hyväksymispäivä arvosana

arvostelija

Ohjelmien oikeaksitodistamisen historia

Tuomo Malinen

Helsinki 15.5.2001

Tietojenkäsittelytieteen historia -seminaari

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Sisältö

1 Johdanto

Tietokoneohjelma voi toimia joko oikein tai väärin. Intuitiiviselta kannalta on selvää, mitä ohjelman toimiminen oikein tai väärin tarkoittaa. Jos yhteenlaskualgoritmi saa syötteekseen kaksi kokonaislukua, sen tulee tulostaa niiden summa, jotta se toimisi oikein – muussa tapauksessa se toimii väärin. Jos käyttöjärjestelmä kaatuu, se toimii väärin. Jos se ei kaadu, se saattaa toimia oikein. Koska on kyse matemaattisluonteisista asioista, niitä voidaan tarkastella myös muuten kuin intuition avulla, formaalisti. Ohjelman oikeellisuus voidaan siten todistaa (jos se ylipäätään toimii oikein). Tässä esitelmässä käsitellään ohjelmien muodollisten oikeellisuustodistusten historiaa.

Ohjelmien formaali oikeellisuuden tarkastelu on oleellinen osa tietojenkäsittelytiedettä. Se on, tietojenkäsittelytieteen lyhyellä aikaskaalalla, suhteellisen tuore tutkimuksen aihe. Idea tietyn ohjelman ominaisuuksien muodollisesta todistamisesta on ilmaistu ensimmäistä kertaa 1960-luvun alussa. Oikeellisuuden tutkimus seurasi käytännön tarpeesta: 1960-luvulla huomattiin, että tuotetut ohjelmistot eivät yksinkertaisesti olleet tarpeeksi luotettavia, mistä seurasi hankaluuksia monella saralla. Alan ensimmäiset tulokset ovat tosin peräisin jo 1930-luvulta: Turing esitti erään oleellisen oikeellisuustodistuksia koskevan seikan vuonna 1936 [Tur36].

Oikeellisuustodistusten historia voidaan jakaa kahteen, enemmän tai vähemmän erillään kehittyneeseen tutkimuksen haaraan. Ensimmäisenä kehittyivät sarjallisten ohjelmien oikeellisuustodistukset, jotka perustuvat ohjelmointikielen käskyjen semantiikan aksiomatisointiin. Nämä todistukset käsittelevät siis tavanomaisia ohjelmia, jotka voidaan määritellä kuvauksena syötteiden joukolta tulosteiden joukolle. Sarjalliset ohjelmat toimivat oikein ainoastaan, jos ne pysähtyvät. Toinen tutkimuksen haara on rinnakkain toimivien algoritmien oikeellisuustodistukset. Nämä algoritmit eroavat laskennan määritelmänsä (ja siten myös oikeellisuustodistustensa) suhteen edellisistä siinä, että mielenkiintoiset suoritukset ovat päättymättömiä (tai ainakin ne voidaan mallintaa päättymättömiksi).

Tarkastellaan yllämainittuja tapauksia erikseen seuraavissa luvuissa, kumpaakin kronologisessa järjestyksessä. Näkökulma pyrkii olemaan tieteenhistoriallinen siinä mielessä, että keskitytään tieteellisten saavutusten kuvaamiseen.

2 Pysähtyvät ohjelmat

Sarjallinen ohjelma saa syötteen, suorittaa laskentaa ja (mielenkiintoisessa tapauksessa) tulostaa jotain ja pysähtyy. Ohjelman tulostus on ohjelman määrittelemän funktion arvo annetulla syötteellä. Ohjelma voi ratkaista jonkin päätösongelman, matemaattisen funktion tai muun laskennallisen ongelman. Tulos voi joko olla haluttu tai ei – jos tulos on kaikilla syötteillä haluttu, sanotaan että ohjelma toimii oikein. Muussa tapauksessa ohjelma toimii väärin. Ohjelma toimii siis oikein jos se toteuttaa juuri halutun funktion. Tässä luvussa käsitellään sarjallisten ohjelmien oikeellisuustodistusten historiaa. Historiallisissa seikoissa nojataan paljolti Griesin esitykseen [Gri81], mutta myös muihin viitattuihin artikkeleihin.

2.1 Turing

Alan Turing esitti ensimmäisen sarjallisten ohjelmien oikeellisuuteen liittyvän tuloksen vuonna 1936: mielivaltaisen, universaalissa laskennan mallissa esitetyn ohjelman suorituksen päättyminen kaikilla syötteillä ei ole ratkeava ongelma [Tur36]. Tästä siis tiedetään, että yleisessä tapauksessa automaattinen ohjelman todistaminen oikeaksi tai vääräksi ei käy päinsä. Mielivaltaisen pysähtyvän ohjelman pysähtymiselle on kuitenkin todistus.

