Tampereen yliopisto

Aleksi Vesterinen

OHJELMIEN EKVIVALENTTIUDEN SELVITTÄMINEN

Informaatioteknologian ja viestinnän tiedekunta
Kandidaattitutkielma
Huhtikuu 2021

TIIVISTELMÄ

Aleksi Vesterinen: Ohjelmien ekvivalenttiuden selvittäminen
Kandidaattitutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Huhtikuu 2021

Ohjelmien ekvivalenttiuden selvittäminen on paljon tutkittu aihe sen monista sovellusalueista johtuen. Sitä voidaan hyödyntää ohjelmien testaamisessa, refaktoroinnissa, verifioinnissa ja kääntäjien tekemien optimointien tarkastamisessa. Ongelma on todistettavasti ratkeamaton, eli yleispätevää ohjelmien ekvivalenttiuden selvittävää algoritmia ei voi olla. Tästä huolimatta monet tutkimusyhteisöt ovat esittäneet tutkimaansa rajalliseen sovellusalueeseen tarkoitettuja menetelmiä ohjelmien ekvivalenttiuden selvittämiseen, ja kehittäneet niihin perustuvia automaattisia työkaluja. Tämä tutkielma on kirjallisuuskatsaus, jossa tarkastellaan näitä menetelmiä. Kaikkia mahdollisia ohjelmien ekvivalenttiuden selvittämisen menetelmiä ei tässä työssä voida esittää. Katsaus on rajattu siten, että tarkasteltavaksi valittiin vain sellaisia menetelmiä, joista on useampia julkaisuja eri tutkijaryhmiltä. Valitut menetelmät ovat bisimilaarisuus, tulo-ohjelmat ja mallintarkastus. Menetelmistä esitetään niiden teoria, toteutus algoritmisesti sekä niiden vahvuudet ja heikkoudet.

Ohjelmien ekvivalenttiuden selvittämisen ongelmasta, sen sovellusalueista ja ekvivalenttiuden selvittämisen menetelmien arvioinnista puhutaan johdantoluvussa. Valittujen menetelmien käsittelyä ennen työssä esitellään ensin vaaditut pohjatiedot ohjelmien ekvivalenttiuden tarkastelemiseen. Ensin kerrotaan, miten ohjelmien toimintaa voidaan kuvata formaalisti, minkä jälkeen määritellään, mitä ohjelmien ekvivalenttiudella tarkoitetaan. Tämän jälkeen esitellään valitut ekvivalenttiuden selvittämisen menetelmät. Yksittäisiä menetelmiä käsittelevissä luvuissa esitellään ensin teoria niiden toiminnalle, minkä jälkeen esitetään algoritminen toteutus ohjelmien ekvivalenttiuden selvittämiseen kyseisellä menetelmällä. Lopuksi kerrotaan menetelmän heikkouksista ja menetelmään liittyvästä julkaistusta kirjallisuudesta.

Työssä nähdään, miten nykyiset menetelmät eroavat toiminnaltaan, ja miten ne soveltuvat eri käyttötarkoituksiin. Menetelmistä selviää, mitä ongelmia niissä on vielä ratkottavana, ja millaista uutta tutkimusta niistä mahdollisesti julkaistaan.

Avainsanat: kirjallisuuskatsaus, kandidaattitutkielma, ohjelmien ekvivalenttius
Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

Sisällysluettelo

1 Johdanto

Ohjelmien ekvivalenttiuden selvittäminen on hyvin tunnettu ongelma tietojenkäsittelytieteissä, ja siihen on esitetty erilaisia menetelmiä. Kiinnostusta aiheeseen on lisännyt sen sovellettavuus eri alueisiin. Kääntäjillä, jotka tekevät muutoksia ohjelmiin optimoimalla niitä, olisi hyvä olla takuu siitä, että nämä muutokset eivät vaikuta ohjelman toimintaan ei-halutulla tavalla (Necula, 2000). Toinen sovellusalue on ohjelmien testaaminen. Yleinen tapa testata ohjelmia on arvata, minkälaisilla syötteillä mahdolliset ohjelmassa olevat virheet tulisivat ilmi, ja tehdä näiden perusteella joukko testejä ohjelmalle. Tapauksissa, joissa tunnetaan jo jokin ratkaisu, voidaan muita toteutuksia testata tarkistamalla ovatko ne ekvivalentteja alkuperäisen ohjelman kanssa. Esimerkki tapauksesta, jossa tällaisella menetelmällä voidaan hyötyä, on opiskelijoiden ohjelmointiharjoitusten tarkistaminen (Clune et al., 2020). Ohjelmien välistä ekvivalenttiutta voidaan käyttää myös formaalissa verifioinnissa. Relationaalisessa verifioinnissa ohjelmien ekvivalenttiudella on keskeinen rooli, sillä ohjelmien ekvivalenttius on erittäin tärkeä relationaalinen ominaisuus.

Tässä työssä tarkastellaan, mitä menetelmiä ohjelmien ekvivalenttiuden selvittämiseen on sekä näiden menetelmien vahvuuksia ja heikkouksia. Ohjelmien ekvivalenttiuden selvittämistä tutkiessa on huomioitava, että mitään yleispätevää ratkaisua kahden ohjelman ekvivalenttiuden selvittämiseen ei ole. Ohjelmien ekvivalenttiuden tarkistaminen Turing-täydellisissä kielissä on ekvivalentti Turingin koneen totaalisuuden (eli pysähtyykö se kaikilla syötteillä) selvittämisen kanssa. Tämä on ratkeamaton ongelma, joten on mahdotonta löytää algoritmi, joka voi selvittää minkä tahansa kahden ohjelman ekvivalenttiuden (Park et al., 2020).

