teisenduste keelt kasutav laiendusfunktsioon ja muud keelega seotud tegevused asuvad moodulifailis *GrbTrFromCode.ml*. Laiendusfunktsioon selles moodulis loeb koodifailist vajaliku info, parsib sealt peatükis 2 kirjeldatud funktsioonide abiga mõlemad *DGFR*-muutujad (mille servadele paneb ka pordid) ja sobitab vasaku *DGFR*-muutuja reaalse *DGFR*'iga ning lisab paremas *DGFR*'is asuvatele uutele tippude dimensioonid ja servade kujutused. Seejärel edastab õigesti valitud tippude ning korrektse koodi korral vajalikud servad raamistikule, mille järel toimub *DGFR*-teisendus. Teisendused märgitakse *XML*-kujul ka logifaili, mille nimi määratakse failis *GrbTrReadModule.ml* asuva muutuja `logFile` juures. Uued failid, mis teisenduste keele programmeerimise käigus on lisatud, on *GrbTrFromCode.ml* teisenduskeele liidestamiseks raamistikuga, *commonFunkctions.ml* erinevate pisifunktsioonide jaoks, *readCode.ml* koodifailist lugemiseks, *parser.ml* teisenduste keelse koodi parsimiseks ja servade genereerimiseks *variableAdjustment.ml* servamuutujate seostamiseks reaalsete servadega, *dimsAndMaps.ml* dimensioonide ja kujutuste seadmiseks, *codeParserInteract.ml* kolme eelmise faili kasutamiseks, ja moodulifail *GrbTrReadModule.ml* liidestamise faili *GrbTrFromCode.ml* väljakutsumiseks. Lisaks on täiendatud ka faile *GrbInteract.ml*, *GrbInput.ml* ja *GrbGraphs.ml*.

3.1. Teisenduste keele liidestamine

Funktsioon `expand` moodulifailis *GrbTrFromCode.ml*, juhib kogu keele abil teisenduse protsessi. Et see töötaks, peab vähemalt üks tipp valitud olema.

Mooduli *GrbTrFromCode.ml* kasutamine mooduli *GrbTrReadModule.ml* kaudu on selle jaoks, et vältida iga teisenduse järel koodifaili sisu redigeerimist. On võimalik kirjutada mooduleid nt *GrbTrReadModule1.ml*, *GrbTrReadModule2.ml*, jne ning nendes asuvate muutujate kolmikud (`codeFile`,`menuItem`,`logFile`) initsialiseerida sõnedekolmikuid vastavalt (`code1`,`item1`,`log1`), (`code2`,`item2`,`log2`), jne.

Funktsioonis `tgr` valmistatakse tippude (3.1) ja servade (3.2) hulgad teisenduste keele jaoks. Sellega loetakse ka teisenduste keelse kood etteantud failist. Iga tipu *n* andmed on

alguses

$$(\ell_1(n), \lambda_1(n), V(n), E(n), J(n), OT(n), nr_i(n), nr_o(n), d(n)), \tag{3.1}$$

kus $\ell_1(n)$ on tipu n identifikaator ja $\lambda_1(n)$ tipul kasutatav operatsioon. $V(n) \in \{\texttt{true}, \texttt{false}\}$ tähendab, kas konkreetne tipp n on *uDraw(Graph)*-aknas valitud. $E(n) \in \{\texttt{true}, \texttt{false}\}$ ja $J(n) \in \{\texttt{true}, \texttt{false}\}$ näitavad, kas on vastavalt valimata eellane või järglane. Komponendid $nr_i(n)$ ja $nr_o(n)$ näitavad vastavalt tippu n sissetulevate ja sellest väljaminevate servade arvu. d on dimensioonid (1.1). Graafi iga serv e on

$$(\ell_1(n_s), \ell_1(n_d), \ell(e), m(e), p(e)), \tag{3.2}$$

kus komponendid $\ell_1(n_s)$ ja $\ell_1(n_d)$ on vastavalt serva alg- ja lõpptipu identifikaatorid, $\ell(e)$ serva identifikaator, $m(e)$ on kujutused (1.3) ning $p(e) \in$ **IPorts** (1.4) port, kuhu serv e lõpptipus suubub.

Tipud (3.1), servad (3.2) ja koodi kolm osa (deklaratsioonid ning vasak ja parem *DGFR*-muutuja) antakse ette funktsioonile `languageUsage` failis *codeParserInteract.ml* mis tagastab muutujatega seostatud servad (3.3) ja tipud (3.4).

Tagastatavad servad on

$$((v(n_s), v(n_d)), (ek(e), \ell(e), m(e), p(e))), \tag{3.3}$$

kus $v(n_s)$ ja $v(n_d)$ on vastavalt alg- ja lõpptipu muutujad $(v(n_s), v(n_d) \in \mathbf{Var} \times \mathbb{N}^*)$, $ek(e)$ on serva liik (3.6), $\ell(e)$ – identifikaator, $(m(e))$ – kujutused (1.3) ning $p(e)$ on port (1.4), millesse serv suubub.

Tipud, mida tagastatakse, on

$$(v(n), (\ell_1(n), \lambda_1(n), nk(n), d(n))), \tag{3.4}$$

kus $v(n) \in \mathbf{Var} \times \mathbb{N}^*$, $\ell_1(n)$ on tipu n identifikaator, $\lambda_1(n)$ operatsioon, $nk(n)$ tipu liik (3.8) ja $d(n)$ dimensioonid (1.1).

Servadest (3.3) ja tippudest (3.4) tehakse funktsiooniga `prepareNodesAndEdges` listid `olevadtipud`, `kaduvadtipud`, `tulevadtipud`, `sisendtipud`, `valjundtipud`, `olevadservad`, `tulevadservad` ja `kaduvadservad`, mille elemendid on *JSON*-objektidele sarnastes kirjetes. Tipukirjetes `tnn` $\in$ **Var** on tipumuutuja nimi, `tindices` $\in \mathbb{N}^*$ – tipumuutuja indeksid, `tliik` – tipu liik, `tid` – tipu identifikaator, `ttyyp` – tipu operatsioon ja `tdim` on dimensioonid (1.1). Servakirjetes `ann` $\in$ **Var** ja `aindices` $\in \mathbb{N}^*$ on vastavalt algtipu muutujanimi ja indeksid, `lnn` $\in$ **Var** ja `lindices` $\in \mathbb{N}^*$ on vastavalt lõpptipu muutujanimi ja indeksid `ep` $\in$ **IPorts** – port, `sliik` – serva liik, `skmap` – kujutused (1.3) ning `eid` on serva identifikaator.

Tulevate tippude ja servade identifikaatorid genereeritakse automaatselt. Liigid listide `olevadtipud`,...,`kaduvadservad` elementidel on alati vastavalt `Olev_tipp`,..., `Kaduv_serv`.

Kui need kolm serva ja viis tipulisti on valmis, siis asutakse nendel olevaid andmeid teisendamiseks kasutama. Teisenduste defineerimise skeem on, et teisendus on antud oma muutujatega, mis väärtustatakse suurest graafist. `paramtype` on nende muutujate tüüp, mis igal teisendusel isemoodi on, ehk seda tüüpi väärtus on muutujanimi. `paramkindtype` on

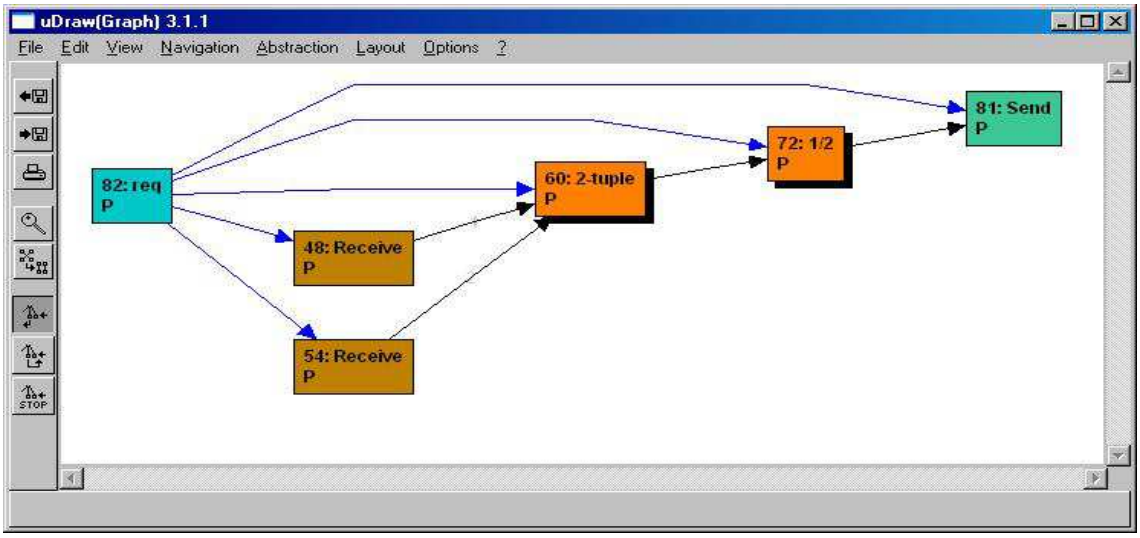
nende muutujate tüüpide tüüp ja `paramvaluetype` on nende muutujate võimalikke väärtuste tüüp (tipp, serv, jne).

Konkreetselt failile *GrbTrFromCode.ml* omased funktsioonid on `dgr` juurde lisatud teisenduste keele rakendamiseks, `prepareNodesAndEdges` ning `string_to_portname`, `string_to_nodename` ja `kind_of_string` vastavalt portide, operatsioonide ja liikide tüübi muutmiseks ning `setInitValues` erinevate väärtute nullimiseks enne teisendamist.

