Tietojenkäsittelytiede **24** Joulukuu 2005 sivut 22–34 
Toimittaja: Jorma Tarhio

# Ohjelmistojen äärellistilaisten, formaalisti oikeiden mallien tuottaminen automaattisesti

Antti Siirtola
Oulun yliopisto
Tietojenkäsittelytieteiden laitos
antti.siirtola@tol.oulu.fi

## 1 Johdanto

Tietokoneiden laskentakapasiteetin kasvaminen on mahdollistanut paitsi yhä laajempien myös rakenteeltaan monimutkaisempien sovellusten suorittamisen ja tuottamisen. Testaamisesta on tullut vaikeampaa ja virheiden havaitsemisesta hankalampaa. Erityisen ongelmallisia perinteisten testausmenetelmien kannalta ovat rinnakkaiset sovellukset, koska tällaisen ohjelman osien suoritukset voivat limittyä keskenään lukemattomin eri tavoin, jolloin sovelluksen jokainen ajokerta saattaa olla erilainen.

Ongelman ratkaisemiseksi on kehitetty formaaleja tarkastus- eli verifiointimenetelmiä, joita käyttäen ohjelman jokainen suoritusvaihtoehto voidaan tutkia. Eräät näistä menetelmistä perustuvat päättelysääntöihin [21, 16], joiden soveltaminen vaatii yleensä hidasta ja virhealtista käsityötä. Siksipä ne eivät sovellu laajamittaiseen todellisten, suurten sovellusten tarkasteluun. Toiset, mallintarkastusmenetelmät [4, 7, 15, 23], sen sijaan ovat automaattisia, mutta ne edellyttävät yleensä tutkittavan sovelluksen tila-avaruuden olevan äärellinen, mikä ei ohjelmistojen kohdalla käytännössä pidä paikkaansa.

Mallintarkastuksen sovellettavuutta rajoittaa lisäksi tilaräjähdysongelma [27]; ohjelman tila-avaruus kasvaa pahimmassa tapauksessa eksponentiaalisesti suhteessa muuttujien ja rinnakkaisten komponenttien määrään, mikä tarkoittaa mallintarkastuksen olevan käyttökelpoinen vain hyvin pieniä sovelluksia tutkittaessa. Ongelman kompensoimiseksi on esitetty useita menetelmiä [3, 7, 27], joista on kyllä hyötyä, mutta ne eivät ratkaise ongelmaa kokonaan. Kokemukseni perusteella järjestelmän voimakkaalta abstrahoinnilta, pelkistetyn mallin rakentamiselta, onkin käytännössä hankala välttyä [25].

Koska todellisuudessa sovellusten tila-avaruus on yleensä ääretön tai ainakin valtavan suuri, järjestelmäkuvausta tai ohjelmakoodia ei voi suoraan syöttää mallintarkastustyökaluun, vaikka se olisikin käännetty ohjelman ymmärtämään muotoon. Luotettavien verifiointitulosten saamiseksi analysointikohteen on kuitenkin vastattava todellisuutta. Siksipä myös mallin rakentaminen on voitava perustella formaalisti. Lisäksi tämä vaihe pitäisi pystyä suorittamaan mahdollisimman automaattisesti inhimillisten virheiden välttämiseksi sekä menetelmän laajamittaisen sovellettavuuden ja käytännön merkityksen takaamiseksi.

Oma tutkimukseni keskittyykin juuri tähän ongelmaan: äärellistilaisten mal-

lien tuottamiseen täysin tai lähes automaattisesti ohjelman kuvauksen perusteella. Lähtökohtanani on keskittää huomio vain muutamaan prosessiin ja kätkeä kaikki tapahtumat, jotka eivät suoraan vaikuta tarkasteltavaan ominaisuuteen, ohjelman oikeellisuuden määritelmään. Mikäli sovellus kohtelee eri prosesseja riittävän yhdenmukaisesti, muutamalle prosessille osoitetut ominaisuudet voidaan yleistää koskemaan myös muita. Varsinainen ongelma on kuitenkin tunnistaa syntaksin perusteella ne tapaukset, joissa muutamaa erikokoista äärellistilaista ohjelmaa tutkimalla voidaan osoittaa ominaisuuksia jopa äärettömän suurelle joukolle samankaltaisia järjestelmiä.

Koska teoriani on pääosin vasta kehitteillä, ongelmakenttää havainnollistetaan esimerkin avulla luvussa viisi, jossa käsitellään palvelin–asiakas-verkon mallintamista ja tämän tarkastelun yleistämistä. Tätä ennen tutustutaan kuitenkin alueella tehtyyn aikaisempaan tutkimukseen ja esitellään käytettävä formalismi. Ongelman täsmällisempi määrittely tehdään seuraavassa luvussa. Lopuksi kerron tutkimukseni taustalla olevista tavoitteista.

# 2 Tutkimusongelma

Abstrahointiongelma voidaan jakaa kahteen osaan: sovelluksen muuttamiseen tila-avaruudeltaan äärelliseksi malliksi, jotta automaattisia verifiointityökaluja voisi yleensä käyttää, ja mallin kutistamiseen tilaräjähdysongelman takia. Vaikka näistä jälkimmäinen on käytännössä erittäin merkittävä ongelma, äärellistilaisten mallien tarkastelu on kompleksisuudestaan huolimatta periaatteessa mahdollista nykyisin automaattisin menetelmin, minkä vuoksi en pidä jälkimmäistä ongelmaa yhtä perustavanlaatuisena kuin edellistä. Lisäksi koosta on järkevää puhua vain äärellisten rakenteiden kohdalla. Siksi keskitynkin ongelmista edelliseen: tila-avaruudeltaan äärellisten mallien tuottamiseen täysin tai lähes automaattisesti ohjelman kuvauksen perusteella.

Käytännön sovellusten tila-avaruuden äärettömyys johtuu siitä, että ohjelmien kuvauksissa esiintyy

1. kokonaislukuarvoisia tai muita muuttujia, joiden arvoalue on rajoittamaton;
2. mielivaltaisen suuria taulukoita tai dynaamisia tietorakenteita kuten listoja ja puita;
3. rajoittamaton määrä samankaltaisia entiteettejä, esimerkiksi olioita tai prosesseja, tai
4. niiden välisessä kommunikoinnissa käytettäviä puskureita, joiden kokoa ei ole määritelty.

