Tietojenkäsittelytiede **27** Joulukuu 2007 sivut 47–76
Toimittajat: Ari Korhonen ja Jorma Tarhio

Rinnakkaisjärjestelmien algoritminen verifiointi*

Antti Valmari
Tampereen teknillinen yliopisto
Ohjelmistotekniikan laitos
Antti.Valmari@tut.fi

Tiivistelmä

Tässä katsauksessa tutustutaan ensin siihen, miten vaikeaa rinnakkaisuutta sisältäviä ohjelmia ja järjestelmiä on saada toimimaan luotettavasti. Sen jälkeen kerrotaan, mitä rinnakkaisjärjestelmien toiminnan automaattisella tarkastamisella käytännössä tarkoitetaan ja mihin se pystyy ja ei pysty. Sitten kerrotaan lyhyesti, minkätyyppisiä kieliä on kehitetty toisaalta järjestelmän ja toisaalta siltä vaadittavien ominaisuuksien kertomiseksi verifiointiautomaatille. Loppuosassa artikkelia tutustutaan automaattista verifiointia vaivaavaan syvälliseen suorituskykyongelmaan ja käydään lyhyesti läpi yli puoli tusinaa keinoa sen lievittämiseksi. Asiat esitellään yleistajuisesti ilman matematiikkaa ja algoritmikuvauksia.

1 Kauhutarina

Kuvitelkaamme, että Harri Hakkerin on laadittava pankkiautomaatin ja pankin keskustietokoneen välille tietoliikenneohjelmisto. Pitkissä johdoissa sattuu helposti monenlaisia häiriöitä, joiden vuoksi sanomia voi kadota matkalla. Tehtävänä on laatia kumpaankin päähän ohjelma siten, että kokonaisuus havaitsee kadonneet sanomat ja tarvittaessa lähettää ne uudelleen. Tällaista ohjelmistoa (tai säännöstöä, jonka mukaan se lähettää ja vastaanottaa sanomia) kutsutaan *tietoliikenneprotokollaksi*. Kutsukaamme pankkiautomaatissa toimivaa ohjelmaa "lähettimeksi" ja pankin keskustietokoneessa toimivaa "vastaanottimeksi".

Tietoliikenneprotokollissa käytetään usein niin sanottuja *kuittauksia*. Joka kerta kun vastaanotin saa sanoman, se lähettää lähettimelle sanoman, jonka merkitys on "kiitos, kuulin viestisi". Joka kerta kun lähetin on lähettänyt sanoman, se odottaa kuittausta ennalta ohjelmoidun ajan, esimerkiksi yhden sekunnin. Jos kuittaus saapuu tänä aikana, lähetin toteaa sanoman välityksen onnistuneen. Jos määräaika menee umpeen ilman kuittauksen saapumista, lähetin lähettää sanoman uudelleen. Tarvittaessa lähetin voi lähettää sanoman kolmannen kerran ja vieläkin useammin.

Tarinamme Harri tekee ohjelmat ja, vastuuntuntoinen kaveri kun on, testaa ne huolellisesti. Testeissä ilmenee omituinen asia: osa viesteistä menee perille kahteen

*Artikkeli on alunperin kirjoitettu vuonna 2002 Tietojenkäsittelytieteen seuran toteutumatta jäänyttä 20-vuotisjuhlakirjaa varten.

kertaan. Siis jos Harri nostaisi tililtään 100 euroa, nosto saattaisi kirjautua kahdesti, jolloin Harrin tililtä vähennettäisiin 200 euroa!

Virhetoiminnan syyksi paljastuu seuraava tapahtumaketju: Lähetin lähettää sanoman. Se menee perille vastaanottimeen, joka lähettää kuittauksen. Kuittaus katoaa matkalla. Lähetin odottaa kuittausta aikansa ja sitten lähettää sanoman uudelleen. Uusintakin menee perille, joten viesti menee perille kahdesti.

Virheen välttämiseksi vastaanottimen on jotenkin kyettävä erottamaan edellisen sanoman uusinta uudesta sanomasta. Sanoman sisällön perusteella tätä ei voi tehdä, koska voi olla, että sama sisältö todella halutaan lähettää kahdesti. Muutoinhan Harri voisi huijata pankkia nostamalla tililtään 100 euroa ja heti perään uudelleen 100 euroa, jolloin pankin tietokone tulkitsisi jälkimmäisen noston edellisen uusinnaksi ja velottaisi tililtä vain 100 euroa.

Tavallisesti tämä ongelma ratkaistaan liittämällä jokaiseen sanomaan vuoronumero. Vastaanotin pitää kirjaa viimeksi saamansa sanoman vuoronumerosta. Se tunnistaa toistuneen sanoman siitä, että vuoronumero ei ole muuttunut viimeksi vastaanotettuun verrattuna. Itse asiassa riittää, että vuoronumeron paikalla on yksi bitti, joka on joka toisessa alkuperäisessä sanomassa 0 ja joka toisessa 1.

Harri teki tämän muutoksen. Nyt ohjelmisto läpäisi perusteelliset testit. Harri vei sen pankkiin ja se otettiin käyttöön.

Puoli vuotta myöhemmin Harri sai pankista vihaisen puhelun. Hyvin pieni osa nostoista oli jäänyt kirjaamatta, ehkä yksi kuukaudessa. Pankin asiantuntijat olivat onnistuneet osoittamaan, että vika on Harrin ohjelmistossa. Vika ilmeni erityisesti perjantaisin ja palkkapäivinä.

Harrilla kesti kauan saada selville, missä vika piili. Nostotapahtumia hävisi seuraavasti. Lähetin lähetti sanoman, ja se meni perille vastaanottimeen. Pankin tietokoneella oli kiirettä, koska nostosanomia tuli tiuhaan eri automaateista, joten se kyllä lähetti kuittauksen, mutta vasta pienen viiveen jälkeen. Tällä välin lähettimen odotusaika kului umpeen, joten se lähetti sanoman uudelleen. Välittömästi tämän jälkeen kuittaus saapui lähettimelle. Automaatti totesi tiedon nostosta menneen perille ja antoi nostajalle hänen pyytämänsä rahat.

Nostaja lähti pois, ja jonossa seuraava alkoi nostaa rahaa. Lähetin lähetti uudesta nostosta kertovan sanoman. Se hävisi matkalla. Edellisen sanoman uusinta oli sillä aikaa ehtinyt vastaanottimeen, joka lähetti siitä kuittauksen taas pienen viiveen jälkeen. Kuittaus saapui perille. Lähetin tulkitsi sen uudesta nostosta kertovan sanoman kuittaukseksi, vaikka se oli todellisuudessa aikaisemman sanoman uusintalähetyksen kuittaus. Niinpä lähetin päätteli tiedon uudesta nostosta menneen perille ja automaatti antoi käyttäjälle rahaa. Todellisuudessa uusi sanoma oli kadonnut, joten tieto nostosta ei kirjautunut pankin keskustietokoneeseen.

* * *

Kertomani tarina on tietenkin tätä kirjoitusta varten keksitty. Oikeasti pankkiautomaatin ja pankin välinen tietoliikenne on huomattavasti monimutkaisempi ja huolellisemmin varmistettu. Tarina ei silti ole pelkkää satua. Siinä kerrotun kaltaisia virheitä tapahtuu usein myös todellisuudessa. Eivätkä tällaiset ongelmat rajoitu tietoliikennejärjestelmiin. Niitä esiintyy kaikkialla, missä tietotekninen järjestelmä sisältää vuorovaikutuksessa olevia omatoimisia osia. Niitä on sattunut muun muassa puhelinverkoissa, junien kulunvalvonnassa, sädehoitolaitteissa ja jopa television kaukosäätimissä.

2 Rinnakkaisjärjestelmät ja niiden virhetoiminnot

Rinnakkaisjärjestelmät (englanniksi concurrent systems) ovat tietoteknisiä järjestelmiä, joissa on ainakin kaksi yhteistoiminnassa olevaa omatoimista osaa. Näitä osia kutsutaan *prosesseiksi*. Harri Hakkerin tapauksessa niitä ovat pankkiautomaatti, pankin keskustietokone ja niiden väliset tietoliikenneyhteydet. (Minkäpä muunkaan "oma toiminto" sanoman kadottaminen on, kuin tietoliikenneyhteyden?) Todellisessa elämässä yksittäinen laite saattaa sisältää useita prosesseja, esimerkiksi yhden vahtimaan näppäimistöä, muutamia huolehtimaan tietoliikenteestä ja niin edelleen.

Rinnakkaisjärjestelmät ovat yleensä *reaktiivisia*. Reaktiivisuus tarkoittaa sitä, että järjestelmä on samanaikaisesti korvat höröllä useampaan kuin yhteen suuntaan. Esimerkiksi päällä oleva kännykkä on koko ajan valmiina sekä vastaanottamaan puheluita että reagoimaan näppäimenpainalluksiin. Reaktiivisuus vaikeuttaa järjestelmän halutun toiminnan määrittelyä. Mitä kännykän tulee tehdä, jos sisään tulee puhelu kesken tekstiviestin kirjoittamisen?

Järjestelmälle voi olla määritelty maksimiaikoja, joiden sisällä sen on ehdittävä tehdä tiettyjä asioita. Esimerkiksi tasoristeyksen ohjauksen on laskettava puomit alas hyvissä ajoin ennen kuin juna saapuu paikalle. Vasteaikavaatimuksia huomioonottaen suunniteltua järjestelmää kutsutaan *reaaliaikajärjestelmäksi*. Reaaliaikajärjestelmät ovat yleensä reaktiivisia ja rinnakkaisia. Joskus reaaliaikaisuutta käytetään järjestelmän sisäisenä ominaisuutena joidenkin toimintojen luotettavuuden varmistamiseksi, mutta yleisesti ottaen parempana pidetään suunnitella järjestelmä siten, että olivatpa ajoitukset mitä tahansa, järjestelmä ei tee muuta väärin kuin että se mahdollisesti myöhästyy määräajoistaan.

Rinnakkaisjärjestelmissä voi esiintyä kaikkia niitä virheitä mitä tietoteknisissä järjestelmissä yleensäkin. Rinnakkaisjärjestelmissä on lisäksi vain niille tyypillisiä virheitä. Ne aiheutuvat prosessien yhteistoiminnan kompastelusta. Ehkä kaikkein dramaattisin esimerkki on *lukkiutuma* (deadlock). Se tarkoittaa tilannetta, jossa jokainen prosessi odottaa, että jokin toinen prosessi tekisi ensin jotain. Lukkiutuma voi syntyä esimerkiksi siten, että prosessi A on varannut kirjoittimen ja yrittää seuraavaksi varata CD-rom-aseman, ja samanaikaisesti prosessi B on varannut CD-rom-aseman ja yrittää varata kirjoittimen. Silloin kumpikin prosessi jää turhaan odottamaan, että toinen saisi työnsä valmiiksi ja vapauttaisi varaamansa laitteen.

Toinen dramaattinen virhe on *pillastuma* (livelock, divergence). Se tarkoittaa tilannetta, jossa järjestelmä touhuaa loputtomiin, mutta ei saa mitään hyödyllistä aikaan. Joissakin sähköpostiohjelmissa on "lomalla"-toiminto, joka päällä ollessaan lähettää jokaiseen saapuvaan viestiin välittömästi vastauksen, jossa voi lukea vaikka "olen lomalla ja käsittelen viestisi palattuani". Mitä tapahtuu, jos lomalle lähtijä kytkee toiminnon päälle ja sitten lähettää viestin kaverilleen, jolla myös on toiminto päällä? Lomalla-toiminnon ensimmäisen toteutuksen kanssa kerrotaan käyneen juuri niin. Nykyisissä toteutuksissa tämä on estetty esimerkiksi siten, että automaattivastausta ei lähetetä samaan osoitteeseen saman vuorokauden aikana kuin kerran, vaikka sieltä tulisi useita viestejä.

Lukkiutumilta voi suojautua lisäämällä järjestelmään ajastimia, jotka herättävät

prosessin ja ohjaavat sen tekemään jotakin muuta, kun se on odottanut liian kauan. Pillastumilta voi suojautua laskemalla tapahtumien toistumiskertoja ja vaihtamalla toimintaa, kun toistoja on kertynyt liikaa. Ei kuitenkaan ole helppoa keksiä, mitä järjestelmän tulisi tehdä, kun odotusaika tai toistojen määrä ylittyy. Pelkkä toiminnan jättäminen kesken johtaa helposti siihen, että järjestelmän eri osille jää eri käsitys siitä, mitä pitäisi tapahtua seuraavaksi. Lopputuloksena voi olla virheellinen kokonaistoiminto. Harrin ohjelmiston ensimmäinen versio oli laadittu sen oletuksen mukaan, että jos kuittausta ei tule, on viesti kadonnut. Jos kuitenkin viesti on mennyt perille ja kuittaus on kadonnut, jää pankin tietokoneelle ja automaatille eri käsitys siitä, onko rahaa annettu. Siksi lähetin lähettää viestin turhaan uudelleen, jolloin tiliä veloitetaan kahdesti.

Harrin ohjelmisto jäi tarinassa virheelliseksi, mutta siitä voidaan tehdä täysin vedenpitävä. Sen verran joudutaan tinkimään, että jos yhteys katkeaa juuri kun pankin keskustietokone on lähettänyt automaatille luvan antaa rahoja, ei keskustietokone voi olla varma, antoiko automaatti rahat vai ei. Tällainen tilanne voidaan kuitenkin varmuudella havaita ja kirjata ylös, minkä perusteella tilit voi tarkistaa jälkikäteen. Vedenpitävä ohjelmisto on jonkin verran tarinassa kuvailtua monimutkaisempi, mutta ei ylettömän monimutkainen.

Asian tekee hankalaksi se, että ihmisten on hyvin vaikeaa erottaa vedenpitävää ohjelmistoa aukollisesta. Jopa ensi näkemältä äärimmäisen yksinkertainen järjestelmä voi sisältää pahan virheen.

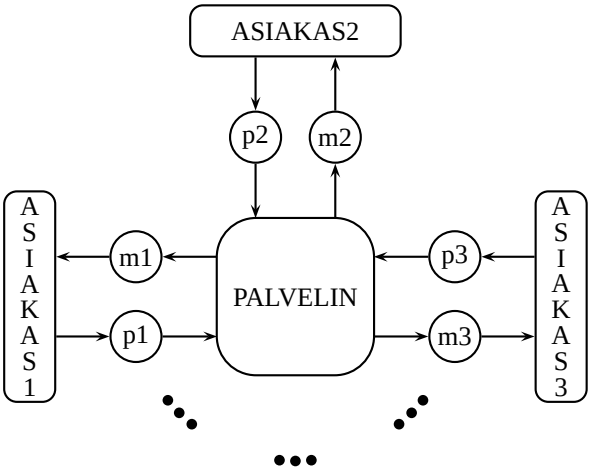
Tarkastellaan esimerkkinä järjestelmää, jonka tehtävänä on varmistaa, että jotakin yhteistä palvelua pääsee käyttämään vain yksi käyttäjä kerrallaan, ja vuorot jaetaan käyttäjille jotenkin oikeudenmukaisesti. Kirjoitin on hyvä esimerkki tällaisesta palvelusta: ei olisi mukavaa, jos samalle paperille tulisi rivejä vuorotellen minun tulostuksestani ja sinun tulostuksestasi. Esimerkin palvelin viestii käyttäjien kanssa siten, että jokaista käyttäjää kohti on kaksi kaksiarvoista muuttujaa nimiltään “p” eli “pyyntö” ja “m” eli “myöntö”. Järjestelmä on esitetty kuvassa 1.

