Spesifioinnin ja verifioinnin perusteet
Luentomateriaali, kevät 2005

Timo Karvi

UNIVERSITY OF HELSINKI
FINLAND

Sisältö

1 Johdanto **1**

1.1 Lähtökohta . . . 1

1.2 Yhteistilaverkko . . . 1

1.3 Ekvivalenssipohjainen verifiointi . . . 2

1.4 Mallintarkistus . . . 4

1.5 Käytännön kokemuksia . . . 4

2 Tilasiirtymäsysteemit **7**

2.1 Vuorottelevan bitin protokolla . . . 7

2.2 Asiakas/palvelin-systeemi . . . 10

2.3 Tilasiirtymäsysteemin määritelmä . . . 11

3 Yhteistilaverkko **13**

3.1 Johdanto . . . 13

3.2 AB-protokollan yhteistilaverkko . . . 13

3.3 Rinnakkaisoperaattori . . . 15

3.4 Rinnakkaisoperaattorin ominaisuuksia . . . 17

3.4.1 Monisynkronointi . . . 17

3.4.2 Synkronoinnin luonne . . . 17

3.4.3 Rinnakkaisoperaattorin liitännäisyys . . . 17

3.5 Yhteistilaverkon käytännön toteutuksesta . . . 19

3.5.1 Yhteistilaverkko tietorakenteena . . . 19

3.5.2 Bittihajautus ja muita vaihtoehtoja . . . 19

4 Mallinnuksen perusasioita **21**

4.1 Epädeterminismi . . . 21

4.2 Kanavat ja ympäristö . . . 22

5 Ekvivalenssit ja verifiointi **27**
5.1 Palvelun kuvaukset 27
5.2 Relaatiot 29
5.3 Suoritusjälkiekvivalenssi 33
5.4 Heikko bisimulaatioekvivalenssi 34
5.5 Heikon bisimulaation laskeminen 39
5.6 Esimerkkejä 43
5.6.1 AB-protokolla 43
5.6.2 FE-protokolla 43
5.7 Johtopäätöksiä ja ongelmia 46

6 Perus-Lotos **49**
6.1 Johdanto 49
6.2 Valmiit prosessit 50
6.3 Etutoiminto (action prefix) 50
6.4 Kätkentä (hiding) 51
6.5 Valinta (choice) 52
6.6 Rinnakkaisoperaattori 53
6.7 Peräkkäisyys 56
6.8 Keskeytys 56
6.9 Operaattoreiden sitovuus 57
6.10 Prosessin kutsu 57
6.11 Exit ja noexit 60
6.12 Esimerkkejä 60
6.12.1 Tuottaja-kuluttaja-spesifikaatio 60
6.12.2 Laskuri 62
6.12.3 Asiakas-palvelin -systeemi 63
6.13 Ohjelmistot 64
6.13.1 ARA 64
6.13.2 Aldebaran 65
6.14 AB-protokolla 65

7 Datan käsittely **69**
7.1 Datan lähetys ja vastaanotto 69
7.2 Parametrit aloituksessa ja lopetuksessa 72

7.3 Vahdit . . . 74
7.4 Yleistetty valinta . . . 75
7.5 Sijoitus . . . 76
7.6 Par-lause . . . 76
7.7 Katsaus tietotyyppeihin . . . 77
7.7.1 Totuusarvot abstraktina tietotyyppinä . . . 77
7.7.2 Termialgebra . . . 78
7.8 Tietotyypin määrittely käytännössä . . . 79
7.8.1 Tietotyypit CADP-ohjelmistossa . . . 81

8 Esimerkkejä 83

8.1 AB-protokolla täydellä Lotoksella . . . 83
8.2 Poissulkemisalgoritmeja . . . 90
8.2.1 Ratkaisu 1 . . . 90
8.2.2 Ratkaisu 2 . . . 93
8.2.3 Dekkerin algoritmi . . . 96

9 Aikalogiikkoja 101

9.1 Johdanto . . . 101
9.2 Kripken struktuurit . . . 102
9.3 Lineaarisen ajan logiikka LTL . . . 104
9.4 Haarautuvan ajan logiikka CTL . . . 106
9.5 Mallintarkistus . . . 108

Luku 1

Johdanto

1.1 Lähtökohta

Tällä kurssilla esitellään muutamia keskeisiä käsitteitä, joihin hajautettujen järjestelmien verifiointi useimmiten perustuu. Aluksi määritellään *siirtymäsysteemit*, joiden avulla esitetään prosessit. Siirtymäsysteemi on suunnattu verkko, jonka kaariin liittyy symboli, tapahtuma. Tapahtuma kuvaa joko kommunikointia toisen prosessin kanssa tai sisäistä laskentaa. Kuvattaessa prosesseja tällä tasolla sisäistä laskentaa ei useinkaan tarvitse esittää tarkasti, vaan sitä voidaan abstrahoida voimakkaasti.

Siirtymäsysteemit täytyy myös esittää jollakin tavalla, jotta ne voidaan antaa tietokoneelle syötteeksi. Tähän on käytössä useita *spesifiointikieliä*. Yksinkertaisimmillaan spesifiointikieli on vain suoraviivainen menetelmä luetella tilat ja niiden väliset siirtymät. Useimmiten kuitenkin kieli sisältää muutakin kuten esimerkiksi mahdollisuuden määritellä sisäistä laskentaa yksityiskohtaisesti ja äärettömienkin siirtymäsysteemien äärellisen kuvauksen. Esimerkki tällaisesta vanhemmasta spesifiointikielestä on Estelle, jossa on paljon piirteitä Pascalista. Nykyaikaisemmat kielet perustuvat *prosessialgebroihin*. Niissä siirtymäsysteemit kuvataan algebrallisella notaatiolla. Tämä johtaa tiiviiseen esitystapaan, jonka syntaksi ja erityisesti semantiikka on täysin formaalisti määritelty. Kurssilla käsitellään eräs tällainen kieli, *Lotos*. Muita prosessialgebroja ovat CCS, CSP ja ACP. On lisäksi huomattava, että on myös muita vaihtoehtoja esittää prosesseja kuin siirtymäsysteemit. Mainittakoon vain Petri-verkot ja temporaalilogiikka. Näitä varten on omat esityskielensä.

1.2 Yhteistilaverkko

Hajautetun algoritmin yksittäinen prosessi on usein varsin yksinkertainen. Sen sijaan usean,yksinkertaisenkin prosessin muodostamaa systeemiä on hankala analysoida. Hajautetussa järjestelmässä syntyy uudenlaisia virhetilanteita kuten *lukkiumia* ja *eläviä lukkiumia*. Hajautetussa järjestelmässä täytyy toisinaan taata myös *reiluus* eli että prosessit saavat resursseja tasapuolisesti käyttöönsä. Kaiken tämän varmis-

taminen on hankalaa perinteisellä testauksella, sillä testattaessa täytyisi kokeilla kaikkia mahdollisia suoritusvaihtoehtoja ja -järjestyksiä, joita on suuri määrä.

Kokonaisen systeemin käyttäytymistä voidaan mallintaa myös yksittäisellä siirtymäsysteemillä. Tällöin systeemin tilaa kuvataan vektorilla, jonka i.s alkio kuvaa i. prosessin tilaa. Kun i. prosessi muuttaa tilaansa, muuttuu samalla yhteistilaa kuvaava vektori: alkio i muuttuu. Prosessin tila voi muuttua sanoman lähettämisen ja vastaanottamisen yhteydessä sekä sisäisen laskennan suorituksessa.

Ensimmäinen ongelma systeemin mallintamisessa yhteistilaverkolla syntyy aidosti rinnakkaisten tapahtumien kohdalla. Jos prosessit i ja j suorittavat tapahtuman samaan aikaan, pitäisi vektorialkioiden i ja j myös muuttua yhdessä askeleessa. Toisaalta prosessien i ja j tapahtumat voivat sattua peräjälkeen, jolloin ensin muuttuu esimerkiksi alkio i ja sen jälkeen alkio j. Pitäisikö kaikki nämä vaihtoehdot ottaa huomioon? Jos näin tehtäisiin mekaanisesti, syntyisi valtava joukko erilaisia suoritusvaihtoehtoja. Tästä syystä yleensä noudatetaan ns. *lomitusemantiikkaa*. Ajatellaan, että rinnakkaiset tapahtumat sattuvat aina jossakin järjestyksessä eikä aidosti rinnakkaisia tapahtumia oleteta sattuvan. Tämä periaate näyttää toimivan hyvin käytännössä ja enemmistö verifiointityökaluista perustuu lomitussemantiikkaan.

Huolimatta edellä esitetystä rajoituksesta erilaisia suoritusvaihtoehtoja syntyy edelleen runsaasti. Puhutaan yhteistilaverkon *tilaräjähdyksestä*. Lomitussemantiikka generoi kaikki erilaiset suoritusjärjestykset. Useimmiten näillä erilaisilla järjestyksillä ei ole merkitystä, vaan kaikki johtavat samaan lopputulokseen. Tätä ei voi kuitenkaan etukäteen päätellä helposti. Tästä syystä täydellisen yhteistilaverkon generointi ei onnistu suurille systeemeille. On esitetty erilaisia tapoja rajoittaa yhteistilaverkon kokoa. Eräs mahdollisuus on jättää generoimatta yhteistilaverkon kaikki tilat ja kulkea siellä vain satunnaisesti valittujen tilojen kautta. On myös huomattava, että aidon eli todellisen rinnakkaisuuden älykäs soveltaminen saattaa joissakin tilanteissa johtaa suppeampaan verkkoon kuin puhdas lomitussemantiikka. Tällä kurssilla ei kuitenkaan puututa näihin menetelmiin.

1.3 Ekvivalenssipohjainen verifiointi

Yhteistilaverkosta nähdään helposti monia protokollavirheitä. Lukkiumat paljastuvat jo yhteistilaverkkoa muodostettaessa. Elävät lukkiumat vastaavat yhteistilaverkon komponentteja, joista ei päästä enää takaisin pääsykliin, esimerkiksi alkutilaan. Tällaiset lukkiumat löydetään vaikkapa syvyyssuuntaisen etsinnän avulla. On kuitenkin virhetilanteita, joita on vaikea nähdä pelkästään yhteistilaverkkoa analysoimalla. Esimerkiksi voi olla haitallista, jos tietyt tapahtumat sattuvat väärässä järjestyksessä. Väärän järjestyksen havaitseminen edellyttää, että yhteistilaverkon suoritusjälkiä verrataan sallittuihin suoritusjälkiin. Sallitut suoritusjäljet täytyy esittää jollakin tavalla.

Keskitymme tällä kurssilla ekvivalenssipohjaiseen verifiointiin. Tällöin lähtökohta on seuraava. Systeemin toimintaa voidaan usein kuvata ulkopuolisen havaitsijan tai

hyväksikäyttäjän kannalta. Ulkopuolinen havaitsija näkee vain sen, mitä tapahtuu systeemin, tai oikeammin tässä yhteydessä protokollan, rajapinnalla sen kommunikoidessa protokollaa käyttävien prosessien kanssa. Rajapinnan toimintaa voidaan myös kuvata siirtymäsysteemillä ja sitä kutsutaan *palveluksi*. Eksivalenssipohjaisessa verifioinnissa pyritään osoittamaan, että protokolla täyttää palvelun. Tämä tapahtuu seuraavasti.

Protokollan toiminta mallinnetaan ensin jollain spesifiointikielellä. Käytännössä tämä tarkoittaa, että jokainen protokollan prosessi, esimerkiksi lähettäjä, vastaanottaja ja ajastin, kuvataan esimerkiksi Lotoksella. Tämän jälkeen muodostetaan systeemiä vastaava yhteistilaverkko, joka on siis iso siirtymäsysteemi. Yhteistilaverkko muodostetaan täysin automaattisesti ohjelmistolla, joka ymmärtää käytettävää spesifiointikieltä. Yhteistilaverkon kaarissa näkyvät tapahtumat, jotka protokolla suorittaa. Näistä tapahtumista piilotetaan kaikki muut kuin rajapinnan tapahtumat. Piilottaminen tarkoittaa, että tapahtuman nimi muutetaan erityiseksi näkymättömäksi tapahtumaksi. Tapahtumaa vastaava kaari jää verkkoon, ainoastaan siihen liittyvä nimi siis muuttuu.

Vastaava tehdään palvelulle. Siis palvelu kuvataan esimerkiksi Lotoksella ja kuvauksesta generoidaan automaattisesti siirtymäsysteemi. Nyt meillä on kaksi siirtymäsysteemiä. Toinen vastaa protokollan toimintaa kaikissa tilanteissa, toinen esittää vaatimuksen, miten protokollan odotetaan toimivan. Ekvivalenssipohjaisessa verifioinnissa käytössä on jokin ekvivalenssi, jonka avulla verrataan siirtymäsysteemeitä: ovatko ne saman toiminnan tuottavia vai ei.