Ohjelmien ekvivalenttiutta selvittävät menetelmät eivät siis voi olla yleispäteviä, mutta niiden luotettavuudesta voidaan olla varmoja. Oikealla menetelmällä saadaan tulos, että kaksi ohjelmaa ovat ekvivalentit, vain jos ne oikeasti ovat. Lisäksi menetelmät usein keskittyvät niihin liittyvien sovellusalueiden erikoistapauksiin. Esimerkiksi kääntäjän tekemällä silmukoiden optimoinnilla saadun ohjelman vertaamisessa alkuperäiseen ohjelmaan optimoinnin korrektiuden varmistamiseksi voidaan keskittyä silmukoiden ekvivalenttiuden osoittamiseen. Menetelmiä voidaan arvioida sen perusteella, kuinka laajalla joukolla ohjelmia ne tuottavat oikean tuloksen, ja kuinka hyvin tämä ohjelmajoukko liittyy haluttuun sovellusalueeseen.

Jotkin menetelmät soveltuvat paremmin tietyille ohjelmointiparadigmoille. Esimerkiksi Gordonin (1995) esittämä bisimulaatioon perustuva menetelmä soveltuu ainoastaan funktionaalisille ohjelmointikielille, joilla ei ole sivuvaikutuksia. Menetelmät eroavat

myös siltä kannalta, kuinka hyvin ne soveltuvat algoritmiseen toteutukseen, ja kuinka tehokkaita nämä toteutukset ovat.

Luvussa 2 esitellään kirjallisuuskatsauksen tulokset. Luvuissa 3 ja 4 esitellään ohjelmien ekvivalenttiuden käsittelemiseen vaadittua pohjatietoa. Ekvivalenttiuden selvittämisen menetelmiä esitellään luvuissa 5, 6 ja 7.

2 Tarkasteltavien menetelmien valinta

Työn tarkoituksena on esitellä, millaista tutkimusta ohjelmien ekvivalenttiuden selvittämisestä on. Monet eri aiheista kiinnostuneet tutkimusyhteisöt ovat tutkineet ja julkaisseet ohjelmien ekvivalenttiuden selvittämisestä. Kuten Strichman (2018) huomauttaa, nämä tutkimusyhteisöt eivät kuitenkaan ole tehneet paljoa yhteistyötä keskenään. Tämä on johtanut siihen, että he ovat itsenäisesti kehittäneet samankaltaisia käsitteitä ja menetelmiä, jotka sopivat juuri heidän tutkimusaiheensa tarkoituksiin. Jotkin näistä käsitteistä vastaavat toisiaan, kuten formaaliverifioinnissa usein käytetyt Kripke-rakenteet ja siirtymäjärjestelmät. Kripke-rakenteista ja siirtymäjärjestelmistä kerrotaan luvuissa 3 ja 7. Tutkimuksen hajanaisuuteen vaikuttaa myös ekvivalenttiuden ratkeamattomuuden ongelma. Yleispätevää menetelmää ekvivalenttiuden selvittämiseen ei voi olla, joten ratkaisujen täytyy olla rajallisia ja tiettyyn tarkoitukseen suunnattuja. Tietty ekvivalenttiuden selvittämismenetelmä voi olla varsin hyödyllinen tietyllä sovellusalueella sopimatta toiselle sovellusalueelle lainkaan. Kaikki tässä työssä esitetyt menetelmät toimivatkin vain tietyissä tapauksissa ja tiettyyn tarkoitukseen.

Käsiteltäviksi menetelmiksi valittiin bisimulaatio, tulo-ohjelmat, ja mallintarkastus. Kaikista näistä menetelmistä löytyy useita artikkeleita. Edellä mainituista syistä johtuen yksittäistä parasta menetelmää ei voida valita. Menetelmiin liittyvissä luvuissa esitellään, mihin ne soveltuvat. Työssä ei ole esitetty kaikkia mahdollisia ekvivalenttiuden selvittämisen menetelmiä, vaan työ rajattiin enemmän tutkittuihin menetelmiin, joista on useampia julkaisuja eri tutkijaryhmiltä. Esimerkiksi Clunen et al. (2020) julkaisema menetelmä eroaa muista, mutta siitä ei ole vielä ollut enempää tutkimusta.

3 Ohjelmien toiminta

Jotta voidaan tarkastella ohjelmien ekvivalenttiutta, on oltava käsitys ohjelmien toiminnasta. Tämän tarjoaa semantiikka. Lyhyesti kuvattuna ohjelmointikielillä, kuten luonnollisilla kielillä, on syntaksi ja semantiikka. Syntaksilla tarkoitetaan sitä, miten symboleita voidaan sallitusti yhdistää ohjelmien (luonnollisissa kielillä lauseiden) luomiseen (Slonneger & Kurtz, 1995). Luonnollisilla kielillä semantiikka antaa syntaktisesti korrektien

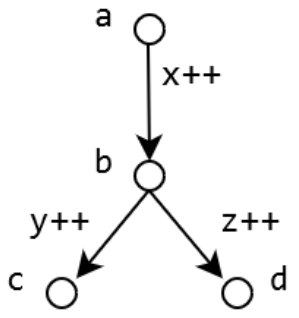
lauseiden merkityksen, kun taas ohjelmointikielillä semantiikka kertoo, miten syntaktisesti korrekti ohjelma toimii, kun se suoritetaan (Slonneger & Kurtz, 1995).