2.2 1960-luku

1960-luvulla ohjelmistojen ja tietokonejärjestelmien koot kasvoivat ja tietokoneiden sovellusalueet monipuolistuivat. Laskennan oikeellisuuden suhteen kriittisten sovellusten yleistyessä havaittiin, että siihen asti käytetyt ohjelmointikäytännöt ja työkalut eivät enää kelvanneet tarpeeksi hyvien ohjelmistojen tuottamiseen. Griesin mukaan alalla vallitsi syvä yhteisymmärrys siitä, että ohjelmistotuotanto oli kriisissä. Kriisin nähtiin johtuvan siitä, että ohjelmointia ei ymmärretty syvällisellä tasolla [Gri81]. Ohjelmointimenetelmien kehittäminen nähtiin ratkaisuna ongelmaan ja ohjelmointiin liittyvän tutkimuksen arvo tunnustettiin tiedeyhteisössä.

Yksi ensimmäisiä kirjallisia osoituksia tietoisuudesta, että ohjelmien oikeellisuus voidaan ja tulisi todistaa täsmällisesti, on seuraava virke John McCarthyn artikkelissa [McC61]: “instead of trying out computer programs on test cases until they are debugged, one should prove that they have the desired properties”. Artikkelissaan McCarthy keskittyy kuitenkin ennemminkin ohjelmien ekvivalenssitarkasteluihin; ideana oli tutkia mahdollisuutta johtaa intuitiivisesti selvistä pidemmistä ohjelmista ekvivalentteja lyhyempiä, tehokkaampia ohjelmia. Tämä vähän hedelmää tuottanut ajatus oli muutenkin yleisesti tutkittu 60-luvulla [Gri81].

Motiivin oikeellisuustutkimuksille synnytti siis käytännön tarve saada aikaan paremmin toimivia ohjelmia. Tieteellistä pohjaa ohjelmien oikeellisuuden formaaleille tarkasteluille loi lingvistiikan kehitys 50- ja 60-luvuilla. Chomskyn kontekstittomat kieliopit vuodelta 1956 sekä Backuksen ja Naurin muutamaa vuotta myöhemmin esittämä Backus-Naur -normaalimuoto olivat sopivia tapoja määritellä ohjelmointikielen syntaksi täsmällisesti. Näitä kehitysaskeleita käytettiin hyväksi kääntäjien kehityksessä, ja syntaksin täsmällinen määrittely lisäsi ymmärrystä ohjelmien toiminnasta. Syntaksin täsmällinen määrittely on myös luonnollisesti välttämätön ehto koneellisesti käännettäväksi tarkoitetun ohjelman oikeellisuustarkastelulle.

Ensimmäinen selvä askel oikeellisuustodistusten suuntaan oli Peter Naurin artikkeli vuodelta 1966, "Proofs of algorithms by general snapshots" [Nau66]. Käsite "general snapshot" tarkoitti jotain ehtoa, joka toteutuu aina ohjelman suorituksen saavuttaessa tietyn kohdan. Naur esitti esimerkin Algol-kielisen ohjelman oikeellisuustodistuksesta ja epäformaalin todistustekniikan [Gri81].

Robert Floydin artikkeli "Assigning meaning to programs" [Flo67] julkaistiin 1967. Kuten nimestä tulee selväksi, Floydin tarkoituksena oli määritellä ohjelmille täsmällinen merkitys eli semantiikka. Ohjelman semanttisen merkityksen voidaan sanoa olevan sen aiheuttamat muutokset käsittelemilleen tietorakenteille. Semantiikan määrittelyä tarvitaan oikeellisuustodistuksissa ohjelman liittämiseen formaalisti muotoiltuihin oikeellisen toiminnan vaatimuksiin. Ilman täsmällistä semantiikan määrittelyä ei ohjelman toiminnasta voida siis sanoa formaalisti mitään. Floydin idea oli siten välttämätön myöhempää kehitystä varten.

Floyd käytti ohjelman merkityksen määrittelemiseen ohjelman käskykaavion kaariin kiinnitettyjä väitteitä. Ideana oli, että kuhunkin kaareen liitetty väite olisi tosi aina kun kaarta vastaava käsky suoritetaan. Ohjelmien suorituksista voitaisiin siis todistaa mielenkiintoisia seikkoja suorituspolun väitteisiin perustuen.

Floyd esitti myös ensimmäisenä *silmukkainvariantin* (*loop invariant*) idean. Silmukkainvariantti on silmukkaan liittyvä väite, joka säilyy totena silmukan jokaisella suorituskerralla. Floydin formalismissa, ohjelman käskykaaviossa, silmukat ovat siis kaavion syklejä. Floyd liitti johonkin silmukan kaareen erityisen väitteen silmukkaa ajatellen, ja todisti, että jos silmukan suoritus alkoi tuosta kaaresta, saavuttaessa suorituksessa uudelleen samaiseen kaareen väite oli jälleen tosi.