Joonisel 3.1 on näidatud, et valitud on *Proj*-tipp 72 ja *Tuple*-tipp 60. Nende vahetud eellased on *Req*-tipp 82 ning *Receive*-tipud 48 ja 54. Ainsaks vahetuks järglaseks on *Send*-tipp 81. Sarnane pilt saadakse, kui lisada faili *GrbInput.ml* funktsioon

```
let projTpl = PL rep(P,send proj(1,2)(tuple(receive,receive));stop);;
```

ja toimida sama moodi kui on kirjeldatud käesoleva töö peatükis 4. Seejärel tuleb graaf avada ja rakendada teisendust *Do for all ...→Various simplifications*. Et ühendada serv otse



Joonis 3.1. Valitud on tipud 72 (*Proj 1 2*) ja 60 (*Tuple 2*)

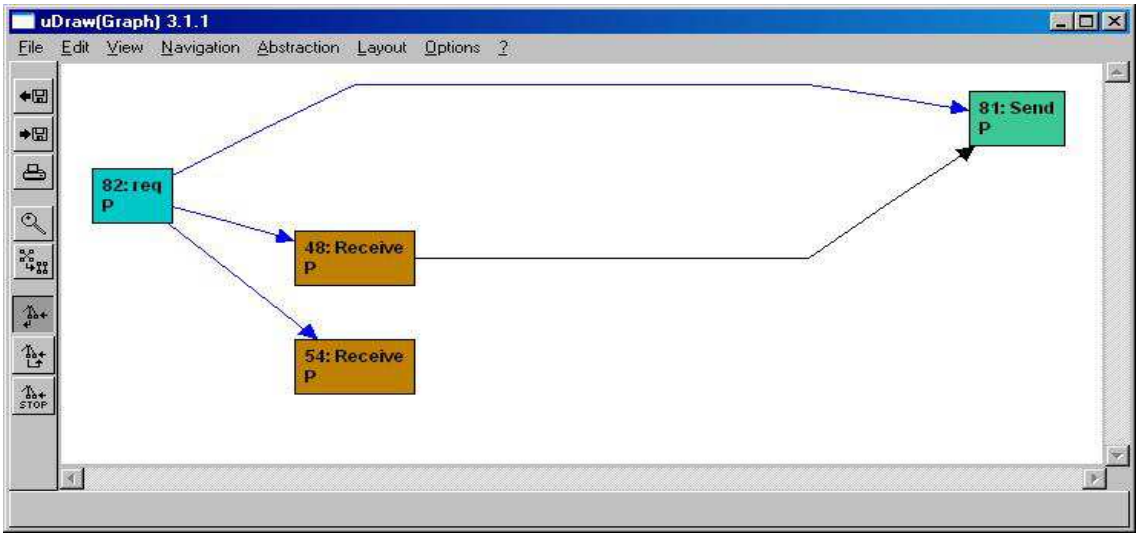
Receive-tipust *Send*-tippu 81, teisendada koodiga

```
{T=Tuple 2;P=Proj 1 2;O=OutputS;
n=2;loop i 1 n (I[i]=InputS;B[i]=InputB);}
{T(B[1];I[1];I[2]);P(B[2];T);O(P);}{O(I[1]);}.
```

Tulemuseks saame joonisel 3.2 näidatud *DGR*'i. Servad, tipud ning vasak ja parem *DGFR*-muutuja koodifailist antakse ette failis *codeParserInteract.ml* asuvale funktsioonile `languageUsage`, mis kasutab vastavaid mooduleid koodist saadud servamuutujate parsimiseks, muutujate sobitamiseks reaalsete andmetega, jt tegevusteks.

3.2. Koodifaili lugemine

Koodifail loetakse failis *ReadCode.ml* asuva funktsiooniga `readCode`, millele antakse ette faili nimi (koos asukoha kaustaga) ning tagastatakse (d, l, r, e), kus $d \in \mathbf{AST}_{\texttt{Decls}}$, $l, r \in$



Joonis 3.2. Tipud (*Proj 1 2*) ja 60 (*Tuple 2*) on eemaldatud

$\mathbf{AST}_{\texttt{Edges}}$ ja *e* on võimalik veateade koodifailist mitte arusaamise kohta.

Teisendustekeelse koodi lugemiseks on tarvis näidata koodifaili nime koos täisteega, mida määratakse *OCaml*-moodulifailis *GrbTrReadModule.ml* muutujaga `codeFile`. Seal tuleb määrata ka omale sobiv menüü kirje `menuItem` ja logifail muutujale `logFile`. See moodulifail kutsub välja mooduli *GrbTrFromCode.ml*.

Koodifaili sisust eraldatakse kolm looksulgude vahel olevat osa – deklaratsioonid, vasak ja parem *DGFR*-muutuja, mis saadetakse parserisse. Parseris töödeldakse info käesoleva töö peatükis 2 kirjeldatud funktsioonidega läbi ja lõpptulemusena tagastatakse servamuutujate (3.5) hulk. Parserisse saadetakse ka reaalsed *DGR*'i servad, et koguda andmeid arvumuutujate automaatseks väärtustamiseks.

3.3. Maatriks

Kui funktsioon `languageUsage` failis *codeParserInteract.ml* on koodi kolm osa, tipud (3.1) ja servad (3.2) kätte saanud, siis alustatakse töötlust. Eraldatakse valitud tipud ning vahetud eellased ja järglased. Nendest sõelutud andmetest tehakse funktsiooniga `makeFragmentTable` *DGFR*-maatriks, mille rea pealkirjadesse pannakse servade algtipud ja veerupealkirjadesse lõpptipud.

3.3.1. Maatriksi genereerimine

Maatriksi genereerimisel vaadatakse servad (3.2) läbi: kui serval vähemalt ühe otstipu identifikaator kuulub *uDraw(Graph)*-aknas valitud tipule, siis serv pannakse vastavasse lahtrisse.

Tabelis 3.1 on joonisel 3.1 näidatud sõltuvusgraafi fragmendi maatriks lihtsal kujul, kus pealkirjades näidatakse tipu identifikaatorit ja operatsiooni ning lahtrites maatriksi elemen-

	(60 : *Tuple* 2)	(72 : *Proj* 1 2)	(81 : *Send*)
(48 : *Receive*)	(*S*, `PortToList(1)`)		
(54 : *Receive*)	(*S*, `PortToList(2)`)		
(60 : *Tuple* 2)		(*S*, `PortFromList(2)`)	
(72 : *Proj* 1 2)			(*S*, `PortText`)
(82 : *Req*)	(*B*, `PortSingleB`)	(*B*, `PortSingleB`)	

Tabel 3.1. Algne *DGFR*-maatriks

tidena servi, milles on andmetüüp (*B*-bitt või *S*-bitijada) ja port.

3.3.2. Maatriksi laiendamine

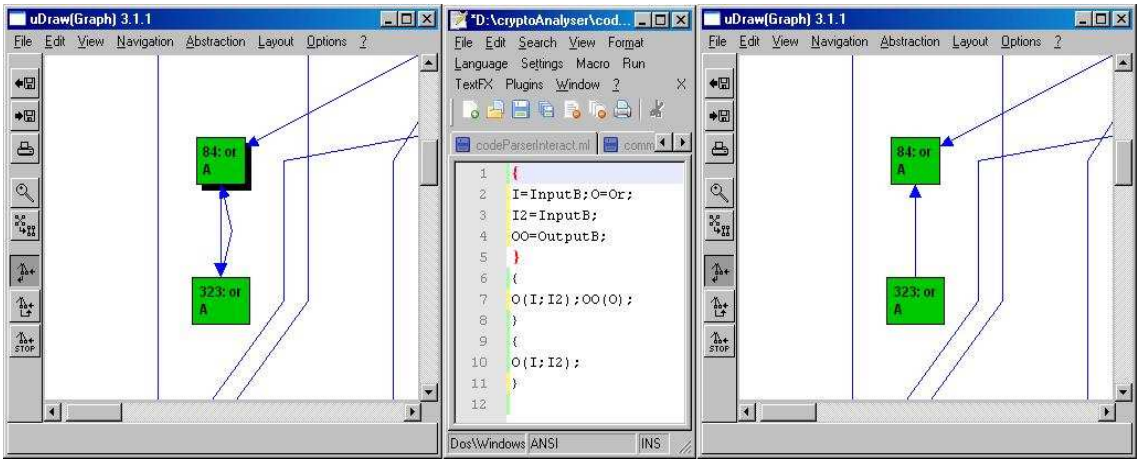
Pärast maatriksi genereerimist see laiendatakse (funktsiooniga `expandMatrix` failis *commonFunkctions.ml*) nii, et igas veerus ja igas reas oleks täpselt üks serv. Laiendatud maatriksil pannakse pealkirjades asuvatele tippudele paralleelselt ka teine operatsioon λ_2. Iga valitud tipu n_v korral võrdub see senise operatsiooniga: $\forall n_c : \lambda_1(n_c) = \lambda_2(n_c)$. Igale sisendtipule n_i pannakse operatsiooniks *InputB* või *InputS* vastavalt sellele, kas tipp on tõeväärtuse või bitijada väljundiga ehk $\forall n_i \lambda_2(n_i) \in \{InputB, InputS\}$. Igale väljundtipule n_o pannakse operatsiooniks *OutputB* või *OutputS* vastavalt sellele, kas sinna suubub tõeväärtuse või bitijada serv ehk $\forall n_o : \lambda_2(n_o) \in \{OutputB, OutputS\}$. Seejärel sorteeritakse laiendatud maatriks sveerud ja read (failis *commonFunkctions.ml* asuva funktsiooniga `quickSort`), et *DGFR*'i väljund- ja sisendtipud läheksid lõppu. Pärast seda sorteerimist pannakse ka identifikaatorile teine juurde (ℓ_2), mis iga valitud tipu n_c korral jääb samaks ($\forall n_c : \ell_1(n_c) = \ell_2(n_c)$) ning sisend- ja väljundtippude jaoks tulevad uued identifikaatorid. Alguses otsitakse originaalsete identifikaatorite seast maksimaalne. Alates maksimaalsest ühe võrra suuremast määratakse sisend- ja väljundtippudele teine identifikaator. Sellega käsitletakse igat sisend- ja väljundtipu esitust vastavalt veeru ja rea pealkirjades omaette tipuna, olgugi et tegelikkuses on mitu sellist tippu üks tipp. Nendel tippudel muudetakse ka väljundite ja sisendite arvud (nr_i ja nr_o) – sisendtippudel on sisend- ja väljundservade arvuks vastavalt 0 ja 1 ning väljundtippudel 1 ja 0. Teine identifikaator pannakse funktsiooniga `setSecondID` failis *codeParserInteract.ml*.

Tabelis 3.2 on näidatud laiendatud maatriks, mis on saadud tabelilt 3.1. Rea ja veeru pealkirjades selles on vastavalt alg- ja lõpptipud näidatud kui ($\ell_1(n), \lambda_1(n), \ell_2(n), \lambda_2(n), nr_i(n), nr_o(n)$) ja servad *e* nende tippude vahel on ($dt(e), p(e)$), kus $dt(e)$ näitab kas on bitijada (*S*) või tõeväärtusserv (*B*). Tabelis on näha, et *Req*-tipp 82 saab sellel teel kaks koopiat, millede teisteks identifikaatoriks saavad 83 ja 84 ning operatsiooniks *InputB*. Ka *Send*-tipp 81 saab selle näite järgi uue identifikaatori (87) ja operatsiooniks *OutputS*.