Aluksi jokaisessa muuttujassa on arvo nolla. Kun käyttäjä haluaa käyttää palvelua, se asettaa pyyntö-muuttujansa arvoon yksi. Palvelin antaa käyttäjälle luvan käyttää palvelua asettamalla käyttäjän myönnön arvoksi yksi. Lopetettuaan palvelun käytön käyttäjä asettaa pyyntönsä nollaksi. Kun palvelin huomaa tämän, se vastaavasti asettaa myönnön nollaksi ja, jos tähän mennessä on tullut muita pyyntöjä, antaa käyttöluvan jollekin muulle käyttäjälle.

Vaikka en ole vielä kuvaillut periaatetta, jolla palvelin jakaa vuoroja jos on monta pyyntöä yhtäaikaa, on järjestelmään jo rakennettu ratkaiseva virhe. Lukija voi yrittää keksiä, mikä se on. Tämä nimenomainen virhe on helppo korjata sitten kun se on löydetty. Mutta miten voidaan olla varmoja, että järjestelmään ei jää piileksimään muita virheitä? Miten voidaan tietää, voiko korjattuun järjestelmään luottaa?

Tavallinen tapa löytää virheitä järjestelmistä on nimeltään *testaus*. Se on järjestelmän toiminnan suunnitelmallista kokeilemista. Testaus löytää suuren osan virheistä, mutta ikävä kyllä osa jää aina piiloon.

Prosessien yhteistoiminnan kompuroinnista aiheutuville virheille on tavallista, että virhe ilmenee vain, jos järjestelmän eri osissa asiat tapahtuvat ajallisesti onnettomissa kohdissa toisiinsa nähden. Tällaiset virheet ovat yleisiä rinnakkais-



ASIAKAS*i*:n koodi

p*i* := 1
odota kunnes m*i* = 1
käytä palvelua
p*i* := 0

Kuva 1: Viallinen vuorojenjakojärjestelmä.

järjestelmissä, mutta ne ilmenevät koeajon tai käytön aikana harvakseltaan. Siksi niitä on vaikea havaita testauksessa. Jos sellainen havaitaan, sen syytä on vaikea jäljittää, koska virhettä on vaikea saada toistumaan tarkempaa tutkimusta varten. Nostotietojen hukkuminen Harri Hakkerin ohjelmistossa on esimerkki tällaisesta virheestä. Nostoja hukkui erityisesti perjantaisin ja palkkapäivinä, koska vain silloin keskustietokone oli niin kiireinen, että se hidasteli riittävästi aiheuttaakseen onnettoman tapahtumaketjun.

Virheiden paljastumistodennäköisyyttä testauksen aikana voidaan nostaa lähettämällä järjestelmälle syötteitä tiuhaan tahtiin. Olettakaamme, että jonkin virheen esiintymistodennäköisyys normaalissa käyttötilanteessa on kerran miljoonaa käyttötuntia kohti, mutta rankemman kuormituksen ansiosta virheitä sattuu testauksen aikana kerran kymmenessätuhannessa tunnissa. Kymmenentuhatta tuntia on niin pitkä aika, että ehkä avaruusluotaimien ohjelmistoja on varaa testata niin kauan, mutta ei tavallisia järjestelmiä. Pankkiautomaatteja on kuitenkin Suomessa lähes kaksituhatta ja nostoja tehdään vuosittain yli kaksisataa miljoonaa [17], joten käyttötunteja kertyy yhteensä miljoonittain, ja virhe ehtii ilmetä useita kertoja vuoden aikana.

Niinpä, vaikka testaus paljastaa virheitä moninkertaisella tehokkuudella tavalliseen käyttöön verrattuna, testauksen kannalta liian harvinaisia virheitä tapahtuu tuotantokäytössä.

3 Verifiointi

Alunperin sanalla “verifiointi” tarkoitettiin järjestelmän todistamista virheettömäksi. Vahvoja takeita virheettömyydestä tarvitaan, jotta jotkin järjestelmät uskallettaisiin ottaa käyttöön. Ei ole vakavaa, jos joka tuhannes kännykkäpuhelu katkeaa kesken, mutta on vakavaa, jos lentokone putoaa joka tuhannennella lennolla!

Aikaa myöten kuitenkin huomattiin, että virheettömyyden osoittaminen on yleensä monessa suhteessa epämielekäs tavoite:

- Todellisia järjestelmiä ei koskaan saada virheettömiksi varsinkaan ensi yrittämällä. Virheellisen järjestelmän todistaminen virheettömäksi ei tietenkään voi eikä saa onnistua, ja liian yksioikoisesti sitä yrittävä verifiointimenetelmä joutuu umpikujaan. Tästä syystä on käytännössä tärkeää, että verifiointimenetelmä pystyy myös sanomaan "järjestelmä on virheellinen" ja tuottamaan tietoa, jonka avulla virheitä voi jäljittää — esimerkiksi kuvauksen virhetilanteeseen johtavasta tapahtumaketjusta. Pikkuhiljaa on alettu ajatella, että tämä on jopa tärkeämpää kuin kyky julistaa järjestelmä virheettömäksi.

- Teoreettinen virheettömyys ei takaa virheettömyyttä käytännössä seuraavasta syystä. Jotta verifiointiautomaatti pystyisi löytämään virheitä tai todistamaan, että niitä ei ole, sen täytyy tietää, minkälainen käyttäytyminen on väärää. Osa virhelajeista on kaikille järjestelmille yhteisiä — esimerkiksi täydellinen mykistyminen kesken kaiken, mutta osa riippuu verifioitavasta järjestelmästä — esimerkiksi onko valojen sytyttäminen jonain hetkenä virhe. Niinpä verifiointiautomaattia käyttävä ihminen joutuu tavalla tai toisella kertomaan, minkälainen käyttäytyminen on virheetöntä. Kertomiseen on pakko käyttää verifiointiautomaatin ymmärtämää, usein kömpelöä kieltä. Niinpä ihminen saattaa esittää ajatuksensa väärin. Sitäpaitsi ihmisten ajatukset eivät aina ole alunperinkään täysin johdonmukaisia.

- On olemassa toinenkin syy, miksi täydellinen varmuus jää haaveeksi. Vaikka ohjelma saataisiin verifioitua, sen suoritusympäristön ei voida edes periaatteessa todistaa toimivan oikein. Sähkönsyöttö voi katketa, kosmisen säteilyn hiukkanen voi muuttaa muistin sisältöä ja niin edelleen. Käytännössä mukana on muitakin epävarmuustekijöitä. Ohjelman suorittaminen esimerkiksi vaatii, että se on ensiksi käännetty konekieliseen muotoon. Tässä tarvitaan käännösohjelmia, joihin ei aina voi täysin luottaa. Ketjun myöhemmät lenkit jäävät siis joka tapauksessa epätäydellisiksi, joten ohjelman absoluuttisen virheettömyyden osoittamiseksi ei kannata nähdä miten paljon vaivaa tahansa.

- Parhaimpienkin tähän mennessä toteutettujen verifiointiautomaattien suorituskyky jättää paljon toivomisen varaa. Poikkeustapauksia esiintyy, mutta pääsääntö on, että nykyisin ei pystytä verifioimaan oikeita ohjelmia, vaan ainoastaan niiden toimintaperiaatteet kuvaavia suunnitelmia. Kattava verifiointi onnistuu rutiininomaisesti vain melko pienille järjestelmille. (Harri Hakkerin järjestelmä on korjattunakin hyvin pieni.) Isompien järjestelmien verifioimisessa käytetään erikoistekniikoita, jotka joskus tehoavat ja joskus eivät. Sen sijaan verifiointia voi käyttää epätäydellisenä virheiden etsimiskeinona hyvin isojenkin järjestelmien tapauksessa.

Näistä syistä sanan "verifiointi" piiriin kuuluvaksi katsotaan nykyisin paitsi ohjelmakoodin tai toimintaperiaatteen kuvauksen oikeaksi todistaminen suhteessa halutun käyttäytymisen kuvaukseen, myös virheiden etsiminen menetelmillä, jotka ainakin periaatteessa kykenisi-

vät myös oikeaksi todistamiseen. Testaus ei ole verifiointia, koska se ei edes periaatteessa kykene todistamaan järjestelmää virheettömäksi.

Automaattista verifiointia kutsutaan myös *mallintarkastukseksi* (model checking). Tässäkin tapauksessa sanan käyttö on laajentunut täsmällisen merkityksensä ulkopuolelle. Alunperin "mallintarkastus" on tarkoittanut sen selvittämistä, toteuttaako annettu rakenne annetun loogisen kaavan. Jos vastaus on "kyllä", niin loogikot sanovat, että rakenne on kaavan malli. Jäljempänä tässä kirjoituksessa puhutaan tila-avaruuksista ja aikalogiikasta. Tila-avaruuden automaattinen tarkastaminen aikalogiikan kaavaa vastaan on yksi verifioinnin valtavirroista ja on mallintarkastusta sanan täsmällisessä merkityksessä. Mutta, kuten todettu, sanaa käytetään nykyisin lähestulkoon kaikesta toiminnasta, jossa jotakin vähänkin tila-avaruuden kaltaista kohdetta verrataan automaattisesti millä tahansa monimutkaisia ominaisuuksia ilmaisemaan kykenevällä kielellä ilmaistuun vaatimukseen.

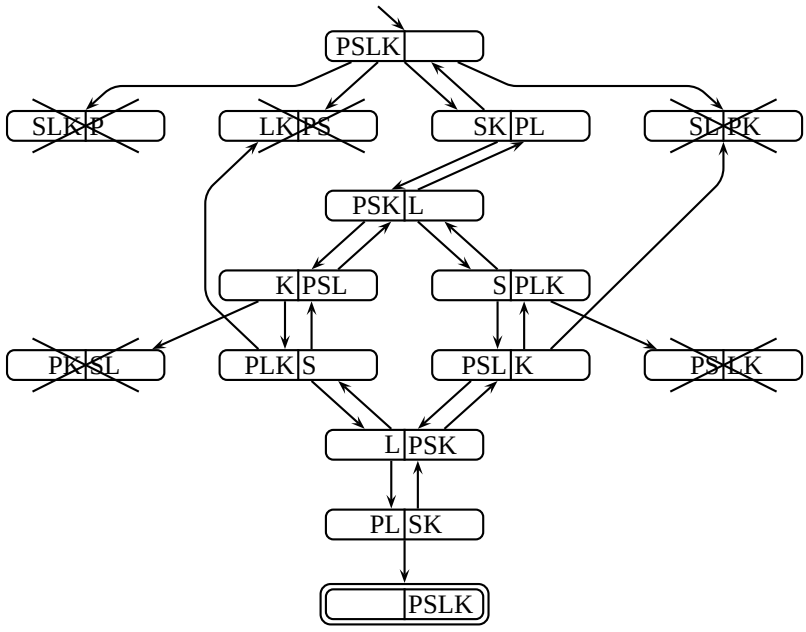
Verifioinnista (tai mallintarkastuksesta) on kaksi käytännön hyötyä. Ensinnäkin se kykenee löytämään käytännössä merkityksellisiä virheitä, jotka jäävät testauksessa löytymättä. Kuten edellä todettiin, tällaisia virheitä on, ja ne ovat aiheuttaneet ikäviä seurauksia.

Toiseksi, koska verifiointi ei yleensä kohdistu todellisiin järjestelmiin vaan niiden suunnitelmiin, verifioida voidaan jo siinä vaiheessa, kun järjestelmästä on olemassa vain suunnitelma. Virheen paljastaminen jo tässä vaiheessa säästää huomattavasti, koska vältetään virheellisen suunnitelman toteuttamisesta aiheutuva hukkatyö. Jos luotetaan vain testaukseen, ei järjestelmän eri osien yhteistoiminnan virheitä voida havaita ennen kuin osat on toteutettu ja liitetty yhteen. Tosin on niin, että suunnitelman verifiointi edellyttää suunnitelman esittämistä tietokoneen luettavassa muodossa, ja sellaista suunnitelmaa voi myös testata.

Verifiointimenetelmiä on kehitetty useisiin eri tehtäviin, mutta parhaiten ne ovat menestyneet rinnakkaisjärjestelmille tyypillisten virheiden etsimisessä ja niiden poissaolon todistamisessa. Tämä johtuu toisaalta siitä, että rinnakkaisjärjestelmille on löydetty helppokäyttöisiä ja helposti automatisoitavia menetelmiä, jotka eivät tehoa muualla; ja toisaalta siitä, että insinöörijärki on rinnakkaisjärjestelmien tapauksessa poikkeuksellisen epäluotettava. Niinpä tarpeet ja mahdollisuudet kohtaavat rinnakkaisjärjestelmissä paremmin kuin muualla.

Erityisen hyvin menetelmät ovat toimineet mikropiirien, kuten mikroprosessoreiden, suunnitelmien verifioinnissa. Tuotteessa olevat virheet ovat mikropiirien valmistajan maineelle ja taloudelle paljon vahingollisempia kuin ohjelmistojen valmistajalle, mikropiireille kun ei voi lähettää jälkikäteen päivityksiä Internetin kautta. Toisaalta, sen verifioiminen, että jokin yksikkö laskee liukulukujen jakolaskuja oikein, on mikropiirien tapauksessa sangen hyödyllinen saavutus, mutta ohjelmien tapauksessa samantasoisen asian verifiointi ei riitä pitkälle.

Rinnakkaisjärjestelmien verifioinnissa kaikkein parhaiten ovat menestyneet niin kutsutut *tila-avaruusmenetelmät*. Järjestelmän tila-avaruus tarkoittaa kaikkia tiloja, joihin järjestelmä voi joutua, sekä kaikkia näiden tilojen välisiä siirtymiä. Kuvassa 2 on esitetty yksinkertainen esimerkki. Tila-avaruusmenetelmien perusideana on muodostaa tila-avaruus osittain tai kokonaan ja analysoida sitä joko jälkikäteen tai muodostamisen aikana. Tila-avaruusmenetelmien etuja ja haittoja käsitellään yksityiskohtaisesti tämän kirjoi-



Kuva 2: Suden, lampaan ja kaalin tehtävän tila-avaruus. Paimenen (P) pitää viedä susi (S), lammas (L) ja kaali (K) joen yli. Veneeseen mahtuu paimenen lisäksi korkeintaan yksi muu. Lammasta ei saa jättää suden tai kaalin seuraan ilman paimenen läsnäoloa, koska silloin susi syö lampaan tai lammas kaalin.

tuksen loppuosassa.

Jonkin verran käytetään myös *teoreemantodistusmenetelmiä*. Niiden perusideana on todistaa järjestelmän suunnitelman toimivuus samaan tapaan kuin matemaatikko todistaa teoreeman, mutta päättelytyö pyritään teettämään tarkoitukseen suunnitelluilla ohjelmistoilla, niin sanotuilla *teoreemantodistimilla*.