Ohjelmointikielien semantiikan määrittelemisen tapoja on monenlaisia. Jotkin semantiikan määrittelytavat helpottavat tiettyjä ekvivalenttiuden selvittämismenetelmiä. Esimerkiksi denotationaalisessa semantiikassa kaksi ohjelmaa, joilla on sama *denotaatio* (denotation) ovat ekvivalentteja (Koutavas & Wand, 2006). Yleisin tapa kuitenkin on operationaalinen semantiikka, joka esiintyy kaikissa tässä työssä esitellyissä menetelmissä.

On huomioitavaa, että kaikki ekvivalenttiuden tyypit eivät vaadi tietoa ohjelman semantiikasta. Yksi tällainen ekvivalenttius on ohjelmien eroaminen vain muuttujan nimien suhteen, eli α-ekvivalenssi, joka voidaan selvittää pelkästä ohjelman syntaksista.

Ohjelmien ekvivalenttiutta tutkittaessa kuvataan usein ohjelmien toimintaa käyttämällä *operationaalista semantiikkaa* (operational semantics). Operationaalisen semantiikan tarkoituksena on kuvata sen lisäksi, mitä ohjelma tekee, miten sen suoriutuminen etenee. Tämä eteneminen kuvataan määrittämällä ensin joukko tiloja, joista yksi toimii aloitustilana. Näiden tilojen välille määritellään siirtymärelaatio, joka liittää tiloja toisiinsa. Ohjelman suorittaminen on loppu, kun on siirrytty tilaan, jolle siirtymärelaatio ei ole määritetty, eli ohjelman lopputilaan (Slonneger & Kurtz, 1995). Nämä tilat voidaan määrittää jonkin abstraktin koneen avulla. Siirtymärelaatio määritellään kuvaamalla, miten yhdestä koneen tilasta siirrytään toiseen. Tätä tilojen joukkoa ja siirtymärelaatiota kutsutaan *siirtymäjärjestelmäksi* (transition system). Kuvassa 1 on esitetty ohjelma ja sen siirtymäjärjestelmä suunnattuna graafina.

```
P: x++; if (*) { y++; } else { z++; }
```



Kuva 1. Ohjelma ja sen siirtymäjärjestelmä suunnattuna graafina.

Rakenteellisessa operationaalisessa semantiikassa (structural operational semantics) ohjelmointikielen osien merkitys annetaan loogisina päättelysääntöinä, jotka määrittävät siirtymärelaation. Tiloina voivat toimia ohjelmointikielen lausekkeesta ja kontekstista

koostuvat parit siten, että konteksti sisältää suorittamiseen vaaditut tiedot, kuten muuttujien nimet (Slonneger & Kurtz, 1995).

Koska rakenteellisessa operationaalisessa semantiikassa semanttiset määritelmät annetaan loogisina sääntöinä, ohjelmien ominaisuuksia voidaan todistaa suoraan kielen rakenneosista (Slonneger & Kurtz, 1995). Tämä on yleinen tapa kuvata ohjelmien toimintaa ekvivalenttiutta käsittelevissä artikkeleissa.

4 Ohjelmien ekvivalenttius

Ohjelmien ekvivalenttiudesta keskusteltaessa on tärkeää määritellä, minkä tyyppisestä ekvivalenttiudesta on kyse. Valitun ekvivalenssirelaation pitää tarkasti kuvata ohjelmien välinen suhde niissä konkreettisesti esiintyvien asioiden, kuten syntaksin, tilojen, tai syötteiden ja tulosteiden avulla, sillä abstraktimmat käsitteet, kuten ohjelmien toteuttamat algoritmit, eivät sovi ekvivalenssirelaation määrittämiseen (Blass et al., 2009).

Tässä työssä keskitytään ekstensionaaliseen ekvivalenssiin, sillä se vastaa hyvin ohjelmien ekvivalenttiuden intuitiivista tulkintaa. Kun tässä työssä viitataan ekvivalenttiuteen ilman tarkempaa määritystä ekvivalenttiuden tyypistä, tarkoitetaan ekstensionaalista ekvivalenssia. Eri lähteissä saatetaan käsitellä eri ekvivalenttiuden tyyppejä. Joskus eri artikkeleissa saatetaan viitata erilaisiin ekvivalenttiuden tyyppeihin samalla nimellä, joten lukiessa on syytä kiinnittää huomiota siihen, minkälaisesta ekvivalenttiudesta on kyse. Seuraavaksi esitellään ekstensionaalinen ekvivalenssi.

Ekstensionaalisella ekvivalenssilla tarkoitetaan sitä, että kaksi ohjelmaa ovat ekvivalentit, jos ne tuottavat kaikilla syötteillä saman tuloksen (Slonneger & Kurtz, 1995). Mitä syötteellä ja tuloksella tarkoitetaan, täytyy määrittää kahdelle ohjelmalle yhteisen semantiikan kautta. Tämä onnistuu samalla ohjelmointikielellä kirjoitetuilla ohjelmilla suoraan ohjelmointikielen semantiikasta, mutta eri kielillä kirjoitetuilla ohjelmilla täytyisi esimerkiksi määrittää ensin yhden ohjelmointikielen semantiikka toisen ohjelmointikielen kielen kautta translationaalisen semantiikan tavoin.