Näistä ensimmäisen ratkaisemiseen on olemassa tehokkaita ja käyttökelpoisia menetelmiä [6, 19, 20, 30]. Lisäksi koen kolme jälkimmäistä ongelmaa mielenkiintoisemmiksi, joten keskityn tutkimuksessani niihin.

Tapauksista kolme jälkimmäistä ovat oikeastaan saman asian eri ilmentymiä; ohjelmassa voi olla mielivaltaisen monta rakenteeltaan samanlaista osaa, jotka ovat tapauksesta riippuen taulukon tai puskurin muistipaikkoja, tietorakenteen solmuja, olioita tai prosesseja. Ongelma tunnetaan myös äärettömän järjestelmäperheen verifiointina tai parametrisenä verifiointiongelmana, jossa samankaltaisten osien määrät ja tätä kautta sovelluksen täsmällinen rakenne määräytyvät parametrien perusteella. Kun parametrit vaihtelevat yli arvoalueidensa, saadaan järjestelmäkuvauksen instansseja, joista jokaisen tila-avaruus on äärellinen ja joista muodostuu ääretön järjestelmäperhe.

Pyrkimyksenäni on kehittää tehokkaita ja käyttökelpoisia algoritmeja järjestelmäperheen tai toisin sanoen parametrien arvoalueiden tuloavaruuden jakamiseen ekvivalenssiluokkiin, jolloin koko perheen mallintarkastus olisi mahdollista tehdä tutkimalla yksi edustaja kustakin ekvivalenssiluokasta. Luonnollisesti mielenkiintoni kohteena ovat sellaiset järjestelmäperheet, jotka voidaan jakaa äärelliseen määrään ekvivalenssiluokkia. Tämä edellyttää tietysti sitä, että ekvivalenssiluokkaan kuuluvat järjestelmät ovat riittävän samanlaisia. Se, mitkä instanssseista voidaan samaistaa keskenään, riippuu puolestaan järjestelmäkuvauksen lisäksi tutkittavasta ominaisuudesta, ohjelman oikeellisuuden määritelmästä.

Mitä yksinkertaisemmasta ominaisuudesta on kysymys, sitä enemmän on ominaisuuden kannalta tarpeetonta tietoa, jota voi hävittää, ja sitä erilaisempia järjestelmiä on mahdollista pitää keskenään ekvivalentteina [27]. Itse keskityn ominaisuuksiin, jotka määrittävät toiminnan kiinteälle määrälle komponentteja ja jotka näin ollen eivät muutu järjestelmän kasvaessa. Esimerkiksi vertaisverkkoja tutkittaessa ominaisuus voisi koskea kahden solmun välistä kommunikointia. Arvioin tällaisten oikeellisuusmääritysten kattavan suuren osan käytännön verifioinnissa tarvittavista ominaisuuksista.

# 3 Aiempia tuloksia

Parametrinen verifiointiongelma on yleisesti ratkeamaton, vaikka järjestelmäperheen jäsenet olisivatkin äärellistilaisia [1]. Joissakin tapauksissa ongelma kuitenkin on ratkeava. Yleisimmät lähestymistavat perustuvat induktion käyttöön ja ekvivalenssiluokkia vastaavien prosessiinvarianttien laskemiseen, joiden ominaisuudet säilyvät muuttumattomina, vaikka tiettyjä komponentteja lisättäisiinkin.

Clarke, Grumberg ja Jha käyttävät verkkokielioppia järjestelmien rakenteen määrittelemiseen ja esittävät menetelmän prosessi-invariantin laskemiseksi [5]. Valmarin lähestymistapa puolestaan toimii joillekin rengasrakenteille [29].

Dathin, Roscoen ja Brookesin lukkiutumien tarkastelussa käyttämä menetelmä ei edellytä verkolta erityistä topologiaa [23]. Kuvauskieltä rajataan sillä tavalla, että vierekkäisiä prosesseja tutkimalla voidaan yleensä selvittää, lukkiutuuko verkko. Tässä suhteessa menetelmä on kiinnostava ja lupaava, mutta epäilen sen soveltuvuutta muiden ominaisuuksien tarkasteluun. Kaikissa näissä menetelmissä on lisäksi sama ongelma: yksittäisten prosessien tila-avaruus tai haarautumisaste ei saa muuttua verkon koon mukana, joten esimerkiksi palvelin–asiakas-verkoille äärellistilaista mallia ei pystytä määrittämään.

Creesen kehittämässä induktiomenetelmässä tätä rajoitetta ei ole [9]. Sitä voi soveltaa useisiin eri verkkotopologioihin, mutta induktiossa käytettävän prosessiinvariantin keksiminen edellyttää tutkittavan järjestelmän perinpohjaista tuntemusta, ja se on laadittava käsityönä. Tämän vuoksi en pidä menetelmää kovin mullistavana, vaikka se onkin muuten automaattinen ja mahdollistaa useiden erityyppisten järjestelmien tutkimisen.

Yksinkertaisimman mutta käyttökelpoisimman menetelmän ovat mielestäni esittäneet Ip ja Dill, jotka abstrahoivat identtisten prosessien tilan periaatteessa tiputtamalla tilavektorista pois useammin kuin kaksi kertaa toistuvat komponentit [18]. Mikäli prosessit ovat äärellistilaisia, abstrakti tila-avaruus lakkaa jossain vaiheessa kasvamasta, vaikka identtisiä prosesseja lisättäisiinkin. Menetelmä mahdollistaa kuitenkin vain yksinkertaisten virhetilanteiden havaitsemisen. Lisäk-

si arvelen pienempiinkin invarianttiprosesseihin päästävän.