Võib esineda juhtumeid, kus mitu väljundtippu ja/või sisendtippu vastavad ühele tegelikule tipule reaalses graafis. Seega igale sisend- ja väljundtipu koopiale saab ühendada kuni ühe serva (joonis 3.3).

	(72, *Proj*12, 72, *Proj*12, 2, 1)	(72, *Proj*12, 72, *Proj*12, 2, 1)	(60, *Tuple2*, 60, *Tuple2*, 3, 1)	(60, *Tuple2*, 60, *Tuple2*, 3, 1)	(60, *Tuple2*, 60, *Tuple2*, 3, 1)	(81, *Send*, 87, *OutputS*, 1, 0)
(72, *Proj*12, 72, *Proj*12, 2, 1)						(*S*, PortText)
(60, *Tuple2*, 60, *Tuple2*, 3, 1)	(*S*, PortFromList(2))					
(82, *Req*, 83, *InputB*, 0, 1)			(*B*, PortSingleB)			
(82, *Req*, 84, *InputB*, 0, 1)		(*B*, PortSingleB)				
(54, *Receive*, 85, *InputS*, 0, 1)					(*S*, PortToList(2))	
(48, *Receive*, 86, *InputS*, 0, 1)				(*S*, PortToList(1))		

Tabel 3.2. Laiendatud *DGFR*-maatriks



Joonis 3.3. Mitu sisend- ja väljundmuutujat võivad viidata ühele tipule

3.3.3. Maatriksist saadud servad

Maatriksist (tabel 3.2) võetakse kõik servad e ja konverteeritakse kujule

$$(n_s, (c(e), \ell(e), ek(e), p(e), m(e)), n_d), \tag{3.5}$$

kus n_s ja n_d on serva e alg- ja lõpptipp, $c(e)$ valikute arv (läheb tarvis muutujatega kokkuviimise juures), $\ell(e)$ on serva e identifikaator, $ek(e)$ on serva liik:

$$ek(e) \in \{\texttt{Kaduv\_serv}, \texttt{Olev\_serv}, \texttt{Tulev\_serv}\}, \tag{3.6}$$

$p(e) \in$ **IPorts** – lõpptipu sisendport (1.4), kuhu serv suubub $m(e)$ serva kujutused (1.3). Servade alg- ja lõpptipud on

$$(v(n), \ell_1(n), \ell_2(n), \lambda_1(n), \lambda_2(n), nk(n), d(n), nr_i(n), nr_o(n)), \tag{3.7}$$

kus andmetest $\ell_1(n)$, $\lambda_1(n)$, $d(n)$, $nr_i(n)$ ja $nr_o(n)$ pärinevad valemist (3.1), $v(n) \in (\mathbf{Var}, \mathbb{N}^*)_\perp$ on muutuja tähis (koht muutujale on pandud, kuna servad on vaja saada samale tüübile

kui parserist tulnud servamuutujad), mis määratakse sobitamise juures, $\ell_2(n)$ ja $\lambda_2(n)$ on vastavalt teine identifikaator ja operatsioon ning $nk(n)$ on tipu liik:

$$nk(n) \in \{\texttt{Kaduv\_tipp}, \texttt{Tulev\_tipp}, \texttt{Olev\_tipp}, \texttt{Väljund\_tipp}, \texttt{Sisend\_tipp}\}. \quad (3.8)$$

Väärtused eksisteerivad alguses iga tipu n komponentidel $\ell_1(n)$, $\ell_2(n)$, $\lambda_1(n)$, $\lambda_2(n)$, $nr_i(n)$, $nr_o(n)$ ja $d(n)$ ning serva e osadel $\ell(e)$, $m(e)$ ja $p(e)$. Teistele tipu ja serva komponentidele määratakse need hiljem.

Kaduvad tipud (nk = `Kaduv_tipp`) ja servad (ek = `Kaduv_serv`) leiduvad ainult vasakus *DGFR*'is. Tulevad tipud (nk = `Tulev_tipp`) ja servad (ek = `Tulev_serv`) leiduvad ainult paremas *DGFR*'is. Olevad tipud (nk = `Olev_tipp`) ja servad (ek = `Olev_serv`) eksisteerivad mõlemas *DGFR*'is. Sisend- ja väljundtipud ($nk \in$ {`Sisend_tipp`, `Valjund_tipp`}), mis eksisteerivad vasakus *DGFR*'is võivad eksisteerida paremas *DGFR*'is. Need tippude (3.8) ja servade (3.6) liigid määratakse pärast reaalse *DGFR*'i ja *DGFR*-muutuja servade omavahel sobitamist funktsiooniga `setKinds` failis *codeParserInteract.ml*.

Ka parseri töö tulemusena tekivad tipud ja servad samade tüüpidena, mis (3.7) ja (3.5), kuid väärtused eksisteerivad igal serval e vaid komponendil $p(e)$ ning igal tipul n vaid komponentidel $v(n)$, $\lambda_2(n)$, $nr_i(n)$ ja $nr_o(n)$. Parser genereerib kaks hulka – vasakut ja paremat *DGFR*-muutujat.

Järgmisena alustatakse parseri vasaku *DGFR*'i servade kokkuviimist reaalse *DGR*'i servadega.

3.4. Muutujate sobitamine tippudega

Muutujad väärtustatakse failis *variableAdjustment.ml* vasaku *DGFR*-muutuja servade sobitamise teel reaalse *DGFR*-servadega. Servade sobitamist alustatakse selle võimalikkuse kontrollimisega, kus vaadatakse, kas servade arvud mõlemas hulgas on võrdsed ning kas iga reaalse *DGR*'i serva tüüp on esindatud vasakus *DGFR*-muutujas ning vastupidi. Vajadusel väljastatakse veateade.

Tüübi all on mõeldud otstippude teisi operatsioone (λ_2), sisendite ja väljundite arve otstippudes ning porte (v.a väljundtippude pordid). Seejärel skanneeritakse mõlemad servad, et panna igale servale e võimaluste arv $c(e)$, mis näitab mitu seda tüüpi serva on vaadeldavas servade hulgas.

Tabelist 3.2 saadud servad on

$$\begin{gathered}
(n_{86}, (1, 63, , \texttt{PortToList(1)}, (P, 1) \to 1), n_{60}) \\
(n_{85}, (1, 64, , \texttt{PortToList(2)}, (P, 1) \to 1), n_{60}) \\
(n_{84}, (1, 108, , \texttt{PortSingleB}, (P, 1) \to 1), n_{72}) \\
(n_{83}, (1, 109, , \texttt{PortSingleB}, (P, 1) \to 1), n_{60}) \\
(n_{60}, (1, 75, , \texttt{PortFromList(2)}, (P, 1) \to 1), n_{72}) \\
(n_{72}, (1, 84, , \texttt{PortText}, (P, 1) \to 1), n_{87})
\end{gathered} \quad (3.9)$$

kus n_{ℓ_2} on servade (3.9) otstipud:

$$\begin{aligned} n_{60} &= (,60,60,\texttt{Tuple 2},\texttt{Tuple 2},,P \to 1,3,1) \\ n_{72} &= (,72,72,\texttt{Proj 1 2},\texttt{Proj 1 2},,P \to 1,2,1) \\ n_{83} &= (,82,83,\texttt{Req},\texttt{InputB},,P \to 1,0,1) \\ n_{84} &= (,82,84,\texttt{Req},\texttt{InputB},,P \to 1,0,1) \\ n_{85} &= (,54,85,\texttt{Receive},\texttt{InputS},,P \to 1,0,1) \\ n_{86} &= (,48,86,\texttt{Receive},\texttt{InputS},,P \to 1,0,1) \\ n_{87} &= (,81,87,\texttt{Send},\texttt{OutputB},,P \to 1,1,0) \end{aligned} \quad . \tag{3.10}$$

Parserist saadud servamuutujad on

$$\begin{gathered} ((I_1,,,,\texttt{Proj 1 2},,[],2,1),(1,,,\texttt{PortToList(1)},[]),(T,,,,\texttt{InputS},,[],0,1)) \\ ((I_2,,,,\texttt{InputB},,[],0,1),(1,,,\texttt{PortToList(2)},[]),(T,,,,\texttt{InputS},,[],0,1)) \\ ((B_2,,,,\texttt{InputB},,[],0,1),(1,,,\texttt{PortSingleB},[]),(P,,,,\texttt{InputS},,[],0,1)) \\ ((B_1,,,,\texttt{Tuple 2},,[],3,1),(1,,,\texttt{PortSingleB},[]),(T,,,,\texttt{InputS},,[],0,1)) \\ ((T,,,,\texttt{InputS},,[],0,1),(1,,,\texttt{PortFromList(2)},[]),(P,,,,\texttt{InputS},,[],0,1)) \\ ((P,,,,\texttt{InputS},,[],0,1),(1,,,,[]),(O,,,,\texttt{OutputS},,[],0,1)) \end{gathered} \tag{3.11}$$

Algul sorteeritakse mõlemad servad funktsiooniga *adjustment*, et paralleelselt asuvad servad sobiksid omavahel võimaluste arvude, sisendite ja väljundite arvu, teiste operatsioonide (λ_2) ning kokkuviidud vasaku *DGFR* muutujaservade ja reaalse *DGFR*'i servade otstippude identifikaatorite järgi. Kõikidel servadel, kus võimaluste arv $c(e) = 1$, saame tipud kohe muutujatega kokku viia, ning servadel, kus $c(e) > 1$, kasutame funktsiooni rekursiivselt. Kui sisendis pole ühtegi serva, kus oleks $c(e) = 1$, siis otsitakse esimene ettejuhtuv serv, kus üks otstipp on muutujaga seotud, millel seostatakse teine. Kui ka sellist serva ei leidu, siis otsitakse esimene serv, kus mõlemad tipud seostamata ja viime kokku neist ühe.

Näiteservade (3.9) ja (3.11) puhul on esimesel iteratsioonil kõigil $c(e) = 1$. Pärast sellist sobitamist saame servade hulga (3.9), mille tipud on juba muutujatega seostatud:

$$\begin{aligned} n_{60} &= (T,60,60,\texttt{Tuple 2},\texttt{Tuple 2},,P \to 1,3,1) \\ n_{72} &= (P,72,72,\texttt{Proj 1 2},\texttt{Proj 1 2},,P \to 1,2,1) \\ n_{83} &= (B_1,82,83,\texttt{Req},\texttt{InputB},,P \to 1,0,1) \\ n_{84} &= (B_2,82,84,\texttt{Req},\texttt{InputB},,P \to 1,0,1) \\ n_{85} &= (I_2,54,85,\texttt{Receive},\texttt{InputS},,P \to 1,0,1) \\ n_{86} &= (I_1,48,86,\texttt{Receive},\texttt{InputS},,P \to 1,0,1) \\ n_{87} &= (O,81,87,\texttt{Send},\texttt{OutputB},,P \to 1,1,0) \end{aligned} \quad .$$

3.5. Liigid ja pordid

Kui vasaku *DGFR*-muutuja servad on seostatud reaalsete andmetega, siis seatakse nende liigid (3.6) ja (3.8), ning servade pordid (1.4), kuhu serv suubub.