Kokemukset ovat osoittaneet, että teoreemantodistimen käyttäjän on formalisoitava melkoinen määrä yksityiskohtia ja ohjattava todistin kädestä pitäen todistuksen läpi. Matemaattisesti vaativaa ihmistyötä tarvitaan siis paljon. Jos järjestelmä on virheellinen, ei sen virheettömyyden ilmaiseva väittämä päde, eikä niin ollen ole todistettavissa. Teoreemantodistimen käyttäjälle tilanne näkyy siten, että todistusta ei saa menemään läpi vaikka kuinka yrittäisi. Käyttäjä saa vihjeitä virheen sijainnista analysoimalla, missä kohdassa todistus jumiutuu. Näin saatava tieto on kuitenkin epäsuoraa. Se palvelee huonosti verifioinnin nykyisin tärkeimmäksi katsottua käyttöä, eli virheiden paljastamista ja jäljittämistä. Sitäpaitsi todistamisen epäonnistuminen ei takaa, että järjestelmä olisi virheellinen — voihan olla, että vikaa on vain käyttäjän todistimelle antamissa määritelmissä ja välitavoitteissa.

Näistä syistä teoreemantodistimien käyttö rinnakkaisjärjestelmien verifioinnissa on jäänyt olennaisesti vähäisemmäksi kuin tila-avaruusmenetelmien. Teoreemantodistimilla on kuitenkin se kiistaton etu, että toisin kuin tila-avaruusmenetelmät, ne ovat periaatteessa

yleispätevä menetelmä, minkä vuoksi niitä voi käyttää joissakin tapauksissa joissa tila-avaruusmenetelmät eivät tehoa.

Teoreemantodistus ja tila-avaruuden käyttö voidaan myös yhdistää, esimerkiksi siten, että teoreemantodistin huolehtii suurista linjoista itse ja hakee yksityiskohtaisiin kysymyksiin vastauksia mallintarkastimelta. Jotkin tila-avaruusmenetelmät esittävät tietoa niin epäsuorassa muodossa, että niiden ja teoreemantodistimien ero hämärtyy. Kohdassa 6.8 käsiteltävät binääripäätöskaaviot ovat esimerkki. Vielä selvempiä esimerkkejä on olemassa, kuten menetelmiä, jotka esittävät äärettömiä luonnollisten lukujen osajoukkoja äärellisten automaattien avulla. Kohdassa 6.9 esitettävä rajoitettu mallintarkastus ei oikeastaan ole tila-avaruusmenetelmä eikä teoreemantodistusmenetelmä, vaan niiden välimuoto.

Rinnakkaisjärjestelmien verifiointiin on kehitetty myös muunlaisia menetelmiä, esimerkiksi lineaarialgebraan perustuvia. Niiden sovellusalue on kapea, minkä vuoksi niiden merkitys on jäänyt vähäiseksi. Lisäksi on mainittava rinnakkaisjärjestelmien suoritusaikoihin liittyvien ominaisuuksien verifioimiseksi kehitetyt, tila-avaruuksiin perustuvat menetelmät. Ne muodostavat aivan oman maailmansa, johon ei valitettavasti voida syventyä tässä kirjoituksessa.

4 Kuvauskielet

Kuvauskieli on kieli, jolla järjestelmä esitetään verifiointiautomaatille. Kuten edellä todettiin, suoraan ohjelmointikielellä esitettyjen järjestelmien automaattinen verifiointi on mahdollista vain poikkeustapauksissa. Ohjelmointikieltä käyttämällä myös menetettäisiin mahdollisuus verifioida jo siinä vaiheessa, kun järjestelmästä on olemassa vain suunnitelma.

Järjestelmien suunnitelmien esittämistä varten on kehitetty useita täsmällisiä kieliä. Käytännössä suunnitelmat esitetään kuitenkin epätäsmällisten kaavioiden ja taulukoiden sekä luonnollisen kielen avulla, ja periaatteessa täsmällisiäkin kieliä käytetään epätäsmällisesti, joten verifioinnin kannalta oleellisten tietojen poimiminen automaattisesti suunnitelmasta on mahdotonta. Silloinkin, kun suunnitelma on täysin täsmällinen, siitä on ennen verifiointia abstrahoitava pois suuri joukko rinnakkaisilmiöiden kannalta epäolennaisia piirteitä, mikä vaatii ihmistyötä. Näistä syistä verifiointimallit esitetään tähän tarkoitukseen erityisesti suunnitelluilla kielillä.

Rinnakkaisjärjestelmien verifiointimallin esittämiseen tarkoitetun kielen täytyy esittää selkeästi järjestelmän jakaantuminen rinnakkaisiin prosesseihin ja prosessien välinen vuorovaikutus. Prosessien sisäisestä toiminnasta on abstrahoitava mahdollisimman tarkoin pois kaikki se, millä ei ole merkitystä vuorovaikutuksen kannalta. Esimerkiksi tietoliikenneprotokollan sisäisistä sanomista esitetään sanoman tyyppi (dataviesti, kuittaus, ...) ja vuoronumero, mutta sanoman varsinainen datasisältö joko puuttuu kokonaan tai se voi saada vain kaksi tai kolme mahdollista arvoa. Vuoronumeroidenkin arvoalue voi olla pienennetty. Kaksi tai kolme datan arvoa riittää hyvin esimerkiksi sen tutkimiseen, voiko protokolla vaihtaa sanomien järjestystä.

Prosessien välisen vuorovaikutuksen mekanismeja on useita erilaisia. Todellisissa järjestelmissä käytetään paljon puskureita, jotka säilyttävät sanomien järjestyksen. Ne ovat kuitenkin verifioinnin kannalta ongelmallisia, koska ne pystyvät varastoimaan paljon tietoa ja siksi kasvattavat rajusti järjestelmän tilojen määrää. Lisäksi puskurin käyttäytyminen

silloin, kun varastointikapasiteetti loppuu kesken, voidaan määritellä monella eri tavalla, eikä todellisen puskurin käyttäytymistä yleensä tiedetä. Niinpä verifiointimallia ei kuitenkaan saada täysin yhtäpitäväksi todellisuuden kanssa.

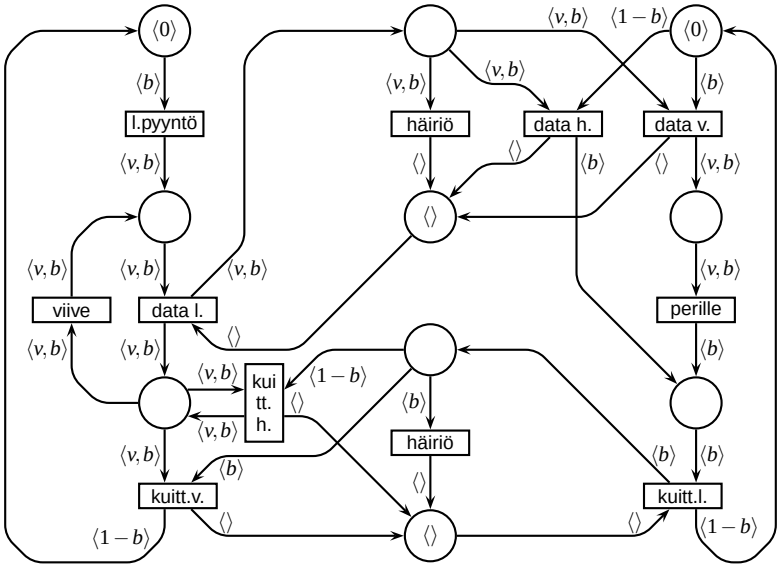
Rinnakkaisjärjestelmien verifioinnissa käytettävissä kuvauskielissä esiintyy useita eri vuorovaikutusmekanismeja. Jotkin tarjoavat ensisijaisesti puskurit, kuten CCITT:n standardoima *SDL*. Toiset, kuten Gerard Holzmannin ATT Bell Laboratoriossa (nykyisin Lucent Bell Laboratories) kehittämä *Promela* [8] tarjoavat monia keskenään tasa-arvoisia mekanismeja, joista puskurit ovat yksi. (Holzmann sai kehittämästään verifiointityökalusta “Spin” vuoden 2001 ACM Software System -palkinnon. Promela on Spinin syöttökieli. Sama palkinto on aiemmin myönnetty muun muassa Unixin, TeXin, PostScriptin, TCP/IP:n ja World-Wide Webin laatimisesta [1].)

SDL:ssä, Promelassa ja monessa muussa kielessä prosessi esitetään joko suoraan tai epäsuorasti niin sanottuna *tilakoneena*. Tilakone koostuu joukosta *tiloja* ja *tilasiirtymiä*. Tilassa prosessi odottaa herätettä toiselta prosessilta, järjestelmän ympäristöstä tai ajastimelta. Herätteen saatuaan prosessi suorittaa tilasiirtymän, jonka aikana se voi lähettää herätteitä muille prosesseille, kirjoittaa yhteiseen muistiin, laskea omien muuttujiensa avulla ja niin edelleen.

Aivan omanlaisensa ovat lukuisat *Petriverkkoihin* perustuvat kielet. Petriverkoissa ei ole valmiina minkäänlaista prosessin esittämiseen tarkoitettua käsitettä. (Sana “prosessi” kyllä esiintyy Petriverkkokirjallisuudessa, mutta aivan eri merkityksessä.) Petriverkkomallin rakentaja joutuu kokoamaan sekä prosessit että niiden väliset vuorovaikutukset alemman tason peruspalikoista, jotka ovat tietoa tallettava *paikka* ja tapahtumaa ja vuorovaikutusta edustava *transitio*. Vaikka tämä voi kuulostaa työläältä, tilakoneiden rakentaminen paikoista ja transitioista on todellisuudessa hyvin helppoa. Petriverkkoja on tosin kritisoitu siitä, että niillä voi luontevasti matkia vain joitakin vuorovaikutusmekanismeja. Nykyisin käytetään niin sanottuja *väritettyjä Petriverkkoja* [10], joilla voi kätevästi esittää järjestelmän datasisällön sopivasti abstrahoituna. Petriverkoilla on havainnollinen graafinen esitystapa (katso kuva 3). Niinpä ne ovat verifioinnissa suosittuja.

Niin sanotut *prosessialgebralliset kielet*, kuten ISO:n standardoima *Lotos* [3], Oxfordin yliopistossa kehitetty *CSP* [7, 19] ja Edinburghin yliopiston *CCS* [16] perustuvat *synkroniseen vuorovaikutukseen*. Siinä prosessit vuorovaikuttavat suorittamalla tilasiirtymän yhtäaikaa. Synkronista vuorovaikutusta käytetään todellisissa järjestelmissä vähän, koska sen toteuttaminen fyysisen välimatkan yli on hyvin vaikeaa. Silti se sopii hyvin verifiointimalleissa käytettäväksi, koska sillä voi helposti matkia muita vuorovaikutusmekanismeja tekemällä tiedon välivarastosta (kuten puskurista) itsenäisen prosessin. Lisäksi synkronisen vuorovaikutuksen teoria on kehitetty erittäin paljon pitemmälle kuin muiden vuorovaikutusmuotojen teoriat, minkä ansiosta synkroniselle vuorovaikutukselle on pystytty kehittämään erinomaisia verifiointimenetelmiä.

Teknillisessä korkeakoulussa Espoossa on käytetty Petriverkkoja rinnakkaisjärjestelmien verifiointiin yli kaksikymmentä vuotta. Monet ulkomaiset Petriverkkotutkijat ovat ottaneet siellä kehitetyn “Prod”-työkalun [18] käyttöönsä. Helsingin yliopistossa on käytetty sekä tilakonepohjaisia että prosessialgebrallisia kieliä. Tampereen teknillisessä yliopistossa



Kuva 3: Vuorottelevan bitin protokollan väritetty Petriverkkomalli. “l.” = lähetys, “v.” = vastaanotto ja “h.” = hylkäys virheellisen vuorottelevan bitin vuoksi. Välitettävä viesti *v* tulee protokollalle transitiolla “l.pyyntö” ja menee määränpäähän transitiolla “perille”. Esimerkiksi jos oikean yläkulman paikassa on $\langle 0\rangle$ ja ylärivin keskimmäisessä paikassa on $\langle v, 1\rangle$, missä *v* on viestin sisältö, niin transitio data h. on suoritettavissa. Se poistaa merkit $\langle 0\rangle$ ja $\langle v, 1\rangle$ sekä asettaa tyhjää datakanavaa edustavaan paikkaan merkin $\langle\rangle$ ja kuittauksen lähettämistä edeltävään paikkaan merkin $\langle 1\rangle$. Samalla tavalla data h. voi siirtää ja muuttaa merkit $\langle 1\rangle$ ja $\langle v, 0\rangle$ merkeiksi $\langle\rangle$ ja $\langle 0\rangle$.

algoritmisen verifioinnin tutkimus aloitettiin vuonna 1992. Tamperelainen työ on jatkoa silloisessa VTT:n Tietokonetekniikan laboratoriossa Oulussa vuonna 1984 aloitetulle tutkimukselle, joka alkoi Petriverkoilla, mutta on suurimman osan aikaa käyttänyt prosessialgebrallisia kieliä. Tampereen ryhmän [22] saavuttamat verifioinnin teoriaa ja algoritmeja koskevat tulokset, joista osa on kehitetty yhteistyössä Helsingin yliopiston kanssa, on otettu maailmalla hyvin vastaan.

5 Oikean käyttäytymisen määritteleminen

5.1 Aikalogiikat

Verifiointiautomaatille on tutkittavan rinnakkaisjärjestelmän lisäksi tavalla tai toisella kerrottava, minkälainen käyttäytyminen on virheellistä. Aluksi tyydyttiin etsimään yksittäisiin ennalta määrättyihin lajeihin kuuluvia virheitä, kuten lukkiutumia tai odottamatta saapuvia sanomia. Varsin pian alettiin sekä kehittää että ottaa käyttöön muualla kehitettyjä ilmaisuvoimaisempia tapoja määritellä virheitä. Suuren suosion — ja hyvillä syillä — saivat

aikalogiikat, joita israelilaiset Zohar Manna ja vuoden 1996 Turing-palkinnon[1] saaja Amir Pnueli[2], yhdysvaltalaiset Ed Clarke ja Allen Emerson sekä monet muut kehittivät 1970-luvun loppupuolelta alkaen.

Pnueli on keskittynyt *lineaarisiin aikalogiikoihin* [14], joilla määritellään ominaisuuksia, jotka järjestelmän jokaisen yksittäisen suorituksen on täytettävä. Lineaarisella aikalogiikalla ei voi puhua vaihtoehtoisista jatkomahdollisuuksista suorituksen ollessa kesken. Clarken ja muiden *haarautuvan ajan logiikoissa* tämä on mahdollista. Lineaarinen aikalogiikka pystyy esimerkiksi toteamaan vain, että joinakin aamuina professori juo kahvia ja joinakin teetä, mutta haarautuva aikalogiikka pystyy kertomaan myös, tekikö professori valintansa kahvin ja teen välillä ennen vai jälkeen kahvioon astumisen.