Oma lähestymistapani muistuttaa Ipin ja Dillin sekä Dathin, Roscoen ja Brookesin menetelmää siinä mielessä, että pyrin tekemään ekvivalenssiluokkiin jaon syntaksin perusteella. Tämä tähtää myös tilaräjähdyksen välttämiseen, sillä ongelmahan syntyy siitä, että muunnettaessa ohjelmakoodi mallintarkastustyökalujen ymmärtämään muotoon saadaan pahimmillaan kooltaan eksponentiaalisesti suurempi tilamalli. Vaikka muunnos joudutaan joka tapauksessa tekemään automaattisen verifioinnin yhteydessä, menetelmä helpottaa ja tehostaa myös muilla menetelmillä tapahtuvaa mallin analysointia. Tähän voidaan joutua, mikäli verifiointi mallin äärellistilaisuudesta huolimatta vie liikaa resursseja. Lisäksi perinteiset tilamallit eivät tarjoa tietoa siitä, miten ohjelman toiminta muuttuu komponenttien määrän mukana, joten tämänkin vuoksi syntaksitasolla toimiminen on perusteltua.

Ongelman voisi kiertää symbolisia tilamalleja käyttäen [19], mutta ainakaan tällä hetkellä työkalut eivät tue niitä. Ongelman ratkaiseminen symbolisten tilamallien tasolla mahdollistaisi kuitenkin tulosten yleistämisen helposti eri kuvauskielille, mutta ainakin aluksi pyrin toimimaan yhden formalismin sisällä.

# 4 Kuvauskieli

Syntaksi ja semantiikka tekevät karkeimman rajauksen tarkasteltaviin järjestelmiin ja ominaisuuksiin, joten niiden valinta ei ole yhdentekevää. Työssäni kuvauskielen merkitys korostuu vielä entisestään, koska tarkastelut tehdään pääsääntöisesti syntaksitasolla. Tarkoituksenani on rakentaa lähtökohtana olevan kuvauskielen päälle rajoitetumpi formalismi, jonka puitteissa käytännön sovellusten mallintaminen on mahdollista ja joka tarjoaa riittävästi lisätietoa, jotta ekvivalenssiluokat voidaan määrittää tehokkaasti syntaksin perusteella.

Perusformalismina toimii funktionaalinen prosessialgebrallinen mallinnuskieli CSP (Communicating Sequential Processes) [23]. Se ei tee eroa järjestelmien ja ominaisuuksien kuvaamisessa, vaan kaikki sen rakenteet ovat prosesseja. Tapahtumapohjaisuutensa vuoksi CSP soveltuu mielestäni tilaperustaisia lähestymistapoja luonnollisemmin ohjelmistojen mallintamiseen. Lisäksi se tarjoaa riittävät rakenteet todenmukaisten järjestelmien kuvaamiseen, abstraktiotason säätelemiseen ja verkon koostamiseen muutamaa osaa kopioimalla. CSP:n mallintarkastus suoritetaan yleensä vertaamalla järjestelmän mallia ominaisuutta kuvaavaan prosessiin.

CSP:n ongelmana teoreettisissa tarkasteluissa on sen löysästi määritelty syntaksi etenkin parametrien kohdalla, joten olen rajoittunut sen yksinkertaiseen, parametrittomaan versioon, josta myös muuttujat on poistettu. Tällainen kieli ei kuitenkaan ole paljon tilamalleja korkeammalla tasolla eikä sillä voi kuvata äärettömiä järjestelmäperheitä. Siksi olen rakentanut parametrittoman version päälle yksinkertaisen tyyppijärjestelmän [12, 19, 20], jolloin CSP toimii lähinnä siltana tilamalleihin ja denotationaalisiin tulkintoihin. Näistä jälkimmäiset kertovat, mitä prosessi tekee, kun taas edelliset keskittyvät myös siihen, miten eteneminen tapahtuu.

CSP:n denotaatioissa prosessit tulkitaan jälkien, estymien ja pillastumien joukkoina. Jälki on prosessin suorittama näkyvien tapahtumien jono. Estymä puolestaan muodostuu jäljestä sekä joukosta tapahtumia, joista prosessi voi kyseisen jäljen suoritettuaan kieltäytyä. Pillastuma taas on sellainen jälki, jonka jälkeen

prosessi voi pyöriä loputtomiin tekemättä kuitenkaan mitään näkyvää.

Jälki- ja estymämallit ovat denotationaalisia semantiikkoja, joissa prosessi tulkitaan vastaavasti sen jälkien joukkona sekä jälkien ja estymien joukkoina. Näissä tulkinnoissa turvallisuusominaisuuksien [2] ja estymämallissa myös lukkiutumien tutkiminen on mahdollista. Käytännössä tämä tarkoittaa, että virheellisten toimintojen suorittaminen voidaan havaita. Se ei kuitenkaan merkitse sitä, että prosessi tekisi oikeita toimenpiteitä; se ei vain tee mitään laitonta.

Mikäli virheiden havaitsemisen ohella halutaan varmistaa ohjelman myös etenevän, on kysymys elävyydestä [2], jolloin täytyy ottaa huomioon myös pillastumat sekä tila-avaruudeltaan rajoittamattomien järjestelmien kohdalla äärettömän pitkät jäljet. CSP:n tapa käsitellä näistä edellisiä on kuitenkin hieman ongelmallinen, koska abstrahoinnin ja kätkennän seurauksena malleihin tulee usein epätodellisia pillastumia ja CSP:n standardisemantiikassa prosessi voi tehdä pillastuman jälkeen mitä tahansa. Siksi elävyysominaisuuksia tutkittaessa on parempi turvautua esimerkiksi CFFD-semantiikkaan [28]. Toinen vaihtoehto on pyrkiä reiluihin malleihin, joissa epätodellisia pillastumia ei esiinny.

Yksi CSP:n tärkeimmistä ominaisuuksista on kuitenkin mahdollisuus kätkeä tarkastelun kannalta epäolennaiset tapahtumat. Tutkimukseni kannalta se tarkoittaa sitä, että sellaisten prosessien, jotka eivät vaikuta suoraan tutkittavaan ominaisuuteen, kaikki tapahtumat muutetaan näkymättömiksi. Myös ekvivalenssiluokkien määrittely on tällöin helppoa; yhteen luokkaan kuuluvilla prosesseilla on kätkennän jälkeen sama denotaatio.