Erilaisia ekvivalensseja on lukuisia. Jo useita vuosia sitten laskettiin, että erilaisia ekvivalensseja ja niiden muunnelmia on yli kolmesataa. Eri ekvivalenssit ottavat huomioon hieman eri asioita. Yksinkertaisimmillaan tarkastellaan vain tapahtumien suoritusjärjestystä. Tällöin ei esimerkiksi kiinnitetä huomiota lukkiumiin. Vaativin eksivalenssi on verkkoisomorfia. Siinä ekvivalenttien verkkojen tulee olla täsmälleen samoja tilojen nimeämistä vaille. On selvää, että protokollan ja palvelun vertailuun tarvitaan jokin näiden ääriesimerkkien välimuoto.

Yksi suosituimmista ekvivalensseista on (heikko) bisimulaatio. Prosessit P ja Q ovat sen mukaan ekvivalentteja, jos toinen pystyy simuloimaan toisen ulospäin näkyviä toimintoja vieläpä niin, että simulointijärjestystä voidaan vaihtaa lennossa. Eli ensin esimerkiksi P simuloi Q:ta, mutta jossain vaiheessa kesken simuloinnin Q alkaakin suorittaa tapahtumiaan ensin ja P rupeaa simuloimaan Q:ta. P:n simulointi alkaa siitä pisteestä, johon P viimeksi jäi suorittaessaan itse ensimmäisenä.

Jotta ekvivalenssi olisi käytännöllinen, täytyy se kyetä laskemaan melko tehokkaasti, matalassa polynomisessa ajassa. Bisimulaatio on tällainen ekvivalenssi. Sen varsinainen laskeminen sujuu nopeasti, mutta se vaati transitiivisen sulkeuman laskemista aluksi tai laskennan varrella. Tämä hidastaa oleellisesti suurten verkkojen bisimulaatioekvivalenssin laskentaa.

Jos laskenta osoittaa, että protokolla ja palvelu ovat ekvivalentit, voidaan protokollaan pitää virheettömänä. Tällaisessa verifioinnissa siis tietokone tekee suurimman osan työstä. Ihmisen tehtäväksi jää spesifikaatioiden kirjoittaminen ja sopivan

ekvivalenssin valitseminen. On myös tilanteita, joissa tarvitaan lisäanalyysiä, koska ekvivalenssi ei kata kaikkia virhetilanteita. Esimerkiksi reiluus saattaa olla asia, joka pitää varmistaa muilla keinoin. Tällöin joudutaan turvautumaan esimerkiksi aikalogiikkaan. Toinen syy, miksi verifointi on käytännössä vaikeampaa kuin yllä on esitetty, johtuu siitä, että käytännön yhteistilaverkot ovat usein liian suuria. Paloittemalla systeemi sopivasti osiin voidaan joskus saada suuretkin verkot verifioiduksi, mutta toisinaan monesti tätä lähestymistapaa ei voida suoraan käyttää. Tällöin jäljelle jää osittainen verifiointi esimerkiksi satunnaiskulun avulla.

1.4 Mallintarkistus

Vaihtoehto ekvivalenssipohjaiselle verifioinnille on *mallintarkistus*. Se tarkoittaa jonkin, yleensä aikalogiikan, käyttöä niin, että logiikan kaavoilla ilmaistaan systeemiltä vaadittavat ominaisuudet. Sen jälkeen rakennetaan systeemistä malli, käytännössä siirtymäsysteemi tai verkko, ja tutkitaan, pätevätkö logiikan kaavat tuossa mallissa. Hyvänä puolena tässä lähestymistavassa on, että voidaan ilmaista helposti vain tiettyjä ominaisuuksia, joita systeemin on noudatettava. Systeemin mallinnuskaan ei välttämättä vaadi niin monien yksityiskohtien mukaan ottamista kuin ekvivalenssipohjainen verifiointi vaatii. Toisaalta mallintarkististuksella ei saada aina niin kattavaa tulosta kuin ekvivalensseja ja palvelukuvauksia käytettäessä.

Erilaisia logiikkoja on monenlaisia. Kaksi tunnetuinta ja useimmiten käytettyä ovat lineaarisen ajan logiikka (LTL) ja haarautuvan ajan logiikka (CTL). Edellisessä väitteet koskevat kaikkia polkuja, jälkimmäisessä voidaan käsitellä vain tiettyjä polkuja ja niiden ominaisuuksia. Logiikat ovat yhteismitattomia siinä mielessä, että kummallakin voidaan ilmaista ominaisuus, jota toisella ei voida ilmaista. Tästä syystä on kehitetty vielä voimakkaampia logiikkoja, joihin kumpikin sisältyy, esimerkiksi CTL*.

Perinteisesti aikalogiikan formalismina on verkko, jonka kaariin ei liity tapahtumia. Sen sijaan tiloilla on ominaisuuksia tai rakennetta. Kuitenkin prosessialgebrojen pohjalta syntyvissä verkoissa tiloilla ei ole rakennetta, mutta kaariin liittyy tapahtuma. Tästä syystä on kehitelty aikalogiikan tapaisia logiikkoja, jotka sopivat hyvin siirtymäsysteemien yhteyteen. Yksi tällainen on ACTL, joka on saatu CTL:stä. Muita ovat Hennessyn ja Milnerin logiikka sekä μ-kalkyyli. Tällä kurssilla käsittelemme lyhyesti vain LTL:ää ja CTL:ää.

1.5 Käytännön kokemuksia

Formaaleja menetelmiä on sovellettu jo yli parikymmentä vuotta. Sovellukset ovat pääsiassa syntyneet yliopistojen tutkimusprojekteissa, mutta viime vuosina myös teollisuus on jossain määrin kiinnostunut formaaleista menetelmistä. Parhaat tulokset teollisuudessa on saatu piirisuunnittelun yhteydestä. Suurilla laitevalmistajilla

kuten Intelillä on verifiointiryhmänsä, jotka testaavat piirien toimintaa. Useimmiten tässä yhteydessä sovelletaan mallintarkistusta.

Myös protokollia on menestyksekkäästi verifioitu. Parhaiten tunnettuja esimerkkejä ovat mm. seuraavat. IEEE Futurebus- ja cache coherence-protokollasta löydettiin joukko aikaisemmin huomaamattomia virheitä. ISDN/SUP-tietoliikenneprotokollasta löydettiin 122 virhettä. Active structural control -järjestelmästä löydettiin suuri virhe, joka olisi voinut pahentaa värähdysten vaikutusta.

Formaaleja menetelmiä on myös sovellettu tietoturvaprotokolliin. Menetelmillä on löydetty joitakin virheitä. Toisaalta tietoturvaa on vaikeampi mallintaa ja analysoida kuin tietoliikenneprotokollia ja hardware-systeemejä. Tästä syystä tietoturva-alalla formaali verifiointi on enemmän tutkimusasteella.

Luku 2

Tilasiirtymäsysteemit

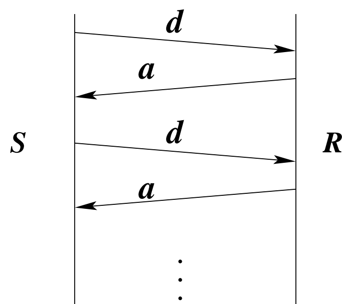
2.1 Vuorottelevan bitin protokolla

Ennen tilasiirtymäsysteemien formaalia määrittelyä näytetään esimerkki, miten käytännön tietoliikenneprotokolla voidaan spesifioida siirtymäsysteemin avulla. *Vuorottelevan bitin protokolla* (alternating bit protocol, AB-protokolla) on yksi niistä pelkistetyistä yksinkertaisista protokollista, joita on tarkasteltu paljon alan tieteellisessä kirjallisuudessa. Se on matemaattisesti elegantti ja logiikaltaan epätriviaali.

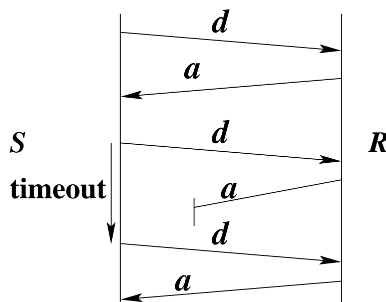
Kysymyksessä on tilanne, jossa lähettäjäprosessi S ja vastaanottajaprosessi R vaihtavat sanomia virhealttiin *vuorosuuntaisen* kanavan välityksellä. Vuorosuuntainen kanava sallii sanomien lähettämisen yhteen suuntaan kerrallaan, mutta ei molempiin suuntiin yhtäaikaa. Yhteys on altis häiriöille, joten kanava voi vääristää tai hukata minkä tahansa sen kautta lähetetyn sanoman. Sanomien järjestys kuitenkin säilyy kanavassa, eikä kanava myöskään monista sanomia.

Sanoman vastaanottaja pystyy erottelemaan vääristyneet sanomat virheettömistä esimerkiksi CRC-kentän avulla niin suurella todennäköisyydellä, että seuraavissa tarkasteluissa voidaan olettaa kaikkien vääristyneiden sanomien paljastuvan.

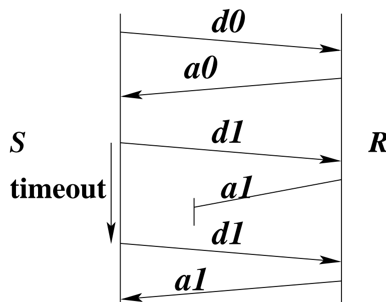
AB-protollan tavoitteena on varmistaa luotettava tiedonsiirto S:ltä R:lle. Selvästikin S:n on saatava palautetta lähettämänsä datasanoman d kohtalosta: onko d saapunut virheettömänä perille vai ei. Palaute saapuu S:lle kuittauksena a, jonka R lähettää. Negatiiviset kuittaukset eivät ole käytössä; vääristynyt sanoma ei aiheuta kuittaustoimenpidettä. Siten protokollan perustoimintaa voidaan kuvata kaaviolla



Lähettäjän täytyy siis odottaa kuittausta, ennen kuin se lähettää uuden sanoman. Edellisen skenaarion mukaan kuittauksen häviäminen aiheuttaa sen, ettei lähettäjä lähetä enää mitään, vaikka uusia datasanomia olisikin odottamassa lähettämistä. Tästä syystä otetaan käyttöön myös ajastin. Se liittyy lähettäjään ja laukeaa, jos kuittausta ei kuulu tietyn ajan kuluessa. Eräs protokollaan liittyvä skenaario voisi olla nyt seuraava:



Kuittaussanomien käyttö tuo mukanaan sanomien monistumisriskin. Jos a häviää kanavassa, S saattaa lähettää datasanoman d uudestaan arvellen sen tulleen hylätyksi linjavirheen vuoksi, vaikka d onkin tullut virheettömänä perille. Jos mitään lisätietoa ei lähetetä, vastaanottaja ei kykene erottamaan duplikaatteja. Tämän sekaannuksen välttämiseksi sanomat varustetaan numerolla. Itse asiassa riittää ottaa käyttöön vain numerot 0 ja 1. Siten edellinen skenaario tuleekin kirjoittaa muodossa



Mallinnetaan seuraavaksi lähettäjä ja vastaanottaja siirtymäsysteemeinä. Mallinnuksessa täytyy ratkaista aluksi muutamia yksityiskohtia:

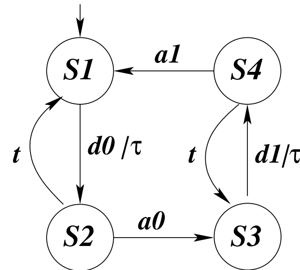
- Merkitäänkö kunkin tapahtuman yhteyteen, onko se lähetys- vai vastaanottotapahtuma?
- Miten ajastin mallinnetaan?
- Miten mallinnetaan sanoman katoaminen tai vääristyminen?

Ratkaistaan mainitut yksityiskohdat seuraavasti. Siirtymäsysteemissä ei merkitä tapahtumiin, ovatko ne lähetyksiä vai vastaanottoja. Sen, kummasta on kysymys, tulee ilmetä asiayhteydestä. Lähetykset ja vastaanotot erotetaan toisistaan vasta sitten, kun siirrytään käyttämään spesifiointikieltä siirtymäsysteemien asemesta.

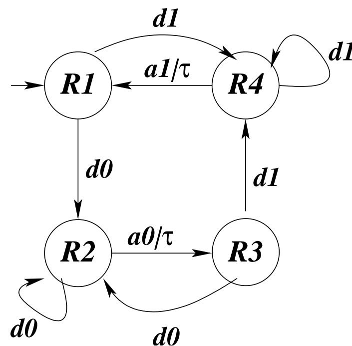
Tässä ensimmäisessä esimerkissä mallinnetaan ajastin yhdellä siirtymällä lähettäjäprosessissa. Myöhemmin siirrymme tapauksiin, joissa ajastin mallinnetaan omana prosessinaan.