Lineaarisella aikalogiikalla esitetyn väittämän paikkansapitävyyden tarkastaminen rinnakkaisjärjestelmän tilaavaruudesta on työlästä (tarkasti sanoen **PSPACE**-täydellistä kaavan pituuden suhteen). Sen sijaan Clarken ja Emersonin kehittämän *CTL*:n (Computation Tree Logic) [4] väittämien paikkansapitävyyden tarkastaminen on laskennallisesti helppoa. Helppous johtuu siitä, että CTL:stä on jätetty pois joitakin lineaarisen aikalogiikan piirteitä. CTL:n ilmaisuvoimasta ei silti tullut käytännön sovelluksia ajatellen kohtuuttoman heikko, koska kyky puhua vaihtoehtoisista tulevaisuuksista lisää ilmaisuvoimaa. CTL onkin saavuttanut suuren suosion. Jos CTL:ään lisätään siitä poistetut lineaarisen aikalogiikan operaattorit, saadaan nimellä CTL^* tunnettu logiikka.

Lineaarisella aikalogiikalla on kuitenkin omat etunsa haarautuviin verrattuna. Jos tutkittavasta järjestelmästä löytyy virhe, se voidaan havainnollistaa kuvailemalla yksittäinen virheellinen suoritus. Haarautuvan ajan logiikassa yksinkertainen kuvailu voi olla mahdotonta. Eikä lineaarisen aikalogiikan työläys sittenkään ole suuri ongelma, sillä lyhyillä kaavoilla työmäärä ei kasva sietämättömäksi ja verifioitavat kaavat ovat yleensä lyhyitä. Niinpä myös lineaarisen aikalogiikan suosio on suuri. Sitäpaitsi vaikka lineaarisuus aiheuttaa tehokkuushaittaa lopullisessa tarkastusvaiheessa, se antaa tärkeitä tehokkuusetuja toisaalla [23]. Logiikoiden välinen paremmuusjärjestys ei siis ole edes tehokkuuden osalta itsestään selvä.

5.2 Ekvivalenssit ja esijärjestykset

Aikalogiikoilla määritellään yksittäisiä ominaisuuksia tyyliin “jos asiakas on tehnyt oman osuutensa nostotapahtumasta, pankkiautomaatti antaa lopulta joko rahaa tai virheilmoituksen” ja “tililtä ei koskaan, häiriötilanteissakaan, veloiteta enempää rahaa kuin asiakkaalle on annettu”. Sallittu käyttäytyminen voidaan määritellä muillakin tavoilla. Eräs paljon tutkittu mahdollisuus on määritellä toinen järjestelmä ikään kuin malliksi ja vaatia, että tutkittava järjestelmä käyttäytyy jossain mielessä “samalla tavalla” tai “ainakin yhtä hyvin” kuin mallijärjestelmä. Jälkimmäinen vaihtoehto tarvitaan seuraavan esimerkin havainnollistamasta syystä. Nimittäin, jos spesifikaatio sallii pankkiautomaatin vastata “yhteyshäiriö” mutta automaatti ei koskaan vastaa niin, niin automaatti ei käyttäydy samalla tavalla kuin spesifikaatiosta tehty mallijärjestel-

[1] Turing-palkinto [1] on nimetty tietojenkäsittelyn teorian pioneeri Alan Turingin (1912–1954) mukaan, ja se on tietojenkäsittelytieteen alan arvostetuin palkinto.

[2] Pnueli kehitti logiikkansa alunperin inhimillistä päättelyä varten.

mä. Olisi kuitenkin hölmöä käyttää automaatin virheelliseksi julistamisen perusteena sitä, että se ei koskaan tee spesifikaation sallimaa mutta epäsuotavaa toimintoa.

Mallijärjestelmä voi olla tavattoman paljon yksinkertaisempi kuin verifioitava järjestelmä, koska sen ei tarvitse olla käytännössä toteutettavissa. Esimerkiksi pankkiautomaatti voi mallijärjestelmässä käsitellä tiliä suoraan, vaikka tosimaailmassa kaiken on tapahduttava epäluotettavan tietoliikenneyhteyden yli. Mallijärjestelmä voidaan myös rajata yksittäiseen näkökulmaan — esimerkiksi vain kaksi asiakasta ja heidän tilinsä, vaikka todellisen järjestelmän on koko ajan käsiteltävä kaikkia tilejä. Mallijärjestelmän laatiminen voi siten olla helppoa.

Malli- ja verifioitavan järjestelmän vertaamiseksi toisiinsa on kehitetty useita niin sanottuja *ekvivalensseja* ja *esijärjestyksiä*. Ekvivalenssi tekee täsmälliseksi "samalla tavalla käyttäytymisen" käsitteen ja esijärjestys käsitteen "käyttäytyy ainakin yhtä hyvin".

Ekvivalenssia tai esijärjestystä käytettäessä pitää sopia, mitkä järjestelmien tapahtumat otetaan vertailussa huomioon. Esimerkiksi pankkijärjestelmän tapauksessa voidaan ottaa huomioon kaikki, mitä automaattia käyttävä asiakas voi välittömästi havaita, sekä hänen tilinsä saldon muutokset. Silloin muun muassa automaatin ja pankin välisen tietoliikenteen tapahtumat jätetään pois. Huomioonotettavia tapahtumia kutsutaan *näkyviksi*, ja muut tapahtumat ovat tietenkin *näkymättömiä*.

Yksinkertaisin ekvivalenssi eli *jälkiekvivalenssi* (englanniksi *trace equivalence*[3]) julistaa järjestelmät samanveroisiksi, jos niiden kaikista mahdollisista suorituksista syntyvät äärelliset näkyvien tapahtumien jonot eli niin sanotut *jäljet* ovat samat. Vastaava esijärjestys toteaa järjestelmän olevan ainakin yhtä hyvä kuin malli on, jos jokainen järjestelmän jälki on myös mallin jälki, mutta ei välttämättä toisinpäin. Jälkiekvivalenssi on melkein sama asia kuin kahden äärellisen automaatin hyväksymien kielten samuus, ja jälkiesijärjestys vastaa sitä, että ensimmäisen automaatin kieli on toisen automaatin kielen osajoukko.

Näissä vertailuissa suorituksiksi lasketaan myös suoritusten keskeneräiset alkuosat. Tämä voi vaikuttaa omituiselta, mutta se yksinkertaistaa asioita. Jos keskeneräisiä suorituksia ei otettaisi huomioon, jouduttaisiin tutkimaan sekä äärettömiä että päättyviä suorituksia. Asiakashan voi palata pankkiautomaatille yhä uudelleen (asiakkaan eliniän rajallisuuden esittäminen ei kuulu verifiointimallin tehtäviin!), mutta toisaalta, järjestelmän lukkiutumiset halutaan havaita, ja niitä vastaavat suoritukset ovat päättyviä. Koska keskeneräiset suoritukset otetaan huomioon, äärettömät suoritukset tulevat riittävässä määrin huomioiduiksi keskeneräisten alkuosiensa välityksellä, joten äärellisten jälkien tutkiminen riittää.

Jälkiekvivalenssi ja -esijärjestys riittävät sen selvittämiseen, voiko järjestelmä joutua tilanteeseen, jossa jotakin luvatonta on tapahtunut. Aikalogiikassa tällaisia ominaisuuksia kutsutaan *turvallisuusominaisuuksiksi*. Esimerkiksi "tililtä ei koskaan veloiteta liikaa" on turvallisuusominaisuus. Turvallisuusominaisuuksien vastakohta on *etenemisominaisuudet* eli *aidot elävyysominaisuudet*, jotka voivat rikkoutua vain siten, että jotakin jää tapahtumatta. "Jokainen nosto kirjataan lopulta" on etenemisominaisuus. Mikään ää-

[3] Verifioinnin yhteydessä saattaa törmätä myös niin sanottuihin Mazurkiewiczin jälkiin, mutta ne ovat eri asia.

rellinen jälki ei riitä todisteeksi siitä, että nosto voi jäädä kirjaamatta, sillä vaikka jälki ei sisältäisi noston kirjausta, voi olla, että nosto kirjattiin hetki sen jälkeen kun jäljen nauhoittaminen lopetettiin.

Jotta järjestelmän ja mallin vertailussa voitaisiin ottaa huomioon myös etenemisominaisuuksia, on kehitetty jälkiekvivalenssia monimutkaisempia ekvivalensseja ja esijärjestyksiä. Niitä ovat kehittäneet erityisesti prosessialgebrojen tutkijat. Ekvivalenssilla olisi hyvä olla niin sanottu *kongruenssiominaisuus*. Se tarkoittaa, että jos järjestelmän osan tilalle vaihdetaan ekvivalentti osa, koko järjestelmän pitää säilyä ekvivalenttina alkuperäisen kanssa. Esijärjestyksen tapauksessa kongruenssiominaisuus vaatii, että järjestelmä ei saa huonontua siitä, että sen jokin osa korvataan paremmalla. Esimerkiksi suksipaketin hinta ei saa nousta, jos asiakas vaihtaa siteet halvempiin. Ilman kongruenssiominaisuutta on vaikea johtaa järjestelmän osia koskevista verifiointituloksista koko järjestelmää koskevaa tietoa.

Jos etenemisominaisuudet halutaan ottaa mielekkäällä tavalla huomioon, on kongruenssiominaisuus odottamattoman vaikea saavuttaa. Tämä on johtanut lukuisten toinen toistaan omituisempien ekvivalenssien ja esijärjestysten kehittämiseen, jotka käsittelevät etenemisominaisuuksia mikä mitenkin. Aivan kuten aikalogiikoita, myös ekvivalensseja ja esijärjestyksiä voidaan muodostaa sekä lineaariselle että haarautuvalle ajalle, mikä on lisännyt sekamelskaa entisestään.

Useimmat ekvivalenssit ja esijärjestykset voidaan katsoa kuuluviksi kolmeen perheeseen: Robin Milnerin *heikko havaintoekvivalenssi* [16] muunnelmineen; *estymäpohjaiset ekvivalenssit*, joista kuuluisin sisältyy Tony Hoaren CSP-teoriaan [7, 19] ja joihin kuuluu myös suomalainen *CFFD* [28]; sekä *haarautuva bisimilaarisuus* [29]. Hoare sai vuoden 1980 Turing-palkinnon, ja Milner sai saman kunnian vuonna 1991.

Heikko havaintoekvivalenssi ja haarautuva bisimilaarisuus sukulaisineen edustavat haarautuvaa aikaa. Niiden perusideana on vaatia, että verrattavat järjestelmät voivat simuloida toisiaan. Järjestelmien tiloille määritellään monta–moneen-vastaavuus siten, että kun toinen tekee jonkin asian jostakin tilastaan alkaen, toinen voi tehdä “samanveroisen” asian kyseisen tilan jokaisesta vastintilasta alkaen, minkä jälkeen järjestelmät päätyvät vastintiloihin. Haarautuva bisimilaarisuus vastaa ilmaisuvoimaltaan melko tarkasti CTL-logiikkaa, ja heikko havaintoekvivalenssi on niitä heikompi.

Estymäpohjaiset ekvivalenssit perustuvat lineaariseen aikakäsitykseen. Ne saadaan edellämainitusta jälkiekvivalenssista lisäämällä informaatiota niin sanotuista *estymistä* (failure), joka on lukkiutuman yleistys. Muutakin lisäinformaatiota voi olla mukana, tärkeimpänä luettelo jäljistä, joiden seurauksena järjestelmä voi pillastua (katso sivu 49). Pnuelin lineaarista aikalogiikkaa vastaa tarkimmin suomalainen NDFD, joka on melkein sama kuin CFFD.

NDFD ja CFFD kehitettiin ratkaisuksi CSP:n ekvivalenssin omituisuuteen, joka juontaa juurensa siitä, että CSP:n ekvivalenssi haluttiin kehittää abstraktilla algebrallisella tasolla, vetoamatta edes tilapäisesti prosessien toiminnan sellaisiin yksityiskohtiin, jotka eivät näy ulos. CSP:n ekvivalenssi ei säilytä mitään tietoa järjestelmän käyttäytymisestä sen jälkeen, kun se on suorittanut pillastumajäljen. Verifiointisovelluksia ajatellen tämä on harmillinen rajoitus. NDFD:n ja CFFD:n määritelmissä vedotaan näkymättömiin yksityiskohtiin, mutta ne pois-

tetaan lopputuloksesta. Oxfordissa on tunnustettu CFFD:n etu ja yritetty saada se aikaan myös CSP:n matematiikalle ominaisin keinoin. Vuonna 2005 Roscoe julkaisi artikkelin [20], jossa johdettiin erittäin paljon CFFD:n kaltainen ekvivalenssi soveltaen CSP-mäisiä keinoja syvällisellä tavalla. CFFD:n ja NDFD:n suhdetta CSP:hen analysoitiin Tony Hoaren eläkkeelle jäännin yhteydessä vuonna 1999 pidetyn juhlaseminaarin eräässä esitelmässä [25].

Ekvivalenssien ja esijärjestysten teoriaa ovat Suomessa menestyksekkäästi tutkineet etenkin Jaana Eloranta, Roope Kaivola, Timo Karvi ja Martti Tienari Helsingin yliopistosta, sekä Antit Puhakka ja Valmari Tampereen teknillisestä yliopistosta.

5.3 Visuaalinen verifiointi

Kuten edeltä käy ilmi, verifioitavien ominaisuuksien määrittelemiseksi on käytettävissä niin voimakkaita keinoja, että ominaisuuden ilmaiseminen ei ole kynnyskysymys. Ikävä kyllä on osoittautunut, että verifioitavien ominaisuuksien keksiminen on vaikeaa. On helppo keksiä irrallisia yksittäisiä ominaisuuksia, mutta on vaikea laatia verifioitavien ominaisuuksien luettelo, jossa kaikki olennainen on mukana. Jos on verifioitu, että tietoliikenneprotokolla ei voi hukata eikä vääristää sanomia, on yhä mahdollista, että se voi esimerkiksi kahdentaa niitä.

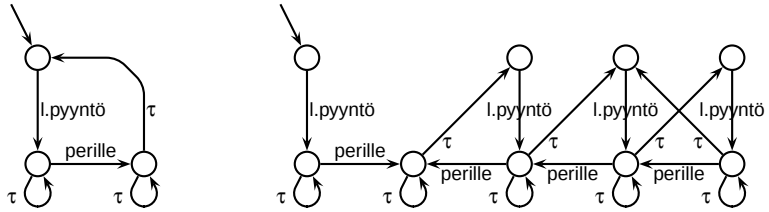
Myös on vaikeaa laatia verifioitavan järjestelmän käyttäjien ja toimintaympäristön mallit siten, että ne eivät vahingossa piilota virhemahdollisuuksia. Tämä pätee erityisesti etenemisominaisuuksille. Esimerkiksi tietoliikenneprotokollassa voi olla virhe, jonka vuoksi sanoma voi pysähtyä matkalle, ellei perässä tule toista sanomaa, joka ikään kuin “työntää” edellistä sanomaa edellään. Tätä virhettä ei havaita, jos tietoliikenneprotokollan asiakas mallitetaan lähettämään päättymätön jono sanomia, kuten usein tehdään.