Ongelman voi yleistää piilottamalla kaikki osat, joiden määrä riippuu parametrista. Mikäli ominaisuus kuitenkin edellyttää joidenkin tällaisten prosessien pysyvän näkyvinä, järjestelmän kuvausta voi algoritmisesti muuttaa sillä tavalla, että parametrista erotetaan kiinteä määrä alkioita yksittäisiksi prosesseiksi, jotka jätetään kätkemättä. Menettelyssä on vain yksi ongelma: järjestelmän kuvaus kasvaa pahimmillaan eksponentiaalisesti parametrista erotettujen prosessien lukumäärän suhteen. Kuitenkin käytännössä määrät ovat yleensä pieniä, joten en pidä ongelmaa kovin vakavana.

# 5 Palvelin–asiakasverkon mallintaminen

Tutkimusaiheeni ja siihen liittyvien kysymysten valaisemiseksi tarkastellaan kuvan 1 mukaista yksinkertaista järjestelmää, jossa asiakas pyytää palvelimelta tietoa $x$, jää odottamaan vastausta $y$ ja sen saatuaan palaa alkutilaan pyytääkseen uutta tietoa. Palvelin puolestaan odottaa ensin pyyntöä $x'$ joltakin asiakkaistaan ja sen tultua lähettää siihen vastauksen $f(x')$. Tämän jälkeen palvelin on valmis käsittelemään uuden pyynnön. Oikeellisuusvaatimuksena on, että pyytäessään tietoa $x$ asiakas saa aina asianmukaisen vastauksen $f(x)$.

CSP-kielellä asiakkaan $i$ voi kuvata prosessina

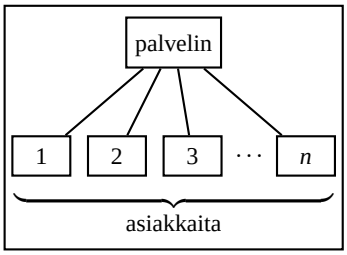
$$C_i = i.request\$x \rightarrow i.reply?y \rightarrow C_i.$$

Symboli \$ tarkoittaa epädeterminististä valintaa, eli asiakas voi pyytää mitä tietoa tahansa. Kysymysmerkki puolestaan merkitsee sitä, että asiakasprosessi hyväksyy minkä tahansa vastauksen, sillä ei ole järkevää olettaa sen tietävän, mikä vastauksen kuuluu olla.

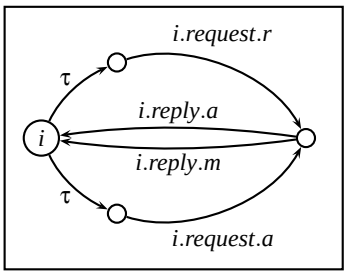
Palvelinprosessin taas voi esittää muodossa

$$S = ?j.request?x' \rightarrow j.reply.f(x') \rightarrow S,$$

missä kysymysmerkit viittaavat jälleen siihen, että palvelin hyväksyy mitä tahansa tietoa $x'$ koskevan pyynnön miltä tahansa asiakkaalta $j$.




Kuva 1: Palvelin–asiakas-järjestelmä




Kuva 2: Abstraktin asiakkaan $i$ tilamalli

Operaattorien erot käyvät ilmi kuvan 2 tilamallista, jossa asiakkaan toiminta on esitetty kahta eri pyyntö- ja vastausviestiä käyttäen. Alkutila on merkitty $i$-kirjaimella, ja $\tau$ tarkoittaa näkymätöntä tapahtumaa, jonka prosessi voi suorittaa milloin tahansa siirtymän alussa ollessaan. Tilamallissa epädeterministinen valinta näkyy kahtena vaihtoehtoisena $\tau$-tapahtumana. Prosessi voi alkutilassa ollessaan suorittaa niistä näkymättömästi kumman tahansa, jolloin asiakas omavaltaisesti päättää mitä tietoa pyytää. Vastaustapahtumien kohdalla tällaista valintamahdollisuutta ei ole, sillä asiakkaalle kelpaa niistä kumpi tahansa.

Näkymättömät siirtyvät ovat yksi syy, jonka vuoksi saman toiminnallisuuden voi esittää usealla erilaisella tilamallilla. Siksi lopullinen tarkastelu tehdäänkin usein jonkin denotaation kautta.

Kun merkitään jonon alkiot kulmasulkeisiin pilkuin erotettuina ja käytetään sanoista *request* ja *reply* vastaavasti lyhenteitä *rq* ja *rp*, niin asiakkaan $i$ jäljet voidaan esittää säännöllisenä ilmaisuna

$$\begin{aligned}&[\,(\langle i.rq.a\rangle + \langle i.rq.r\rangle)\cdot\\&\quad(\langle i.rp.a\rangle + \langle i.rp.m\rangle)\,]^*\cdot\\&(\langle\rangle + \langle i.rq.a\rangle + \langle i.rq.r\rangle).\end{aligned}$$

Ne saadaan prosessin tilamallista alkutilasta lähtevinä näkyvien tapahtumien äärellisinä jonoina.

Estymät puolestaan ovat muotoa

$$\begin{aligned}&(t_e, X \setminus \{i.rq.a\}),\\&(t_e, X \setminus \{i.rq.r\}),\\&(t_o, X \setminus \{i.rp.a, i.rp.m\}),\end{aligned}$$

missä $t_e$ ja $t_o$ ovat vastaavasti parillisen ja parittoman pituiset jäljet ja $X$ kaikkien tapahtumien joukon, $\Sigma$:n, osajoukko. Tässä tapauksessa

$$\begin{aligned}\Sigma = \{\, &j.rq.r, j.rq.a, j.rp.m, j.rp.a \mid\\ &j = 1, 2, \ldots, n\}.\end{aligned}$$

Myös estymistä käy ilmi, että asiakas voi valita pyyntöviestin mutta ei vastausta. Pillastumia asiakkaalla ei sen sijaan ole, sillä sen tilamallissa ei ole yhtään $\tau$-siirtymistä muodostuvaa äärettömän pitkää polkua.