Sanoman katoaminen mallinnetaan ottamalla käyttöön ns. *sisäinen tapahtuma* τ. Se sallii prosessin siirtyä toiseen tilaan ilman, että muut prosessit näkevät mitään tapahtumia. Sisäisellä siirtymällä on tärkeä rooli jatkossa muissakin yhteyksissä.

Lisäksi käytämme epädeterminismiä. Näillä sopimuksilla lähettäjä S on tilasiirtymäkaaviona seuraava:



Vastaanottaja R:

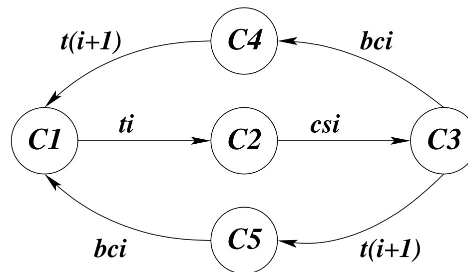


Tässä vaiheessa tarkastelemme vain yksittäisiä prosesseja. Myöhemmin nähdään, miten koko systeemin toimintaa analysoidaan.

2.2 Asiakas/palvelin-systeemi

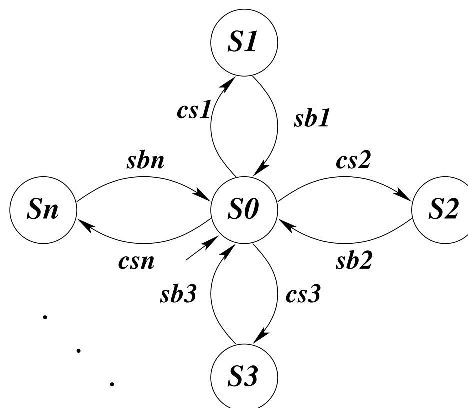
Tarkastelemme vielä toista tapausta, joka koostuu mielivaltaisen monesta prosessista. Järjestelmä koostuu palvelinprosessista S ja n:stä asiakkaasta C_i, $i = 1, \cdots, n$. Asiakas C_i pyytää palvelua S:ltä sanomalla csi. S lähettää vastauksen sbi puskuriin B_i, josta asiakas voi noutaa vastauksen sanoman bci avulla. Protokolla noudattaa round robin-periaatetta. Tämä tarkoittaa, että vuoromerkki kiertää asiakkaalta toiselle. Aina kun asiakas saa vuoromerkin, se pyytää palvelua. Tämän jälkeen se luovuttaa vuoromerkin seuraavalle. Vuoromerkki toteutetaan sanomien avulla. Kun asiakas C_i vastaanottaa sanoman ti, se voi pyytää palvelua. Tämän jälkeen se lähettää sanoman $t(i+1)$ asiakkaalle C_{i+1}. Yhteenlasku tässä tapauksessa tulkitaan niin, että $n+1 = 1$.

Asiakas C_i siirtymäsysteeminä:

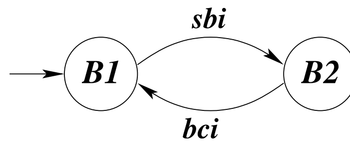


Alkutila on C1 paitsi C_1:ssä, jossa se on C2. Tällöin round robin alkaa C_1:stä.

Palveluprosessi S:



Puskuriprosessi B_i:



2.3 Tilasiirtymäsysteemin määritelmä

Lähtökohtana on malli, jossa hajautettu järjestelmä koostuu kahdesta tai useammasta *prosessista*. Prosessilla tarkoitetaan tässä yhteydessä suurin piirtein samaa kuin käyttöjärjestelmien yhteydessä:

- Prosessi on ohjelma, jonka suoritus on alkanut, mutta ei vielä pysähtynyt.
- Samasta koodista voidaan synnyttää useita prosesseja.
- Prosessi etenee diskreetein askelin tilasta toiseen; tila määräytyy muuttujien arvoista ja seuraavaksi suoritettavasta käskystä (tai käskyistä – sallimme kuvauksissa epädeterminismin).

Prosessin eteneminen tapahtuu suorittamalla jokin tapahtuma, joka voi olla esimerkiksi

- sisäinen laskenta (muuttujien päivitys yms.),
- sanoman lähetys tai vastaanotto,
- jokin muu tapahtuma, jossa on mukana toisia prosesseja.

Esimerkki viimeksi mainitusta on mm. synkronointi. Synkronointia voi tietenkin tapahtua myös lähetyksen ja vastaanoton yhteydessä.

Yleensä hajautettujen järjestelmien kuvaus siirtymäsysteemeillä tapahtuu korkealla abstraktiotasolla. Se tarkoittaa käytännössä, että huomiota kiinnitetään ennen kaikkea prosessien keskinäiseen kommunikointiin. Prosessien sisäistä toimintaa pyritään pelkistämään usein mahdollisimman paljon. Spesifiointikielet tosin tarjoavat välineitä monipuoliseen sisäisen laskennan kuvaamiseenkin, kun taas tavallisen siirtymäsysteemin yhteydessä sisäinen laskenta täytyy häivyttää melko täydellisesti.

Määritelmä 1 Siirtymäsysteemi *on rakenne* $(S, A, \longrightarrow, s_0)$*, missä*

- S *on tilojen joukko;*
- A *on tapahtumien joukko, joka myös sisältää näkymättömän tai sisäisen tapahtuman* τ*;*
- $\longrightarrow \subset S \times A \times S$ *on siirtymärelaatio;*
- s_0 *on alkutila.*

Usein S ja A ovat äärellisiä, mutta periaatteessa ne voivat olla myös numeroituvasti äärettömiä. A:n alkiot edustavat tapahtumia, joiden yhteydessä prosessit eli tässä tilasiirtymäsysteemit vaihtavat tietoa. Huomattakoon, että tässä formalismissa ei yksittäisen prosessin yhteydessä erotella, onko kysymyksessä lähetys vai vastaanotto. Siirtymäsysteemi voitaisiin määritellä niin, että tämä otettaisiin huomioon, mutta tavoitteenamme on Lotosta vastaavat siirtymäsysteemit, joissa riittää yllä esitetty rakenne.

Siirtymäsysteemistä voidaan lukea, miten prosessi etenee suorituksen aikana. Toiminta lähtee liikkeelle alkutilasta s_0. Siirtymärelaatio $\longrightarrow$ määrittelee, mitkä vaihtoehdot tilassa tulevat seuraavaksi suoritusvuoroon. Tilasiirtymäsysteemien käyttö hajautetujen järjestelmien kuvaamisessa käy parhaiten selville esimerkeistä, joista pari on jo esitelty.

Ensimmäisen luvun lopuksi esitellään vielä merkintöjä, joita tarvitaan jatkossa.

1. Jos siirtymäsysteemissä $(s_1, a, s_2) \in \longrightarrow$, niin tavallisesti kirjoitetaan $s_1 \overset{a}{\longrightarrow} s_2$.
2. Jos tilasta s_1 alkaa τ-polku

   $$s_1 \overset{\tau}{\longrightarrow} s_2 \overset{\tau}{\longrightarrow} \cdots \overset{\tau}{\longrightarrow} s_n,$$

   niin merkitään $s_1 \overset{\varepsilon}{\Longrightarrow} s_n$. Merkintä $\overset{\varepsilon}{\Longrightarrow}$ käsittää myös tapauksen, että tilasta ei siirrytä mihinkään. Siis aina pätee $s_1 \overset{\varepsilon}{\Longrightarrow} s_1$.
3. Merkintä $s_1 \overset{\tau}{\Longrightarrow} s_n$ tarkoittaa, että $s_1 \overset{\varepsilon}{\Longrightarrow} s_k \overset{\tau}{\longrightarrow} s_r \overset{\varepsilon}{\Longrightarrow} s_n$.
4. Tapahtumajono $u = a_1 a_2 \cdots a_n$ johtaa tilasta r tilaan s, jos on olemassa polku

   $$r \overset{\varepsilon}{\Longrightarrow} r_1 \overset{a_1}{\longrightarrow} s_1 \overset{\varepsilon}{\Longrightarrow} r_2 \overset{a_2}{\longrightarrow} s_2 \overset{\varepsilon}{\Longrightarrow} \cdots \overset{\varepsilon}{\Longrightarrow} r_n \overset{a_n}{\longrightarrow} s_n \overset{\varepsilon}{\Longrightarrow} s.$$

   Tällöin merkitään $r \overset{u}{\Longrightarrow} s$.

Luku 3

Yhteistilaverkko

3.1 Johdanto

Edellä olemme mallintaneet yksittäisiä prosesseja. Ne tosin lähettävät sanomia toisilleen, mutta prosessien kuvauksesta ei mitenkään näe koko systeemin toimintaa kokonaisuutena. Osoittautuu, että koko systeemiäkin voidaan kuvata yhdellä siirtymäsysteemillä, joka on saatu prosessien siirtymäsysteemeistä. Puhutaan ns. *yhteistilaverkosta*.

Yhteistilaverkon tilaksi tulee prosessiverkkojen tilojen vektori. Siten yhteistilaverkon yksi tila kuvaa, missä tiloissa erilliset prosessit voivat olla suorituksen aikana. Ennenkuin täsmällinen määritelmä voidaan antaa, on otettava kantaa, millä tasolla rinnakkaisuutta eli samanaikaisuutta mallinnetaan. Periaatteessa on kaksi vaihtoehtoa.

Jos mallinnus perustuu *todelliseen rinnakkaisuuteen*, mallista näkyy, mitkä tapahtumat sattuvat tapahtumaan rinnakkain ja mitkä peräkkäin. Todelliseen rinnakkaisuuteen perustuvia malleja on yritetty rakentaa, mutta ne johtavat monimutkaiseen formalismiin ja hankaliin algoritmeihin. Periaatteessa tämä mallinnusperiaate on kuitenkin tärkeä.

Toinen tapa mallintaa rinnakkaisuutta perustuu *lomitukseen*. Tällöin oletetaan, että kaksi tapahtumaa voidaan aina asettaa ajalliseen järjestykseen eli mitään rinnakkaisuutta ei itse asiassa tapahdu. Tämä periaate näyttää aluksi epäuskottavalta, mutta se toimii hyvin käytännössä. Melkein kaikki verifiointiohjelmistot perustuvat tälle idealle, sillä määritelmät ja algoritmit ovat tällöin melko yksinkertaisia. Seuraavassa esitettävä yhteistilaverkko perustuu lomitukseen.

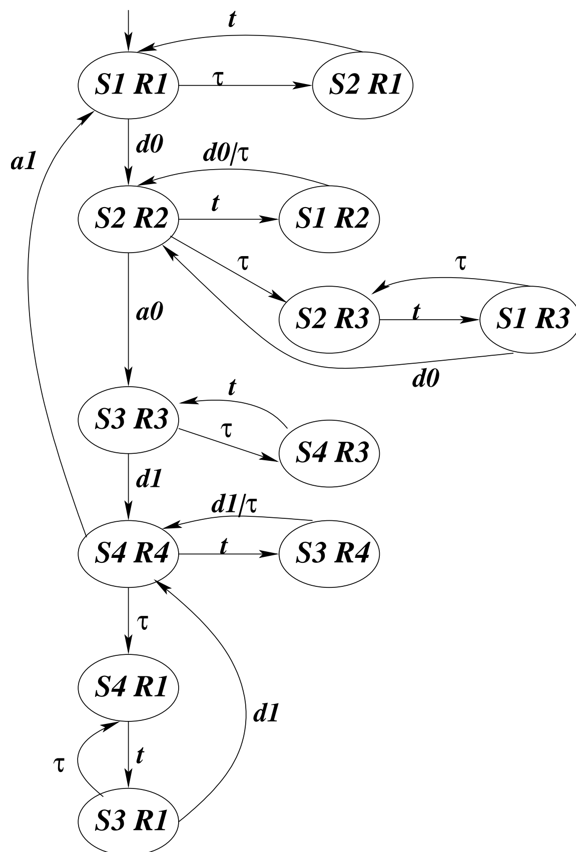
3.2 AB-protokollan yhteistilaverkko

Näytetään ensin AB-protokollan avulla konkreettisesti, mitä yhteistilaverkko merkitsee. AB-protokollassa siis lähettäjä S lähettää sanomia $d0$ ja $d1$ vastaanottajalle

R. Vastaavasti R lähettää kuittauksia $a0$ ja $a1$ S:lle. Nyt oletetaan, että lähetykset ja vastaanotot tapahtuvat synkronoidusti (tight coupling, rendezvous). Tämä tarkoittaa, että kun lähettäjä lähettää, vastaanottajan on oltava valmis ottamaan lähetys vastaan. Jos vastaaottaja ei ole valmis, ei lähettäjä voi lähettää. Siis esimerkiksi sanoman $d0$ lähetyksen yhteydessä lähettäjä siirtyy uuteen tilaan siirtymällä $d0$ ja samalla vastaanottaja siirtyy uuteen tilaan myöskin siirtymällä $d0$.