Etenemisominaisuuksien verifiointia varten on usein tehtävä niin sanottuja *reiluusoletuksia*, jotka lupaavat, että järjestelmän eri osia suoritetaan jossain mielessä tasapuolisesti. Reiluusoletukset ovat usein vaikeita hahmottaa ja siksi vaikeita asettaa oikein.

Näistä syistä verifiointiautomaatteja on aina käytetty myös analyysin välineinä: automaatille on esitetty järjestelmän toimintaa koskevia kysymyksiä sitä mukaa kuin niitä tulee mieleen ilman, että verifioitavien ominaisuuksien luetteloa on muodostettu etukäteen. Tavoitteena ei ole järjestelmän osoittaminen virheettömäksi, vaan järjestelmän toimintaa koskevan hyödyllisen palautteen saaminen vähällä vaivalla.

Viimeksi kuluneen runsaan kymmenen vuoden aikana on erityisesti Tampereen teknillisessä yliopistossa tutkittu lähestymistapaa, joka voidaan katsoa satunnaisen kyselyn ja kattavan verifioinnin välimuodoksi [27]. Siinä verifiointiautomaatille kerrotaan vain, mitkä järjestelmän yksittäiset tapahtumat ovat kiinnostavia. Automaatti tuottaa järjestelmän käyttäytymisestä eräänlaisen kyseisille tapahtumille projisoidun tiivistelmän ja esittää sen kuvana. Tiivistelmän muodostaminen perustuu prosessialgebrallisiin ekvivalensseihin ja kutistusalgoritmeihin. Käyttäjän tehtävänä on tulkita kuva ja päättää, hyväksyykö hän siinä esiintyvän käyttäytymisen.

Jos käytetään lineaariseen aikaan perustuvaa ekvivalenssia ja tapahtumat valitaan sopivasti, kuva voi olla samanaikaisesti pieni ja hyvin informatiivinen. Esimerkiksi tietoliikenneprotokollasta saadaan paljon tietoa muodostamalla kuva, jossa näkyvät vain sanoman välitystehtävän anto lähetinpäässä ja sanoman toi-



Kuva 4: Vuorottelevan bitin protokollan ehjän (vasemmalla) ja Harri Hakkerin käyttämän (oikealla) version visualisoitu abstrahoitu käyttäytyminen. "τ"-kaaret esittävät näkymättömien tapahtumien välillisiä vaikutuksia.

mittaminen asiakkaalle vastaanotinpäässä. Sanomien katoamiset, kahdentumiset, matkalle pysähtymiset ja niin edelleen vihjaavat olemassaolostaan aiheuttamalla kuvaan omituisuuksia, joita käyttäjän on vaikea olla huomaamatta kuvaa tutkiessaan.

Esimerkki tästä on kuvassa 4. Kuvan oikea puoli paljastaa silkalla monimutkaisuudellaan, että jotain on vialla. Oikeaa puolta tutkimalla selviää, että ensimmäinen viesti menee perille (olettaen, että kuvassa τ-silmukoina ilmenevä pillastumisen mahdollisuus vältetään), mutta sen jälkeiset lähetyspyynnöt voivat pelkkien näkymättömien tapahtumien välityksellä johtaa tilaan, jossa ainoa mahdollinen seuraava tapahtuma on uusi lähetyspyyntö. Toisin sanoen sanomia voi kadota.

Tämän *visuaaliseksi verifioinniksi* kutsutun menetelmän suurin etu on, että käyttäjän ei tarvitse missään vaiheessa määritellä sallitun ja kielletyn käyttäytymisen välistä rajaa formaalisti. Niinpä menetelmä vaatii vähemmän koulutusta kuin tavanomainen verifiointi. Visuaalisen verifioinnin tuottamien kuvien on kokeissa havaittu kiinnittävän käyttäjän huomiota sellaisiinkin käyttäytymisen piirteisiin, joiden olemassaoloa käyttäjä ei ole aiemmin tiedostanut. Niinpä se paljastaa virheitä, jotka jäävät tavallisilta verifiointimenetelmiltä huomaamatta, koska käyttäjä ei hoksaa esittää oikeaa kysymystä. Toki käyttäjä voi tulkita visuaalisen verifioinnin tuottaman kuvan väärin, mutta käyttäjä voi tehdä virheitä myös formalisoidessaan oikean ja väärän käyttäytymisen välistä rajaa tavanomaiselle verifiointimenetelmälle. Niinpä visuaalisen verifioinnin luotettavuuden ei voi väittää olevan huonompi eikä parempi kuin tavallisen verifioinnin.

Koska valinta oikean ja virheellisen välillä jää käyttäjän tehtäväksi, visuaalinen verifiointi ei ole verifiointia sanan täsmällisessä merkityksessä. Menetelmän kehittäjät uskalsivat kuitenkin ottaa nimityksen käyttöön, sillä visuaalinen verifiointi käyttää samoja algoritmeja ja tuottaa suunnilleen yhtä luotettavia tuloksia kuin tavanomainen verifiointi.

6 Tilaräjähdysongelma

6.1 Tilaräjähdysongelman syy

Edellä todettiin, että parhaidenkin tähän mennessä toteutettujen verifiointiautomaattien suorituskyky jättää paljon toivomisen varaa. Tähän on syvällinen syy: tyypillisistä verifiointitehtävistä voidaan todistaa, että ne ovat laskennallisesti varsin vaativia. Melkeinpä mikä tahansa synkronisesti vuorovaikuttavien rinnak-

kaisten tilakoneiden käyttäytymistä koskeva kysymys on **PSPACE**-kova. Käytännössä tämä tarkoittaa, että tehtävän ratkaisemiseksi tarvittavan ajan määrä kasvaa eksponentiaalisesti tehtävän koon funktiona.[4] Myös muistin käyttö kasvaa käytännössä eksponentiaalisesti. Tiedetään, että vähemmälläkin muistilla tultaisiin toimeen, mutta ei tiedetä, miten sen voisi tehdä ilman, että ajan kulutus kasvaisi kohtuuttomasti.

Muilla mallinnusformalismeilla verifiointikysymykset ovat yleensä ainakin yhtä työläitä. Jos vuorovaikutusmekanismiksi vaihdetaan järjestyksen säilyttävä rajattoman kapasiteetin puskuri, muuttuvat kysymykset algoritmisesti ratkeamattomiksi. Se tarkoittaa, että kysymyksen yleispätevästi ratkaisevaa ohjelmaa ei voi olla olemassa.

Nämä tulokset ovat huonoimman tapauksen tuloksia. Ne tarkoittavat, että jokainen verifiointialgoritmi on toivottoman hidas ainakin joillakin syötteillä, mutta ne eivät kerro, millä syötteillä niin käy ja onko niitä tiheässä vai harvassa.

Verifioinnin periaatteellinen vaikeus ilmenee eri menetelmissä eri tavoin. Teoreemantodistimilla se ilmenee tarvittavan ihmistyön suurena määränä. Tila-avaruusmenetelmissä se ilmenee tila-avaruuksien suurina kokoina. Jokainen järjestelmään lisättävä prosessi kasvattaa tilojen määrän tyypillisesti moninkertaiseksi. Niinpä tila-avaruuden koko kasvaa eksponentiaalisesti prosessien määrän funktiona, ja pienelläkin järjestelmällä voi olla miljoonia, miljardeja tai vielä valtavasti enemmän tiloja.

Tämän niin sanotun *tilaräjähdysongelman* hillitsemiseksi on kehitetty monenlaisia tehokkaita keinoja, joista jotkin ovat hyvin nerokkaita. Tässä yhteydessä voidaan esitellä niistä vain muutamia. Rajoitettua mallintarkastusta lukuunottamatta kaikkia jäljempänä esiteltyjä menetelmiä ja eräitä muitakin on käsitelty perusteellisemmin katsausartikkelissa [24].

Laskennallisen vaativuuden teoriasta seuraa sellainenkin johtopäätös, että jos **NP** $\neq$ **PSPACE** — niin kuin laajalti uskotaan, niin rinnakkaisjärjestelmissä on virheitä, jotka eivät voi ilmetä ennen kuin suoritus on kestänyt pitkään. Tällaisia virheitä on vaikea havaita etukäteen millään keinolla, mutta ne voivat ilmetä käytännössä, sillä esimerkiksi Internet on ollut toiminnassa jo vuosikymmeniä. Internetin tapauksessa tästä ilmiöstä ei kannata murehtia, sillä siinä on varmasti yllin kyllin vähemmän eksoottisia virheitä. Avaruusluotaimen suunnittelijan näkökulmasta tilanne voi olla toinen.

6.2 Symmetriat

Eräs ensi näkemältä melko ilmeinen ja sen vuoksi usein ehdotettu keino on hyödyntää järjestelmässä esiintyviä *symmetrioita*. Ajatuksena on, että jos esimerkiksi pankkiautomaatit ovat keskenään samanlaisia, niin verifioinnin aikana riittää tietää, että *jokin* niistä on antamassa rahaa, ei tarvitse tietää *mikä*.

Valitettavasti symmetrioiden hyödyntämistä vaikeuttaa se, että järjestelmien tilat ovat usein huomattavasti vähemmän symmetrisiä kuin järjestelmä itse. Vaikka ei tarvitse tietää, *mikä* pankkiautomaatti on missäkin tilassa, täytyy kuitenkin pitää kirjaa siitä *kuinka moni* on missäkin tilassa, mikä kasvattaa tutkittavien tilanteiden määrää. Myös verifioitava ominaisuus saattaa rikkoa symmetrian: ei riitä havaita, että jonkin tilin saldo pieneni, vaan sal-

[4] Vaikka on hyviä syitä uskoa, että kaikki **PSPACE**-kovat tehtävät vaativat eksponentiaalisesti aikaa, ei asiaa ole onnistuttu sitovasti todistamaan. Laskennallisesti vaativien tehtävien teoriaa on käsitelty mukavalla tavalla kirjassa [5].

don pitää pienetä sillä tilillä, jolta nosto tapahtui. Lisäksi symmetrioiden hyödyntäminen hävittää eräiden ominaisuuksien verifioinnissa välttämätöntä tietoa. Näiden vaikeuksien vuoksi symmetrioiden käytön idea on pirstoutunut moneksi eri menetelmäksi, joista muutamia on esitelty kirjoissa [10, osa 2] ja [4].

Symmetrioista saatava hyöty riippuu symmetrian määrästä. Peilisymmetria pystyy enimmillään puolittamaan tilojen määrän. Ympyräsymmetria jakaa tilojen määrän parhaassa tapauksessa prosessien määrällä. Se on tilojen määrän eksponentiaalisen kasvun rinnalla vain pieni parannus, mutta saattaa kuitenkin riittää kasvattamaan suurimman verifioitavan järjestelmän kokoa muutamalla prosessilla. Symmetriasta saadaan todella suurta hyötyä vasta, jos järjestelmässä on joukko prosesseja, jotka ovat täysin samanveroisessa asemassa sekä suhteessa toisiinsa että suhteessa muuhun järjestelmään. Näin on esimerkiksi tähtikytkentäisessä järjestelmässä, jossa on yksi keskusasema tai palvelin, jolla on useita asiakkaita, joiden välillä ei ole suoria kytkentöjä.

Rajoitetun sovellettavuutensa ja heikohkon kutistustehonsa vuoksi symmetriamenetelmää ei ole toteutettu läheskään kaikissa verifiointityökaluissa. Symmetriat eivät silti ole jääneet vajaalle käytölle, sillä ihmiset ovat hyviä hyödyntämään niitä epäformaalisti verifiointimalleja laatiessaan.

6.3 Datariippumattomuus

Symmetrioiden kanssa hieman samankaltainen menetelmä on *datariippumattomuuden* hyödyntäminen. Sen lähtökohtana on havainto, että monet rinnakkaisjärjestelmät varastoivat tai siirtävät dataa ilman, että katsovat sen sisältöä. Niinpä, jos järjestelmä toimii oikein jollakin data-arvolla, se toimii oikein millä tahansa data-arvolla. Näin ollen riittää, että järjestelmää verifioitaessa on käytössä muutama erisuuri data-arvo. Enemmän kuin yksi data-arvo tarvitaan, jotta data-alkioiden katoaminen, kahdentuminen ja vaihtuminen voitaisiin havaita.

Datariippumattomuus voi ilmetä myös siten, että yksittäisillä data-arvoilla on merkitystä järjestelmän käyttäytymiselle, mutta tutkittava ominaisuus ei riipu siitä, kuinka monta eri data-arvoa on käytössä, kunhan niitä on vähintään jokin määrä. Esimerkiksi eräistä tietoliikenneprotokollista voidaan yksinkertaisen, helposti automatisoitavan päättelyn ja tavallisen automaattisen verifioinnin yhdistelmällä todistaa, että protokollan käyttäjilleen tuottama palvelu ei riipu uudelleenlähetysten määrästä. Toisin sanoen, käyttäytyminen ei riipu siitä, kuinka monta kertaa protokollan lähetin yrittää sanoman lähetystä ennen kuin se luovuttaa ja ilmoittaa, että sanomaa ei saada läpi — kunhan se yrittää ainakin kerran.

Järjestelmän toiminta voi olla riippumatonta paitsi data-arvoista tai niiden määrästä, myös samanlaisten, jonkin säännöllisen hahmon mukaisesti toisiinsa kytkettyjen prosessien määrästä. Tällöin riittää verifioida järjestelmä jollakin pienellä samanlaisten prosessien määrällä, mutta tulokset pätevät mille tahansa isommalle määrälle. Muuttujat voidaan tulkita prosesseiksi, joten tavallinen taulukko on erikoistapaus säännöllisen hahmon mukaisesti toisiinsa kytketyistä prosesseista.

Datariippumattomuutta hyödynnetään yleensä ihmistyönä: verifiointityökalun käyttäjä päättelee riittävän data-arvojen määrän (tai prosessien määrän tai taulukon koon) ja karsii data-arvot (tai prosessit tai taulukon alkiot) muutamaan jo verifiointimallia laatiessaan. Datariippumattomuuden käyttöä on pyritty automa-

tisoimaan toisaalta johtamalla teoreemoja, joista riittävän arvojen määrän voi suoraan katsoa (kirjan [19] luvussa 15.3 on esimerkkejä), ja toisaalta kehittämällä automaattisia abstrahointimekanismeja, jotka karsivat liiat data-arvot kun järjestelmän kuvausta käännetään verifiointityökalulle. Molemmilla alueilla on edistytty, mutta ainakin vielä nykyisin tulokset ovat verifiointityökalun käyttäjän näkökulmasta keskeneräisiä. Ehkä pisimmällä on G. Holzmannin Lucent Bell Laboratoriossa kehittämä järjestelmä, jossa ennalta määrätyn koodauskäytännön mukainen ja vuorovaikutustoiminnot ennalta määrättyjä aliohjelmia kutsumalla toteuttava C-kielinen tietoliikenneohjelma käännetään käyttäjän antaman vihjekokoelman avulla verifiointimalliksi [9].