Lopullisen, kuvan 1 mukaisen järjestelmän koostaminen tehdään synkronoimalla palvelimen ja asiakkaiden suorittamat tapahtumat. Palvelin voi siis suorittaa tapahtuman $l.request.z$ vain, jos asiakas $l$ on myös halukas tekemään niin, jolloin molemmat suorittavat tapahtuman yhtä aikaa. Asiakkaiden välillä synkronointia ei tehdä, sillä niiden ei tarvitse, eivätkä ne saakaan, kommunikoida keskenään. CSP:llä tämä ilmaistaan kirjoittamalla

$$G = S \| (C_1 \|\|\| C_2 \|\|\| \cdots \|\|\| C_n),$$

missä kaksi pystyviivaa tarkoittaa synkronista ja kolme asynkronista yhdistämistä.

Nyt tarvitaan vielä tutkittavaa ominaisuutta esittävä prosessi

$$P = \$k.request\$x'' \rightarrow k.reply.f(x'') \rightarrow P.$$

Se eroaa palvelimen kuvauksesta vain siinä, että ulkoinen valintaoperaattori "?" on korvattu epädeterministisellä vastineellaan. Ominaisuusprosessin voi tulkita sillä tavalla, että pyysipä mikä tahansa asiakas mitä tahansa tietoa, se saa haluamansa.

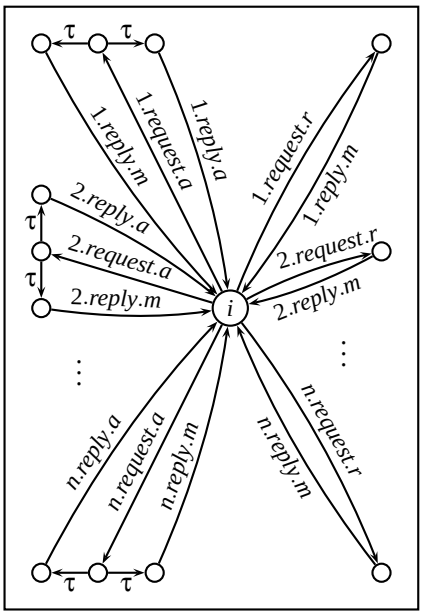
Jos ominaisuusprosessi osoittautuu vähintään yhtä abstraktiksi kuin järjestelmän kuvaus eli CSP:n notaatiota käyttäen

$$P \sqsubseteq G,$$

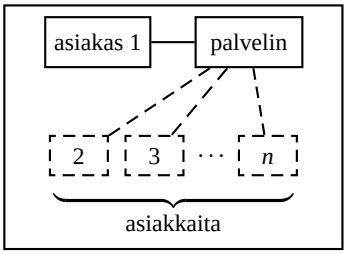
niin järjestelmän sanotaan toteuttavan ominaisuuden. Tämän voi ymmärtää myös sillä tavalla, että $G$ on $P$:n toteutus eli $P$:n deterministisempi versio.

Vaikka tarkasteltavat prosessit ovat todella yksinkertaisia, edellä esitetty kysymys järjestelmän $G$ oikeellisuudesta ominaisuuden $P$ mielessä on mallintarkastuksen kannalta epätriviaali. Tämä johtuu siitä, että jo näinkin pienessä ja yksinkertaisessa mallissa on useita rajoittamattomia parametreja. Näitä ovat pyyntöjä vastausviestien sekä asiakasprosessien lukumäärät. Puskuroimalla palvelimen ja asiakkaiden väliset viestintäkanavat saataisiin helposti vielä yksi tekijä lisää.

Ongelmista kaksi ensimmäistä voidaan ratkaista rajoittumalla oikeellisuustarkasteluissa yhteen mielivaltaiseen pyyntöön $r$ ja siihen liittyvään vastaukseen $m = f(r)$, jolloin data-abstraktiota [6] käyttäen kaikki muut viestit voidaan kuvata yhdeksi alkioksi $a$. Tämä aiheuttaa luonnollisesti muutoksia myös prosessien toimintaan; palvelimen vastaus pyyntöön $a$ ei ole enää yksikäsitteinen, sillä onhan mahdollista, että on olemassa myös jokin muu pyyntöviesti, johon oikea vastaus on $m$. Tämä epädeterministisyys käy ilmi abstraktin palvelimen tilamallista kuvassa 3.




Kuva 3: Abstrahoidun palvelimen tilamalli




Kuva 4: Palvelin ja yksi kätkemättä jätetty asiakas

Myös asiakkaiden määrän rajoittamiseen voisi tässä tapauksessa soveltaa joitakin aiemmin esiteltyjä menetelmiä. Oma lähestymistapani perustuu kuitenkin siihen, että kaikki määrältään parametroidut prosessit, tässä tapauksessa asiakkaat, kätketään. Ajatuksessa ei ole sinänsä mitään uutta tai ihmeellistä, eikä ominaisuuden $P$ tutkiminenkaan onnistu suoraan, mutta

jos asiakkaista yksi erotetaan parametrista, tarkastelu voidaan rajata siihen ja yrittää yleistää myöhemmin koskemaan myös muita asiakkaita.

Muutoksen jälkeen päädytään kuvan 4 mukaiseen järjestelmään, jossa asiakas 1 on parametrista erotettu prosessi ja muut asiakkaat on kätketty. Syntaksitasolla muutokset koskevat palvelinta

$$S' = (?j.request?x' \rightarrow j.reply.f(x') \rightarrow S) \\ \Box (1.request?x' \rightarrow 1.reply.f(x') \rightarrow S),$$

jossa $j$ saa nyt arvoja kahdesta $n$:ään, ja järjestelmän kuvausta

$$G' = S' \| (C_1 ||| C_2 ||| \cdots ||| C_n) \setminus \{|2,3,\ldots,n|\}.$$

Tässä tapauksessa neliö tarkoittaa palvelimen olevan valmis käsittelemään yhtä lailla sekä parametrista erotetun että muiden asiakkaiden pyyntöjä ja kenoviiva on kätkentäoperaattori, joka piilottaa kaikki 2-, 3-, …, $n$-alkuiset tapahtumat. Kannattaa huomata, että asiakkaan 1 erottaminen parametrista koskee ainoastaan palvelimen kuvausta; tilamalli säilyy entisellään, joten palvelimen toiminta ei muutu.