Operationaalisen semantiikan avulla syöte ja tulos voidaan määrittää siten, että abstraktin koneen tiloissa on säiliö syötteelle ja tulokselle. Näin ollen ekstensionaalisesti ekvivalenteilla ohjelmilla täytyy olla lopputilassa sama tulossäiliö, jos niillä on alkutilassa sama syötesäiliö. Churchill et al. (2019) määrittelevät ekvivalenssin tällä tavoin oikeilla x86-64-tietokoneilla suoritettaville C-kielen ohjelmille koneen rekisterien ja keon arvojen suhteen. Ekstensionaalinen ekvivalenssi on ekvivalenssin käsitteenä riittävän laaja, että se sisältää muut yleisesti käytetyt ekvivalenttiuden tyypit. Jos kahden ohjelman välillä on jonkin tyypin ekvivalenssi, pätee niiden välillä todennäköisesti ekstensionaalinen ekvivalenssi. Ekstensionaalista ekvivalenssia on kutsuttu myös η-ekvivalenssiksi.

5 Bisimilaarisuus

Bisimilaarisuuden osoittaminen on yksi tutkituimmista tavoista osoittaa ohjelmien ekvivalenttius. Bisimilaarisuus ekvivalenttiuden osoittamisen menetelmänä esitellään kahdessa osassa. Luvussa 5.1 määritellään bisimilaarisuus ja luvussa 5.2 kuvataan, mitä menetelmiä on bisimilaarisuuden osoittamiseen algoritmisesti.

5.1 Bisimilaarisuuden teoria

Bisimilaarisuus (bisimilarity) on ohjelmien toimintaan perustuva relaatio. Kaksi ohjelmaa voidaan osoittaa ekvivalenteiksi osoittamalla, että niiden välillä pätee tämä relaatio (Pitts, 1997). Gordon (1995) määrittelee bisimulaation relaationa ohjelmien operationaalisesta semantiikasta saatujen siirtymäjärjestelmien välisenä relaationa. Hänen mallissaan siirtymäjärjestelmän tiloina toimii kyseisellä ohjelmointikielellä kirjoitetut ohjelmat. Alkutila on ohjelma itse, ja siirtymärelaatio liittää tilat siten, että saatu seuraava tila on ohjelma yhden reduktioaskeleen jälkeen. Malli toimii sivuvaikutuksettomille funktionaalisille kielille. Nyt yksittäiselle ohjelmalle voidaan sen alkutilasta lähtien muodostaa *jäsennyspuu* (derivation tree), jonka kaaret ovat nimettyjä siirtymiä ja solmut ohjelmia. Siirtymät nimetään niihin liittyvän toiminnan perusteella. Tämän jälkeen voidaan määrittää seuraavanlainen relaatio ohjelmien välillä:

$$Kun\ a\ ja\ b\ ovat\ ohjelmia\ ja\ \alpha\ siirtymä, niin\ a \sim b\ joss$$

$$(1)\ a \xrightarrow{\alpha} a' \Rightarrow \exists b' \left(b \xrightarrow{\alpha} b' \wedge a' \sim b' \right)$$

$$(2)\ b \xrightarrow{\alpha} b' \Rightarrow \exists a' \left(a \xrightarrow{\alpha} a' \wedge a' \sim b' \right)$$

Bisimilaarisuus on suurin tällainen relaatio. Tämä relaatio liittää toisiinsa ohjelmat, jotka tuottavat isomorfisesti samat puut. Solmuissa olevat ohjelmat voivat syntaktisesti erota toisistaan (Gordon 1995).

Bisimilaarisuus eli suurin edellä annetut ominaisuudet toteuttava relaatio voidaan määritellä seuraavien funktioiden avulla:

$$[S] = \left\{ (a,b) |\ a \xrightarrow{\alpha} a' \Rightarrow \exists b' \left(b \xrightarrow{\alpha} b' \wedge a' \sim b' \right) \right\}$$

$$\langle S \rangle = [S] \cap [S^{-1}]^{-1}$$

Jos jokin relaatio S on funktion $\langle - \rangle$ kiintopiste, se toteuttaa edellä annetut ominaisuudet. Koska $\langle - \rangle$ on monotoninen, niin Knasterin-Tarskin lauseen mukaan sillä on olemassa suurin kiintopiste, joka on suurin relaatio, joka toteuttaa edellä annetut ominaisuudet (Gordon 1995).

Bisimulaation määrittely sivuvaikutuksellisille kielille vaatii ohjelmassa esiintyvien arvojen säiliön huomioimisen. Koutavas ja Wand (2006) esittelevät tällaisen bisimulaation sivuvaikutuksellisille kielille.