Floyd esitti myös ajatuksen ohjelmointikielen määrittelystä todistustekniikoiden avulla. Tämä vaikutti Hoaren muutaman vuoden päästä julkaistuihin ideoihin.

2.2.1 Hoare

Tony Hoaren artikkeli "An Axiomatic Basis for Computer Programming" [Hoa69] julkaistiin vuonna 1969. Siinä esiteltiin ensimmäistä kertaa idea ohjelmointikielen käskyjen aksiomatisoinnista. Jos tietyn ohjelmointikielen lauseiden ja kontrollirakenteiden semantiikka määritellään aksiomaattisesti, kyseisellä kielellä esitettyjen ohjelmien merkitystä voidaan tarkastella formaalisti. Ohjelmointikielen lauseita ovat siis esimerkiksi sijoituslauseet, kontrollirakenteita käskyjen suoritus peräkkäin sekä valinta- ja silmukkarakenteet. Hoare esitti artikkelissaan yksinkertaisen ohjelmointikielen, aksiomatisoi sen ja esitti muutamia johtosääntöjä uusille tosille lauseille. Lauseiden ja kontrollirakenteiden lisäksi täytyy tietysti määritellä aksiomaattisesti kielen perusoperaatiot (esimerkiksi modulo-aritmetiikka luonnollisille luvuille), jotta kielellä kirjoitetuilla ohjelmilla olisi jotain käyttöä.

Tarkastellaan hiukkasen Hoaren ideoita. Otetaan esimerkiksi sijoituslause, jonka semantiikka määritellään aksioomamallilla

$$\vdash P_0\{x := f\}P,$$

jossa x on muuttujan nimi ja f kielen lauseke. P_0 saadaan P:stä korvaamalla x:n kaikki esiintymät f:llä (oikeastaan määritellään siis aksioomien joukko $\{x \mid x$ vastaa syntaktisesti merkkijonoa "$\vdash P_0\{x := f\}P$"$\}$).

Aksioomaa voidaan soveltaa siten, että jos oletetaan P:n olevan tosi sijoituksen jälkeen, P_0:n tiedetään olevan tosi ennen sijoitusta. Ylläoleva aksioomamalli käsittelee siis tarkasteltavien predikaattien syntaksia: sijoituslauseen tapauksessa lausetta edeltävästä predikaatista korvataan tietyt merkit toisilla, jotta lauseesta saataisiin tosi.

Yleisemmin ottaen tästä säännöstä seuraa Hoaren todistustapa: määritellään muuttujien arvot, joiden halutaan vallitsevan ohjelman suorituksen loppuessa, ja todistetaan ohjelman oikeellisuus ikäänkuin takaperin. Jos totuusarvoinen vakio *true* implikoi ohjelman ensimmäistä lausetta edeltävän tilan, ohjelma toimii oikein eli saavuttaa halutun lopputilan.

Hoaren artikkeli oli teoreettinen uranavaus; määritelty ohjelmointikieli on suppea ja esitetty todistusmenetelmä hankala. Hoare ei myöskään esittänyt aksioomia tai todistusmenetelmiä ohjelman suorituksen päättymisen todistamiseksi. Hoare käsittelee vain *osittaista oikeellisuutta*: osittainen oikeellisuus on oikeellisuutta sillä ehdolla, että ohjelma pysähtyy.

Mitään käytännön todistustyötä helpottavia sääntöjä tai todistustapoja ei myöskään esitellä, vaikka Hoare viittaakin niiden kehitysmahdollisuuteen. Kuitenkin jossain mielessä kaikki olennainen sarjallisten ohjelmien oikeellisuustodistuksiin liittyvä seuraa Hoaren artikkelista.

2.3 1970-luku

1970-luvulla tutkimus jatkui Hoaren jäljillä. Sekä Hoare itse että muut jatkoivat yleisen teorian kehittelyä aksiomatisoimalla monimutkaisempia rakenteita, kuten proseduurikutsuja ja kehittyneitä tietorakenteita. Artikkelissa [HW73] esitettiin Pascalin aksiomatisoinnin yhteydessä idea taulukon tulkinnasta funktiona. Igarashi, London ja Luckham toteuttivat Pascalin aksiomatisointiin liittyen automaattisen todistusjärjestelmän [ILL75]. Tutkimus on jatkunut 1980-luvun puolelle perusteorian selvittämisen ja monimutkaisempien ohjelmarakenteiden määrittelyn merkeissä.