Myös oikeellisuusvaatimusta voisi periaatteessa käsitellä vastaavalla tavalla. Menetelmäni ajatuksena on kuitenkin kiinnittää huomio vain kiinteään määrään prosesseja ja pyrkiä yleistämään näille osoitetut ominaisuudet koskemaan myös muita. Tämän vuoksi on perusteltua kirjoittaa oikeellisuusvaatimus suoraan kätkemättömän asiakkaan näkökulmasta, jolloin se saadaan muotoon

$$P' = 1.request\$x'' \rightarrow 1.reply.f(x'') \rightarrow P'.$$

Kyseessä ei ole täysin sama ominaisuus kuin

$$P \setminus \{|2,3,\ldots,n|\},$$

mutta $P'$ on kuitenkin sellainen, joka ainakin itselläni tulee ensimmäisenä mieleen menetelmää suoraan soveltaessani. Edelleen $P'$ on näistä kahdesta estymämallin mielessä abstraktimpi, eli jos se on voimassa, järjestelmä toteuttaa molemmat ominaisuudet. Lisäksi $P'$ on yksinkertaisempi ja asiakkaiden määrästä riippumaton.

Prosessin $G'$ toimintaan asiakkaiden määrä sen sijaan vaikuttaa kätkennästä huolimatta. Kun tämä parametri saa kaikki mahdolliset arvonsa, syntyy äärettömän monta samankaltaista järjestelmää eli ääretön prosessiperhe. Käytetään näistä merkintää $G'_n$, missä alaindeksi kertoo kätkettävien asiakkaiden lukumäärän. Nyt kysymys kuuluu, mitkä prosesseista $G'_0, G'_1, G'_2, \ldots$ ovat samanlaisia. Mikäli erilaisia järjestelmiä voidaan osoittaa olevan vain äärellinen määrä, ne ovat äärellistilaisia ja ne pystytään vielä nimeämään, voidaan ääretön määrä mallintarkastustehtäviä suorittaa äärellisessä ajassa.

Järjestelmän formaali tarkastelu osoittaa, että tässä tapauksessa prosessiperheen jäsenet tai oikeastaan parametrit kooltaan jakautuvat kahteen ekvivalenssiluokkaan

$$\{0\} \text{ ja } \{1,2,3,\ldots\}$$

käytettäessä CSP:n estymä- tai standardisemantiikkaa. Siis jos

$$P' \sqsubseteq G'_0 \text{ ja } P' \sqsubseteq G'_1,$$

niin järjestelmä toimii oikein asiakkaiden määrästä riippumatta. Erityisesti, jos vertailuista jälkimmäinen on voimassa, niin silloin järjestelmä toimii oikein, kun kätkettäviä asiakkaita on vähintään yksi, ja päinvastoin.

Kannattaa huomata, että oikeellisuus koskee tässä tapauksessa vain kätkemätöntä prosessia. Tulos voidaan kuitenkin yleistää koskemaan myös muita asiakkaita, mikäli palvelin kohtelee niitä samalla tavalla kuin näkyvää. Tämä ominaisuus tunnetaan paremmin symmetriana [8, 10,

17], ja se käy hyvin ilmi palvelimen tilamallista kuvassa 3.

Symmetrioita on tutkittu tilaperustaisten lähestymistapojen yhteydessä hyvinkin kattavasti, mutta transitiopohjaista symmetriaa koskevia artikkeleita on olemassa vain muutama [13, 14, 22]. Siksi olen laajentanut symmetrian käsitteen CSP:n estymämalliin [26]. Määritelmäni on osoittautunut samaksi kuin Goldsmithin ja Moffatin [14].

Taustalla on jokin perusjoukko, jonka suhteen valittu permutaatioryhmä määrää symmetrian käsitteen. Jotta permutaatioita voisi soveltaa mihin tahansa CSP-rakenteeseen, permutaatiot laajennetaan tapahtumiin ja tätä kautta prosesseihin sekä niiden jälkiin ja estymiin sillä tavalla, että ainoastaan perusjoukon alkioita siirretään.

Prosessi $P$ on $\Pi$-symmetrinen, jos $P$:n jäljen ja estymän kuvaaminen ryhmän $\Pi$ permutaatiolla tuottaa aina vastaavasti $P$:n jäljen ja estymän. Esimerkiksi palvelinprosessi on $Sym(I)$-symmetrinen, missä $Sym(I)$ muodostuu kaikista joukon $\{1,2,\ldots,n\}$ permutaatioista. Edelleen jokainen asiakas yhdistetään palvelimeen samalla tavalla, jolloin myös koko järjestelmä on $Sym(I)$-symmetrinen. Tämä tarkoittaa sitä, että yhdelle asiakkaalle osoitetut ominaisuudet koskevat prosessin tunnusta vaille myös kaikkia muita. Järjestelmän oikeellisuuden toteamiseksi riittää siis prosessien $P'$ ja $G'$ vertailu.

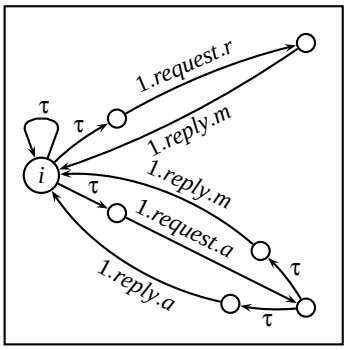
Olen rajoittanut yksinkertaista CSP-kieltä sillä tavalla, että ainoastaan valitun perusjoukon suhteen symmetristen prosessien määritteleminen on mahdollista. Tätä rajoitettua kieltä käyttämällä pystytään kuitenkin esittämään mikä tahansa estymämallin mielessä symmetrinen prosessi [26]. Niinpä rajoitetun kielen puitteissa toiminnallisen symmetrian toteaminen palautuu helposti ratkeavaksi syntaksin tarkastamiseksi. Yleisesti CSP-prosessin symmetrisyyden selvittäminen estymämallissa on kuitenkin **NP**-kova ongelma eli ainakin yhtä vaikea kuin kysymys, onko verkossa Hamiltonin polkua [11]. Vastaavien tulosten pitäisi olla mahdollisia myös rakentamalleni tyypitetylle CSP-kielelle.