Datariippumattomuuden hyödyntäminen käsityönä on luontevaa siksikin, että verifiointimallin laatija joutuu melkein aina abstrahoimaan, muuten verifiointityökalu tukehtuu jo ensi vaiheessa. Niinpä on luontevaa ensin verifioida järjestelmä yhdellä data-arvolla tai toistuvalla prosessilla tai taulukon alkiolla. Jos se onnistuu, määrää kasvatetaan kahteen, kolmeen ja niin edelleen, kunnes verifiointityökalu tukehtuu tai uutta tietoa järjestelmästä ei enää saada. Näin voi tehdä ja tehdään, vaikka järjestelmä ei olisikaan datariippumaton. Silloin verifiointia ei saada täysin kattavaksi, mutta, kuten edellä todettiin, täysin kattava verifiointi on joka tapauksessa yleensä liian vaativa tavoite.

Monen tämänkaltaisen tekniikan yhtäaikaisesta käytöstä on oppikirjamainen esimerkki kirjan [19] luvussa 15.3.3 sekä kirjoituksessa [26]. Suomeksi alaa on käsitelty kirjoituksessa [21].

6.4 Tila-avaruuden vaiheittainen muodostaminen

Edellä puhuttiin prosessien vertaamisessa käytettävien ekvivalenssien ja esijärjestysten *kongruenssiominaisuudesta*. Kongruenssiominaisuus tekee mahdolliseksi järjestelmän tila-avaruuden rakentamisen vaiheittain siten, että jaetaan järjestelmä osiin, muodostetaan jokaiselle osalle tila-avaruus, kutistetaan kunkin osan tila-avaruus ekvivalenssin määrittelemät ominaisuudet säilyttäen ja lopuksi yhdistetään osatila-avaruudet kokonaisuuden tila-avaruudeksi. Lopputulos voi olla valtavan paljon pienempi kuin aito tila-avaruus, mutta on silti sen kanssa ekvivalentti ja niin ollen tuottaa samat vastaukset verifiointikysymyksiin. Osan tila-avaruus voidaan muodostaa samalla periaatteella jakamalla osa vielä pienempiin osiin.

Tällainen *tila-avaruuden vaiheittainen muodostaminen* on osoittautunut erittäin tehokkaaksi keinoksi vastustaa tilaräjähdystä. Sitä käytettiin verifiointityökalussa tiettävästi ensimmäisen kerran Ranskassa 1980-luvun lopulla, ja sen jälkeen sitä on tutkittu paljon varsinkin Euroopan eri maissa Suomi mukaan lukien. Sen ehkä ainoa merkittävä heikkous on, että sen yhteydessä on vaikea käsitellä asianmukaisesti aikalogiikassa yleisesti käytettyjä reiluusoletuksia (katso sivu 61). Tosin Antti Puhakan väitöskirjassa vuodelta 2004 saavutettiin tällä saralla edistystä.

Klassisessa ohjelmien oikeaksi todistamisessa käytetään usein apuna niin sanottuja *invariantteja*. Invariantti on väittämä, josta voidaan tietyllä periaatteella päätellä, että se on aina voimassa, kun ohjelman suoritus on tietyssä kohdassa. Invarianttien muodostaminen on usein työlästä (mikä on yksi pääsyy siihen, että teoreemantodistimien käyttö verifioinnis-

sa on työlästä).

Tila-avaruuden vaiheittaisessa muodostamisessa voidaan käyttää apuna *rajapintaprosesseja*. Ne muistuttavat hieman invariantteja sikäli, että niiden avulla verifiointityökalun käyttäjä voi esittää väittämän, jonka hän uumoilee olevan aina voimassa tietyssä kohdassa järjestelmää. Tällaisen väittämän voimassaolo riippuu usein järjestelmän osan vuorovaikutuksesta muiden osien kanssa. Niinpä ympäristöstään irrotettu järjestelmän osa voi joutua tilanteisiin, joissa väittämä ei päde. Tämä voi kasvattaa sen tila-avaruutta merkittävästi. Tällaiset tilanteet leikkautuvat pois sitten kun osan tila-avaruus yhdistetään muiden osien tila-avaruuksiin. Silloin on kuitenkin jo myöhäistä: niiden muodostamisen vaiva on jo nähty.

Tämä turha työ voidaan välttää rajapintaprosesseja käyttämällä. Verifiointityökalu voi pienentää osan tila-avaruutta jo sitä muodostaessaan jättämällä pois tilanteet, joissa rajapintaprosessin esittämä väittämä on rikkoutunut. Jos väittämä pitää paikkansa koko järjestelmän tasolla, niin verifiointityökalu tuottaa samat lopulliset tulokset kuin ilman rajapintaprosessia, mutta vähemmällä vaivalla. Jos väittämä ei pidäkään paikkaansa, verifiointityökalu havaitsee lopuksi että näin on ja varoittaa käyttäjää, että verifiointitulokset ovat sen vuoksi puutteelliset. Käyttäjän ei siis tarvitse tietää etukäteen, pitääkö väittämä paikkansa.

Vaiheittaisesta tila-avaruuden muodostamisesta on kehitetty muunnelma, joka symmetriamenetelmän tavoin hyödyntää sitä tosiasiaa, että monessa järjestelmässä on useita keskenään samanlaisia osia — esimerkiksi monta samanlaista pankkiautomaattia. Siinä järjestelmään lisätään samanlaisia osia yksi kerrallaan, kunnes järjestelmän käyttäytymisen projektio tietyille näkyville tapahtumille ei enää muutu. Nämä näkyvät tapahtumat sisältävät tarkasteltavien ominaisuuksien kannalta olennaisten tapahtumien lisäksi rajapinnan, johon osat lisätään. Tällä muunnelmalla on useita kertoja onnistuttu verifioimaan järjestelmä siten, että tulokset pätevät riippumatta samanlaisten osien määrästä. On siis verifioitu kokonainen ääretön järjestelmäperhe rajallisella työllä. Tämä menetelmä voidaan katsoa esimerkiksi datariippumattomuudesta.

Vaiheittaista tila-avaruuden muodostamista muunnelmineen on käsitelty oppikirjamaisesti artikkelissa [26].

6.5 Lennosta verifiointi

Vaikutukseltaan jonkin verran rajapintaprosessien kaltainen on *lennosta verifiointina* tunnettu lähestymistapa. Lennosta verifiointi tarkoittaa, että tarkastettava ominaisuus otetaan tavalla tai toisella huomioon jo tila-avaruuden muodostamisen aikana ja muodostaminen lopetetaan heti kun havaitaan virhe.

Jos esimerkiksi Harri Hakkerin pankkiohjelmiston mallista verifioidaan lennosta, että jokaista nostoa vastaa samansuuruinen saldon väheneminen ja päinvastoin, niin tila-avaruuden muodostaminen lopetetaan heti, kun nosto jää kirjaamatta ensimmäisen kerran. Tavallisessa verifioinnissa tila-avaruuden muodostaminen ei loppuisi tähän vaan jatkuisi kenties hyvinkin pitkään, tuottaen sekä tiloja, jotka edustavat tapahtumien kehitystä virheen jälkeen, että tiloja, jotka edustavat niitä vaihtoehtoisia tapahtumakulkuja, joita ei ollut vielä tutkittu kun virhe ilmeni ensimmäisen kerran. Virheen ensimmäisen ilmenemisen jälkeen tutkittavia tiloja on monessa tapauksessa huomattavasti enemmän kuin sitä ennen tutkittavia. Verifioimalla lennosta säästetään siis huomattavasti aikaa ja muistia verrattuna siihen,

että tila-avaruus muodostetaan ensin kokonaan ja virheet paljastetaan tutkimalla valmista tila-avaruutta.

Osa lennosta verifioinnin tekniikoista kykenee ainakin osittain estämään niiden tilojen muodostamisen, joilla ei ole merkitystä tutkittavan ominaisuuden kannalta. Jos esimerkiksi halutaan tutkia, käynnistääkö ydinvoimalan säätöohjelmisto varmasti turvajärjestelmän ennen reaktoria, niin tutkimista ei tarvitse jatkaa tiloista, joissa turvajärjestelmä on käynnistetty. Osa lennosta verifioinnin menetelmistä tekee tämän automaattisesti tutkittavan ominaisuuden ilmaisevan mallijärjestelmän tai aikalogiikan kaavan pohjalta. Verifiointimallin tekijäkin voi toki ohjelmoida tällaisen rajoituksen suoraan malliinsa, mutta automatiikka tekee sen varmemmin oikein.

Lennosta verifiointiin on lukuisia eri tekniikoita. Monien yksittäisten ominaisuuksien, kuten lukkiutumattomuuden, tarkastaminen lennosta verifioimalla on tietysti helppoa. Prosessialgebrojen tapauksessa esijärjestykset yhdistettynä Ed Brinksman vuonna 1987 ehdottamaan *kanonisten testaajien teoriaan* tarjoavat luontevan keinon monimutkaisten ominaisuuksien verifiointiin lennosta. Lineaarisen aikalogiikan kaavoina annetut väittämät voi kääntää niin sanotuiksi *Büchi-automaateiksi*, joita on kohtalaisen helppo käsitellä tila-avaruuden muodostamisen aikana, kuten Moshe Vardi ja Pierre Wolper ehdottivat vuonna 1986. Heille myönnettiin vuoden 2000 Gödel-palkinto tästä ja eräistä tähän liittyvistä, teoreettisemmista keksinnöistään [6]. Heidän nimissään on muitakin automaattisen verifioinnin keskeisiä tuloksia.

6.6 Tiivistetty tila bittivektorin indeksinä

Tilaräjähdystä voi hillitä myös valitsemalla tila-avaruudelle tavallisesta poikkeava esitystapa. Gerard Holzmann ehdotti vuonna 1987 tila-avaruuden esittämistä suurena bittitaulukkona, jota indeksoidaan tilan kuvauksesta hajautusfunktiolla muodostetulla tiivistelmällä [8]. Aluksi taulukon kaikki bitit ovat nollia. Sitä mukaa kuin tiloja löytyy, niitä vastaavat bitit asetetaan ykköseksi.

Koska taulukon indeksi on muodostettu hajautusfunktiolla tiivistämällä, monta eri tilaa kuvautuu samalle bitille. Tästä seuraa, että tila saatetaan tulkita jo löydetyksi, vaikka se on todellisuudessa uusi. Käyttäytymisen jatko kyseisestä tilasta alkaen jää silloin tutkimatta. Holzmannin menetelmä ei siis takaa tulosten kattavuutta, eikä ole verifiointimenetelmä sanan normaalissa merkityksessä.

Holzmannin menetelmän kyky löytää virheitä on kuitenkin samantapainen kuin verifiointimenetelmien. Se on hyvä menetelmä silloin, kun tavoitteena on täydellisen kattavuuden sijasta tutkia niin paljon kuin tietyn ennalta määrätyn muistin rajoissa on mahdollista. Holzmannin menetelmä valikoi tutkimatta jäävät tila-avaruuden alueet melko satunnaisesti. Lisäksi menetelmä käyttää sille annettua muistia todella tehokkaasti: jos bittitaulukko täyttyy edes puoliksi, muistia tarvitaan vain kaksi bittiä tutkittua tilaa kohti! On tosin huomattava, että kuten tila-avaruusmenetelmät yleensä, myös Holzmannin menetelmä tarvitsee merkittäviä määriä lisää muistia pitääkseen kirjaa keskeneräisestä työstä.

Koska Holzmannin menetelmä jättää melko mielivaltaisesti tiloja tutkimatta, voisi luulla, että sitä ei voi käyttää monimutkaisten, useista tiloista ja niiden keskinäisistä suhteista riippuvien

ominaisuuksien tarkastamiseen. Näin ei ole. Holzmann on itsekin kehittänyt menetelmänsä kanssa yhteensopivia monimutkaisten ominaisuuksien tarkastusalgoritmeja. Courcoubetiksen, Vardin, Wolperin ja Yannakakiksen nerokas algoritmi vuodelta 1990 yhdisti Büchi-automaatit Holzmannin menetelmään ja siten toi lineaarisen aikalogiikan kokonaisuudessaan Holzmannin menetelmän piiriin.

6.7 Tapahtumien riippumattomuuden hyödyntäminen

Jos rinnakkaisjärjestelmän osat voivat edetä yhtäaikaa ilman, että ne vuorovaikuttavat toisiinsa, niin tila-avaruus kasvaa nopeasti. Jos esimerkiksi neljä osaa voi kukin suorittaa viisi askelta toisistaan riippumatta, niin yhteensä syntyy $(5+1)^4 =$ 1296 tilaa. Etenemisjärjestyksellä ei kuitenkaan yleensä ole verifioinnin kannalta merkitystä, joten riittäisi tutkia ne tilat, jotka saataisiin yhdellä etenemisjärjestyksellä. Niitä on vain 21 kappaletta.

Tämä ilmiö on tunnettu niin kauan kuin rinnakkaisjärjestelmiä on tutkittu. Yksinkertaisissa tapauksissa se on eliminoitavissa niin sanotulla *atomisuuden harventamisella*: jos jono prosessin suoritusaskelia on riippumaton muista prosesseista, jonon askeleet voidaan yhdistää yhdeksi isoksi askeleeksi, joka voidaan vielä yhdistää jonoa edeltävän askeleen kanssa. Tällainen yhdistäminen on tavallista jo mallia laadittaessa.

Atomisuuden harventaminen on kuitenkin vasta ensiapu, sillä sitä voi käyttää vain kun askel on riippumaton kaikista muista prosesseista. Usein askel riippuu yhdestä tai kahdesta muusta prosessista, mutta on riippumaton lopuista. Jos tila-avaruus muodostetaan vaiheittain, voidaan prosessiryhmän riippumattomuutta muusta maailmasta hyödyntää kun ryhmän tila-avaruus on muodostettu. Itse asiassa prosessiryhmän tila-avaruuden kutistamisessa on paljolti kyse juuri tästä. Tämäkään keino ei kuitenkaan aina riitä.

Yleispätevämmän keinon hyödyntää tapahtumien riippumattomuutta automaattisesti julkaisivat toisistaan riippumatta Antti Valmari vuonna 1988, edellämainitun Pierre Wolperin oppilas Patrice Godefroid 1990 sekä Doron Peled 1993. (Peledin työ pohjautui Shmuel Katzin ja hänen työhönsä vuodelta 1988, jossa samankaltaisia ajatuksia sovellettiin käsikäyttöiseen verifiointiin.) Heidän menetelmänsä ovat keskenään varsin samankaltaisia. Samat ja muut tutkijat ovat sittemmin kehittäneet niitä edelleen.

Valmari kutsui menetelmäänsä *itsepäisiksi joukoiksi* (stubborn sets), Godefroid käytti nimeä “persistent sets” ja Peled “ample sets”. Omituiselta kuulostava nimi aiheutuu siitä, että itsepäinen (tai persistent tai ample) joukko edustaa rinnakkaisjärjestelmän osaa, jonka seuraavaan askeleeseen ympäristö voi vaikuttaa hyvin vähän. Se siis tekee mitä tahtoo muista piittaamatta. Itsepäiset joukot vaihtuvat sitä mukaa kuin suoritus etenee, minkä vuoksi itsepäisten joukkojen menetelmä pystyy koko ajan hyödyntämään juuri senhetkisessä tilassa vallitsevia riippumattomuuksia.