Tarkastellaan kuitenkin vielä erästä olennaista kehitystä. 1970-luvulla vallitsivat ristiriitaiset mielialat ohjelmantodistusten suhteen. Ylipäätään kiisteltiin siitä, mitä tuli pitää *todistuksena*: käsintodistuksia vai mekaanisesti suoritettuja. Artikkeli [MLP79] sisältää selvityksen esitetyistä näkökannoista.

Monet pitivät Hoaren todistusmenetelmiä vaikeakäyttöisinä. Tutkijat kävivät argumentaatiota Hoaren silmukkainvarianttien tarpeellisuuden puolesta ja vastaan [Gri81]. Valmiiden ohjelmien todistusta pidettiin lähestulkoon mahdottomana, joten todistus ja ohjelma piti työstää samanaikaisesti. Mitään metodologiaa tähän ei kuitenkaan esitetty ennen vuotta 1975, jolloin Edsger Dijkstran artikkeli "Guarded commands, nondeterminacy and the formal derivation of programs" [Dij75] julkaistiin.

2.3.1 Dijkstra

Dijkstra tarkoitti termillä "derivation of programs" ohjelmien johtamista täsmällisen "laskuopin" avulla. Laskuopin muodosti sääntöjoukko, jota noudattamalla ongelman ratkaiseva ohjelma kirjoitetaan, ja jota noudattaen tiedetään valmiin ohjelman olevan oikeellinen. Dijkstra perusti työnsä Hoaren esittelemälle kielen aksiomatisoinnin idealle.

Dijkstran menetelmässä ohjelman ja sitä myötä todistuksen konstruoinnissa on keskeistä *heikoimman esiehdon (weakest precondition)* käsite. Määritellään heikoin esiehto informaalisti. Olkoon S jokin ohjelma jollain ohjelmointikielellä, jonka semantiikka on määritelty, ja R jälkiehto, eli predikaatti jonka halutaan toteutuvan ohjelman suorituksen jälkeen. Heikoin esiehto ohjelmalle S ja jälkiehdolle R on se predikaatti P, jolle pätee

$$\vdash P\{S\}R,$$

ja joka on heikoin kaikista lauseen toteuttavista predikaateista P. Dijkstra käytti heikoimmasta esiehdosta ohjelmalle S ja jälkiehdolle R merkintää $wp(S, R)$. $wp(S, R)$:n loogisia seurauksia eivät siis ole mitkään muut predikaatit P, joille pätee $P\{S\}R$. Intuitiivisesti heikoin esiehto määrää, minkä asiantilojen tulee "vähintään" olla totta ennen ohjelman S suoritusta, jotta sillä saavutettaisiin haluttu lopputulos. Käyttämällä heikoimman esiehdon määritelmää, ohjelman semantiikan määrittely voidaan korvata sille nimenomaisesti määritellyllä *predikaattimuuntimella*, joka on säännöstö heikoimman esiehdon muodostamiseksi. Käytännössä predikaattimuunnin määritellään kullekin ohjelmointikielen käskylle ja kontrollirakenteelle, ja näistä määritelmistä seuraavat predikaattimuuntimet kaikille kielellä esitetyille ohjelmille.

Dijkstra esittelee heikoimman esiehdon idean myös toisella, hyvin selkeällä tavalla kirjassaan [Dij76]. Ohjelmakoodin sijaan tarkastellaan abstraktiota sen suorituksista, eräänlaista äärellistä automaattia. Heikoin esiehto $wp(S, R)$ tietylle lauseelle S, automaatin kontekstissa käsitettynä, määrittelee kaikkien niiden tilojen joukon, joista lause S lukemalla päästään jälkiehdon R toteuttavaan tilaan.

Ohjelmien kehittäminen heikoimpien esiehtojen avulla perustuu ohjelman jälkiehdon (eli siis laskennan halutun tuloksen) määrittelyyn, ja sopivien, ohjelmointikielessä määriteltyjen predikaattimuuntimien (kokemuksen ja sääntöjen ohjaamaan) kokeiluun. Predikaattimuuntimet käsittelevät Hoaren sijoitusaksiooman tapaan predikaattien syntaksia. Oleellista on siis löytää predikaattimuuntimia ja sitä myötä ohjelmointikielen lauseita, joilla ohjelman loppuehtoa voidaan heikentää kunnes saavutetaan kaikista predikaateista heikoin, totuusarvoinen vakio $true$ (jos ohjelman alkutilassa tiedetään pätevän jonkin $true$:ta vahvemman predikaatin Q, riittää saavuttaa predikaatti, joka on Q:n looginen seuraus).

Dijkstran tutkimus oli myös lyhyehkö askel ohjelmien mekaanista konstruktiota kohti. Eräs Dijkstran menetelmän seurauksista on Dijkstran itsensä mukaan kirkastunut ymmärrys siitä, mikä ohjelmoinnista on mekaanista suorittamista ja mikä luovaa työtä [Dij76].