Sisäinen siirtymä τ ei aiheuta tilanvaihtoja toisissa prosesseissa. Samoin ajastintapahtuma t aiheuttaa siirtymän vain lähettäjäprosessissa.

Yhteistilaverkon tila on AB-protokollan tapauksessa pari (Si, Rj), missä Si on lähettäjän ja Rj vastaanottajan tila. Piirretään nyt protokollan yhteistilaverkko. Ajastimesta ei oleteta mitään. Siten se voi laueta turhan aikaisin, mutta tämän ei pitäisi johtaa virheisiin.



Yhteistilaverkosta nähdään, ettei lukkiumia esiinny. Siitä näkyy myös protokollan perussykli. Erilaiset tiedonsiirto- ja ajastinvirheet näkyvät poikkeamina perussyklistä, mutta jokaisessa tilanteessa päästään takaisin perussykliin. Siten yhteistilaverkon tarkastelun perusteella protokolla näyttää toimivan oikein.

3.3 Rinnakkaisoperaattori

Seuraavaksi määrittelemme yhteistilaverkon formaalisti rinnakkaisoperaattorin avulla. Olkoot P ja Q siirtymäsysteemejä. Oletetaan, että P:n tilat ovat $P_1, P_2, \cdots, P_m$ ja Q:n tilat $Q_1, Q_2, \cdots, Q_n$. Koko systeemin alkutila olkoon (P_1, Q_1). Yhteistilaverkon tilajoukon muodostavat parit (P_i, Q_j), $i = 1, \cdots, m$, $j = 1, \cdots, n$. Kirjoitamme $P_i \xrightarrow{a} P_{i'}$, jos P:n tilasta P_i on siirtymä tapahtumalla a P:n tilaan $P_{i'}$. Vastaavasti Q:n tapauksessa.

Prosessien P ja Q yhteistilaverkko $P|[a_1, \cdots, a_k]|Q$ määritellään nyt formaalisti antamalla säännöt, joiden mukaan asetetaan siirtymät tilasta toiseen. Yllä tapahtumat $a_1, \cdots, a_k$ ovat *synkronointitapahtumia*, joiden yhteydessä prosessit synkronoituvat ja suorittavat tapahtuman yhtäaikaa. Jos toinen prosesseista ei voi suorittaa synkronointitapahtumaa, ei toinenkaan saa sitä suorittaa. Muita tapahtumia prosessit voivat suorittaa vapaasti toisistaan riippumatta.

Rinnakkaisoperaattorin $|[a_1, \cdots, a_k]|$ avulla määritellään nyt siirtymät tilojen (eli tilaparien) välille seuraavasti. Aluksi vaaditaan, että $\tau \neq a_i$ kaikilla $i = 1, \cdots, k$. Kun tilapariin (P_i, Q_j) sovelletaan rinnakkaisoperaattoria, merkitään synkronointitapahtumat näkyviin kirjoittamalla $P_i|[a_1, a_2, \cdots, a_k]|Q_j$. Seuraavien sääntöjen perusteella saadaan määrättyä siirtymät parista (P_i, Q_j) (pari kirjoitetaan siis muotoon $P_i|[a_1, a_2, \cdots, a_k]|Q_j$) pariin $(P_{i'}, Q_{j'})$ (eli $P_{i'}|[a_1, a_2, \cdots, a_k]|Q_{j'}$).

1. Jos $a \in \{a_1, \cdots, a_k\}$, $P_i \xrightarrow{a} P_{i'}$ ja $Q_j \xrightarrow{a} Q_{j'}$, niin
$$P_i|[a_1, a_2, \cdots, a_k]|Q_j \xrightarrow{a} P_{i'}|[a_1, a_2, \cdots, a_k]|Q_{j'}.$$

2. Jos $a \notin \{a_1, \cdots, a_k\}$ ja $P_i \xrightarrow{a} P_{i'}$, niin
$$P_i|[a_1, a_2, \cdots, a_k]|Q_j \xrightarrow{a} P_{i'}|[a_1, a_2, \cdots, a_k]|Q_j.$$

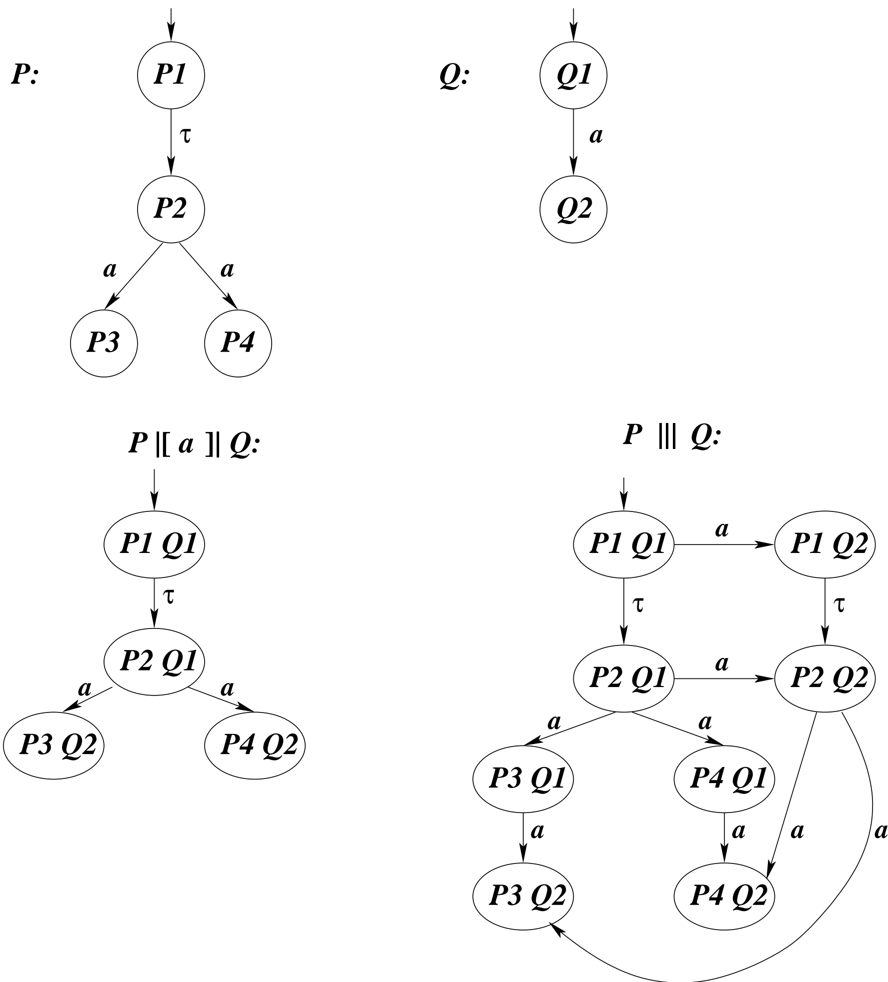
3. Jos $a \notin \{a_1, \cdots, a_k\}$ ja $Q_j \xrightarrow{a} Q_{j'}$, niin
$$P_i|[a_1, a_2, \cdots, a_k]|Q_j \xrightarrow{a} P_i|[a_1, a_2, \cdots, a_k]|Q_{j'}.$$

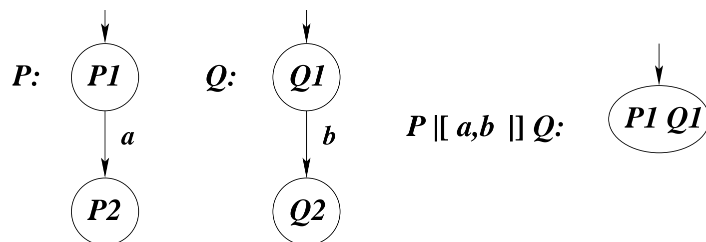
Sovelluksissa synkronointijoukko valitaan tilanteen mukaan. Jos $k = 0$ eli yhtään tapahtumaa ei ole mukana, puhutaan täydellisestä lomituksesta. Sille käytetään merkintää $|||$. Jos synkronointijoukkoon halutaan kaikki näkyvät tapahtumat, käytetään merkintää $||$.

Lopulliseen yhteistilaverkkoon otetaan tietenkin vain yhteistilat, jotka voidaan saavuttaa alkutilasta. Se merkitään kirjoittamalla rinnakkaisoperaattori synkronointijoukkoineen prosessien nimien väliin. Edellä generoitu AB-protokollan yhteistilaverkko vastaa siis seuraavaa rinnakkaisoperaattorilla saatua lauseketta:

$$S|[d0, d1, a0, a1]|R$$

Seuraavassa on muutamia esimerkkejä lisää:





3.4 Rinnakkaisoperaattorin ominaisuuksia

3.4.1 Monisynkronointi

Rinnakkaisoperaattori tukee usean prosessin yhtäaikaista synkronointia. Esimerkiksi lausekkeessa

```
P |[a]| (Q |[a]| R)
```

toiminto a voi esiintyä Q:ssa ja R:ssä vain, jos molemmat prosessit osallistuvat siihen. Toisaalta

```
Q |[a]| R
```

on myös siirtymäsysteemi, joten a voi tapahtua siinä (ja siis Q:ssa ja R:ssä) ja P:ssä vain, jos molemmat osallistuvat siihen. Siten a:han osallistuvat kaikki kolme prosessia.

3.4.2 Synkronoinnin luonne

Synkronointi on symmetristä. Ei erotella, kuka aloittaa synkronoinnin ja kuka on vastaanottaja. Tämä tulee erityisen selvästi esille siinä, että sanoman välitykseen osallistuvat sekä lähettäjä että vastaanottaja tasavertaisina.

Synkronointi on nimetöntä. Synkronointiin valmis oleva prosessi tarjoaa synkronointia ympäristön kanssa kykenemättä kuitenkaan ohjaamaan synkronointitarjousta millekään tietylle prosessille. Systeemin rakenne ratkaisee, mitkä prosessit osallistuvat synkronointitapahtumaan.

3.4.3 Rinnakkaisoperaattorin liitännäisyys

Yleisesti ottaen rinnakkaisoperaattori ei ole liitännäinen eli ei päde

$$P \,|A_1|\, (Q \,|A_2|\, R) = (P \,|A_1|\, Q) \,|A_2|\, R.$$

Seuraavissa tapauksissa liitännäisyys on kuitenkin voimassa:

1. *CSP:n tapaus.* Olkoon P, Q ja R prosesseja ja A_P, A_Q, A_R prosessien toimintojoukot. Tällöin

   $$P\,|A_P \cap (A_Q \cup A_R)|\,(Q\,|A_Q \cap A_R|\,R) \equiv (P\,|A_P \cap A_Q|\,Q)\,|(A_P \cup A_Q) \cap A_R|\,R,$$

   missä '$\equiv$ tarkoittaa, että vastaavat siirtymäsysteemit ovat tilojen nimeämistä vaille samat.

2. Mielivaltaisella synkronointijoukolla B

   $$P\,|B|\,(Q\,|B|\,R) \equiv (P\,|B|\,Q)\,|B|\,R.$$

3. Synkronointijoukoilla B_1 ja B_2 pätee

   $$P\,|B_1|\,(Q\,|B_2|\,R) \equiv (P\,|B_1|\,Q)\,|B_2|\,R,$$

   jos $A_P \cap B_2 = \emptyset$ and $A_R \cap B_1 = \emptyset$.

Perustelut. Kohdassa 1 riittää osoittaa, että jokaista siirtymää toisessa systeemissä vastaa sama siirtymä toisessakin. Joudutaan tarkastelemaan monia vaihtoehtoja.

a) Oletetaan ensin, että $a \notin A_P \cap (A_Q \cup A_R)$ ja

$$P\,|A_P \cap (A_Q \cup A_R)|\,(Q\,|A_Q \cap A_R|\,R) \xrightarrow{a} P'\,|A_P \cap (A_Q \cup A_R)|\,(Q\,|A_Q \cap A_R|\,R).$$

Tällöin $a \in A_P$ ja siten $a \notin A_Q \cup A_R$. Siis $a \notin (A_P \cup A_Q) \cap A_R$ ja $a \notin A_P \cap A_Q$. Mutta tässä tapauksessa myös

$$(P\,|A_P \cap A_Q|\,Q)\,|(A_P \cup A_Q) \cap A_R|\,R \xrightarrow{a} (P'\,|A_P \cap A_Q|\,Q)\,|(A_P \cup A_Q) \cap A_R|\,R.$$

b) $a \notin A_P \cap (A_Q \cup A_R)$, $a \notin A_Q \cap A_R$ ja

$$P\,|A_P \cap (A_Q \cup A_R)|\,(Q\,|A_Q \cap A_R|\,R) \xrightarrow{a} P\,|A_P \cap (A_Q \cup A_R)|\,(Q'\,|A_Q \cap A_R|\,R).$$

c) $a \notin A_P \cap (A_Q \cup A_R)$, $a \in A_Q \cap A_R$ ja

$$P\,|A_P \cap (A_Q \cup A_R)|\,(Q\,|A_Q \cap A_R|\,R) \xrightarrow{a} P\,|A_P \cap (A_Q \cup A_R)|\,(Q'\,|A_Q \cap A_R|\,R').$$

d) $a \in A_P \cap (A_Q \cup A_R)$ ja $a \in A_Q \cap A_R$.

e) $a \in A_P \cap (A_Q \cup A_R)$ ja $a \notin A_Q \cap A_R$.