Paitsi että rinnakkaisjärjestelmän osat voivat vuorovaikuttaa, tiettyjen tapahtumien keskinäisellä järjestyksellä saattaa olla välitöntä merkitystä verifioitavan ominaisuuden kannalta. Tämän vuoksi itsepäisten joukkojen menetelmästä on jouduttu kehittämään useita versioita erilaisia ominaisuuksien perheitä varten. Esimerkiksi lineaariselle aikalogiikalle on oma menetelmänsä ja CTL:lle toinen. Mitä pienempää ominaisuusperhettä varten menetelmä on kehitetty, sitä tehokkaammin

se pystyy pienentämään tila-avaruutta.

Menetelmien kirjoa on lisännyt se, että itsepäisten joukkojen muodostaminen on jossain määrin konstikasta, minkä vuoksi aina ei ole selvää, miten itsepäinen joukko olisi paras muodostaa. Varsinkin Peled on suosinut yksinkertaisia, helposti ohjelmoitavia, nopeita tapoja, Valmarin ehdotettua monimutkaisia tapoja, joita ei ole ihan helppo ohjelmoida. Monimutkaisemmat tavat ovat yleensä hieman hitaampia, mutta niillä löydetään parempia itsepäisiä joukkoja.

Jokin itsepäisten joukkojen kaltaisten menetelmien versio on toteutettu muun muassa Holzmannin Spin-työkalussa ja Teknillisen korkeakoulun Prodissa [18]. Lievästi vanhentunut katsaus itsepäisten joukkojen kaltaisiin menetelmiin sisältyy artikkeliin [24]. Siinä käytetään Teknillisen korkeakoulun tutkija Marko Rauhamaan kehittämää eleganttia ajattelutapaa, jonka Kimmo Varpaaniemi osoitti varsin yleispäteväksi.

Tapahtumien riippumattomuutta on hyödynnetty myös Godefroidin keksimässä *unijoukkojen menetelmässä*, joka sopii hyvin täydentämään itsepäisiä joukkoja. Sen teoriaa ei ole kehitetty kovin pitkälle.

Kokonaan oma lukunsa on Ken McMillanin 1992 julkaisema ja erityisesti Javier Esparzan edelleen kehittämä “unfolding” eli “auki kelaus” -menetelmä, jossa tila-avaruuden tilalla on monimutkainen haarautuva rakenne, jossa eri prosessien osuudet rinnakkaisjärjestelmän suorituksesta pidetään osittain erillään. Suomessa sitä ovat tutkineet erityisesti Johan Lilius ja Keijo Heljanko.

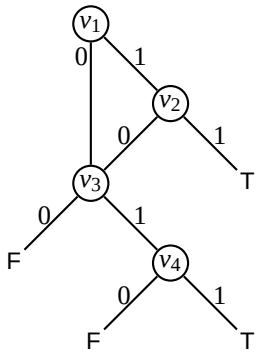
6.8 Binääripäätöskaaviot

Erityisesti mikropiirien verifioinnissa on käytetty paljon niin sanottuja *normalisoituja järjestettyjä binääripäätöskaavioita* (reduced ordered binary decision diagram, ROBDD, OBDD tai vain BDD). BDD on suunnattu silmukaton graafi, joka esittää propositiologiikan kaavan, jossa on muuttujasymboleita. Propositiologiikan kaava on lauseke, jonka muuttujat, välitulokset ja lopputulos saavat arvonsa totuusarvojen joukosta $\{\mathsf{False}, \mathsf{True}\}$ eli $\{0, 1\}$. Propositiologiikan kaavoissa esiintyy yleensä operaattoreita “$\wedge$” (ja), “$\vee$” (tai) ja “$\neg$” (ei). Muitakin operaattoreita voidaan käyttää.

Kuvassa 5 on eräs BDD. BDD:n solmun nimenä voi olla False, True tai muuttujasymboli. False- ja True-solmuista ei lähde kaaria eteenpäin. Jokaisesta muuttujasymbolilla nimetystä solmusta lähtee kaksi kaarta, joita voi kutsua nimillä *0-kaari* ja *1-kaari*. Kutsumme näitä solmuja *sisäsolmuiksi*. Muuttujasymboleille on määritelty järjestys. Jos kaari yhdistää kaksi sisäsolmua, niin alkupäässä olevan muuttujasymbolin tulee olla järjestyksessä ennen loppupään symbolia.

Yksi BDD:n solmuista on nimeltään *juuri*. BDD:n esittämän lausekkeen arvo annetuilla muuttujasymboleiden arvoilla voidaan laskea aloittamalla juuresta, etenemällä 0- ja 1-kaaria pitkin sen mukaan mikä arvo kulloinkin kohdattua sisäsolmua vastaavalla muuttujasymbolilla on, ja katsomalla, päädytäänkö False- vai True-solmuun.

BDD:t *normalisoidaan* ennen käyttöä, mikä tarkoittaa seuraavaa. Rakenteeltaan samanlaiset osat yhdistetään ja siis talletetaan muistiin vain yhtenä kappaleena. Lisäksi ne sisäsolmut ohitetaan ja poistetaan, joista lähtevät kaaret päättyvät samaan solmuun. Normalisointi voi kuulostaa monimutkaiselta, mutta todellisuudessa se saadaan tehtyä tehokkaasti etenemällä “takaperin”. Ensin sulautetaan kaikki False-solmut yhdeksi solmuksi ja kaikki True-solmut yhdeksi solmuksi. Sitten nor-



Kuva 5: BDD, joka esittää kaavan $(v_1 \wedge v_2) \vee (v_3 \wedge v_4)$.

malisoidaan aina sellaiset sisäsolmut, joista lähtevien kaarten toisessa päässä olevat solmut on jo normalisoitu. Usein BDD normalisoidaan sitä mukaa kuin se syntyy jonkin laskutoimituksen tuloksena.

Jos on annettu kaksi mahdollisesti normalisoitua BDD:tä b_1 ja b_2 sekä muuttujasymboli v, niin on helppo muodostaa ja normalisoida BDD:t, jotka edustavat lausekkeita $b_1 \wedge b_2$, $b_1 \vee b_2$, $\neg b_1$ sekä väittämiä $v = 0$ ja $v = 1$.

Kaksi propositiologiikan kaavaa esittää samaa loogista funktiota, jos ja vain jos niitä esittävät normalisoidut BDD:t ovat rakenteeltaan samanlaiset. Normalisointi voidaan viedä niin pitkälle, että jos kahdella eri BDD:llä on rakenteeltaan samanlainen osa, niin nekin sulautetaan yhteen. Jos näin tehdään, niin on äärimmäisen helppo testata, esittääkö kaksi BDD:tä samaa funktiota: testataan vain, ovatko niiden juuret sama solmu.

NP-täydellisyyden teoriaa tunteva lukija voi tässä vaiheessa ihmetellä, eikö kerrotusta seuraa, että BDD:illä voi tarkastaa aina tehokkaasti, onko annettu propositiologiikan kaava tyydytettävissä. Senhän uskotaan olevan mahdotonta. Asian selitys on siinä, että esimerkiksi lauseketta $b_1 \wedge b_2$ edustavan BDD:n koko on (todistettavasti) huonoimmillaan verrannollinen alkuperäisten BDD:ien kokojen tuloon eikä summaan.

BDD:iden käyttö verifioinnissa perustuu siihen, että propositiologiikan kaava voidaan tulkita joukon kuvaukseksi olettamalla, että alkiot esitetään määrämittaisina bittivektoreina ja jokainen muuttujasymboli edustaa yhtä bittiä alkion esityksessä. Rinnakkaisjärjestelmän tilaavaruuden keskeisin osa on niiden tilojen joukko, joihin järjestelmä voi toimiessaan joutua. Joukon esittäminen verifioinnissa normaalisti käytettävillä tavoilla (muun muassa hajautustaulu ja binääripuu) vie sitä enemmän muistia, mitä enemmän alkioita joukossa on. BDD:ille tämä ei päde. Niinpä BDD:illä on mahdollista esittää monia erittäin suuria joukkoja kohtuullisessa määrässä muistia.

Mahdollisuudesta esittää erittäin suuri tilojen joukko ei olisi paljoa iloa, jos tilat pitäisi lisätä joukkoon yksi kerrallaan, kuten normaaleissa verifiointimenetelmissä tehdään. BDD:itä käytettäessä tämä ongelma ratkaistaan esittämällä myös järjestelmän *tilasiirtymärelaatio* BDD:nä. Tilasiirtymärelaatio on abstrakti

yhteenveto järjestelmän ohjelmakoodista. BDD:illä verifioitaessa järjestelmää ikään kuin suoritetaan yhtäaikaa kaikista siihen mennessä löydetyistä tiloista alkaen. Sen vuoksi uusia tiloja löydetään monta kerrallaan, ja suurikin tila-avaruus voidaan tutkia kohtuullisessa ajassa.

Joskus BDD:itä käytetään esittämään niiden tilojen joukko, joihin tutkittava järjestelmä voi joutua. Tavallisempaa on kuitenkin esittää niiden tilojen joukko, joista alkaen järjestelmä voi päätyä johonkin kiellettyyn tilanteeseen. Se muodostetaan aloittamalla kiellettyjen tilojen kuvauksesta ja ikään kuin suorittamalla järjestelmää takaperin. Lopuksi tätä joukkoa verrataan järjestelmän mahdollisiin alkutiloihin. Tätä lähestymistapaa kutsutaan *symboliseksi mallintarkastukseksi*.

BDD:illa voi siis käsitellä tila-avaruuksia, jotka ovat monta kertaluokkaa suurempia kuin mihin päästään tavallisilla esitystavoilla. Valitettavasti tässäkin paratiisissa on käärme. Informaatioteoriasta seuraa, että olipa joukon esitystapa tietokoneen muistissa mikä tahansa, niin vain häviävän pieni osa suurista joukoista voidaan esittää pienellä muistilla. BDD:iden menestys riippuu siitä, esittävätkö ne pienellä muistilla juuri ne joukot, joita verifioinnissa tarvitaan.

Tulokset ovat olleet vaihtelevia. Varsinkin tilasiirtymärelaatioista tulee hyvin usein sietämättömän suuria BDD:ita. Tämän ongelman ratkaisemiseksi on kehitetty monenmoisia vippaskonsteja, joiden ansiosta BDD:t on saatu toimimaan kohtalaisen hyvin varsinkin digitaalipiirien verifioinnissa. Ohjelmien verifioinnissa BDD:iden menestys on ollut heikompi.

BDD:iden käyttö verifioinnissa on Ed Clarken oppilaan Ken McMillanin ajatus vuodelta 1987. Perusasiat on esitelty kirjassa [4], ja [15] on syvällisempi katsaus aihealueeseen.

6.9 Rajoitettu mallintarkastus

Rajoitettu mallintarkastus esiteltiin vuonna 1999 ja on sen jälkeen saavuttanut suuren suosion. Varsinaisesti sillä tarkoitetaan menetelmiä, joissa järjestelmän toimintaa kuvaavaa mallia rajoitetaan siten, että se esittää järjestelmän toiminnan vain ennalta asetettuun suoritusaskelmäärään saakka. Rajoitettu mallintarkastus ei siis voi paljastaa virheitä, jotka ilmenevät vasta pitkän suorituksen jälkeen. Vastapainoksi se tarjoaa suorituskykyetuja ja soveltuu tavallista paremmin ohjelmien tarkastukseen, myös muiden kuin rinnakkaisohjelmien.

Rajoitetussa mallintarkastuksessa ohjelman kaikkien suoritusten alkuosa käännetään tyypillisesti propositiologiikan kaavaksi. Rikkaampaakin logiikkaa voidaan käyttää, mutta se on harvinaista.

Kunkin muuttujan arvon esittämiseen varataan niin monta propositiosymbolia kuin muuttujan esittämiseen tarvitaan bittejä. Aina kun muuttujaan sijoitetaan arvo tai testataan esimerkiksi **if**-lauseen ehtoa, otetaan käyttöön uudet propositiosymbolit ja lisätään kaavaan osa, joka ilmaisee, miten niiden arvot suhtautuvat vanhojen propositiosymboleiden arvoihin. Tämä kaavanosa on itse asiassa suoritetun lauseen tai testin toiminnan kuvaus ilmaistuna logiikan kielellä. Kaavaan liitetään myös osa, joka määrittelee muuttujien alkuarvot, sekä osa, joka kertoo, mitä ei saa tapahtua.

Sen jälkeen käytetään niin sanottua *toteutuvuustarkastinta* (satisfiability solver tai sat solver) etsimään propositiosymbolien arvoyhdistelmä, joka toteuttaa saadun kaavan. Tällainen arvoyhdistelmä kuvaa virheellisen suorituksen. Jos sellaista ei ole, niin suoritukset ovat virheet-

tömiä mallituksessa käytettyyn askelmäärään saakka.

Propositiologiikan kaavan toteutuvuuden tarkastaminen on klassinen **NP**-täydellinen tehtävä, ja käännöksessä syntyvä kaava on kooltaan valtava, joten lähestymistapa saattaa kuulostaa toivottomalta. Mutta tässäkin pätee se verifioinnissa monesti havaittu ilmiö, että menetelmä toimii käytännössä paljon paremmin kuin vaativuusteoreettinen, huonointa tapausta kuvaava tulos ennustaa. Nykyaikaiset toteutuvuustarkastimet tutkivat kaavaa tarkoituksenmukaisessa järjestyksessä ja käyttävät monenlaisia heuristiikkoja, joiden ansiosta ne saattavat selvitä jopa satojatuhansia muuttujia sisältävistä kaavoista.

PSPACE-täydellinen verifiointitehtävä on siis korvattu **NP**-täydellisellä tehtävällä. Käytännössä hyöty tulee siitä, että vaikka toteutuvuuden tarkastus tarvitsee paljon aikaa, se ei tarvitse kovin paljoa muistia. Tavalliset tilaavaruusmenetelmäthän käyttävät eksponentiaalisesti muistia.

Rajoitettua mallintarkastusta on esitelty katsauksessa [2]. Toteutuvuustarkastinten nykytilannetta on tarkasteltu suomeksi artikkelissa [11]. Artikkeli sisältää myös lyhyen kuvauksen rajoitetusta mallintarkastuksesta. Toni Jussila väitteli rajoitetusta mallintarkastuksesta Teknillisessä korkeakoulussa vuonna 2005.

7 Verifioinnin näkymiä

Kuten edeltä epäilemättä kävi ilmi, automaattinen verifiointi on vaikeaa. Vaikka alalla on lähes kolmenkymmenen viime vuoden aikana tehty lukuisia nerokkaita keksintöjä, verifiointiautomaatit ovat yhä liian tehottomia ja vaikeakäyttöisiä annettavaksi jokaisen insinöörin käteen.

Verifiointiautomaattien kaupallinen käyttö on pisimmällä mikropiirien suunnittelussa. Syitä tähän pohdittiin sivulla 53. Esimerkiksi Intelin suhtautuminen verifiointiin muuttui merkittävästi myönteisemmäksi, kun sen Pentium-prosessorin liukulukuaritmetiikasta löytyi vuonna 1994 kiusallinen ja yhtiölle todella kalliiksi käynyt virhe “Pentium FDIV bug”. Mikropiiriteollisuus käyttää ennenkaikkea BDD-pohjaisia menetelmiä. Se ei yleensä pyri kattavaan verifiointiin, vaan käyttää verifiointia testauksen, simuloinnin ja muiden menetelmien täydentäjänä.