3 Pysähtymättömät ohjelmat

Tavanomaisen, pysähtyvän laskennan ohella voidaan määritellä muunlaisiakin haluttuja laskentoja, sekä löytää niille käyttöä. Tietokonejärjestelmien kehittymisen myötä on ilmennyt tarvetta suorittaa päättymättömiä, silti oikeellisia laskentoja. Vaikka mitään matematiikasta tai tosielämästä kumpuavia ongelmia harvemmin on mieltä pyrkiä ratkaisemaan ikuisilla silmukoilla, ongelmien ratkaisemisen kehys eli tietokonejärjestelmä tarvitsee niitä toimiakseen. Esimerkkinä tästä ovat vaikkapa käyttöjärjestelmät, jotka reagoivat syötteisiinsä ja joiden laskenta ei saavuta mitään mielenkiintoista lopputilaa.

Rinnakkaisjärjestelmän halutun laskennan määritelmä eroaa siis tavanomaisesta, kuvauksesta syötteiden joukolta tulosteiden joukolle, koska niiden oikeelliset laskennat ovat pääasiassa päättymättömiä. Koska laskenta ei saavuta lopputilaa, ei ole myöskään tarkoituksenmukaista määritellä mitään sarjallisten ohjelmien todistuksissa käytetyn kaltaisia loppuehtoja, joiden tulisi täyttyä laskennan päätyttyä. Rinnakkaisjärjestelmän oikeellisuus määräytyy sen yksittäisten suoritustilojen, sekä sen äärellisten ja äärettömien osasuoritusten perusteella.

Mielenkiinto rinnakkaisten algoritmien oikeellisuutta kohtaan on myös peräisin 1960-luvulta. Ajan formaalit kehitykset ohitetaan lyhyehkösti, koska ne eivät tuoneet muassaan mitään tärkeitä todistusmetodologioita. Tässä työssä keskitytään kahteen myöhemmin esiteltyyn olennaisesti uuteen ideaan, jälkimmäinen näistä vuodelta 1981. Tämänkin jälkeen tutkimusta on tehty (ja tehdään edelleen), mutta merkittäviä uusia todistusparadigmoja ei ole esitetty. Esityksessä nojataan viitattuihin artikkeleihin ja erääseen alueelle luotuun katsantoon [CW96].

3.1 1960-luku

Rinnakkaisuuden tuottamat ongelmat havaittiin 1960-luvulla muun muassa käyttöjärjestelmien kehitystyön yhteydessä. Ongelmista seurasi tarve määritellä rinnakkaisjärjestelmiä muodollisesti. Yksi tapa formalisoida rinnakkaisjärjestelmä on esittää se Petri-verkkona – määritelmä Petri-verkoille esitettiin vuonna 1962. Myös yksittäisiä ongelmia tutkittiin: Dijkstra esitti vuonna 1965 ensimmäisen ratkaisun usean prosessin poissulkemisongelmalle [Dij65]. Poissulkemisen ongelmasta keskusteltiin vuosikymmenen puolivälissä runsaasti, ratkaisuja esittivät muitten muassa Dekker ja Knuth ("lopullisen" ratkaisun ongelmaan esitti

Peterson vasta 1981). Dijkstra esitteli semaforin idean, sekä tunnetun aterioivien filosofien ongelman vuonna 1968. Kuitenkaan mitään ratkaisevaa kehitystä ei saavutettu, tutkimus oli lähinnä yksittäisten ongelmien hajanaista tarkastelua.

3.2 1970-luku

Hoaren ideat perinteisten ohjelmien todistukseen liittyen vaikuttivat myös rinnakkaisten ohjelmien oikeellisuustutkimukseen. Hoare esitteli itse erilaisia rakenteita prosessien kontrolloimiseen, kuten monitorit vuonna 1973. Nämä erilaisten kontrollirakenteiden semanttiset määrittelyt eivät kuitenkaan ole yleisiä oikeellisuustodistuksen tapoja, toisin kuin Turing-täydellisen sarjallisen ohjelmointikielen määrittely sarjallisten ohjelmien tapauksessa. Hoare ja muut vuosikymmenen alkupuolen tutkijat käsittelivät lähinnä funktionaalisia, eli jonkin tietyn tuloksen laskevia ohjelmia [Pnu77]; näiden oikeellisuus määräytyy siis osittain lopputilan oikeellisuuden perusteella. Siten 1970-luvunkin kehitykset olivat sangen epäyleisiä ja epäkäytännöllisiä esimerkiksi käyttöjärjestelmien verifiointiin.

Yleisen tavan määritellä päättymättömän ohjelman oikeellisuus (ja todistaa se) esitti Amir Pnueli vuonna 1977.