Kohta 2) todistetaan samoin kuin kohta 1), mutta kohdassa 2) on vähemmän vaihtoehtoja ($a \in B$, $a \notin B$). Kohta 3) on kohdan 1) modifikaatio.

3.5 Yhteistilaverkon käytännön toteutuksesta

3.5.1 Yhteistilaverkko tietorakenteena

Yhteistilaverkkoa käytetään kahdella tavalla. Joissakin sovelluksissa riittää kulkea yhteistilaverkossa muodostamatta sitä kokonaan. Toisissa sovelluksissa yhteistilaverkko on generoitava kokonaan. Ongelmana tällöin on usein, että verkko on suuri. Itse asiassa monet käytännön ohjelmistot ja protokollat johtavat niin suureen verkkoon, ettei sitä voida muodostaa. Erityisesti jos yhteistilaverkon avulla haluttaisiin analysoida tietoliikenneprotokollien yhteistoimintaa usealla kerroksella, joudutaan tavallisesti vaikeuksiin. Sen sijaan monia yhden kerroksen protokollia on voitu menestyksellisesti analysoida.

Yhteistilaverkko on tavallisesti harva. Siis tiloista lähtee vain muutamia siirtymiä. Siten matriisiesitykset eivät tule kysymykseen, vaan lähtökohtana on vieruslistaesitys. Verkon generointi tapahtuu normaalisti syvyyssuuntaisesti. Otetaan siis lähtökohdaksi alkutila, joka on tilapari. Generoidaan kaikki tilaparit, joihin päästään alkutilasta. Viedään nuo tilaparit pinoon. Jatketaan sen jälkeen seuraavasti kunnes pino on tyhjä: Otetaan pari pinosta, generoidaan kaikki siirtymät parista toisiin pareihin, viedään uudet parit pinoon ja vedetään kaaret. Asiaa mutkistaa hieman se, että on aina tarkistettava, onko generoitu pari uusi vai vanha. Jos generoitava verkko esitetään myös vieruslistojen avulla, solmujen nimet ovat siinä kokonaislukuja, eivät pareja. Siten joudutaan etsimään, vastaako annettua paria jokin verkossa jo oleva koknaisluku vai ei. Jos ei,niin parille on annettava seuraava vapaana oleva luku. Tämä johtaa yleensä hajautukseen.

3.5.2 Bittihajautus ja muita vaihtoehtoja

Koska yhteistilaverkko on suuri, tarvitaan myös suuri hajautustaulukko. Olisi houkuttelevaa päästä käyttämään hyväksi virtuaalimuistia, mutta tässä on omat ongelmansa. Nimittäin parit ja kokonaisluvut tulevat vastaan sekalaisessa järjestyksessä. Tämä johtaa siihen, että virtuaalimuistin sivuja joudutaan jatkuvasti hakemaan levyltä. Se hidastaisi toimintaa liiaksi. Siis yleensä pyritään keskusmuistiratkaisuihin.

G. Holzmannin ratkaisu oli käyttää *bittihajautusta*. Tällöin pari, tai yleensä yhteistilaa kuvaava tieto, tulkitaan bittijonoksi ja se taas edelleen kokonaisluvuksi. Varataan nyt totuusarvoinen taulukko, jonka koko on sellainen, että suurin yhteistilaa vastaava bittijono luvuksi tulkittuna kuuluu vielä taulukon indekseihin. Tätä taulukkoa voidaan käyttää tehokkaasti hajautukseen, eikä sen alkioiden tieto vie kuin yhden bitin.

Itse asiassa koko siirtymäsysteemi voidaan esittää vaihtoehtoisella tavalla. Boolen funktioille on kehitetty tietyissä tilanteissa kompakti esitysmuoto ja samaa ideaa voidaan soveltaa myös siirtymäsysteemeihin. Tällöin puhutaan ns. BDD:stä (binary decision diagram). Tämäkään ratkaisu ei ole universaali ratkaisu yhteistilaverkon kokoon: joissakin tilanteissa BDD:t auttavat pienentämään esitysmuotoa, mutta ei

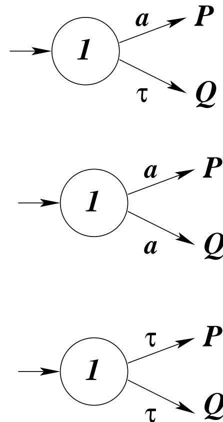
läheskään aina. Monet algoritmit toimivat lisäksi suoraviivaisemmin tavallisen vieruslistaesityksen pohjalta kuin BDD:n pohjalta.

Luku 4

Mallinnuksen perusasioita

4.1 Epädeterminismi

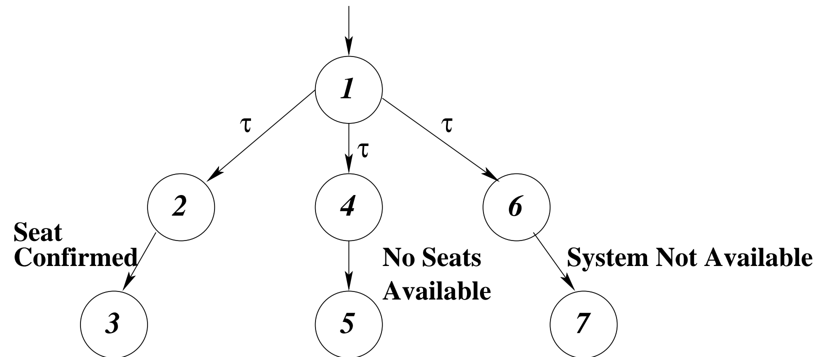
Olkoon P ja Q siirtymäsysteemejä ja a toiminto. On kolme tapaa mallintaa epädeterminismiä siirtymäsysteemeillä:



Tapauksessa 1 ympäristöllä on mahdollisuus vaikuttaa systeemiin, jos a tapahtuu "ennen"sisäistä tapahtumaa τ. Tapauksessa 2 systeemi päättää ympäristön reagoinnin yhteydessä, kumman a:n se valitsee. Tapaus 3 ilmaisee, että systeemi päättää sisäisesti, käyttäytyykö se P:n vai Q:n mukaisesti.

Esimerkiksi tietoliikennekanavaa mallinnettaessa on tarkoituksenmukaista valita vaihtoehto 3, jos P merkitsee sanoman välittämistä ja Q sanoman häviämistä. Ympäristö, tässä tapauksessa siis lähettäjä ja vastaanottaja, eivät voi tällöin vaikuttaa häviääkö sanoma vai ei; se on kokonaan kanavan sisäinen asia. Jos käytettäisiin vaihtoehtoa 1, ympäristö voi vaikuttaa kanavan käyttäytymiseen, mikä joissakin tilanteissa johtaa systeemin virheelliseen toimintaa mallinnusvirheen takia.

Toisena esimerkkinä on lentokoneen paikanvarausjärjestelmä, jonka erästä toimintoa kuvaa siirtymäsysteemi:



Eli paikanvarauspyynnön tulos on täysin epämääräinen asiakkaan näkökulmasta, sillä normaali asiakas ei tiedä, kuinka systeemi on rakennettu tai onko vapaita paikkoja jäljellä. Asiakkaan kyvyttömyyttä vaikuttaa systeemiin mallinnetaan sisäisillä tapahtumilla.

4.2 Kanavat ja ympäristö

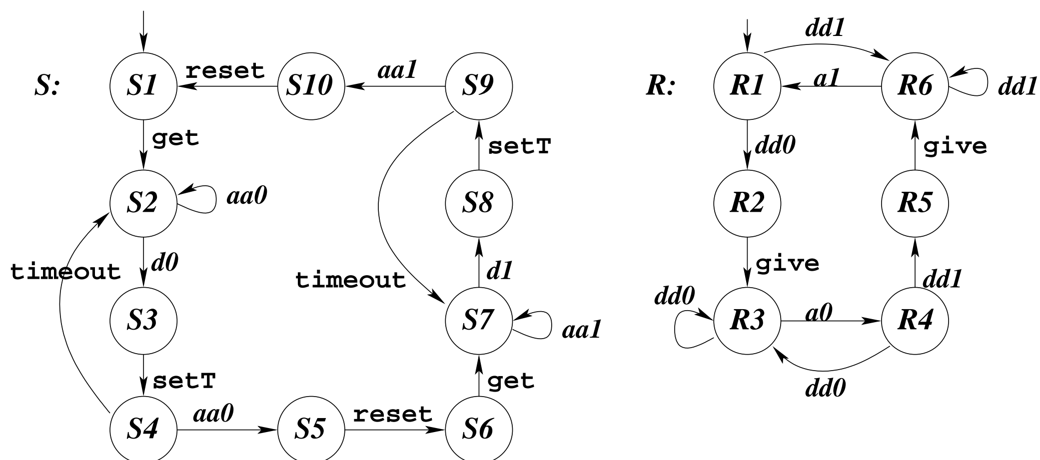
Kommunikointi tietokoneverkoissa ei yleensä ole synkronista. Kuitenkin rinnakkaisoperaattori vaatii synkronista tiedonvälitystä. Asynkroninen tiedonvälitys saadaan aikaan spesifioimalla kanava siirtymäsysteeminä. Prosessi lähettää sanoman kanavaan synkronisesti ja jatkaa sitten toimintaansa. Toinen prosessi ottaa sanoman kanavasta synkronisesti sitten, kun se sille sopii. Tällä tavalla saadaan aikaan kahden prosessin välille asynkroninen sanomanvälitys. Hintana on tosin se, että yhteistilaverkko kasvaa. Tämä riippuu toisinaan voimakkaasti siitä, kuinka monta sanomaa kanavaan voidaan lähettää ilman, että sieltä poistuu sanomia.

Toisinaan on tarpeen mallintaa myös varsinaisen protokollan ympäristöä. Esimerkiksi AB-protokollassa voidaan olettaa, että lähettäjä saa datasanomia `get`-sanomassa ympäristöltä ja vastaanottaja luovuttaa vastaanottamansa datasanomat muodossa `give` omalle ympäristölleen.

Mallinnamme nyt AB-protokollan oikeaoppisemmin kanavien ja ympäristöjen avulla. Lisäksi mallinnetaan ajastin erillisenä prosessina. Tehdään seuraavat sopimukset:

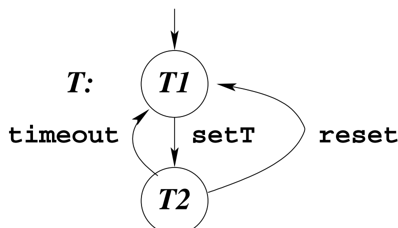
- Lähettäjä lähettää kanavaan sanomia $d0$ ja $d1$.
- Vastaanottaja ottaa kanavasta sanomia $dd0$ ja $dd1$.
- Vastaanottaja lähettää kanavaan sanomat $a0$ ja $a1$.
- Lähettäjä vastaanottaa kanavasta sanomat $aa0$ ja $aa1$.
- Lähettäjä ja ympäristö kommunikoivat synkronisesti sanoman `get` avulla.
- Vastaanottaja ja ympäristö kommunikoivat synkronisesti sanoman `give` avulla.
- Lähettäjä virittää ajastimen synkronisesti tapahtuman `setT` avulla.
- Ajastin ilmoittaa laukeamisesta lähettäjälle synkronisesti tapahtuman `timeout` avulla.
- Lähettäjä ilmoittaa ajastimelle synkronisesti sanomalla `reset`, että ajastin voi palata alkutilaan.

Ohessa on prosessit siirtymäsysteeminä. Kanavan kohdalla on oletettu, että vain yksi sanoma voi olla kanavassa kerrallaan. Samalla sanoman katoaminen tai vääristyminen on laitettu kanavan tehtäväksi, ei lähettäjäprosessin.