Ominaisuuksien laajentaminen koskemaan järjestelmäperheen erikokoisia jäseniä eli ekvivalenssiluokkien määrittäminen syntaksin perusteella on sen sijaan osoittautunut astetta hankalammaksi ongelmaksi. Esimerkin pohjalta tehtyjä havaintoja voidaan kuitenkin jonkin verran yleistää, sillä kokeiden perusteella asiakkaiden määrä jakautuu ekvivalenssiluokkiin $\{0\}$ ja $\{1,2,3,\ldots\}$ aina, kun palvelin muodostuu silmukasta, jossa se käsittelee yhtä kätkettävää asiakasta kerrallaan. Samat ekvivalenssiluokat saadaan myös siinä tapauksessa, että palvelin käsittelee jokaisen asiakkaan vuorollaan tai niitä kaikkia omissa säikeissään toisista riippumatta.

Tällaisia rakenteita voi edelleen yhdistellä peräkkäin esimerkiksi niin, että aluksi palvelin lähettää kättelyviestin jokaiselle asiakkaalle, sen jälkeen ne käsittelevät tietoa omaan tahtiinsa yhdessä palvelimen kanssa ja kommunikoinnin päätteeksi jokaisen asiakkaan kanssa vaihdetaan vielä lopetusviestit. Lisäksi voidaan sallia vain yhden mielivaltaisesti valitun asiakkaan kanssa keskustelu, mikäli lopuksi ainakin toinen osapuolista päätyy samaan tilaan kuin ennen kommunikointia.

On mielenkiintoista, että jossain tilanteissa pienempi järjestelmä toimii suuremman reiluna mallina. Esimerkkijärjestelmän ekvivalenssiluokkia edustavat prosessit $G'_0$ ja $G'_1$, joista jälkimmäisen tilamalli on esitetty kuvassa 5. Siinä on nähtävissä pillastuma, yhdestä $\tau$-siirtymästä muodostuva silmukka, joka puuttuu pro-

sessin $G'_0$ tilamallista. Pillastuma aiheutuu siitä, että palvelimen ei koskaan tarvitse valita asiakkaan 1 pyyntöä, vaan se voi suorittaa tapahtumia pelkästään näkymättömien prosessien kanssa.




Kuva 5: Abstraktin järjestelmän $G'_1$ tilamalli

Tällaiset abstrahoinnin ja kätkennän aiheuttamat pillastumat edustavat harvoin todellisia ilmiöitä, minkä vuoksi ne yleensä eivät ole kiinnostavia ja niistä on pelkkää haittaa. Mikäli palvelimen kuitenkin oletetaan kohtelevan asiakkaitaan reilusti eli yhdenkään asiakkaan pyyntöä ei voida lykätä loputtomiin, prosessien $G'_0, G'_1, G'_2, \ldots$ tilamallit ovat samanlaisia ja saadaan ainoastaan yksi ekvivalenssiluokka. Tässä tapauksessa pienempi järjestelmä toimii siis suuremman reiluna mallina.

Myös tätä tarkastelua voi jonkin verran yleistää. Nyt on kuitenkin syytä olettaa, että kaikkia asiakkaita ei kätketä, vaan niistä yksi jätetään näkyväksi. Kun oletetaan, että palvelin on reilu, niin yhden asiakkaan verkossa tehdyt tarkastelut voidaan yleistää suurempiin järjestelmiin, mikäli palvelin toteuttaa ekvivalenssiluokkien $\{0\}$ ja $\{1,2,3,\ldots\}$ yhteydessä mainitut rajoitukset ja yhden asiakkaan verkko ei lukkiudu eikä siinä ei ole pillastumia.

Kannattaa huomata, että yleistykset on tehty osin intuitiopohjalta ja niiden formalisointi on vielä kesken. On mahdollista, että palvelimen käyttäytymisen lisäksi myös välitettävän tiedon tyyppiä täytyy erikoistapauksissa rajoittaa. Tämä onkin lähitulevaisuudessa yksi tutkimuskohteistani.

# 6 Visio

Työtäni motivoi ajatus saada menetelmä toimimaan useissa käytännön kannalta tärkeissä verkkotopologioissa tai arkkitehtuurimalleissa. Tällöin suuret ja monimutkaisetkin järjestelmäkuvaukset voisi jakaa äärelliseen määrään tila-avaruudeltaan äärellisiä prosesseja, kun menetelmää soveltaisi rekursiivisesti edeten sovelluksen rakenteessa ylätasolta alaspäin.

Todennäköisesti suurille järjestelmille lasketut invarianttiprosessit olisivat kuitenkin mallintarkastustarkoituksiin liian suuria, mutta äärellistilaisille abstraktioille on käyttöä muuallakin kuin verifioinnin piirissä. Järjestelmää analysoitaessa tai simuloitaessa virheiden havaitseminen tehostuu ja sen toiminnan ymmärtäminen helpottuu, kun käytössä on pieni malli. Simulointi tulee kyseeseen esimerkiksi silloin, kun verifiointi mallin äärellistilaisuudesta huolimatta veisi liikaa resursseja. Lisäksi suurten järjestelmien tapauksessa malleja voisi soveltaa intuitiopohjalta ohjelmistojen testiympäristöjen määrittämiseen.

Näin yleiset tulokset ovat kuitenkin valtavan työn takana ja kokonaisten järjestelmäkuvausten abstrahointi automaattisesti vielä kaukana. Joillakin osa-alueilla tämä voi silti olla piankin totta. Esimerkiksi tietoturvaprotokollien analysointi kuvauksen perusteella on jo nyt mahdollista [24].