3.3 Pnueli

Vuonna 1977 julkaistiin Amir Pnuelin artikkeli “The temporal logic of programs”. Pnueli esittää artikkelissaan tietokonejärjestelmän suoritukselle uuden formaalin abstraktion, jossa on mahdollista todistaa luontevasti päättymättömien laskentojen ominaisuuksia. Formalismina käytettiin *temporaalilogiikkaa*, ja suorituksen nähtiin olevan temporaalilogiikan malli.

Temporaalilogiikka on erikoistapaus modaali- eli monimallilogiikasta. Modaalilogiikassa kaavan mallin muodostavat monta maailman tilaa tavanomaisten loogisten kielien yksitilaisen mallin sijaan. Temporaalilogiikassa modaalilogiikan maailman tiloille määritellään jokin järjestys. Jos tiloja tarkastellaan seuraavuusjärjestyksessä, voidaan tilan muutos siirryttäessä tilasta seuraajaansa nähdä maailman tilan muutoksena ajan kuluessa. Maailman tila voidaan käsittää vaikkapa ohjelman muistialueen sisällöksi, maailman tilan muutos on siten ohjelman muistialueen sisällön muuttumista ohjelman suorittaessa käskyjään. Tämä on monessa mielessä luonnollinen abstraktio ohjelman suoritukselle.

Pnuelin idea oli, että temporaalilogiikan kieltä voitaisiin käyttää sekä pysähtyvien että pysähtymättömien ohjelmien oikeellisuustodistuksiin. Tarkastellaan seuraavassa yksimallisten logiikoiden rajoituksia tavanomaisissa ohjelmamalleissa tapahtuvissa todistuksissa.

Perinteisellä logiikalla voidaan ilmaista luontevasti perinteisten, pysähtyvien ohjelmien osalta osittainen oikeellisuus: jos ohjelman suoritus päättyy ja tulostaa vastauksen ongelmaan, vastaus on oikein. Sitävastoin täydellistä oikeellisuutta (vaikka se voidaan perinteisellä logiikalla ja sopivilla todistuskonstruktioilla todistaa) ei ole yleisessä tapauksessa kätevää ilmaista yksimallisessa logiikassa. Rinnakkaisjärjestelmissä tavanomaisella logiikalla voidaan ilmaista turvallisuusominaisuudet, kuten prosessien keskinäisen poissulkemisen toteutuminen ja lukkiumattomuus. Perinteiset logiikat eivät kuitenkaan tavallisia ohjelman abstraktioita käytettäessä tarjoa selviä, yleisiä tapoja määrätä, että rinnakkaisjärjestelmän laskennan tulee edetä tai perinteisen ohjelman suorituksen päättyä.

Rinnakkaisjärjestelmien päättymättömissä suorituksissa halutaan määrätä laskennan edistymisestä, siis yleisessä tapauksessa asioiden keskisestä seuraavuudesta. Jos on kyse jonkin resurssin tasapuolisesta jakamisesta, halutaan ilmaista, että pyyntö päästä käyttämään resurssia johtaa vuoron saamiseen. Jos ohjelmalle annetaan syötteitä, on tavanomaista vaatia, että syötteeseen annetaan vaste. Halutaan siis muotoilla täsmällisesti luonnollisella kielellä seuraavasti ilmaistavia ominaisuuksia: "jos a tapahtuu, niin b tulee tapahtumaan joskus".

Ominaisuuden muotoilevassa lauseessa käytetään sanaa "joskus" ilmaisemaan halutun tapahtumisen hetkeä. Voitaisiin myös määritellä, että tapahtuman tulee tapahtua jollakin tietyllä hetkellä. Tarkasta suorituksen hetkestä ei kuitenkaan olla välttämättä kiinnostuneita: jos halutaan ohjelman suorituksen päättyvän, tarkalla ajanhetkellä jolloin se päättyy ei useinkaan ole väliä – tarpeeksi tarkka arvio seuraa ohjelman aikavaativuudesta. Samoin jos jaetaan resurssin käyttövuoroja, usein riittää tietää, että vuoro saadaan joskus; arvio hetkestä "joskus" seuraa usein järjestelmäkohtaisista, oikeellisuusmääritelmän kannalta epämielenkiintoisista ominaisuuksista.