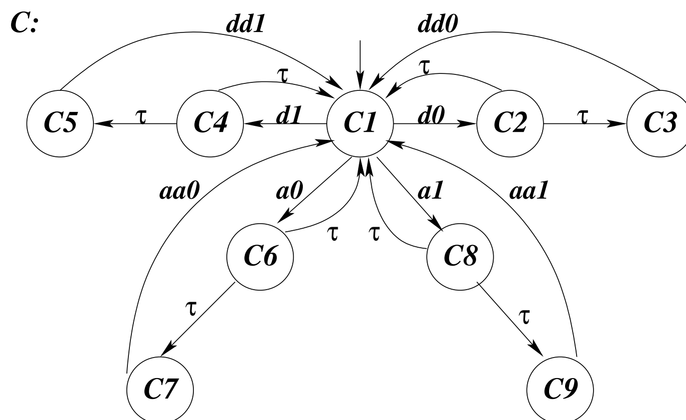


Lähettäjäprosessiin on tullut muutama uusi siirtymä (tilat S2 ja S7), joissa otetaan vastaan kuittaus. Tämä johtuu kanavasta ja hätäilevästä ajastimesta. Nimittäin ajastin voi laueta, vaikka kuittaus onkin tulossa, tosin vasta kanavassa. Jotta kanava voisi luovuttaa sanomansa ja ottaa vastaan uusia sanomia, lähettäjän on otettava vastaan se, mitä kanavasta on tulossa. Jos näin ei tehtäisi, seuraisi lukkiuma.

Ajastinprosessi voidaan esittää kahden tilan avulla.



Kanavaprosessi on mallinnettu siten, että ympäristö ei voi vaikuttaa siihen katoaako sanoma vai ei. Kanavaprosessi siis vastaanottaa datasanoman numerolla 0 tapahtumalla $d0$ ja toimittaa sen eteenpäin tapahtumalla $dd0$. Vastaavasti kanava vastaanottaa datasanoman numerolla 1, kuittaussanoman numerolla 0 ja 1 tapahtumilla $d1$, $a0$ ja $a1$. Kanava toimittaa nämä sanomat eteenpäin tapahtumilla $dd1$, $aa0$ ja $aa1$.



AB=((S |[timeout, reset, setT]| T)

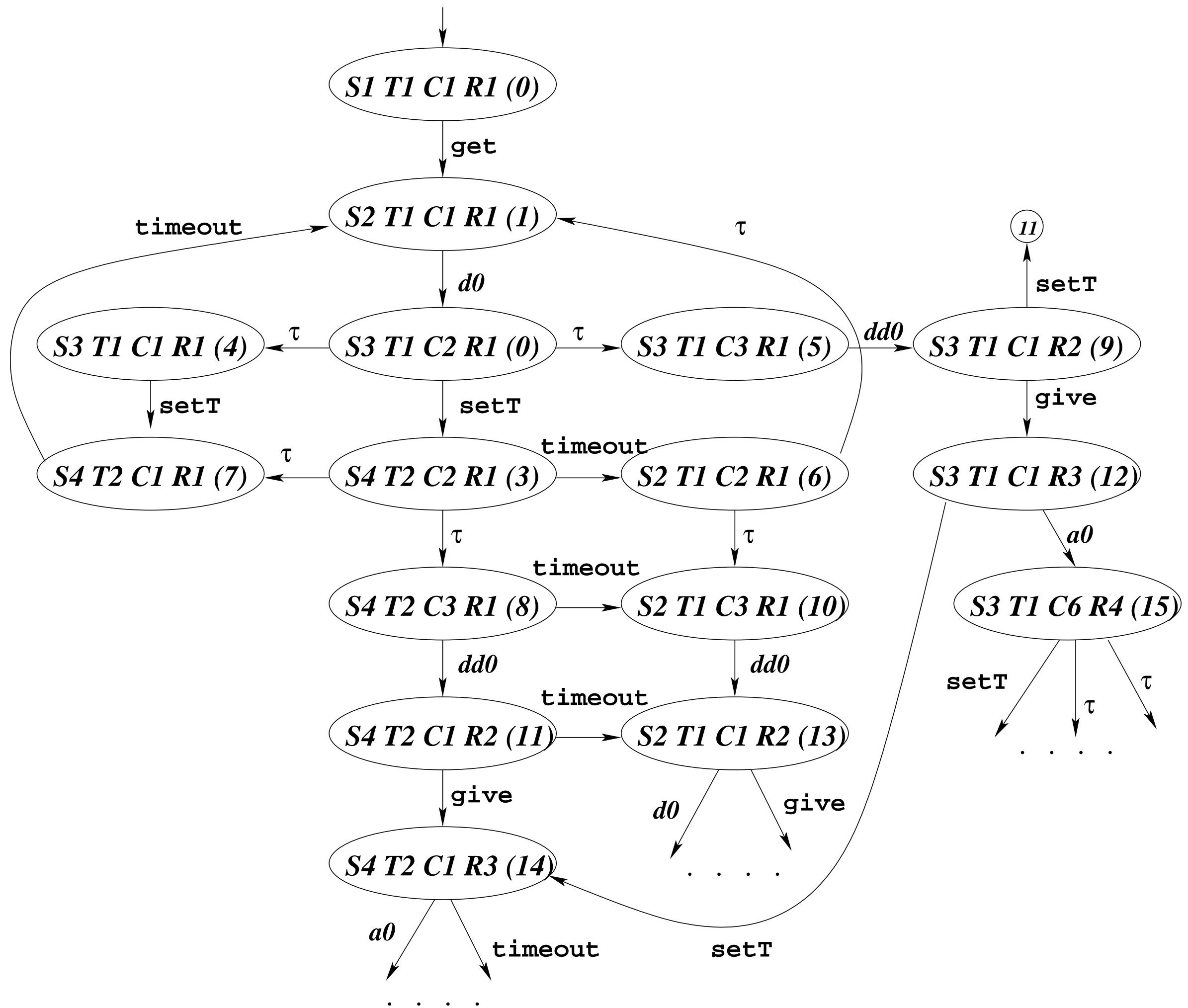
|[*d0,d1,aa0,aa1*]|

C)

|[*dd0,dd1,a0,a1*]|

R

Yhteistilaverkko muodostuu nyt neljän prosessin yhdistelmästä. Se on oleellisesti monimutkaisempi kuin aiemman protokollaversion yhteistilaverkko, joten sen piirtäminen käsin kokonaan ei ole järkevää. Ohessa on sen alkua.



Luku 5

Ekvivalenssit ja verifiointi

5.1 Palvelun kuvaukset

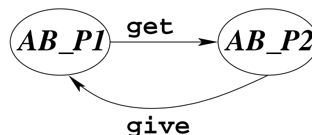
Hajautetun järjestelmän prosessien kuvauksista voidaan mekaanisesti generoida yhteistilaverkko. Jos verkko on pienehkö, alle miljoona tilaa, on mahdollista käydä verkko systemaattisesti läpi ja etsiä virheitä, esimerkiksi lukkiumia, eläviä lukkiumia (solmusta ei ole polkua perussykliin) yms. On kuitenkin melko vaikeaa tällä tavoin löytää kaikkia virheitä. Esimerkiksi sanomia voi kadota, vaikka ei jouduttaisikaan lukkiumiin, sama sanoma voidaan luovuttaa kaksi kertaa käyttäjälle jne.

Jos verkko on liian suuri tai ääretön, voidaan kuitenkin suorittaa satunnaiskulku verkossa. Jos spesifikaatiossa on virhe, se yleensä esiintyy monessa kohdassa verkkoa (lomitussemantiikka!). Vaikka solmuista käytäisiin läpi vain 5%, paljastuu kokemuksen mukaan suurin osa virheistä.

Jos kuitenkin halutaan verifioida spesifikaatio täydellisemmin, tarvitaan toisenlainen lähestymistapa. Prosessialgebrojen yhteydessä tavallisimmin käytetään ekvivalensseihin ja temporaalilogiikkaan perustuvia verifiointimenetelmiä. Käsittelemme tällä kurssilla pelkästään ekvivalenssimenetelmää.

Ekvivalenssiin perustuvassa verifioinnissa keskeinen käsite on *palvelun kuvaus.* Tällä tarkoitetaan siirtymäsysteemiä, joka kuvaa sitä palvelua, jonka protokolla antaa käyttäjälle (ympäristölle, havaitsijalle). Tarkastellaan kahta esimerkkiä.

AB-protokolla tarjoaa tiedonsiirtopalvelun. Protokolla ottaa vastaa datapaketteja ympäristöltä (ylemmältä kerrokselta) ja välittää ne vastaanottavalle osapuolelle (ylempi kerros, ympäristö). AB-protokollan palvelu on siten helppo kuvata:



Ekvivalenssiin perustuvassa verifioinnissa verrataan nyt varsinaisen AB-protokollan yhteistilaverkkoa palvelukuvauksen verkkoon. Jos ne ovat tietyssä mielessä samoja, AB-protokollaa voidaan pitää oikeana, ts. se tekee sen, mitä sen odotetaan tekevän. Verkkojen vertailussa täytyy selvästikin abstrahoida jompaa kumpaa tai molempia verkkoja. Ekvivalenssin määrittelyssä päätetään, millä tarkkuustasolla verkkoja vertaillaan.

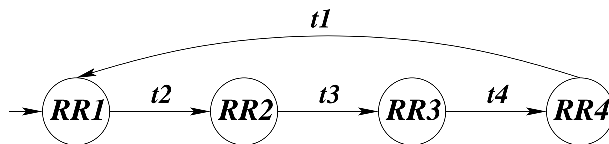
Toisena esimerkkinä tarkastellaan luvussa 1 esiteltyä asiakas/palvelin-systeemiä, kun asiakkaita on 4. Koko systeemiä vastaa rinnakkaisoperaattorin avulla annettu prosessilauseke:

```
SystemRR :=

Server
   | [cs1, sb1, cs2, sb2, cs3, sb3, cs4, sb4] |
((Client1  |[bc1]| Buffer1  )
   | [t1, t2] |
((Client2  |[bc2]| Buffer2 )
   | [t3] |
((Client3 |[bc3]| Buffer3 )
   | [t4] |
(Client4 |[bc4]| Buffer4 ))))
```

Oletetaan nyt, että meitä kiinnostaa round robin- periaatteen toteutuminen systeemissä. Periaate toteutuu, jos ti:t tapahtuvat järjestyksessä

$$t2,\ \ t3,\ \ t4,\ \ t1.$$

Siten round robin- periaatetta kuvaa prosessi



Voisimme nyt generoida prosessin `SystemRR` yhteistilaverkon ja selvittää, tapahtuvatko ti:t mainitussa järjestyksessä. Jos emme halua tutkia verkkoa käsin, meidän pitäisi kirjoittaa ohjelma, joka selvittäisi asian. Tähän kuluisi aikaa.

Nopeampi menetelmä on verrata prosessia `SystemRR` prosessiin `RR`. Siis round robin- periaate toteutuu systeemissä, jos prosessin `SystemRR` toiminta sopivasti abstrahoituna vastaa prosessin `RR` toimintaa. Tässä tapauksessa sopiva abstraktio on sellainen, että muutetaan kaikki muut tapahtumat näkymättömiksi tapahtumiksi τ lukuunottamatta tapahtumia $t1$, $t2$, $t3$ ja $t4$. Jos tämän muutoksen jälkeen kuljetaan polkuja prosessin `SystemRR` yhteistilaverkossa, niin tapahtumien ti tulisi esiintyä poluilla RR:n järjestyksessä, eikä muita näkyviä tapahtumia esiinny lainkaan.

Prosessialgebrojen vahvana puolena on, että voidaan määritellä täsmällisesti useita eri tarkoituksiin soveltuvia ekvivalensseja, jotka voidaan tehokkaasti laskea prosessialgebrallisesti määritellyille prosesseille. Yksinkertaisin ekvivalenssi *suoritusjälkiekvivalenssi*. Eräs perustavimmista ekvivalensseista on *heikko bisimulaatioekvivalenssi*, joka riittää useimpiin tarkoituksiin. Se on tehokkaasti laskettavissa ja se on yleensä toteutettu kaikissa yleiskäyttöisissä verifiointiohjelmistoissa. Käsittelemme tällä kurssilla pelkästään näitä ekvivalensseja.

5.2 Relaatiot

Relaation määritelmä

Olkoon A ja B joukkoja. Jokaista joukkoa $R \subset A \times B$ sanotaan *relaatioksi joukosta A joukkoon B*. Joukko

$$M_R = \{\, x \in A \mid \exists\, y \in B \ \text{ siten että } \ (x, y) \in R \,\}$$

on relaation R *määrittelyjoukko* ja joukko

$$A_R = \{\, y \in B \mid \exists\, x \in A \ \text{ siten että } \ (x, y) \in R \,\}$$

sen *arvojoukko*. Jos $R \subset A \times A$, toisin sanoen jos R on relaatio joukosta A joukkoon A, niin sanotaan lyhyemmin, että R on *joukon A relaatio*. Jos R on joukon A relaatio ja jos $(x, y) \in R$, niin yleensä merkitään xRy.