Tippudele `InputB` ja `InputS` pannakse liigiks `Sisend_tipp`, tippudele `OutputB` ja `OutputS` – `Väljund_tipp`. Vasaku DGFR'i sisetippudele (need, mis pole ei sisendega väljundtipud) `Olev_tipp` ja `Kaduv_tipp` vastavalt sellele, kas tipp eksisteerib ka

paremas *DGFR*'is või mitte. Paremas *DGFR*'is võivad sisetippudel olla `Olev_tipp` või `Tulev_tipp` vastavalt sellele, kas tipp eksisteerib vasakus *DGFR*'is või mitte. Ka servadele tulevad analoogsed liigid, et vasakus *DGFR*-muutujas võivad olla liigiks `Olev_serv` või `Kaduv_serv` ning paremas `Olev_serv` või `Tulev_serv`, kus olevad servad eksisteerivad mõlemas *DGFR*-muutujas. Liike seatakse failis *codeParserInteract.ml* asuva funktsiooniga `setKinds`.

Portide määramisest mitteväljundtippu suubuvatele servadele oli juttu alamjaotises 2.3.13. Parseri juures `OutputB`- ja `OutputS`-tippudesse suubuvatele servadele portide panek oli algul võimatu, kuna vasaku *DGFR*-muutuja servade genereerimise käigus ei ole teada, mis tipp see tegelikult on. Pärast vasaku *DGFR*'i muutujate seostamist reaalse graafi valimata järglastippudega on `OutputB`- ja `OutputS`-tippudesse suubuvate servade pordid teada. Nendesse väljundtippudesse suubuvatest servadest võetakse kõik portide andmed ja pannakse paremas *DGFR*'is tulevatele servadele, mille lõpus on väljundtipud.

3.6. Dimensioonid ja kujutused

Selles jaotises kirjeldatakse, kuidas tekivad *DGFR*-teisenduse käigus uute tippude dimensioonid ja servade kujutused algtipu dimensioonist lõpptipu omasse. Esimesena määratakse paremas *DGFR*'is tulevate servade olevate otstippude (alg- ja lõpptipp) andmed, mis leitakse vasakust *DGFR*'ist ning tulevate tippude esimesed operatsioonid. Lisaks sellele eemaldatakse paremast *DGFR*'ist olevad servad, kuna need juba eksisteerivad vasakus, ja teised servad liidetakse vasaku *DGFR*'iga, et tekiks üks ühine servade list. Sellest servade listist leitakse kõik tipud. Järgmisena viiakse tipud ja servad andmestruktuuri vastavalt (3.4) ja (3.3).

Enne raamistikule edastamist seatakse ka tulevate tippude dimensioonid ja servade kujutused, mis toimub failis *dimsAndMaps.ml*. Programm toetab osapoole nime piires kuni ühe dimensiooni seadmist. Näiteks ei saa rakendada dimensioone $P_1 \to 2, P_2 \to 3$, aga sobib küll selline näide: $P_1 \to 1, P_2 \to 1, P_3 \to 1, P_4 \to 1$. Dimensioonide ja kujutuste osas tekitatakse fragmendist piisavalt koopiaid, et igas koopias esineks oma dimensioonides leiduv osapoole nimi. Uuele lõpptipule lisatakse dimensioonid vaid siis, kui sellesse vähemalt ühe saabuva serva algtipp on dimensiooniga. Tulevale servale pannakse kujutus, kui selle algtipp on vastava dimensiooniga.

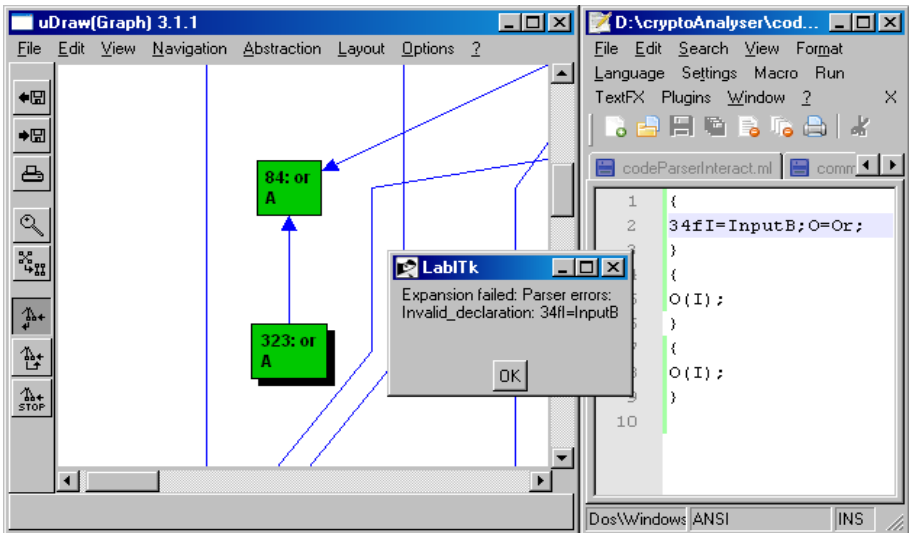
Algoritm vaatab erinevate osapoole nimedega dimensioone eraldi: tekitab servade hulgas vajaliku arvu koopiaid, milles igaühes esinevad vaid oma nimega dimensioonid ja kujutused. Seejärel paneb tulevatele servadele kujutused ja otstippude dimensioonid. Lõpuks ühildab tulevad servad ja tipud, kus on vastavalt leitud uued kujutused ja dimensioonid.

3.7. Teisenduste keele poolt genereeritavad veateated

Parserist tulevad veateated, kui kasutatav muutuja on deklareerimata, leidub mittenaturaalarvulisi indekseid, üritatakse jagada nulliga, esineb ebasobivaid arvu- ja tipu avaldisi, operatsioone ja vale arv parameetreid, automaatse deklareerimise juures ei õnnestu leida arvavaldise väärtust, esineb mittesobivaid sõltuvusi, servade hulki ja deklaratsioone

ning mingi lõpptipu kasutamine rohkem kui üks kord, vale pordi määramine. Kontrollitakse ka, et servade lõpptippudeks ei satuks eeldatavasti sisenditeta tippe ja algtippudeks väljunditeta tippe.

Igal semantikat defineerival funktsioonil on omad veateated, mida *uDraw(Graph)* näitab eraldi tekstikastina (joonis 3.4). Peale semantikafunktsioonide veateadete on ka teisi



Joonis 3.4. Näide veateatest

veateateid. Teisenduskeeles kontrollitakse erinevaid asju: kas iga paremas *DGFR*'is esinev sisend- ja väljundtipp esineb ka vasakus *DGFR*'is, kas kõigil muutujatel on väärtused, kas iga tipp on seotud vähemalt ühe servaga ja kas etteantud nimega koodifail eksisteerib. Võrreldakse ka vasakut ja reaalset *DGFR*'i, et mõlemas oleks sama palju servi ning iga serv ühes *DGFR*'is kattuks etteantud parameetrite (sisendite ja väljundite arvud otstippudel, operatsioonid) järgi teise *DGFR*'i vähemalt ühe servaga ja vastupidi. Kontrollitakse ka seda et ei valita tippe operatsioonidega `Send`, `Req`, `Receive`, `Begin` ja `End`. Veateateid genereeritakse lisaks juhtudel, kui muutujate ja reaalsete andmete kokkuviimine on võimatu või dimensioone ja kujutusi (jaotised 1.6 ja 1.7) ei saa määrata. Kõik võimalikud veateated on ära toodud Lisas 2.

4. peatükk

Krüptoanalüsaatori kasutamine

Selles peatükis kirjeldatakse *Krüptoanalüsaatori* käivitamist ja selleks ettevalmistamist. Tehakse läbi ka lühike näide sõltuvusgraafi transformeerimise kohta peamiselt teisenduste keele abil. Turvalisuse tõestuse asemel on näidetes püütud keskenduda *DGR*'i tippude ja servade ümberpaigutamisele teisenduste keele abil. Teisenduste käigud on piltidena salvestatud Lisas 3 asuva *CD*-plaadi kausta *ptk4.2.2*.

4.1. Ettevalmistus ja käivitamine

Ligikaudne installatsiooni käik on eraldi välja toodud ka käesolevas töös. Iga punkti juures toodud allika mittetoimimise korral saab vastava tarkvara ka kaasasolevast *CD-ROM*'ist, või *Interneti* otsimootorite [11], [28], [23] jt kaudu. Erineva vajaliku tarkvara paigaldamine toimub operatsioonisüsteemil *Windows XP* umbes sellises järjekorras:

1. Selle operatsioonisüsteemi jaoks on tarvis hoolduspaketi *Windows XP Service Pack 2 - SP2* [25] olemasolu.
2. *OCaml* [17], soovitatavalt *MSVC*-põhise pordiga.
3. *Visual C++ Express Edition 2005* [27], mille installimine nõuab mitu tundi aega.
4. *Microsoft Windows Server 2003 SP1 Platform SDK* [26]. Seal asuvad vajalikud teegifailid.
5. *The Microsoft Macro Assembler* [24].
6. *TCL/TK version 8.4* [3].
7. *FlexDLL* [10].
8. Analüsaatori kompileerimiseks *OMAKE* [2].
9. Graafide visualiseerimise tarkvara *uDraw(Graph)* [1].

Rakenduse kompileerimiseks on soovitatav kasutada faili *BuildGrb.cmd*, mis kõik vajalikud kompileerimise käsud välja kutsuks. Soovitatav on selles failis muuta kopeerimise käsku *COPY GRB.OPT "C:\Documents and Settings\user\GRB.EXE"*, kus *uDraw(Graph)* käivitamise ajal on. Üle tuleb vaadata ka faili *setenv.cmd* sisu, et seal märgitud teed oleks kooskõlas tegelike nimetatud programmi asukohtadega oma arvutis.