Yllä epämuodollisesti määritelty ominaisuus viittaa kahteen ajanhetkeen, nimittäin niihin, jolloin a ja b tapahtuvat. Temporaalilogiikassa tämänkaltaiset yhtä tai useampaa ajanhetkeä koskevat ominaisuudet voidaan määritellä helposti ja yleisesti. Temporaalilogiikan kieleen kuuluu tavanomaisten loogisten operaattoreiden lisäksi temporaalioperaattoreita, joiden merkitys on juuri määritellä useampien ajanhetkien välisiä suhteita. Usein määritellään esimerkiksi temporaalioperaattorit $\Diamond$, joka luetaan "joskus", sekä $\Box$, joka luetaan "aina". Yllämainittu ominaisuus voidaan nyt määritellä kaavalla

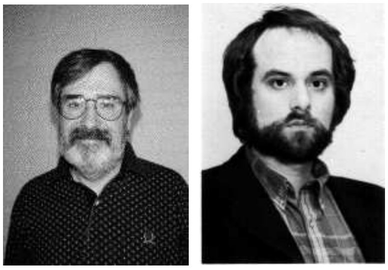
$$\Box(a \Rightarrow \Diamond b),$$

joka luetaan "aina täytyy päteä: jos a tapahtuu, b tulee tapahtumaan joskus".

Pnuelin menetelmä perustuu verifioitavan järjestelmän temporaaliloogiseen aksiomatisointiin, kullekin järjestelmälle määritellään siis järjestelmäkohtaiset aksioomat. Aksioomat puuttuvat tässäkin tapauksessa, kuten Hoaren aksioomat ja Dijkstran predikaattimuuntimet, oikeellisen toiminnan määrittelevien loogisten lauseiden syntaksiin. Ohjelman todistus on siis myös Pnuelin menetelmällä sen oikeellisen toiminnan määräävän kaavan syntaktista muuntelua.

Pnueli viittaa artikkelissaan myös äärellistilaisten järjestelmien verifiointiin, ja toteaa että äärellistilaisen järjestelmän verifiointi temporaalilogiikan kaavan suhteen on mielivaltaisessa tapauksessa ratkeava ongelma. Järjestelmä voidaan verifioida propositionaalisessa temporaalilogiikassa, koska äärellinen tila-avaruus voidaan karakterisoida äärellisellä määrällä propositioita, ja kaavan toteutuvuus voidaan tarkastaa joka tilassa erikseen. Tässä voidaan tunnistaa *mallintarkastukseen* viittaavia piirteitä. Mallintarkastus on viimeinen suuri idea verifioinnin saralla; se luetaan yleensä Clarken ja Emersonin nimiin.

3.4 Clarke ja Emerson



Mallintarkastuksen idea on rakentaa äärellinen malli tietokonejärjestelmästä, ja tarkastaa toteutuvatko halutut ominaisuudet mallissa. Koska malli on äärellinen, minkä tahansa loogisella kielellä esitettävän lauseen toteutuvuus siinä on ratkeava ongelma. Mallintarkastuksen idea esitetään artikkelissa “Synthesis of synchronization skeletons for branching time temporal logic” [CE81] (samaan ajatukseen päätyivät riippumattomasti Queille ja Sifakis vuonna 1982). Termi “mallintarkastus” *(model checking)* on Clarken ja Emersonin perua. [CW96]

Artikkelissaan Clarke ja Emerson mallintavat rinnakkaisjärjestelmän äärellisenä siirtymäjärjestelmänä. Järjestelmän haluttu toiminta esitetään temporaalilogiikan kaavana. Järjestelmän malli on itse asiassa suunnattu verkko, joka sisältää polkuina järjestelmän kaikki mahdolliset suoritukset. Jos oikeellisen toiminnan kertova temporaalilogiikan kaava toteutuu kaikissa näissä suorituksissa, järjestelmä toimii oikein. Mallintarkastuksen keskeinen etu logiikan kielellä tapahtuviin todistuksiin verrattuna on tämän tarkastuksen automatisoinnin (ja ymmärryksen) helppous: tarkastelu voidaan palauttaa yksinkertaisten verkkoalgoritmien soveltamiseen. Riittää esimerkiksi jakaa verkko vahvasti yhtenäisiin komponentteihin ja tutkia näitä, tai etsiä verkosta tietynlaisia syklejä.

Mallintarkastus voidaan siis automatisoida helposti. Sillä on myös toinen olennainen etu kaavatodistuksiin verrattuna. Jos järjestelmän todistus oikeaksi jossain logiikassa ei onnistu, voi olla kyse kolmesta asiasta: todistusta ei ole olemassa, todistus ei onnistu inhimillisistä syistä, tai järjestelmä toimii väärin. Todistusta ei välttämättä ole olemassa jos käytetyn formalismin aksiomatisointi ei ole täydellinen, eli todistusta oikeellisen toiminnan ilmentävälle kaavalle ei voida muodostaa kyseisessä formalismissa. Riippuen menetelmästä, ensimmäinen syy voidaan ehkä sulkea pois, mutta toisten poissulkemiseen tarvitaan joko paljon aikaa ja vaivaa (onnistuneeseen todistukseen), tai vastaesimerkki (joka pitää siis saada aikaan manuaalisesti).