5.2 Bisimilaarisuuden osoittaminen algoritmisesti

Perinteiset menetelmät bisimilaarisuuden osoittamiseen äärellisille siirtymäjärjestelmille algoritmisesti perustuvat *karkeimman osituksen ongelman* (relational coarsest partitioning problem) ratkaisemiseen (Aceto et al., 2011). Ekvivalenssirelaatio tilojen joukolle Pr voidaan esittää joukon Pr osituksena $\{B_0, \dots, B_k\}, k \geq 0$ siten, että

$$(1) B_i \cap B_j = \emptyset \; kaikilla \; 0 \leq i < j \leq k$$

$$(2) \; Pr = B_1 \cup B_2 \cup \cdots \cup B_{k-1} \cup B_k$$

Joukot B_i ovat kyseisen osituksen määrittämän ekvivalenssirelaation ekvivalenssiluokkia, ja niitä kutsutaan usein lohkoiksi. Kun π ja π' ovat joukon Pr osituksia, joukkoa π' sanotaan joukon π *tihennykseksi* (refinement), jos jokaiselle lohkolle $B' \in \pi'$ on olemassa lohko $B \in \pi$ siten että $B' \subseteq B$. Kanellakis-Smolka sekä Paige-Tarjan algoritmit aloittavat jostakin osituksesta π_{alku} ja laskevat tihennyksiä tälle ositukselle lähestyen bisimilaarisuutta (Aceto et al., 2011). Tällä tavoin toimivia algoritmeja kutsutaan *ositustihennysalgoritmeiksi* (partition refinement algorithms). Seuraavaksi esitellään Kanellakis-Smolka algoritmi esimerkkinä tämän tyyppisistä algoritmeista.

Kanellakis-Smolka algoritmi suorittaa tihennyksen *halkaisijoiden* (splitter) perusteella. Osituksen π lohko $B_j \in \pi$ on halkaisija lohkolle $B_i \in \pi$, jos joillain lohkon B_i tiloilla on jokin siirtymä α johonkin tilaan lohkossa B_j ja joillain ei. Algoritmi halkaisee lohkon B_i seuraavasti, kun lohkolla B_i on siirtymällä α halkaisija B_j.

$$B_i^1 = \left\{ s | s \in B_i \wedge \exists s' \in B_j \left(s \xrightarrow{\alpha} s' \right) \right\}$$

$$B_i^2 = B_i \setminus B_i^1$$

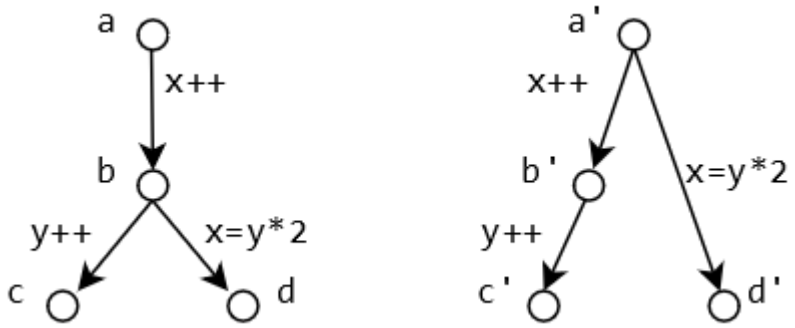
Halkaiseminen antaa uuden osituksen $\pi' = \{B_1, \dots, B_i^1, B_i^2, \dots, B_k\}$. Algoritmi iteroi tällä tavoin joukkojen halkaisemista, kunnes saatua ositusta ei voi enää tihentää. Tätä tuloksena saatua ositusta kutsutaan *karkeimmaksi vakaaksi osituksekseksi* (coarsest stable partition). Tämän osituksen lohkoina ovat syötteenä annetun siirtymäjärjestelmän bisimilaarisuuden ekvivalenssiluokat, kun alkuositukseksi valitaan $\pi_{alku} = \{Pr\}$ (Aceto et al., 2011). Jos tutkittavat ohjelmat ovat a ja b, ja ne ovat algoritmin antamassa osituksessa samassa lohkossa, ohjelmat ovat ekvivalentteja.

Ositustihennysalgoritmit vaativat, että niille annetaan syötteenä valmiiksi rakennettu siirtymäjärjestelmä (Aceto et al., 2011). Tämä vaatii, että annettu siirtymäjärjestelmä on äärellinen, eli sillä on äärellinen määrä solmuja ja siirtymiä. Jotkin siirtymäjärjestelmät voivat kuitenkin olla äärettömiä (Gordon, 1995). Joissain tapauksissa ääretön siirtymäjärjestelmä on bisimilaarinen äärellisen kanssa ja voidaan minimoida äärelliseksi, mitä on ehdotettu mahdolliseksi ratkaisuksi ongelmaan (Aceto et al., 2011).

Bisimulaatiota on tutkittu myös epädeterministisille siirtymäjärjestelmille. Groote, Rivera Verduzco ja de Vink (2018) määrittelevät artikkelissaan *probabilistisen siitymäjärjestelmän* (probabilistic labeled transition system) sekä *probabilistisen bisimulaation*

(probabilistic bisimulation) käsitteet ja antavat ositustihennysalgoritmin tämän kaltaisen bisimulaation löytämiseen tämän kaltaisille siirtymäjärjestelmille.

```
P: while(true) { x++; if (*) { y++; } else { x=y*2; } }
P': while(true) { if (*) { x++; y++;} else { x=y*2; } }
```



Kuva 2. Kaksi ekvivalenttia ohjelmaa, jotka eivät ole bisimilaarisia (Park et al., 2020)

Klassiset bisimulaation muodot rajaavat ekvivalenttiuden usein liian tiukasti ollakseen hyödyllisiä ekvivalenttiuden osoittamiseen ohjelmien välillä. Etenkin koodikääntäjien tekemät optimoinnit voivat muuttaa ohjelman haarautumisrakennetta, mikä johtaa klassisen bisimilaarisuuden häviämiseen (Park et al., 2020). Kuvassa 2 näkyy esimerkki tällaisesta tapauksesta. Kuvan siirtymäjärjestelmät vastaavat ohjelmien silmukoiden sisältöä. Park et al. (2020) esittävät artikkelissaan ratkaisuksi ongelmaan leikkausbisimulaation käsitteen, jossa bisimulaatio yhdistetään muiden ekvivalenttiuden osoittamisen menetelmien kanssa.