Linux-operatsioonisüsteemil töölesaamine on läbi proovitud *Ubuntu 11.04* baasil. *Systems→Administration→Synaptic package manger*, sisestada peakasutaja parool ning tõmmata ja paigaldada vähemalt programmid *camlp4*, *ocaml*, *omake*, *ocamlpp4-extra* ja *tk8.5*. *tk8.5* toimima saamiseks tuleb see linkida: *sudo ln -s /usr/lib/libtk8.5.so.0 /usr/lib/libtk8.5.so*. *Linux*-operatsiooni puhul on kompileerimine pisut lihtsam – kasutada terminali käsku *omake*.

Kompileerimine (nii *Windows*'i kui ka *Linux*'i peal) võtab tavaliselt mõne minuti. See võib ka "kinni jääda", mille lahendamiseks on tegevus katkestada ja alustada uuesti. Et teisenduste keelt kasutada, tuleb enne kompileerimist määrata failis *GrbTrReadModule.ml* sisestatud koodi jaoks sobiv muutuja `codeFile` ja logifaili jaoks muutuja `logFile` väärtus.

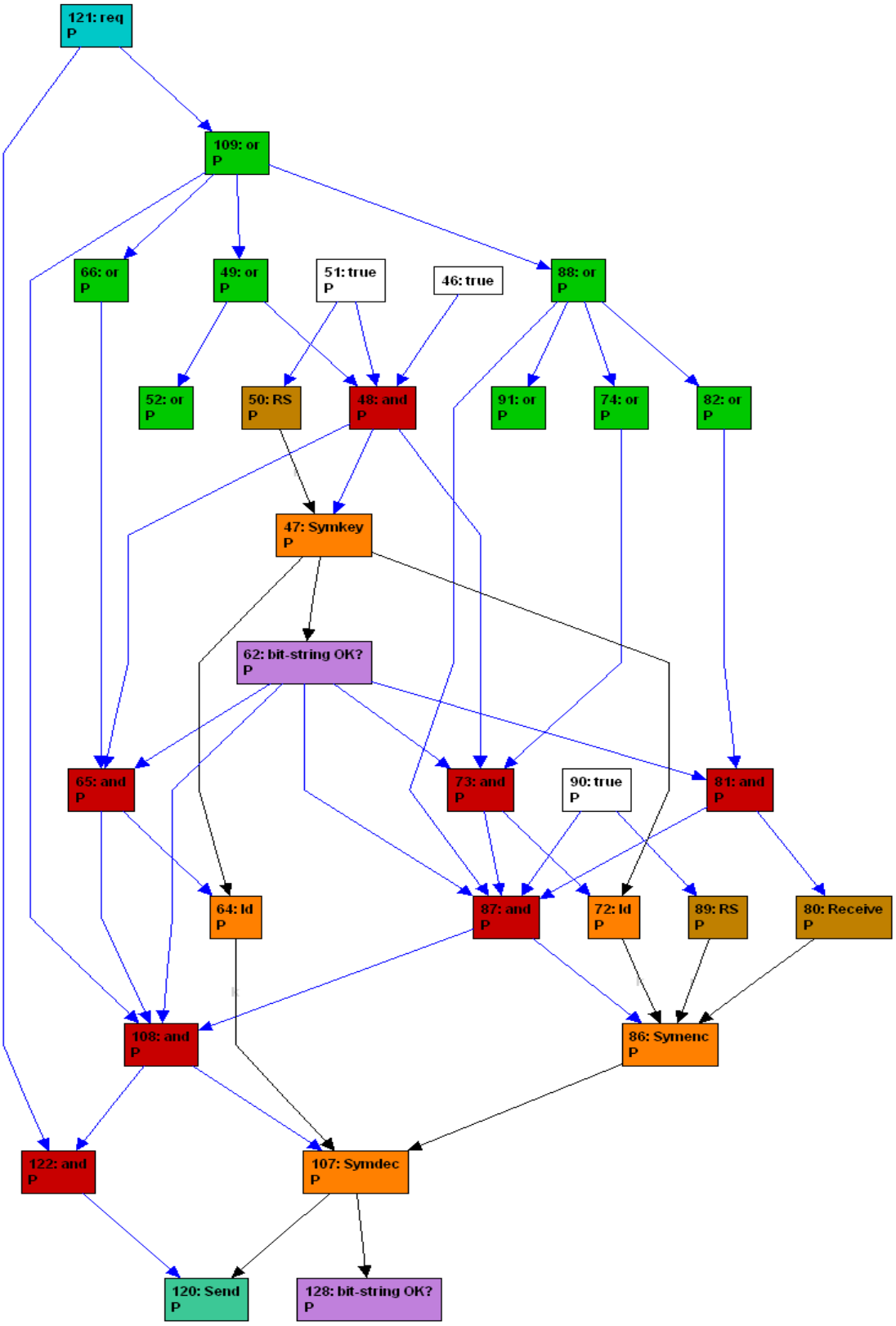
Kui programm on kompileeritud, siis *Linux*'is tuleb võtta tarkvara *uDraw(Graph)*, pakkida see lahti soovitud kausta (nt */home/user/uDrawGraph-3.1*), panna faili */home/user/.profile* rida "*export UDG_HOME=/home/user/uDrawGraph-3.1*", liikuda uuesti avatud terminalis kausta */home/user/uDrawGraph-3.1* ja käivitada käsuga "*./uDrawGraph*". *Windows*'is on see programm tarvis installida ja kävitada *Start*-menüü pealt.

Pärast programmi *uDraw(Graph)* käivitamist saab *uDraw(Graph)* aknas võtta *File→Connect Application* ja navigeerida analüsaatori kausta ning valida *grb.opt* või *grb.exe*. Seejärel ilmub aken *Connect Applcation Option*, kus valida *UNIX-Pipes* ja vajutada nupule *Connect*.

Et saada kätte soovitud sõltuvusgraafi esitust, tuleks valida sobiv protokoll menüüst *Edit→Predefined protocols*. Avanenud graafilt valida sobivad tipud, kirjutada teisendustekeelne kood faili (nt *code.txt*), mis on määratud failis *GrbTrReadModule.ml* muutuja `codeFile` järele. Seejärel saab keele kasutust proovida menüüst *Edit→Use transformation language*.

4.2. Sõltuvusgraafi transformeerimine

Käesolevas jaotises käsitletakse uue krüptoprotokollile vastava sõltuvusgraafi lisamist ja selle kallal töötamist. Analüsaatorile andmiseks pole paremat viisi protokolli sisestamiseks, kui *OCaml*-koodi täiendamine kahes moodulifailis. Siin on piirdutud primitiivse *HelloWorld*-stiilis näitega. Selleks on proovitud krüpteeringu ja lahtikrüpteeringu eemaldamist $dec(k,(enc(r,k,x))) \rightarrow x$,. Katsetamiseks tuleb genereerida soovitud katsegraaf (joonis 4.1), mille transformeerimisel on pööratud tähelepanu sellele, kuidas rakendada teisendusekeelseid koode. Lisas 3 olevas koodis on see näide olemas. Teise näitena on toodud sõltuvusgraafi *Just two encryptions* transformeerimine.



Joonis 4.1. Esialgne näitesõltuvusgraaf

4.2.1. Sõltuvusgraafi genereerimine

Kõigepealt genereerime graafi, mille peal teisendusi katsetada. Selleks tuleb sisestada faili *GrbInput.ml* funktsioon

```
let remCrypt = PL
   rep(P, k := symkey; send symdec(k,symenc(k,receive)); stop);;
```

mis pannakse kirja allikas [20] toodud parseri keeles.

Et seda krüptoprimitiivi graafi avada, tuleb täiendada ka faili *GrbInteract.ml*. Algusesse on tarvis defineerida menüü kirje. Näiteks `let mitem_myKryPrim = "menu_myKryPrim";;`. Funktsioonis `initmenus` lisada rida

```
   ",menu_entry(\"" ^ mitem_myKryPrim ^
      "\",\"Simple Crypto Primitive\")" ^
```

ja panna kirje `mitem_myKryPrim` ka listi, rea `activemenus := UDrawParamSet.union (UDrawParamSet.from_list` järel. Viimsena lisada read

```
   | Term ("menu_selection", [Quoted s]) when s = mitem_myKeyPrim ->
      guardreplgraph (fun () -> currgr := prtogr remCrypt)
```

funktsiooni `mainloop` sisse. Pärast faili *GrbInteract.ml* täiendamist tuleb raamistik uuesti kompileerida ning avada *uDraw(Graph)* ja *File→Connect Application*, navigeerida kompileeritud rakenduse (n. *GRB.exe*, *GRB.run*). Pärast *GRB*-faili avamist ilmub aken, kus valida *UNIX-pipes* ja vajutada nuppu *Connect*. Pärast seda, kui *uDraw(Graph)* on rakendusega ühendatud, tuleb võtta menüü *Edit→Predefined protocols* ja vaadata, kas leidub sealt graaf *Simple Crypto Primitive*. Kui on, siis avada graaf (joonis 4.1). Graafi korrastamiseks on soovitatav *uDraw* aknas hiire parema klõpsuga hüpikmenüüst valida *Improve All*. Nüüd saab graafi teisendada. Selle menüü alt saab kätte kõik genereeritud *DGR*'id.

4.2.2. Teisenduste keele kasutamine

Enne teisenduste keele kasutamist eemaldame n-ö surnud tipud (*Edit→Remove dead nodes*).

Järgmisena saame andmevood suunata otse, mis läbi *Id*-tippude kulgevad. Järjestikku *edit→Pass over an Id-node* ja *edit→Remove dead nodes* saame teha sama koodiga:

```
{m=2; loop i 1 m (ID[i]=Id;B[i]=InputB;O[i]=OutputS;S[i]=InputS)}
{loop i 1 m (ID[i](B[i];S[i]);O[i](ID[i]))}{loop i 1 m (O[i](S[i]))}
```

Valime *Id*-tipud 64 ja 72. Mitme tipu korraga valimisel on kaks võimalust: esimeseks lohistada ala üle soovitud tippude ja teiseks valida tipud ükshaaval, samal ajal hoides all *Shift*-klahvi. Teisendus on korrektne, kuna tõese kontrollsõltuvuse korral väljastatakse selle sisend. Kood tuleb kirjutada faili *code.txt* (vt jaotist 4.1). Ja teisenduste keele abil *edit→Use transformation language*.

Seejärel valime tipud 107 (*Symdec*) ja 86 (*Symenc*) ning kasutame koodi alamjaotises 2.1.2, et eemaldada need kaks tippu ja ühendatakse serv *receive*-tipust 81 *Send*-tippu 120. Sellega sooritme teisenduse $dec(k, (enc(r, k, x))) \rightarrow x$.