Vaikka suuriakin läpimurtoja tehtäisiin, formaalit menetelmät jäänevät kuitenkin korkeintaan muutamien erityisalojen käyttöön, joilla verifioinnin kustannukset lasketaan pienemmiksi kuin ohjelman virheellisen toiminnan seuraukset. Tällaisia kohteita voisivat olla esimerkiksi ajoneuvojen ja tuotantolaitosten ohjausjärjestelmät, sotilaalliset ja lääketieteen sovellukset sekä laajasti käytetyt protokollat. Näissä kaikissa ohjelmistovirhe voi aiheuttaa ihmishengen menetyksen tai valtavia muita tappioita. Menetelmien yleistymiseen vaikuttavat kuitenkin tutkimuksellisten saavutusten lisäksi myös poliittiset päätökset sekä vallitseva asenne, se, minkä verran siedämme ohjelmistovirheitä ympärillämme.

# 7 Kiitokset

Kiitän väitöskirjatyöni ohjaajia, Juha Kortelaista ja Antti Valmaria, heidän esittämistään parannusehdotuksista artikkelin ensimmäiseen versioon.

# Viitteet

[1] K. R. Apt, D. C. Kozen (1986) Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, volume 22, number 6, s. 307–309. Elsevier Science.

[2] B. Alpern, F. B. Schneider (1985) Defining liveness. *Information Processing Letters*, volume 21, number 4, s. 181–185. Elsevier Science.

[3] R. E. Bryant (1986) Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, Series C, volume 35, number 8, s. 677–691. IEEE Computer Society.

[4] A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri (1999) NuSMV: a new symbolic model verifier. *Proceedings of the 11th International Conference on Computer Aided Verification; Lecture Notes in Computer Science*, volume 1633, s. 495–499. Springer-Verlag.

[5] E. M. Clarke, O. Grumberg, S. Jha (1997) Verifying parameterized networks. *ACM Transactions on Programming Languages and Systems*, volume 19, issue 5, s. 726–750. ACM Press.

[6] E. M. Clarke, O. Grumberg, D. E. Long (1994) Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, volume 16, issue 5, s. 1512–1542. ACM Press.

[7] E. M. Clarke, O. Grumberg, D. A. Peled (1999) *Model Checking*. MIT Press, Cambridge, Massachusetts, 314 s.

[8] E. M. Clarke, T. Filkorn, S. Jha (1993) Exploiting symmetry in temporal logic model checking. *Proceedings of the 5th International Conference on Computer Aided Verification; Lecture Notes In Computer Science*, volume 697, pp. 450–462. Springer-Verlag.

[9] S. J. Creese (2001) *Data Independent Induction: CSP Model Checking of Arbitrary Sized Networks*. Ph.D. thesis. Oxford University, Computing Laboratory, 146 s.

[10] E. A. Emerson, A. P. Sistla (1996) Symmetry and model checking. *Formal Methods in System Design*, volume 9, number 1, pp. 105–131. Kluwer Academic Publishers.

[11] M. R. Garey, D. S. Johnson (1979) *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, San Francisco, 338 s.

[12] J.-Y. Girard, P. Taylor, Y. Lafont (1990) *Proofs and Types*. Cambridge University Press, 175 s.

[13] P. Godefroid (1999) Exploiting symmetry when model-checking software. *Proceedings of IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification*, s. 257–275. Kluwer Academic Publishers.

[14] M. Goldsmith, N. Moffat (2004) *Automated Compression Techniques*; Forward project deliverable D5. http://www.forward-project.org.uk/PDF_Files/D5.pdf (haettu 17.11.2005).

[15] G. J. Holzmann (2003) *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 596 s.

[16] M. R. A. Huth, M. D. Ryan (2000) *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 387 s.

[17] C. N. Ip, D. L. Dill (1996) Better verification through symmetry. *Formal Methods in System Design*, volume 9, number 1, s. 41–75. Kluwer Academic Publishers.

[18] C. N. Ip, D. L. Dill (1996) Verifying systems with replicated components in Murφ. *Proceedings of the 8th International Conference on Computer Aided Verification; Lecture Notes in Computer Science*, volume 1102, s. 147–158. Springer-Verlag.

[19] R. S. Lazić (1999) *A Semantic Study of Data Independence with Applications to Model Checking*. Ph.D. thesis. Oxford University, Computing Laboratory, 242 s.

[20] R. S. Lazić, D. Nowak (2000) A unifying approach to data-independence. *Proceedings of the 11th International Conference on Concurrency Theory; Lecture Notes in Computer Science*, volume 1877, s. 581–596. Springer-Verlag.

[21] Z. Manna, A. Pnueli (1995) *The Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 512 s.

[22] F. Michel, P. Azéma, F. Vernadat (1996) Permutable agents in process algebras. *Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems; Lecture Notes in Computer Science*, volume 1055, s. 187–206. Springer-Verlag.

[23] A. W. Roscoe (1997) *The Theory and Practice of Concurrency*. Prentice Hall, 565 s.

[24] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, A. W. Roscoe (2001) *The Modelling and Analysis of Security Protocols: The CSP Approach*. Addison-Wesley, 300 s.

[25] A. Siirtola (2003) *Rinnakkaisten järjestelmien mallintaminen aikalogiikkaa käyttäen*. Diplomityö. Oulun yliopisto, sähkö- ja tietotekniikan osasto, 141 s.

[26] A. Siirtola (2005) Verkkosovellusten äärellistilaisten formaalien mallien tuottaminen automaattisesti. *Tietojenkäsittelytieteen päivät 2005*, s. 163–167. Oulun yliopistopaino, Oulu.

[27] A. Valmari (1998) The state explosion problem. *Lecture Notes in Computer Science*, volume 1491, s. 429–528. Springer-Verlag.

[28] A. Valmari (2000) A chaos-free failures divergences semantics with applications to verification. *Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare; Millennial Perspectives in Computer Science*, s. 365–382. Palgrave-Macmillan.

[29] A. Valmari (2001) Composition and abstraction. *Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes; Lecture Notes in Computer Science*, volume 2067, s. 58–99. Springer-Verlag.

[30] P. Wolper (1986) Expressing interesting properties of programs in propositional temporal logic. *Proceedings of the 13th Annual ACM Symposium on Principles of Programming Languages*, s. 184–193. ACM Press.