6 Tulo-ohjelmat

Tulo-ohjelman rakentaminen ja sille pätevien invarianttien osoittaminen on uudempi menetelmä ekvivalenttiuden osoittamiseen. Luvussa 6.1 esitellään, mitä tulo-ohjelmat ovat, ja luvussa 6.2 kuvataan ohjelmien ekvivalenttiuden osoittamiseen tulo-ohjelmia hyödyntävien järjestelmien toimintaa.

6.1 Tulo-ohjelmien teoria

Tulo-ohjelma (product program) on ohjelma, joka suorittaa kahden ohjelman suoritusaskeleita synkronisesti. Tulo-ohjelmilla voidaan osoittaa ominaisuuksia tietylle ohjelmalle tai eri ohjelmien välisiä ominaisuuksia. Esimerkkinä ohjelman ominaisuudesta voidaan mainita deterministisyys, kun taas ekvivalenttius on ohjelmien välinen ominaisuus. Ekvivalenttiuden selvittämisessä tällä menetelmällä luodaan ensin tulo-ohjelma kahdelle oh-

jelmalle asettamalla niiden semanttisesti, ja joissain yksinkertaisissa tapauksissa syntaktisesti, toisiaan vastaavat osat linjaan. Tämän jälkeen riittää osoittaa tälle tulo-ohjelmalle invariantti, joka takaa ohjelmien ekvivalenttiuden (Barthe et al., 2011).

Barthe et al. (2011) antavat kuvassa 3 näkyvän esimerkin yksinkertaisesta tulo-ohjelmasta.

```
Ensimmäinen ohjelma:    Toinen ohjelma:        Tulo-ohjelma:
i:=0;                   j:=1;                  i:=0; x+=i; i++; j:=1;
while (i≤N) do          while (j≤N) do         while (i≤N) do
  x+=i;                   y+=j;                  y+=j; j++;
  i++;                    j++;                   x+=i; i++;
```

Kuva 3. Kaksi ohjelmaa ja niiden tulo-ohjelma (Barthe et al., 2011).

Ensimmäisen ohjelman ensimmäinen iteraatio on tulo-ohjelmassa siirretty silmukan ulkopuolelle. Ohjelmien ekvivalenttiuden osoittamiseen tässä esimerkissä riittää osoittaa, että tulo-ohjelmalla pätee silmukassa invariantti $i = j \wedge x = y$ (Barthe et al., 2011).

6.2 Ekvivalenttius algoritmisesti tulo-ohjelmilla

Jotta tulo-ohjelmia hyödyntämällä voidaan algoritmisesti tarkistaa ohjelmien ekvivalenttius, on voitava ensin algoritmisesti rakentaa tulo-ohjelma, sekä valita ja todistaa sopiva invariantti. Tulo-ohjelmaa rakennettaessa on löydettävä ohjelmien toisiaan vastaavat osat, mikä voidaan tehdä suoraan niiden syntaktisen rakenteen perusteella yksinkertaisissa tapauksissa, kuten tapauksissa, joissa kahden ohjelman silmukat tekevät saman määrän iteraatioita. Tämä menetelmä ei kuitenkaan toimi monissa tapauksissa, joissa syntaktinen rakenne eroaa, kuten kääntäjien tekemissä silmukoiden optimoinneissa. Tällaisia optimointeja ovat *silmukan iteraatioiden purkaminen* (loop unrolling) ja *vektorisaatio* (vectorization) (Churchill et al., 2019). Churchill et al. (2019) esittävät menetelmän tulo-ohjelman rakentamiseen ennustamalla testisyötteiden avulla, mitkä testattavien ohjelmien osat vastaavat semanttisesti toisiaan.

Tässä menetelmässä ohjelmille tehdään ensin testitapauksia, jonka jälkeen testit suoritetaan ohjelmille, mistä saadaan ohjelmille suoritusjäljet ja luodaan *suoritusjälkien tasaus* (trace alignment). Suoritusjälkien tasaus on joukko koneen tiloja, jotka liitetään toisiinsa. Tämän perusteella luodaan tulo-ohjelma, jonka jälkeen haetaan ekvivalenttiuden osoittamiseen sopivat invariantit. Suoritusjälkien tasaus tehdään valitsemalla ensin *tasauspredikaatti* (alignment predicate). Tasauspredikaatti määrittää, mitkä koneen tilat tulee liittää toisiinsa. Churchillin et al. (2019) esittämässä järjestelmässä käyttäjien antamien testitapauksien perusteella järjestelmä luo joukon kandidaattitasauspredikaatteja,

joista jokaiselle yritetään luoda *ohjelmatasausautomaatti* (program alignment automaton), joka määrittää tulo-ohjelman. Järjestelmä koittaa löytää näille ohjelmatasausautomaateille sopivat invariantit. Jos jokin saatu ohjelmatasausautomaatti toteuttaa tietyt ominaisuudet, kuten sen, että alkuperäisissä ohjelmissa ei ole polkuja, joita ohjelmatasausautomaatti ei sisällä, niin ohjelmien ekvivalenttius on osoitettu. Se, että pätevätkö nämä ominaisuudet saadulle ohjelmatasausautomaatille, voidaan tarkistaa automaattisesti SMT-ratkaisijalla. SAT- (propositional satisfiability problem) ja SMT-ratkaisijat (satisfiability modulo theories) ovat työkaluja, joilla voidaan tarkistaa propositiologiikan lauseiden toteutuvuus. SMT-ratkaisijat voivat lisäksi käsitellä suuruusvertailua, aritmetiikkaa ja kvanttoreita (de Moura & Bjørner, 2008).