Eemaldame n-ö surnud *RS*-tipu 89 ning koodiga `{T=True;O=OutputB;}{O(T)}{}` ükshaaval *True*-tipud 46 ja 90. Korrektsus tuleneb lausest $x \wedge true \equiv x$.

Eemaldame koodiga `{I=InputB;OR=Or;O=OutputB;}{OR(I);O(OR)}{}` ükshaaval ühe sisendi ja ühe väljundiga *Or*-tipud identifikaatoritega 82, 74, 66 ja 88 (joonis 4.1) kuna neid tippe läbivad vood on üleliigsed.

Järgmisena kasutame koodi

```
{I=InputB;O1=Or;O2=Or;OO1=OutputB;OO2=OO1;}
{O1(I);OO1(O1);OO2(O2);O2(O1);}{O1(I);OO2(O2);O2(O1);}
```

ja valime *Or*-tipud 109 ja 49. Selle teisenduse järel eemaldub tipust 109 väljuv serv *OutputB*-tipuna kasutatavasse *And*-tippu 108. Ka see on korrektne teisendus, sest eemaldatavas servas ja *And*-tipust 65 tulevast servast on mõlemate tõeväärtused alati samad. Pärast seda teisendust võtame *Or*-tipud 49 ja 109 vahelt välja koodiga `{I=InputB;O=OutputB;OR=Or;}{OR(I);O(OR);}{O(I);}`, et kontrollvood liiguksid otse.

True-tipust *And*-tippu ühendatud serva eemaldame koodiga

```
{T=True;R=RS;O=OutputS;O3=O;O2=OutputB}{R(T);O2(T);O(R)}{R(T);O(R)}
```

ning valida tuleb tipud 51 (*True*) ja 50 (*RS*), sest vaid *True*-tipu valiku korral rakendus ei tea, kumba serva eemaldada ja teeb seda mittesoovitud servaga.

Eemaldame üleliigseid *And*-tippe. Kõigepealt võtame maha serva *IsOK*-tipust 62 *And*-tippu 108 ja kood on

```
{A1=And;A2=And;A3=And;O=OutputB;loop i 1 6 (I[i]=InputB)}
{A1(I[1];I[2]);A2(A1;I[3];A3);A3(I[4];I[5];I[6]);O(A2)}
{A1(I[1];I[2]);A2(A1;;A3);A3(I[4];I[5];I[6]);O(A2)}
```

ning valida *And*-tipud 65, 87 ja 108.

Järgmisena eemaldame *And*-tippude 81 ja 87 vahelt serva. Kindlasti valida kaks tippu, sest valides näiteks 81 ja kasutades koodi `{A=And;I=InputB;O1=OutputB;O2=OutputB;}{A(I);O1(A);O2(A);}{A(I);O2(A);}` siis rakendus võib eemaldada vale väljuva serva. Seepärast tuleks lisaks sellele tipule täpsustamiseks valida ka teine tipp, näiteks *And*-tipp 87 (*Receive*-tippu ei saa valida, kuna see ei saa kuuluda *DGFR*'i). Seega valime tipud 81 ja 87 ja kasutame koodi

```
{n=2;loop i 1 n (O[i]=OutputB;A[i]=And);loop i 1 3 (I[i]=InputB)}
{A[1](I[1]);A[2](A[1];I[2];I[3]);loop i 1 n (O[i](A[i]))}
{A[1](I[1]);A[2](    ;I[2];I[3]);loop i 1 n (O[i](A[i]))}.
```

Järgmine üleliigne serv, mida eemaldada on *IsOK*-tipust 62 *And*-tippu 87. Ka siin ei soovitata valida ainult seda *And*-tippu, sest rakendus võib soovitud sisendserva asemel võtta *And*-tipust 73. Seega valime tipud 87 ja 73 ning kasutame koodi

```
{loop i 1 3 (I[i]=InputB); loop i 1 2 (A[i]=And);O=OutputB;}
{A[1](loop i 1 2 (I[i]));A[2](A[1];I[3]);O(A[2])}
{A[1](loop i 1 2 (I[i]));A[2](A[1];    );O(A[2])}.
```

Nagu näha, on muutujad `I[3]` ning üks muutujatest `I[1]` või `I[2]` seotud ühe ja sama tipuga (antud juhul *IsOK*-tipuga 62). See tuleneb sellest, et iga väljund- ja sisendtipu muutuja laseb läbi täpselt ühe serva.

Valime *And*-tipud 81 ja 87 ja eemaldame koodiga

```
{n=2;loop i 1 n (I[i]=InputB;O[i]=OutputB;A[i]=And;);}
{loop i 1 n (A[i](I[i]);O[i](A[i]))}{loop i 1 n (O[i](I[i]))}
```

need tipud, et servad otse ühendada.

Valime *And*-tipud 65 ja 108, kasutame koodi

```
{loop i 1 3 (I[i]=InputB);A1=And;A2=And;O=OutputB;}
{A1(I[1];I[2]);A2(I[3];A1);O(A2)}{A1(I[1];I[2]);O(A1)}
```

et eemalduks tipp 108 ja tekiks otseserv tipust 65 tippu 122.

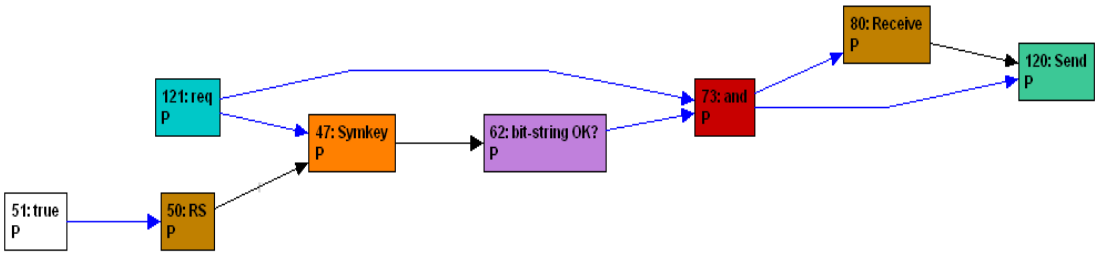
Valime *And*-tipud 65 ja 122 ja eemaldame koodiga

```
{A1=And;A2=And;I1=InputB;I2=InputB;I3=InputB;O=OutputB}
{A1(I1;A2);A2(I2;I3);O(A1)}{A1(  ;A2);A2(I2;I3);O(A1)}
```

üleliigse serva *Req*-tipust 121 *And*-tippu.

Valime *And*-tipu 122 ja eemaldame selle koodiga `{I=InputB;O=OutputB;A=And}{A(I);O(A)}{O(I)}`, et läbi selle kulgev kontrollvoog otseks teha.

Viimasena kasutame näitekoodi alamjaotises 2.1.3 (ainuke erinevus on `m=2` ja `n=2`) ja valime *And*-tipud 65 ja 73, et üks neist eemaldada.



Joonis 4.2. Teisendusi läbinud graaf (joonis 4.1)

Pärast neid teisendusi on graaf näidatud joonisel 4.2. See oli lihtne näide erinevate teiseduste sooritamiseks.

Kokkuvõte

Erinevaid krüptograafilisi protokolle luuakse üle maailma iga päev juurde paljudes valdkondades, nagu näiteks pangandus, e-valimiste organiseerimine, *Interneti*-poed, erinevate taristute (nt transport ja pilvarvutused) töödekorralduses kasutatavad infosüsteemid jne. Paralleelselt muutuvad ka ründed keerukamaks ja neid lisandub järjest. Rünnete peamisteks eesmärkideks on omakasupüüdlikkus, mis tahes süsteemi töö halvamine jt. Seepärast on tarvis loodavate krüptoprotokollide turvalisuses veenduda. Selle üks võimalikke abivahendeid võib olla teisenduste keelega varustatud sõltuvusgraafidel põhinev *Krüptoanalüsaatori* rakendus.

Käesoleva töö eesmärk oli teisenduste keele disainimine, programmeerimine ja *Krüptoanalüsaatorile* juurde lisamine. Üldiselt see eesmärk on täidetud ja teisenduste keel on abiks *DGFR*'ide teisendamisel. Teisenduste jaoks ei lähe alati uute *OCaml*-moodulite kirjutamist tarvis. Mis tahes teisenduse ettevalmistuseks kuluv aeg peaks vähenema märgatavalt – mitmesajarealise *OCaml*-mooduli asemel saab sobiva teisenduse jaoks kirjutada alla kümnerealise teisenduste keelse koodi.

Osa ülesandeid teisenduste keele juures jäi ka tegemata, kuna need oleksid nõudnud rohkem aega, kui käesoleva töö kirjutamine võimaldas. Näiteks teisenduste keeles tuleks veel täiendada arvumuutujate deklareerimist, et nendele leitakse väärtused automaatselt igasugusel võimalusel, mille implementeerimine võib olla üsna raske. Täiendada tuleks ka tulevate tippude dimensioonide kujutiste panekut, et teisenduste keel toetaks ka ühest suurema arvu osapoole nimega dimensioone. Samuti oleks tarvis lisada keele abil tehtud teisenduste tagasivõtmine, salvestatud faili lugemine, et sama protokolli uurimist jätkata järgmisel päeval. Ka lisada salvestatud teisenduste n-ö maha mängimine, mis näitaks kogu teisenduse käiku ette teistele krüptoloogia huvilistele.

Autor tänab Peeter Lauda abi ja nõuannete eest.

Adding a transformation language to the *Cryptoanalyser*

Master's thesis

Ivo Seeba

Abstract

In the last few decades information technology has developed a lot. Infrastructures using information systems are in wide use. Unfortunately there are people sometimes attacking different information systems with selfish goals. Attacks are becoming more and more complicated. Different cryptographic protocols — methods of classified communication between parties — have been invented to protect against the attacks. The security of cryptographic protocols against various attacks, and the preservation of security properties by these protocols has to be studied.

There are several different methods for proving the security of cryptographic protocols — circuits, mathematical proofs (making use of number theory, theory of probability, complexity theory, algebra, analysis, etc.), game-based techniques [4, 29] and dependency graphs [19, 30, 18], similar to the programs of the *LabVIEW* environment [7, 8]. In game-based proofs, a sequence of games is generated where the first game corresponds to the real and the last one to the ideal protocol. The neighbouring games are indistinguishable for a computationally bounded attacker. The distinguishability of the games follows from the security properties the cryptographic primitives are assumed to satisfy [29].