Ekvivalenssirelaatiot

Määritellään ensin muutamia käsitteitä, joista on hyötyä ekvivalenssirelaation käsittelyssä.

Joukot A ja B ovat *alkiovieraat*, jos $A \cap B = \emptyset$. Jos $\mathcal{I}$ on joukko, jonka alkiot ovat joukkoja, niin $\mathcal{I}$ on alkiovieras, jos kaksi joukkoon $\mathcal{I}$ kuuluvaa joukkoa ovat aina alkiovieraat.

Joukon X osajoukkojoukko $\mathcal{H}$ on joukon X *ositus*, jos se täyttää seuraavat ehdot:

H1. Jokainen $A \in \mathcal{H}$ on epätyhjä,

H2. $\mathcal{H}$:n joukkojen yhdiste on X,

H3. $\mathcal{H}$ on alkiovieras.

Osituksella on läheinen yhteys ekvivalenssirelaatioon, joka määritellään seuraavassa.

Määritelmä 2 *Joukon X relaatio R on* ekvivalenssi, *jos se täyttää seuraavat ehdot:*

E1. *aRa jokaisella $a \in X$ (refleksiivisyys);*

E2. *jos aRb, niin myös bRa (symmetrisyys);*

E3. *jos aRb ja bRc, niin myös aRc (transitiivisuus).*

Jos $a \in X$, niin joukko $R(a) = \{x \in X \mid aRx\}$ on alkion a ekvivalenssiluokka *ekvivalenssin R suhteen.*

Lause. *Jos R on joukon X ekvivalenssi, niin sen kaikkien eri ekvivalenssiluokkien joukko X/R on joukon X ositus. Joukon X alkioille a ja b on aRb aina ja vain kun a ja b kuuluvat samaan ekvivalenssiluokkaan.*

Todistus. Käydään todistus läpi, vaikka sama tehdään myös matematiikan peruskursseilla.

Olkoon $a \in X$. Koska aRa, niin $a \in R(a)$. Tästä seuraa, että jokainen ekvivalenssiluokka on epätyhjä ja että ekvivalenssiluokkien yhdiste on X. Ehdon H3 osoittamiseksi riittää näyttää, että kaksi annettua R-ekvivalenssiluokkaa ovat joko identtiset tai alkiovieraat. Oletetaan sitä varten, että $R(a) \cap R(b) \neq \emptyset$, jolloin on olemassa alkio $c \in R(a) \cap R(b)$. Olkoon $x \in R(a)$ eli aRx. Koska $c \in R(a)$, niin aRc ja siis myös cRa, sillä ekvivalenssirelaatio on symmetrinen. Koska cRa ja aRx, niin transitiivisuuden perusteella cRx. Koska $c \in R(b)$, on toisaalta bRc. Saatu tulos osoittaa, että $R(a) \subset R(b)$. Täsmälleen samalla tavalla nähdään, että $R(b) \subset R(a)$. Näinollen $R(a) = R(b)$ ja kaikkien R-ekvivalenssiluokkien joukko on siis joukon X ositus.

Olkoon aRb. Tällöin $b \in R(a)$, joten a ja b kuuluvat samaan ekvivalenssiluokkaan $R(a)$. Oletetaan kääntäen, että a ja b kuuluvat samaan ekvivalenssiluokkaan $R(c)$. Tällöin $a \in R(c) \cap R(a)$ ja siis $R(c) \cap R(a) \neq \emptyset$, joten lauseen alkuosan nojalla $R(c) = R(a)$. Tästä seuraa, että $b \in R(c) = R(a)$ eli aRb. Myös jälkimmäinen väite on siten oikea. □

Lause. *Olkoon $\mathcal{H}$ joukon X ositus. Jos joukon X alkioille asetetaan $aR_{\mathcal{H}}b$ aina ja vain kun a ja b kuuluvat samaan joukkoon $U \in \mathcal{H}$, niin $R_{\mathcal{H}}$ on sellainen joukon X ekvivalenssi, että kaikkien $R_{\mathcal{H}}$-ekvivalenssiluokkien joukko on $\mathcal{H}$.*

Todistus. Olkoon $a \in X$. Koska $\mathcal{H}$ on joukon X ositus, on olemassa joukko $U \in \mathcal{H}$, jolla $a \in U$. Nyt a ja a kuuluvat samaan osituksen joukkoon U, joten $aR_{\mathcal{H}}a$. Eli refleksiivisyys on osoitettu.

Olkoon $aR_{\mathcal{H}}b$. Tällöin a ja b kuuluvat samaan joukkoon $U \in \mathcal{H}$ ja siis myös $bR_{\mathcal{H}}a$. Eli symmetrisyys on osoitettu.

Olkoon sitten $aR_{\mathcal{H}}b$ ja $bR_{\mathcal{H}}c$. Tällöin on olemassa sellaiset joukot U ja $V \in \mathcal{H}$, että a ja $b \in U$ sekä b ja $c \in V$. Koska $b \in U \cap V$, on $U \cap V \neq \emptyset$ ja siis $U = V$, koska $\mathcal{H}$ on alkiovieras. Näinollen a ja c kuuluvat samaan joukkoon $U = V \in \mathcal{H}$, joten $aR_{\mathcal{H}}c$. Transitiivisuus on siten osoitettu ja $R_{\mathcal{H}}$ on siis ekvivalenssirelaatio.

Olkoon $R_{\mathcal{H}}(a)$ mielivaltainen $R_{\mathcal{H}}$-ekvivalenssiluokka. Koska $\mathcal{H}$:n joukkojen yhdiste on X, on olemassa sellainen $U \in \mathcal{H}$ että $a \in U$. Jos $x \in U$, niin a ja x kuuluvat samaan joukkoon $U \in \mathcal{H}$, joten $aR_{\mathcal{H}}x$ eli $x \in R_{\mathcal{H}}(a)$. Olkoon kääntäen $x \in R_{\mathcal{H}}(a)$ eli $aR_{\mathcal{H}}x$. Tällöin a ja x kuuluvat samaan joukkoon $V \in \mathcal{H}$. Koska $a \in U \cap V$, on $U \cap V \neq \emptyset$ ja siis $U = V$, koska $\mathcal{H}$ on alkiovieras. Näinollen $x \in V = U$. Saadut tulokset osoittavat, että $R_{\mathcal{H}}(a) = U$. Kääntäen, jos $U \in \mathcal{H}$, niin $U \neq \emptyset$. Jos $a \in U$, niin edellisen tuloksen nojalla $R_{\mathcal{H}}(a) \in \mathcal{H}$. Koska $R_{\mathcal{H}}(a) \cap U \neq \emptyset$, niin on siis $U = R_{\mathcal{H}}(a)$. Näinollen yhtyy kaikkien $R_{\mathcal{H}}$-ekvivalenssiluokkien joukko joukkoon $\mathcal{H}$. □

Transitiivinen sulkeuma

Olkoon R relaatio joukossa V. Relaation potenssit määritellään seuraavasti:

$$\begin{aligned} R^0 &= \{(a,a) \mid a \in V\}, \\ R^1 &= R, \\ R^2 &= \{(a,c) \mid \exists b \in V: \ aRb \text{ ja } bRc\}, \\ R^n &= R(R^{n-1}), \ n > 2. \end{aligned}$$

Transitiivinen sulkeuma määritellään nyt relaation potenssien avulla. Relaation R *refleksiivinen transitiivinen sulkeuma* R^* on joukko

$$R^* = \bigcup_{i=0}^{\infty} R^i,$$

ja *transitiivinen sulkeuma* R^+ on joukko

$$R^+ = \bigcup_{i=1}^{\infty} R^i.$$

Purkamalla auki määritelmiä nähdään, että aR^*b, jos on olemassa V:n alkiot $a = c_1, c_2, \cdots, c_n = b$, joilla $c_i R c_{i+1}$, $i = 1, \cdots, n-1$.

Relaatio R joukossa V voidaan esittää suoraviivaisesti suunnattuna verkkona: verkon solmujoukko on V ja jos aRb, niin (a,b) on verkon kaari. Verkossa transitiivisella sulkeumalla on havainnollinen tulkinta. Nimittäin R^+ tarkoittaa kaikkia sellaisia pareja $(a,b) \in V \times V$, että a:sta on polku b:hen verkossa R. Vastaavasti R^* on R^+ lisättynä kaarilla jokaisesta solmusta solmuun itseensä.

Nyt voidaan formuloida transitiivisen sulkeuman laskentaongelma. On annettu relaatio R verkkona. Laske R^+ tai R^* verkkona. Relaation verkkoesitys voi perustua joko vierusmatriisiin tai vieruslistaan.

Tarkastellaan ensin vierusmatriisiesitystä. Eräs parhaiten tunnettuja transitiivisen sulkeuman laskenta-algoritmeja on Warshallin algoritmi. Siinä oletetaan, että relaatio on annettu $n \times n$-vierusmatriisina M. Algoritmissa muodostetaan matriisi C, joka kuvaa transitiivista sulkeumaa.

```
   Warshall(M)

   begin

1.   C := M;
2.   for i := 1 to n do C[i,i] := true; end for;
  // rivi 2 jää pois, jos lasketaan vain
  // transitiivista sulkeumaa.
3.   for k := 1 to n do
4.      for i := 1 to n do
5.         for j := 1 to n do
6.            if C[i,k] = true ja C[k,j] = true then
                 C[i,j] := true;
              end if;
           end for;
        end for;
     end for;
   end.
```

Lause 1 *Warshallin algoritmi laskee oikein refleksiivisen transitiivisen sulkeuman ja vaatii ajan $\mathcal{O}(n^3)$ ja tilan $\mathcal{O}(n^2)$.*

Monissa sovelluksissa verkko on varsin suuri ja harva, joten vierusmatriisi ei tule kysymykseen. Transitiivisen sulkeuman laskeminen vieruslistaesityksen perusteella on kuitenkin periaatteessa yksinkertaista: aloitetaan syvyyssuuntainen etsintä verkon jokaisesta solmusta ja vedetään kaari aloitussolmusta jokaiseen saavutettavaan solmuun. Tällaisen menetelmän aikavaatimus on $\mathcal{O}(|V| \times (|E| + |V|))$ eli sama kuin Warshallin algoritmissa, jos verkko on lähes täydellinen ($|E| = |V|^2$). Ongelmana tässä yksinkertaisessa lähestymistavassa on se, että samoja reittejä saatetaan kulkea moneen kertaan. Tehokkaampaan tulokseen päästään, jos vahvasti yhtenäiset komponentit otetaan huomioon. Komponentin sisällä transitiivisen sulkeuman kaaret ovat jokaisen solmuparin välillä. Sen jälkeen tarvitsee tutkia kaaret komponenttien välillä. Idea on suoraviivainen, mutta tähän ideaan perustuvat algoritmit näyttävät olevan melko monimutkaisia. Yhtenä syynä on, että vahvasti yhtenäisten

komponenttien etsintäalgoritmi on monimutkainen, erityisesti oikeellisuustodistuksensa osalta. Tällä kurssilla nämä kehittyneemmät sulkeuma-algoritmit sivuutetaan. Mainitaan vain muutamien algoritmien viitteet:

- Eve ja Kurki-Suonio, On computing the transitive closure of a relation. Acta Informatica 8, 303-314, 1977.

  Eräs ensimmäisiä sulkeuma-algoritmeja, joissa on sovellettu vahvasti yhtenäisiä komponentteja.

- Nuutila, An efficient transitive closure algorithm for cyclic digraphs. Information Processing Letters 52, 207-213, 1994.

  Esimerkki uudemmasta sulkeumatutkimuksesta.

- Sippu, Soisalon-Soininen, Parsing Theory, 37-60, Springer-Verlag 1988.

  Kirjatason esitys tehokkaasta sulkeuma-algoritmista.

Transitiivisen sulkeuman laskeminen etukäteen ei aina ole välttämätöntä, vaikka sovelluksessa sulkeumaa tarvittaisiinkin. Joissakin tilanteissa sulkeuma voidaan *laskea lennosta* (on the fly). Tämä tarkoittaa, että kun tarvitaan sulkeuman kaaria jostain solmusta, käynnistetään syvyyssuuntainen etsintä tuosta solmusta ja etsitään kaikki tarvittavat polut. Saattaa vaikuttaa, että menetelmä on käyttökelvoton, varsinkin jos joudutaan käynnistämään etsintä samasta solmusta moneen kertaan. Kuitenkin on sovelluksia, joissa tämä lähestymistapa johtaa jopa parempaan tulokseen kuin sulkeuman laskeminen ennakolta. Konkreettisessa tilanteessa on siten analysoitava huolellisesti, mikä lähestymistapa on tehokkain.