Tulo-ohjelmat ohjelmien ominaisuuksien osoittamisen apuvälineenä on vielä vähemmän tunnettu menetelmä. Barthe et al. (2011) esittävät teorian ohjelmien ominaisuuksien selvittämiseen tulo-ohjelmilla, mutta eivät anna toteutusta tulo-ohjelmien automaattiselle rakentamiselle, kuten Churchill et al. (2019). Dahiya ja Bansal (2017) antavat algoritmin menetelmään, jossa luodaan tulo-ohjelman toiminnan kuvaava *yhteisten siirtymien kaavio* (joint transfer function graph). Heidän menetelmänsä vaatii, että tutkittavan ohjelman haarautumisehdot voidaan todistaa yhtäpitäviksi toisen tutkittavan ohjelman haarautumisehtojen kanssa. Churchillin et al. (2019) esittämässä menetelmässä tätä ei vaadita, mikä sallii useampien ekvivalenssien havaitsemisen, mutta heidän menetelmänsä vaatii ennalta annettuja testitapauksia tasauspredikaattien päättelemiseen. Felsing et al. (2014) taas esittävät erilaisen invariantteja hyödyntävän strategian. Heidän menetelmässään tulo-ohjelman määrittämisen ja sen invarianttien selvittämisen sijaan etsitään kahden ohjelman silmukoille yhteisiä invariantteja. Heidän menetelmänsä ei kuitenkaan toimi tapauksissa, joissa ohjelmien silmukoiden iteraatioiden määrällä ei ole suoraa suhdetta.

7 Mallintarkastus

Ohjelmien ekvivalenttiuden ongelmaa voidaan käsitellä verifiointiongelmana, jolloin ekvivalenttius todistetaan osoittamalla, että kaksi ohjelmaa toimivat ekvivalenttiuden takaavan spesifikaation mukaisesti. Tältä kannalta ohjelmien ekvivalenttiutta voidaan lähteä selvittämään *mallintarkastuksella* (model checking) (Clune et al., 2020). Esimerkkinä tällaisesta lähestymistavasta on Clarke et al. (2004) julkaisema C-ohjelmien automaattiseen verifiointiin tarkoitettu työkalu, joka voi tällä tapaa verrata C-ohjelmia. Mallintarkastuksessa tarkastettava järjestelmä mallinnetaan äärellisenä automaattina, ja spesifikaatio formalisoidaan kirjoittamalla määritellyt ominaisuudet temporaalilogiikan lauseiksi. Automaatin saavutettavat tilat käydään läpi järjestelmän ominaisuuksien tarkastamiseksi (Biere et al., 2003). Järjestelmän esitys äärellisenä automaattina voidaan tehdä *Kripke-*

rakenteena (Kripke structure). Kripke-rakenteet muistuttavat siirtymäjärjestelmiä, mutta niiden siirtymät eivät ole nimettyjä. Sen sijaan niiden tilat sisältävät temporaalilogiikalla analysoitavaa tietoa. Jokainen Kripke-rakenne voidaan kuitenkin esittää siirtymäjärjestelmänä, ja jokainen siirtymäjärjestelmä Kripke-rakenteena (Karvi, 2009).

Mallintarkastusmenetelmiä on monenlaisia. Clarke et al. (2004) julkaisemassa C-ohjelmien automaattisen verifikaation toteutuksessa käytetään *rajattua mallintarkastusta* (bounded model checking). Rajatussa mallintarkastuksessa haetaan vastaesimerkkiä spesifikaation ominaisuuksille suorituksissa, joiden siirtymien määrän rajaa jokin vakio. Vakiota kasvatetaan, kunnes virhe löytyy tai törmätään vakion ylärajaan. Vastaesimerkin haku tapahtuu luomalla siirtymärelaation perusteella temporaalilogiikan lause, jota tutkimalla voidaan määrittää, onko olemassa suorituspolku, joka rikkoo spesifikaation ominaisuuksia. Tämä lause voidaan antaa SAT-ratkaisijalle syötteeksi. Jos saatu logiikan lause toteutuu, on löydetty vastaesimerkki ja voidaan todeta, että spesifikaation määrittämät ominaisuudet eivät toteudu (Biere et al., 2003).

Suurin ongelma mallintarkastuksessa on järjestelmien tilojen eksponentiaalinen kasvu, mikä rajoittaa sen käyttöä suurille järjestelmille (Biere et al., 2003). Vaikka ohjelmien ekvivalenttiutta voidaan käsitellä formaaliverifikaation erikoistapauksena, yksi formaaliverifikaatiossa tutkittu alue on, miten verifiointijärjestelmien toimivuutta voidaan parantaa hyödyntämällä ohjelmien ekvivalenttiuden osoittamismenetelmiä (Strichman, 2018). On siis mahdollista, että ohjelmien ekvivalenttiuden ongelmaa tullaan jatkossa tutkimaan formaaliverifioinnin edistämiseksi eikä toisinpäin.