Intelillä on nykyisin merkittävä verifiointiyksikkö, ja suomalainen Roope Kaivola on siellä tärkeässä roolissa. Kaivolan ryhmä verifioi merkittävän kokoisia mikroprosessorin osakokonaisuuksia kattavasti käyttäen itse kehittämäänsä menetelmää ja sen apuna muun muassa BDD-teknologiaa. Kaivolan ryhmän menetelmä perustuu verifioitavan kohteen huolelliseen ymmärtämiseen ja vaatii paljon ihmistyötä, mutta lopputuloksena on “resepti”, jonka mukaan kohde voidaan verifioida täysin koneellisesti.

Ohjelmistopuolella kokemukset ovat olleet, että melkein aina kun verifiointiautomaatteja on käytetty, on löydetty ennen havaitsemattomia virheitä. On selvää, että vaikka verifiointiautomaattien teho ei riitä nykyaikaisten isojen järjestelmien täysimittaiseen verifiointiin, niiden teho ylittää selkeästi sekä katselmoivien insinöörien että testausmenetelmien kyvyn löytää rinnakkaisuuden hallintaan liittyviä virheitä. Tästä huolimatta automaattista verifiointia käytetään ohjelmistoteollisuudessa vähän, ja se yleistyy hitaasti. Suomalaisista yrityksistä automaattista verifiointia on käyttänyt lähinnä Nokian tutkimuskeskus.

Teollisuuden haluttomuuden selityksenä saattaa olla se tosiasia, että helppokäyttöisimpienkin verifiointiautomaattien käyttö vaatii erikoiskoulutettuja ihmisiä.

Perusteellisella testauksella saadaan yleisimmät virheet karsittua, ohjelmien käyttäjät ovat oppineet sietämään harvemmin esiintyviä virheitä, eikä verifiointi kuitenkaan poistaisi kuin tietynlajisia virheitä eikä niitäkään kaikkia. Verifioinnin tuomasta lisäavusta ei siis kannata maksaa mitä tahansa. Kuten aina uusilla teknologioilla, kustannukset tulisivat kuitenkin olennaisesti laskemaan, jos käyttö alkaisi yleistyä. Ehkä teollisuuden suhtautumisessa on mukana ripaus ennakkoluuloisuuttakin, onhan verifiointi luonteeltaan kovin toisenlaista kuin päivittäinen ohjelmistotyö.

Verifiointiosaamista on viime aikoina sovellettu muun muassa Ranskassa ja Suomessa myös testausmenetelmien kehittämiseen. Julkaisut [13, 12] ovat suomenkielisiä esimerkkejä tästä. Vielä on liian varhaista sanoa, tuleeko teollisuus saamaan merkittävästi hyötyä tätä kautta. Suomessa tämän alan edelläkävijä on Conformiq-niminen yritys. Sen teknologiajohtaja Antti Huima kutsuttiin pitämään toinen vuoden 2007 Testcom/Fateskonferenssin pääesitelmistä. Kyseessä on edistyneisiin testausmenetelmiin keskittyvä konferenssi.

Vaikka verifiointitutkimuksen välitön hyöty Suomelle on siis jäänyt pieneksi, on siitä todennäköisesti ollut merkittävää välillistä hyötyä sillä tavalla kuin hyvällä perustutkimuksella on. Tietojenkäsittelytieteen osa-alueena verifiointi on mitä mainioin. Sen kansainvälistä tieteellistä merkitystä todistavat sen tutkijoiden saamat arvovaltaiset palkinnot, joista edellä on kerrottu.

Verifiointitutkimuksessa sovelletaan muun muassa logiikan, automaattiteorian, algoritmiikan, laskennallisen vaativuuden teorian sekä ohjelmointikielten suunnittelun ja kääntämisen teorian tuloksia, minkä vuoksi verifiointitutkija joutuu hankkimaan sekä hyvät pohjatiedot teoreettisesta tietojenkäsittelytieteestä että kyvyn laatia tehokkaita poikkeuksellisen monimutkaisia ohjelmia. Verifiointitutkijan tuloksia mitataan viime kädessä verifiointikokeilla, minkä vuoksi hänen täytyy omaksua pragmaattisempi asenne kuin puhtaan teoreetikon. Näin ollen verifiointitutkimus on omalta osaltaan ylläpitämässä ja kehittämässä tietojenkäsittelyteorian ja sen soveltamisen osaamista Suomessa.

Viitteet

[1] http://www.acm.org/awards/ [15.6.2007] (ACM:n palkintojen kotisivu, jonka kautta löytyvät muun muassa Turing-palkinto (taward.html) ja Software System -palkinto (ssaward.html).)

[2] Biere, A., Cimatti, A., Clarke, E. M., Strichman, O., & Zhu, Y.: “Bounded Model Checking”. *Highly Dependable Software*, Advances in Computers 58, Elsevier 2003, ss. 118–149. (Rajoitetun mallintarkastuksen keksijöiden laatima yleisesittely.)

[3] Bolognesi, T. & Brinksma, E.: “Introduction to the ISO Specification Language LOTOS”. *Computer Networks and ISDN Systems* 14 (1987), ss. 25–59. (Helppolukuinen johdatus Lotos-kieleen.)

[4] Clarke, E. M., Grumberg, O. & Peled, D. A.: *Model Checking*. The MIT Press 1999, 314 s. (Aikalogiikoihin (varsinkin CTL:ään) ja BDD-teknologiaan keskittyvä verifiointimenetelmien oppikirja.)

[5] Garey, M. R. & Johnson, D. S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979, 340 s. (Mainio ja korkeasta iästään huolimatta yhä pätevä johdatus laskennallisesti vaativien tehtävien maailmaan.)

[6] http://www.eatcs.org/activities/awards/goedel.html [15.6.2007] (Gödel-palkinnon kotisivu. Palkinnon jakaa European Association for Theoretical Computer Science.)

[7] Hoare, C. A. R.: *Communicating Sequential Processes*. Prentice-Hall 1985, 256 s. (Klassinen, mutta vanhentunut perusteos CSP-teoriasta. Soveltajia ajatellen kirja on kovin abstrakti. Teoreetikoiden kannalta kirja on niukka.)

[8] Holzmann, G. J.: *Design and Validation of Computer Protocols*. Prentice-Hall 1991, 500 s. (Verifiointiin painottuva yleisesitys tietoliikenneprotokollien suunnittelusta. Kirja selostaa erinomaisesti Holzmannin omia siihenastisia tuloksia, joista useilla on ollut suuri merkitys automaattisen verifioinnin kehittymiselle. Muiden asioiden kohdalla kirja on heikkotasoinen ja sisältää olennaisia virheitä.)

[9] Holzmann, G. J.: "From Code to Models". Kutsuttu esitelmä, *Proceedings of 2nd IEEE International Conference on Application of Concurrency to System Design*, Newcastle upon Tyne, U.K., IEEE Computer Society 2001, ss. 3–10. (Esittelee järjestelmän, joka kääntää C-kielisiä tietoliikenneohjelmia verifiointimalleiksi käyttäjän antamien abstrahointisääntöjen avulla.)

[10] Jensen, K.: *Coloured Petri Nets*. *Volume 1: Basic Concepts* (2nd ed.), 1997, 234 s. *Volume 2: Analysis Methods* (2nd ed.), 1997, 174 s. *Volume 3: Practical Use*, 1997, 265 s. EATCS Monographs in Theoretical Computer Science, Springer-Verlag. (Helppotajuinen väritettyjen Petriverkkojen perusoppikirjasarja. Osassa 2 käsitellään automaattisia verifiointimenetelmiä, ja osassa 3 käydään läpi todellisia käyttöesimerkkejä.)

[11] Järvisalo, M.: "Lauselogiikan toteutuvuustarkastus: käytännönläheistä teoriaa". *Tietojenkäsittelytiede* 22, Joulukuu 2004, ss. 47–63. (Esittelee toteutuvuustarkastimen tehtävän, kaksi käyttöaluetta, tekniikkaa, arviointimenetelmiä ja muunnelmia. Sisältää laajan lähdeluettelon.)

[12] Kellomäki, T.: "Historialaajennus mallipohjaisessa testauksessa". *Tietojenkäsittelytiede* 24, Joulukuu 2005, ss. 8–21. (Mallipohjaisessa testauksessa käytetään samantapaisia spesifikaatioita kuin verifioinnissa. Niistä saadaan testisyötteet automaattisesti, ja myös vasteiden oikeellisuus tarkastetaan automaattisesti. Artikkeli vertaa vaihtoehtoisia tapoja löytää virheitä, joiden ilmenemiseksi ei riitä, että alkuperäinen spesifikaatio katetaan täydellisesti mutta vain kertaalleen.)

[13] Kervinen, A. & Virolainen, P.: "Konferenssiprotokollan automaattinen testaus". *Tietojenkäsittelytiede* 22, Joulukuu 2004, ss. 35–46. (Esittelee kokeen, jossa todellisella ohjelmointikielellä kirjoitettu ja käyttöjärjestelmän tasolla rinnakkain ajettu ohjelmisto testattiin automaattisesti abstraktia tilakonemuotoista spesifikaatiota vastaan. Spesifikaatio muodostettiin verifioinnille tyypillisin keinoin yksinkertaisista osista.)

[14] Manna, Z. & Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems, Volume I: Specification*. Springer-Verlag 1992, 427 s. (Helppotajuinen lineaarisen aikalogiikan perusoppikirja. Sarjan jatko-osat keskittyvät ihmistyönä tapahtuvaan verifiointiin.)

[15] Meinel, C. & Theobald, T.: "Ordered Binary Decision Diagrams and Their Significance in Computer-Aided Design of VLSI Circuits". *Bulletin of the European Association for Theoretical Computer Science* 64, 1998, ss. 171–187. (Syvällisempi ja monipuolisempi katsaus BDD-maailmaan kuin kirjassa [4]. Keskittyy piirisuunnittelusovelluksiin.)

[16] Milner, R.: *Communication and Concurrency*. Prentice-Hall 1989, 260 s. (CCS-

kieltä sekä vahvaa ja heikkoa havaintoekvivalenssia käsittelevä perusteos.)

[17] http://www.pankkiyhdistys.fi/ [15.6.2007]

[18] http://www.tcs.hut.fi/Software/prod/ [15.6.2007] (Prod-työkalun kotisivu.)

[19] Roscoe, A. W.: *The Theory and Practice of Concurrency*. Prentice-Hall 1998, 565 s. (CSP-teorian yleisesitys. Sisältää sekä soveltajia että teoreetikkoja varten kirjoitetut jaksot ja esittelee CSP:tä varten kehitetyn FDR-nimisen verifiointityökalun.)

[20] Roscoe A. W.: "Seeing Beyond Divergence". *Communicating Sequential Processes, The First 25 Years*, Lecture Notes in Computer Science 3525, Springer-Verlag 2005, ss. 15–35. (Pohtii CFFD-semantiikan CSP-mäisen määrittelemisen ongelmaa ja johtaa CFFD:tä erittäin paljon muistuttavan, alunperin Antti Puhakan ja Antti Valmarin vuonna 1999 julkaiseman ekvivalenssin CSP-mäisesti. Matemaattisesti haastavaa luettavaa.)

[21] Siirtola, A.: "Ohjelmistojen äärellistilaisten, formaalisti oikeiden mallien tuottaminen automaattisesti". *Tietojenkäsittelytiede* 24, Joulukuu 2005, ss. 22–34. (Tarkastelee datariippumattomuuden hyödyntämistä ensin yleisesti ja sen jälkeen esittelee kirjoittajan oman lähestymistavan.)

[22] Tampereen teknillinen yliopisto, ohjelmistotekniikan laitos: *Verification Algorithm Research Group*. http://www.cs.tut.fi/ohj/VARG/ [15.6.2007] (Ryhmän kotisivu.)

[23] Valmari, A.: "Failure-based Equivalences Are Faster Than Many Believe". *Proceedings of Structures in Concurrency Theory 1995*, Springer-Verlag "Workshops in Computing" series, 1995, ss. 326–340. (Kyseenalaistaa kirjoittamisajankohtanaan yleisesti vaikuttaneen näkemyksen, jonka mukaan verifiointi on haarautuvan ajan tapauksessa laskennallisesti helpompaa kuin lineaarisen ajan tapauksessa. Herätti jonkin verran kiivasta mutta rehtiä keskustelua.)

[24] Valmari, A.: "The State Explosion Problem". *Lectures on Petri Nets I: Basic Models*, Lecture Notes in Computer Science 1491, Springer-Verlag 1998, ss. 429–528. (Monipuolinen katsaus tilaräjähdyksen hillitsemiseksi kehitettyihin keinoihin ja niiden teoriaan. Sisältää laaja-alaisen tiiviin rinnakkaisjärjestelmien algoritmisen verifioinnin menetelmien yleisesittelyn.)

[25] Valmari, A.: "A Chaos-Free Failures Divergences Semantics with Applications to Verification". *Millennial Perspectives in Computer Science, Proceedings of the 1999 Oxford–Microsoft Symposium in Honour of sir Tony Hoare*, Palgrave 2000, ss. 365–382. (Näyttää, miten CFFD-semantiikka putkahtaa esiin yrityksestä johtaa CSP:n kaltainen semantiikka, jossa kongruenssiongelma on ratkaistu verifiointisovellusten kannalta mielekkäämmin kuin CSP:ssä. Esittelee CFFD:n, NDFD:n ja lineaarisen aikalogiikan välisen yhteyden.)

[26] Valmari, A.: "Composition and Abstraction". Cassez, F., Jard., C., Rozoy, B. & Ryan, M. (toim.): *Modelling and Verification of Parallel Processes*. LNCS Tutorials, Lecture Notes in Computer Science 2067, Springer-Verlag 2001, ss. 58–99. (Lyhyt oppikirjamainen johdatus prosessialgebralliseen verifiointiin. Painottaa tehostettuja verifiointimenetelmiä, jotka perustuvat tila-avaruuden vaiheittaiseen muodostamiseen ja osatila-avaruuksien kutistamiseen.)

[27] Valmari, A. & Setälä, M.: "Visual Verification of Safety and Liveness". *Proceedings of Formal Methods Europe '96: Industrial Benefit and Advances in Formal Methods*, Lecture Notes in Computer Science 1051, Springer-Verlag 1996, ss. 228–247. Saatavana myös TTKK:n

Ohjelmistotekniikan laitoksen raporttina nro 17, 1995. (Visuaalisen verifioinnin ja sen taustateorian esittely käyttäen esimerkkinä Petersonin keskinäisen poissulkemisen algoritmia.)

[28] Valmari, A. & Tienari, M.: "Compositional Failure-Based Semantic Models for Basic LOTOS". *Formal Aspects of Computing* (1995) 7: 440–468. (CFFD- ja NDFD-teorian peruslähde.)

[29] van Glabbeek, R. J. & Weijland, W. P.: "Branching Time and Abstraction in Bisimulation Semantics". *Journal of the ACM*, 43(3) 1996, ss. 555–600. (Haarautuvan bisimilaarisuuden vuoden 1989 alkuperäisartikkelin lehtiversio.)