Mallintarkastusohjelma palauttaa joko “kyllä” tai “ei” riippuen siitä, toteutuuko kaava mallissa. Sopivaa algoritmia soveltamalla saadaan kuitenkin myös muuta oleellista tietoa: vastaesimerkki järjestelmän oikeellisuudelle. Koska tila-avaruus on äärellinen, löytyy aina polku virhetilaan tai virhesyklin alkuun (äärettömät suoritukset ovat äärellisessä tila-avaruudessa syklisiä). Tarkastelemalla vastaesimerkkiä voidaan nähdä miksi ohjelma toimii väärin, eli minkä virheen järjestelmä tekee. Tätä suoritusjälkeä tarkastelemalla virheen korjaaminen on yleensä helppoa.

Mallintarkastuksella on äärellisiin järjestelmiin keskittymisen lisäksi yksi vakava haittapuoli: menetelmä on laskennallisesti raskas. Vaikka tila-avaruuden koon suhteen mallintarkastus voidaan suorittaa lineaarisesti, järjestelmän ohjelmakoodin suhteen on kyse PSPACE-täydellisestä ongelmasta. Vuoden 1981 jälkeen yksi pysähtymättömien ohjelmien oikeellisuustodistusten tutkimuksen pääsuuntauksista on ollut pyrkimys kiertää tätä laskennan vaativuutta.

4 Yhteenveto

Tässä esitelmässä on tarkasteltu tietokoneohjelmien oikeellisuustodistuksien historiaa. Tietokoneilla pystytään ratkomaan suhteellisen kompleksisia ongelmia, joiden ratkaisemiseen esitetyn neuvon eli algoritmin oikeellisuus on intuitiivisella tasolla käsittämättömissä. Tietokoneohjelma on sopivasti määriteltynä matemaattinen, eksakti objekti, jonka tunnetussa suoritusympäristössä suorittamia toimintoja voidaan tarkastella formaalisti. Voidaan siis myös todistaa, että formaalisti esitetty ohjelma toimii toisen formaalin seikan, eli oikeellisen toiminnan määrittelyn mukaan. Tämän asian todistamiseen on kehitetty paljon menetelmiä, joita tässä esityksessä on tarkasteltu.

Oikeellisuustodistusten tutkimisella on lähtökohtansa sekä käytännön tarpeessa, että teoreettisesti mielenkiintoisessa ongelmakentässä. Tietokoneiden yleistymisen ja ohjelmien monimutkaistumisen myötä ohjelmien oikeellisuus tuli monien intressien kannalta kriittiseksi asiaksi, joten oikeellisuuden varmistaminen (ja tähän keinona sen todistaminen) tuli mielekkääksi tutkimuksen kohteeksi.

Lähteet

CE81 Clarke, E. & Emerson, E.: Synthesis of synchronization skeletons for braching time temporal logic. *Lecture Notes in Computer Science 131*, Springer-Verlag, 1981.

CW96 Clarke, E. & Wing, J.: Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 1996.

Dij65 Dijkstra, E.: Solution of a Problem in Concurrent Programming Control. *Communications of ACM*, 8(9), 1965.

Dij75 Dijkstra, E.: Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM 18*, 1975, ss. 453-457.

Dij76 Dijkstra, E.: *A discipline of programming.* Prentice-Hall, 1976.

Flo67 Floyd, R.: Assigning meanings to programs. *Proc. Symposium on Applied Mathematics 19, Mathematical Aspects in Computer Science*, American Mathematical Society, 1967, ss. 19-32.

Gri81 Gries, D.: *The Science of Programming.* Springer-Verlag, 1981, ss. 294-301.

Hoa69 Hoare, C.: An axiomatic basis for computer programming. *Communications of the ACM 12*, 1969, ss. 576–580, 583.

HW73 Hoare, C. & Wirth, N.: An Axiomatic definition of the programming language Pascal. *Acta Informatica 2*, 1973, ss. 335-355.

ILL75 Igarashi, S., London, R. & Luckham, D.: Automatic Program Verification I: A Logical Basis and its Implementation. *ACTA Informatica 4*, 1975, ss. 145-182.

McC61 McCarthy, J.: A basis for a mathematical theory of computation. *Proc. Western Joint Comp. Conf.*, 1961.

MLP79 De Millo, R., Lipton, R. & Perlis, A.: Social processes and proofs of theorems and programs. *Communications of the ACM 22*, 1979, ss. 271-280.

Nau66 Naur, P.: Proofs of algorithms by general snapshots. *Nordisk tidskrift for informationsbehandling 6(4)*, 1966, ss. 310-316.

Pnu77 Pnueli, A.: The temporal logic of programs. *IEEE 18th Annual Symposium on Foundations of Computer Science*, 1977, ss. 46-57.

Tur36 Turing, A.: On computable numbers, with an application to the *Entscheidungsproblem. Proc. London Mathematical Society 42*, 1936, ss. 230-265.