8 Yhteenveto

Tutkielmassa esiteltiin kirjallisuudessa esiintyviä ekvivalenttiuden selvittämisen menetelmiä ja niiden ominaisuuksia. Bisimilaarisuus on paljon tutkittu aihe ja yksi varhaisimmista menetelmistä ohjelmien ekvivalenttiuden osoittamiseen. Kuitenkin sen käytännöllinen soveltaminen ohjelmien ekvivalenttiutta selvittävissä järjestelmissä on ollut rajallista. Tästä huolimatta bisimilaarisuus on edelleen suosittu tutkimusaihe ja siitä ilmestyy uutta tutkimusta, kuten Parkin et al. (2020) julkaisema artikkeli. Tulo-ohjelmat ovat uudempi menetelmä, joka perustuu invarianttien löytämiseen kahden ohjelman osien välille. Menetelmä soveltuu etenkin muunnettujen silmukoiden kannalta eroavien ohjelmien ekvivalenttiuden selvittämiseen ja siten soveltuu kääntäjien tuottamien optimoitujen ohjelmien korrektiuden osoittamiseen. Mallintarkastusta formaaliverifiointiin on tutkittu paljon ja sitä voidaan soveltaa ekvivalenttiuden selvittämiseen. Tutkimusta, joka keskittyy tähän sovellusalueeseen, on kuitenkin paljon vähemmän kuin tutkimusta formaaliverifioinnista yleisesti.

Käsitellyt menetelmät havainnollistavat, kuinka eri ekvivalenttiuden selvittämismenetelmät soveltuvat eri käyttötarkoituksiin. Vaikka tiettyjä yhtenäisyyksiä menetelmien välillä voidaan havaita, niiden toimintaperiaatteet eroavat merkittävästi.

Lähdeluettelo

Slonneger, K., & Kurtz, B. L. (1995). Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach. Addison-Wesley Pub. Co.

Churchill, B., Padon, O., Sharma, R., & Aiken, A. (2019). Semantic Program Alignment for Equivalence Checking. *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation.*

Necula, G. C. (2000). Translation Validation for an Optimizing Compiler. *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation - PLDI '00*. https://doi.org/10.1145/349299.349314

Clune, J., Ramamurthy, V., Martins, R., & Acar, U. A. (2020). Program Equivalence for Assisted Grading of Functional Programs. *Proceedings of the ACM on Programming Languages*, *4*(OOPSLA), 1–29. https://doi.org/10.1145/3428239

Gordon, A. D. (1995). Bisimilarity as a Theory of Functional Programming. *Electronic Notes in Theoretical Computer Science*, 1, 232–252.

Blass, A., Dershowitz, N., & Gurevich, Y. (2009). When are Two Algorithms the Same? *The Bulletin of Symbolic Logic*, *15*(2), 145–168. https://doi.org/10.2178/bsl/1243948484

Pitts, A. (1997). Operationally-based Theories of Program Equivalence. *Semantics and Logics of Computation*, 241–298. *https://doi.org/10.1017/cbo9780511526619.007*

Barthe, G., Crespo, J., & Kunz, C. (2011). Relational Verification Using Product Programs. *International Conference on Formal Methods.*

Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., & Ulbrich, M. (2014). Automating Regression Verification. In *Automated Software Engineering* (Vol. 14, ASE '14, pp. 349-360). New York, New York: Association for Computing Machinery.

Aceto, L., Ingolfsdottir, A., & Srba, J. (2011). The Algorithmics of Bisimilarity. In D. Sangiorgi & J. Rutten (Eds.), *Advanced Topics in Bisimulation and Coinduction* (Cambridge Tracts in Theoretical Computer Science, pp. 100-172). Cambridge: Cambridge University Press. doi:10.1017/CBO9780511792588.004

Groote, J., Rivera Verduzco, H., & De Vink, E. (2018). An Efficient Algorithm to Determine Probabilistic Bisimulation. *Algorithms, 11*(9), 131. doi:10.3390/a11090131

Park, D., Rosu, G., Adve, V. S., & Kasampalis, T. (2020). *Cut-Bisimulation and Program Equivalence*. Unpublished manuscript, University of Illinois at Urbana-Champaign.

Biere, A., Cimatti, A., Clarke, E. M., Strichman, O., & Zhu, Y. (2003). Bounded Model Checking. *Advances in Computers, 58*, 118-148.

Karvi T. (2009). Introduction to Specification and Verification [Lecture Notes]. University of Helsinki

Strichman, O. (2018). Special Issue: Program Equivalence. *Formal Methods in System Design, 52*(3), 227-228. doi:10.1007/s10703-018-0318-y

Koutavas, V., & Wand, M. (2006). Small Bisimulations for Reasoning About Higher-Order Imperative Programs. *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL'06*. doi:10.1145/1111037.1111050

De Moura, L., & Bjørner, N. (2008). Z3: An Efficient SMT Solver. *Tools and Algorithms for the Construction and Analysis of Systems,* 337-340. doi:10.1007/978-3-540-78800-3_24