Dependency graphs (DG) are a recent method for proving the security of cryptographic protocols. A dependency graph is a directed graph where each node is an operation outputting a value and each edge is a dependency through which the computed values are transmitted from the nodes that computed them to the nodes that make use of them. In contrast to *LabVIEW* where the edges carry integers, floating-point numbers, strings, etc., the dependency graphs explored in this work only carry bit-strings or booleans. Cryptographic games are translated into dependency graphs; the *DG*-transformation makes use of subgraphs called fragments (*DGF*-s), replacing a fragment with a different, but computationally indistinguishable one. *DG*'s are often infinite, as the number of protocol executions by a party is not bounded, but each node corresponds to a single operation only. Such graphs can also be represented finitely, introducing replication indices to the vertices. *Dependency Graph Representation (DGR)* transformations are also expressed through *Dependency Graph Fragment Representations – DGFR*.

Several cryptographic protocol analysers have been created. *CryptoVerif* [5] is an example of an analyser using game-based techniques. A dependency-graph based analyser also exists, created by P. Laud, I. Tšahhirov and E. Jaaniso [19, 18, 20]. The analyser is written in *OCaml*. It is used together with the program *uDraw(Graph)* where nodes and edges can be chosen as inputs to transformations. Each transformation is implemented in its own module consisting of several hundred lines of *OCaml*-code. The goal of this work was the addition of a separate language of transformations to the analyser, allowing transformations to be represented in just a few lines. The work describes the formal grammar of the language, its semantics, implementation and the usage together with *uDraw(Graph)*. Examples of usage are included in the thesis.

Kirjandus

[1] uDraw(Graph) - The powerful solution for graph visualization. Bremen, 2005. http://www.informatik.uni-bremen.de/uDrawGraph/en/download/download.html – viimati vaadatud 13.06.2011.

[2] OMake, 2010. http://omake.metaprl.org/download.html – viimati vaadatud 13.06.2011.

[3] ActiveTCL. Tcl/tk version 8.4, 2000. http://www.activestate.com/activetcl/downloads – viimati vaadatud 13.06.2011.

[4] M. Bellare and P. Rogaway. Code-Based Game-Playing Proofs and the Security of Triple Encryption, 2004. http://eprint.iacr.org/2004/331.pdf – viimati vaadatud 13.06.2011.

[5] B. Blanchet. Cryptoverif: Cryptographic protocol verifier in the computational model. http://www.cryptoverif.ens.fr/ – viimati vaadatud 13.06.2011.

[6] B. Blanchet. Proverif: Cryptographic protocol verifier in the formal model. http://www.proverif.ens.fr/ – viimati vaadatud 13.06.2011.

[7] National Instruments Corporation. Keskkond LabVIEW. http://www.ni.com/labview – viimati vaadatud 13.06.2011.

[8] National Instruments Corporation. *LabVIEW manuaal*. http://www.chem.unc.edu/courses/742L/labfiles/EXPERIMENT0_Labview_5_manual.pdf – viimati vaadatud 13.06.2011.

[9] P. Manoury ja B. Pagano E. Chailloux. *Developing Applications With Objective Caml*. O'Reilly France, paris, 2000. http://caml.inria.fr/pub/docs/oreilly-book/ – viimati vaadatud 13.06.2011.

[10] A. Frisch. FlexDLL. http://alain.frisch.fr/flexdll.html – viimati vaadatud 13.06.2011.

[11] Google. Otsimootor Google, 1998. http://www.google.com.

[12] A. D. Gordon and A.Š. A. Jeffrey. Typing Correspondence Assertions for Communication Protocols. In *Proc. Mathematical Foundations of Programming Semantics*, Electronic Notes in Computer Science. Elsevier, 2001. Full version appeared in Theoretical Computer Science 300. http://ect.bell-labs.com/who/ajeffrey/papers/mfps01.pdf – viimati vaadatud 13.06.2011.

[13] A. D. Gordon and A.Š. A. Jeffrey. Typing One-to-One and One-to-Many Correspondences in Security Protocols. In *Proc. Int. Software Security Symp.*, volume 2609 of *Lecture Notes in Computer Science*, pages 263–282. Springer-Verlag, 2002. http://ect.bell-labs.com/who/ajeffrey/papers/isss02.pdf – viimati vaadatud 13.06.2011.

[14] J. Henno. *Formaalsed Keeled, Grammatikad ja Translaatorid*. Tallinna Tehnikaülikool, Informaatikainstituut, 2006.

[15] J. Hickey. *Introduction to Objective Caml*. Cambridge University Press, 2007. http://files.metaprl.org/doc/ocaml-book.pdf – viimati vaadatud 13.06.2011.

[16] N. Hirokawa. Programming in OCAML. Technical report, Institute of Computer Science University of Innsbruck, 2006. http://cl-informatik.uibk.ac.at/teaching/ss06/ocaml/schedule.php – viimati vaadatud 13.06.2011.

[17] INRIA. Programmeerimiskeel OCaml. Prantsusmaa, 2010. http://caml.inria.fr/download.en.html – viimati vaadatud 13.06.2011.

[18] P. Laud ja I. Tšahhirov. Symmetric encryption in automatic analyses for confidentiality against active adversaries. *2004 IEEE Symposium on Security and Privacy toimetistes, lk. 71-85*, 2004. http://vvv.cs.ut.ee/~peeter_l/research/IEEESP04long.ps.gz – viimati vaadatud 13.06.2011.

[19] P. Laud ja I. Tšahhirov. Application of Dependency Graphs to Security Protocol Analysis. In *Symposium on Trustworthy Global Computing (TGC 2007) toimetistes, lk. 294-311*, 2007. http://vvv.cs.ut.ee/~peeter_l/research/tgc07.ps.gz – viimati vaadatud 13.06.2011.

[20] E. Jaaniso. Bakalaureusetöö: Mängude jadal põhineva krüptograafiliste protokollide analüsaatori täiendamine. Tartu Ülikool, Arvutiteaduse instituut, 2009.

[21] P. Laud. Symmetric encryption in automatic analyses for confidentiality against active adversaries. *2004 IEEE Symposium on Security and Privacy toimetistes, lk. 71-85*, 2004. http://www.cs.ut.ee/~peeter_l/research/IEEESP04.ps.gz – viimati vaadatud 13.06.2011.

[22] P. Laud. Symmetric encryption in automatic analyses for confidentiality against active adversaries. *2004 IEEE Symposium on Security and Privacy toimetistes, lk. 71-85*, 2004. http://vvv.cs.ut.ee/~peeter_l/research/IEEESP04long.ps.gz – viimati vaadatud 13.06.2011.

[23] Microsoft. Otsimootor Bing. http://www.microsoft.com/en/us/default.aspx.

[24] Microsoft. The Microsoft Macro Assembler. http://www.microsoft.com/downloads/en/confirmation.aspx?FamilyID=7a1c9da0-0510-44a2-b042-7ef370530c64 – viimati vaadatud 13.06.2011.

[25] Microsoft. Windows XP Service Pack 2: XP SP2. http://www.softwarepatch.com/windows/winxpsp2-security.html – viimati vaadatud 13.06.2011.

[26] Microsoft. Microsoft Windows Server 2003 SP1 Platform SDK, 2006. http://download.microsoft.com/download/a/5/f/a5f0d781-e201-4ab6-8c6a-9bb4efed1e1a/PSDK-x86.exe – viimati vaadatud 13.06.2011.

[27] MS Visual Studio. Visual c++ 2005 express edition. http://download.microsoft.com/download/8/3/a/83aad8f9-38ba-4503-b3cd-ba28c360c27b/ENU/vcsetup.exe – viimati vaadatud 13.06.2011, 2006.

[28] Overture Services. Otsimootor AltaVista, 1995. http://www.altavista.com/.

[29] V.Šhoup. Sequences of games: a tool for taming complexity in security proofs, 2004. http://eprint.iacr.org/2004/332.pdf – viimati vaadatud 13.06.2011.

[30] I. Tšahhirov. *Security Protocols Analysis in the Computational Model- Dependency Flow Graphs-Based Approach (Turvaprotokollide analüüs arvutuslikul mudelil - sõltuvusgraafidel põhinev lähenemisviis)*. PhD thesis, Tallinna Tehnikaülikool, 2008. – viimati vaadatud 13.06.2011.

Lisad

Lisa 1. Tippude operatsioonid
Lisa 2. Teisenduste keele veateated
Lisa 3. *CD-ROM* täiendatud tarkvaraga

Lisa 1
Tippude operatsioonid

Selles lisas tuuakse ära nimekiri tippudel olevatest operatsioonidest, mida teisenduste keel toetab. Operatsiooni järel sulgudes on sisendpordid, mille külge on ühendatud servad nende algtippudest – sisenditest. Tippude operatsioonid koos sisendportidega järgnevas nimekirjas on kujul <*operatsioon*>($port_1, \ldots, port_m$) ning teisenduse koodi kirjutamisel on oluline sisendite järjekord, sest see määrab ära, millise tipu väärtus konkreetsesse porti tuleb. Operatsioone, mida toetab lisatud teisenduste keel ja mis võivad kuuluda vasakusse ja paremasse *DGFR*-muutujasse, on toodud järgmises loetelus.

1. *RS*(`PortSingleB`) – On juhuslike bittide allikas. Kõik juhuslikkust kasutavad operatsioonid saavad oma juhuslikud bitid mingi *RS*-tipu käest.

2. *Nonce*(`PortSingleB`, `PortRS`) – Genereerib mingi juhuarvu (põhimõtteliselt lihtsalt väljastab *RS*-tipust tulevad juhuslikud bitid ja võib ka neid kuidagi formaatida).

3. *Constant*^{n}(`PortSingleB`) – Konstantse väärtuse väljastamiseks bitijadana. $n \in \mathbb{N}$ on konstandi väärtus.

4. *Keypair*(`PortSingleB`, `PortRS`) – Krüpteerimise võtmepaari genereerimiseks.

5. *Pubkey*(`PortSingleB`, `PortRS`) – Avaliku võtme genereerimiseks. Kokkukuuluvad *Keypair*- ja *Pubkey*-tipud tulevad samast *RS*-st.

6. *Pubenc*(`PortSingleB`, `PortRS`, `PortPubkey`, `PortText`) – Teate krüpteerimiseks avaliku võtmega, kus sisendid on kontrollsõltuvus, juhuslikud bitid, avalik võti ja krüpteeritav tekst.

7. *Pubdec*(`PortSingleB`, `PortPubenc`, `PortEncKP`) – Lahtikrüpteerimine avaliku võtmega. Sisse tulevad servad on tavaliselt kontrollsõltuvus ja servad tippudelt *PubEnc* ja *Keypair*.

8. *PubencZ*(`PortSingleB`, `PortRS`, `PortPubkey`) – Sama mis tipp `Pubenc`, kuid selle vahega, et krüpteeritava teate asemel on nullidest koosnev bitijada.

9. *Symkey*(`PortSingleB`, `PortRS`) – Sümmeetrilise võtme genereerimiseks.

10. *Symenc*(PortSingleB, PortRS, PortSymkey, PortText) – Teate krüpteerimiseks sümmeetrilise võtmega, kus sisendid on kontrollsõltuvus, juhuslikud bitid, sümmeetriline võti ja krüpteeritav tekst. Kokkukuuluvad tipud *Symkey* ja *Symenc* tulevad alati samast *RS*-st.

11. *SymencZ*(PortSingleB, PortRS, PortSymkey) – Sama mis *Symenc,* kuid selle vahega, et krüpteeritava teate asemel on nullidest koosnev bitijada.

12. *Symdec*(PortSingleB, PortSymenc, PortSymkey) – Lahtikrüpteerimine sümmeetrilise võtmega, kus peale kontrollsõltuvuse sisendid on tipud *SymEnc* ja *Symkey*.

13. *SigVer*(PortSingleB, PortRS) – Allkirja- ja verifitseerimisvõtme paari genereerimiseks.

14. *Verkey*(PortSingleB, PortSigKP) – Tipust *SigVer* verifitseerimisvõtme eraldamiseks võtmepaarist.

15. *Signature*(PortSymkey, PortRS, PortSigKP, PortText) – Allkirja genereerimiseks, kus sisenditeks on kontrollsõltuvus, juhumündid ning allkirja- ja verifitseerimisvõtme paar.

16. *SignedMsg*(PortSingleB, PortSignature) – Allkirja all olev teade, kus sisendid on kontrollsõltuvus ja allkiri (ehk signatuur).

17. *TestSig*(PortSigkey, PortSignature) – Allkirja verifitseerimine etteantud võtme ja allkirjaga ning tagastab *true,* kui allkiri kehtib.

18. $Tuple^n$(PortSingleB, $\mathtt{PortToList}_1, \ldots, \mathtt{PortToList}_n$) – Mitme sisendi liitmiseks korteešiks, kus portidesse saabuvad kontrollsõltuvus ja n bitijada serva. Tipp saab sisendid $x_1, x_2, \ldots, x_n$ ja konstrueerib korteeži $(x_1, x_2, \ldots, x_n)$.

19. $Proj^n_m$(PortSingleB, $\mathtt{PortFromList}_1, \ldots, \mathtt{PortFromList}_n$) – Teostab vastupidist operatsiooni *Tuple*-tipule. Sisendiks võtab kontrollsõltuvuse ja mingi bitijada vektori (mis ise on samuti bitijada) ja väljastab vektori komponendi indeksiga m eeldusel, et vektor on pikkusega n.

20. *Secret*(PortSingleB) – Modelleerib protokolli kaitstavat salateadet.

21. *Merge*(PortSingleB, PortMerge) – Väljastatakse andmesisend, kui kõik andmesisendid omavahel on võrdsed ja kontrollsõltuvus tõene, muudel juhtudel $\perp$.

22. *Id*(PortSingleB, PortText) – Väljastatakse sama andmesisendi väärtus siis ja ainult siis, kui kontrollsõltuvus on tõene.

23. *And*(PortStrictB,PortStrictB,...) – Toimib konjunktsioonina ja sisendportide arv võib olla kui tahes suur.

24. *Or*(PortUnstrB,PortUnstrB,...) – Toimib disjunktsioonina ja sisendportide arv võib olla kui tahes suur.

25. *True*() – Väljastab konstantse tõeväärtuse *true*.

26. *False*() – Väljastab konstantse tõeväärtuse $false$.

27. *Error*() – Väljastab konstantse väärtuse $\bot$.

28. *TTT*(`PortSingleB`) – Tähendab *"True to Top"*. Sellel on üks tõeväärtuseline sisend. Kui see sisend saab tõeseks, siis on tipu väärtuseks *TOP* ja ründaja märkab seda. Seda tippu kasutatakse, kui on vaja graafi sisse otse kirja panna, et seal teatud asjad juhtuda ei saa.

29. *IsOK*(`PortText`) – Tipp, mis väljastab *true*, kui sisend pole $\bot$, muidu $false$.

30. *IsEq*(`PortCompare`) – Tagastatakse *true*, kui porti saabuvad servade väärtused on võrdsed ja erinevuse korral $false$.

31. *IsNeq*(`PortCompare`) – Tagastatakse $false$, kui porti saabuvad servade väärtused on on võrdsed ja erinevuse korral *true*.

32. *LongOr*(`PortSingleB`) – Sama mis *Or*, kuid sellele sobib lõputu arv sisendservi. Kasutatakse lõpmatutel graafidel, kus replitseeritud tippudest tulevate sisendite arv lõpmatu.

33. *LongAnd*(`PortSingleB`) – On analoogne tipuga *LongOr*, kuid vahe on vaid selles et toimib kui *And*.

34. *ShortMux*(`PortMUX`) – Valib sissetulevate väärtuste $x_1, x_2, \ldots, x_n$ seast ühe välja, mis ei ole $\bot$, ja väljastab selle. Kui kõik väärtused on sisendiga $\bot$, siis väljastab $\bot$. Kui mitu tükki ei ole $\bot$, siis on see märk ebakonsistentsusest ning tipp lõpetab arvutused kogu graafi peal. Ründaja saab teada, et jõuti ebakonsistentsuseks loetavasse olukorda.

35. *LongMux*(`PortText`, `PortSingleB`) – Sama mis `ShortMux`, kuid selle sissetulevate servade arv on lõputu.

36. *ShortNand*(`PortNand`) – On kombinatsioon tippudest *And* ja *TTT*. Selle abil saab väljendada, et mingist hulgast asjadest ei saa kõik korraga tõesed olla.

37. *LongNand*(`PortSingleB`) – On kombinatsioon tippudest *LongAnd* ja *TTT*. Selle tipu mõte on see, et kõigist temasse sissetulevatest servadest saab ülimalt üks tõene olla (kui on rohkem, siis on see märk ebakonsistentsusest).

Tipud operatsioonidega *Long<Op>* tõmbavad koordinaate kokku. Graafi esituses on selline tipp ühe sisendiga, lõpmatus graafis on tema vasteks *<Op>*.

Lisa 2
Teisenduste keele veateated

Veateated, mida lisatud teisenduste keel genereerida võib on järgmised:

1. *Error: a Begin-/End-/Req-/Recieve-/Send-node was chosen* – Valitakse vähemalt üks tipp nimetatute seas.
2. *Undefined variable: <muutuja>* – Kasutatakse defineerimata muutujat.
3. *Invalid syntax: <avaldis>* – Avaldises *<avaldis>* on süntaksi viga (nt `Ss+e**`).
4. *Error: in node <tipp>, the indices of the elements of the array <indeksid> are non-positive* – Indeksite seas leidub mittenaturaalarvulisi väärtusi.
5. *Error: Division by zero: <avaldis>* – Proovitakse jagada nulliga.
6. *Error, invalid node expression: <vale tipp>"* – Pole ei tipumuutuja ega tipp (nt *And* või *A* asemel *Annnd*).
7. *Error, superfluous parameters: <tipp>* – Parameetrita tipul on pandud mingi parameeter.
8. *Error, in node constant <n>: the number of parameters has to be one* – *Tuple*- või *Constant*-tipu parameetrite arv ei ole 1.
9. *Error, parameters must be positive:* – Leidub mittenaturaalarvulisi parameetreid
10. *Error, invalid parameters in Proj . . .* – *Proj*-tipul on mittesobivad parameetrid. Peab olema kaks naturaalarvulist, neist esimene mittesuurem kui teine.
11. *Cannot find any value* – Rakendus ei suutnud arvumuutujatele väärtust leida automaatse väärtustamise (alamjaotis 2.3.9) korral.
12. *Invalid variable: <muutuja> in loop <muutuja> 2 4 (..)* – Tsüklilauses on ebaõige muutujanimi.
13. *Invalid expression: <avaldis> in loop i <avaldis> 4 (A[i]=RS)* – Tsüklilauses on ebaõige avaldis.
14. *Invalid declaration: <deklaratsioon>* – Deklaratsioonis on viga.

15. *Error, the body of the loop is not between parentheses <subSources>* – tsüklilause alamavaldis või -deklaratsioon ei ole sulgude vahel

16. *"Invalid dependency <Lõpptipp>(..;«vale algtipp>";..) "* – Algtipu muutuja ei ole õige.

17. *Error, some indices of the node <tipumuutuja> are non-positive* – Tipumassiivi elemendil leidub mittenaturaalarvulisi indekseid.

18. *Node <V(...)> used several times* – lõpptipu avaldise korduv kasutamine *DGFR*'i piires (nt `V(A);...;V(B)`).

19. *Error: The edge <algtipu muutuja>:<op> – <port> –> <lõpptipu muutuja>:<op> has wrong port type* – Sobimatu port. Bitijada serv on tõeväärtuse pordis või vastupidi.

20. *Error: In the left/right DGFR, the node V=InputB/InputS cannot be the destination of some edge* – Serva lõpus leidub sisendtippe.

21. *Error: In the left/right DGFR, the node V=OutputB/OutputS cannot be the source of some edge* – Serva lõpus leidub sisendtippe.

22. *Error: some nodes in real and left DGFRs could not be unified* – Muutuja ja reaalsete servade kokkuviimise käigus ilmneb et tippe omavahel kokku viia ei saa.

23. *Error: no path with given coordinate mapping <kujutus> found from node <algtipp> to node <lõpptipp>* – Tulevate servade kujutuste ja tippude dimensioonide määramine on võimatu.

24. *Error: Cannot find file "<fail>". Set corrrect file name in GrbTrReadModule.ml "let codeFile = <file name>"* – Veateade, kui faili ei leitud.

25. *Error: Place content like "..{<declarations>}... {<left DGFR>}...{<right DGFR>}..." i.e. brackets "..{..}..{..}..{..}.."* – Koodifail ei sisalda täpselt kolme paari looksulge koos nende vahel olevate deklaratsiooni ja mõlema *DGFR*'iga.

Lisa 3
CD-ROM täiendatud tarkvaraga

CD-ROM sisaldab krüptoanalüsaatori nii varasemat (*grb090210.tar.gz*) kui ka uut versiooni (*cryptoAnalyser.zip*) koos sinna lisatud teisenduste keelega. Lisaks on *CD-ROM*'is ka käesoleva töö fail *PDF*-formaadis, peatükis 4 kirjeldatud *DGR*-pildid iga teisenduse järel ning muu vajalik tarkvara *Windows*-operatsioonisüsteemis installeerimiseks.