5.3 Suoritusjälkiekvivalenssi

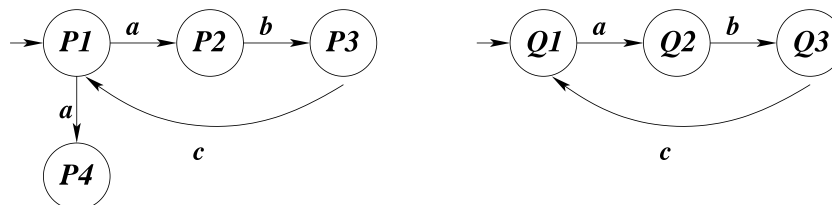
Yksinkertaisin ekvivalenssi perustuu tapahtumajonojen vertailuun. Olkoon A tapahtumien joukko. Seuraavassa oletetaan, että kaikkien prosessien tapahtumat kuuluvat tähän joukkoon.

Määritelmä 3 *Olkoon $u \in (A\backslash\{\tau\})^*$ tapahtumajono. Jono u on prosessin P* suoritusjälki *(trace), jos $P \overset{u}{\Longrightarrow} P'$ jollakin prosessilla P'. Merkitään P:n kaikkien suoritusjälkien joukkoa symbolilla* $\mathrm{tr}(P)$.

Määritelmä 4 *Prosessit P ja Q ovat* suoritusjälkiekvivalentit, *$P \approx_{tr} Q$, jos* $\mathrm{tr}(P) = \mathrm{tr}(Q)$.

Selvästi $\approx_{tr}$ on ekvivalenssirelaatio. Se on myös kompositionaalinen rinnakkaisoperaattorin suhteen. Eli jos $P \approx_{tr} P'$ ja $Q \approx_{tr} Q'$, niin $P|[a_1, \cdots, a_n]|Q \approx_{tr} P'|[a_1, \cdots, a_n]|Q'$ (harjoitustehtävä).

Jos $P \approx_{tr} Q$, niin P:ssä voi olla lukkiumia, vaikka Q:ssa ei niitä olisikaan. Esimerkiksi prosessit P ja Q alla ovat suoritusjälkiekvivalentit:



Yleensä ajatellaan, että lukkiumat ovat kaikkein vakavimpia virheitä hajautetuissa järjestelmissä. Tämän vuoksi suoritusjälkiekvivalenssi harvoin tulee kysymykseen ainoana perusteena vertailla protokollaa ja palvelua. Suoritusjälkiekvivalenssi paljastaa kuitenkin melko tehokkaasti muita virhetyyppejä. Lisäksi lukkiumat on helppo tarkistaa jo verkon generoinnin yhteydessä. Siten suoritusjälkiekvivalenssia voidaan käyttää hyväksi silloin tällöin. Tulemme soveltamaan sitä tutkiessamme erilaisia ratkaisuja keskinäisen poissulkemisen ongelmaan. Edelleeen suoritusjälkiekvivalenssi toimii lähtökohtana kokonaiselle ekvivalenssiryhmälle, johon kuuluvat mm. testi- ja estymäekvivalenssit.

5.4 Heikko bisimulaatioekvivalenssi

Tavoitteena on määritellä verifiointiin sopiva ekvivalenssirelaatio siirtymäsysteemien joukkoon. Bisimulaatioekvivalenssi on Milnerin kehittelemä ja Parkin viimeistelemä ekvivalenssi 70- ja 80-lukujen vaihteesta. Jos P ja Q ovat prosesseja, niin ekvivalenssin ideana on simuloida P:n näkyvien tapahtumien suoritusta Q:ssa ja päinvastoin. Jos simulointi onnistuu koko ajan, prosessit ovat ekvivalentteja, muuten eivät. Täsmälliseen määritelmään tarvitaan apukäsitteitä.

Olkoon a tapahtuma, $a \neq \tau$. Palautetaan mieleen merkintä, että

$$P \stackrel{a}{\Longrightarrow} P',$$

jos on olemassa siirtymäketju

$$P = P_1 \stackrel{\tau}{\longrightarrow} P_2 \stackrel{\tau}{\longrightarrow} \cdots \stackrel{\tau}{\longrightarrow} P_k \stackrel{a}{\longrightarrow} P_{k+1} \stackrel{\tau}{\longrightarrow} P_{k+2} \stackrel{\tau}{\longrightarrow} P_{k+3} \stackrel{\tau}{\longrightarrow} \cdots \stackrel{\tau}{\longrightarrow} P_{k+m} = P',$$

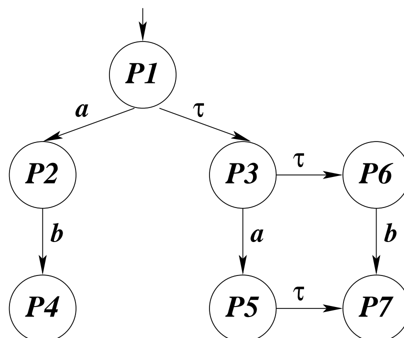
$k \geq 1$, $m \geq 0$. Toisin sanoen $P \stackrel{a}{\Longrightarrow} P'$, jos on olemassa polku P:n alkutilasta P':n alkutilaan ja yksi polun kaarista sisältää toiminon a, muut, 0 tai useampi kaari, τ:n. Voidaan myös kirjoittaa

$$P \stackrel{\varepsilon}{\Longrightarrow} P'$$

jos $P = P'$ tai on olemassa τ-siirtymien ketju

$$P = P_1 \stackrel{\tau}{\longrightarrow} P_2 \stackrel{\tau}{\longrightarrow} \cdots \stackrel{\tau}{\longrightarrow} P_k = P',$$

$k > 1$.



Esimerkki. Tarkastellaan yllä olevaa tilasiirtymäsysteemiä

Sen tilasta $P1$ on seuraavat polut: $P1\stackrel{\varepsilon}{\Longrightarrow}P6$, $P1\stackrel{a}{\Longrightarrow}P2$, $P1\stackrel{a}{\Longrightarrow}P5$, $P1\stackrel{a}{\Longrightarrow}P7$, $P1\stackrel{b}{\Longrightarrow}P7$, $P1\stackrel{\varepsilon}{\Longrightarrow}P3$, $P1\stackrel{\varepsilon}{\Longrightarrow}P1$. □

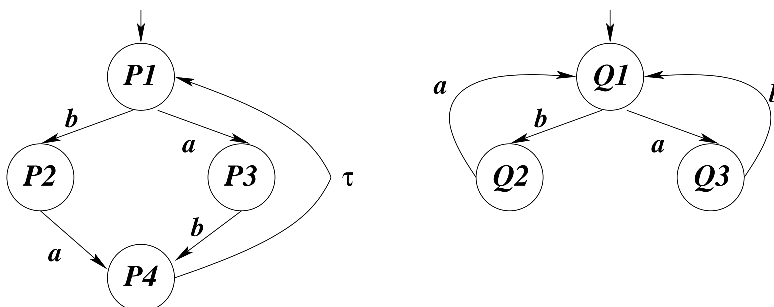
Määritelmä. *Olkoot P ja Q prosesseja ja A P:n ja Q:n toimintojen joukko. Prosessit P ja Q ovat* heikosti bisimilaariset*, $P \approx_{wbis} Q$, jos on olemassa sellainen prosessipareista koostuva joukko $\mathcal{R}$ (*heikko bisimulaatio*), että kaikilla toiminnoilla $a \in (A \setminus \{\tau\}) \cup \{\varepsilon\}$ pätee:*

1. *$(P, Q) \in \mathcal{R}$;*

2. *jos $(P_1, Q_1) \in \mathcal{R}$ ja $P_1 \stackrel{a}{\Longrightarrow} P_2$, niin on olemassa Q_2, jolla $Q_1 \stackrel{a}{\Longrightarrow} Q_2$ ja $(P_2, Q_2) \in \mathcal{R}$;*

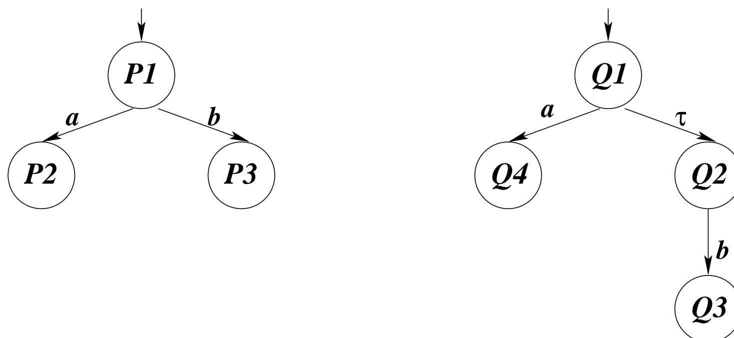
3. *jos $(P_1, Q_1) \in \mathcal{R}$ ja $Q_1 \stackrel{a}{\Longrightarrow} Q_2$, niin on olemassa P_2, jolla $P_1 \stackrel{a}{\Longrightarrow} P_2$ ja $(P_2, Q_2) \in \mathcal{R}$.*

Esimerkki. Seuraavat prosessit ovat heikosti bisimilaariset eli $P \approx_{wbis} Q$:



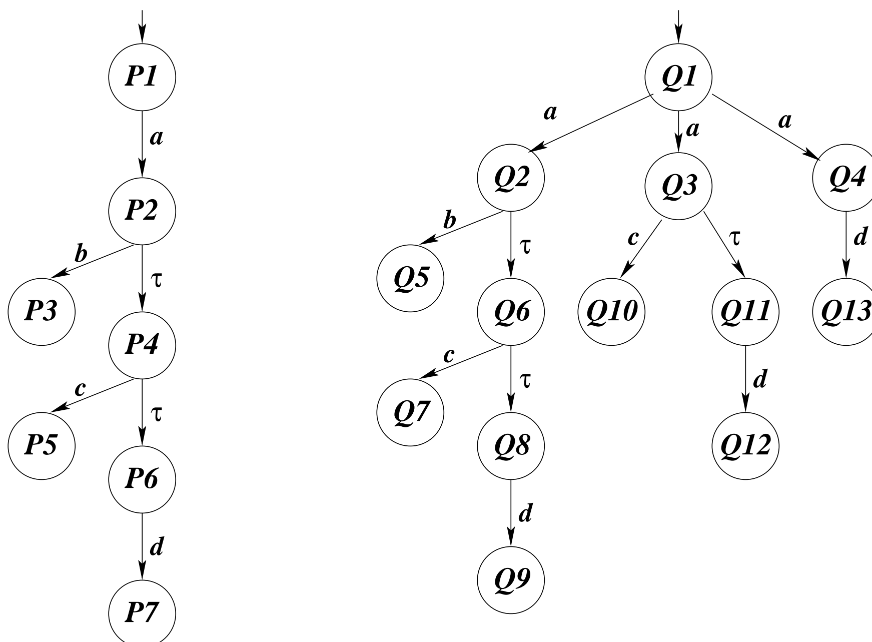
sillä $\mathcal{R} = \{(P1, Q1), (P2, Q2), (P3, Q3), (P4, Q1)\}$ on heikko bisimulaatio ja $(P, Q) \in \mathcal{R}$ $(P = P1,\ Q = Q1)$. □

Esimerkki. Seuraavat prosessit eivät ole heikosti bisimilaariset:



Jos nimittäin yritetään muodostaa heikko bisimulaatiorelaatio $\mathcal{R}$, niin parin $(P1, Q1)$ täytyy kuulua relaatioon. Käytetään seuraavaksi määritelmän ehtoa 3: Q tekee siirtymän tilasta $Q1$ tilaan $Q2$ sisäisellä tapahtumalla. Ainoa tapa simuloida tätä P:n osalta on, että P pysyy tilassa $P1$. Siis parin $(P1, Q2)$ tulee kuulua myös relaatioon $\mathcal{R}$. Käytetään tämän jälkeen pariin $(P1, Q2)$ määritelmän ehtoa 2: P siirtyy $P1$:stä a:lla $P2$:een. Nytpä Q ei voikaan simuloida tätä siirtymää, sillä $Q2$:sta ei lähde yhtään a-siirtymää. Siis heikkoa bisimulaatiorelaatiota ei voi olla olemassa P:n ja Q:n välillä, joten prosessit eivät ole heikosti bisimilaariset. □

Esimerkki. Seuraavat prosessit ovat heikosti bisimilaariset:



Heikko bisimulaatio on seuraava

$$
\begin{array}{l}
(P1, Q1),\ \ (P2, Q2),\ \ (P4, Q3),\ \ (P6, Q4),\\
(P3, Q5),\ \ (P4, Q6),\ \ (P5, Q7),\ \ (P6, Q8),\ \ (P7, Q9)\\
(P5, Q10),\ \ (P6, Q11),\ \ (P7, Q12),\ \ (P7, Q13)
\end{array}
$$

□

Lause. *Relaatio $\approx_{wbis}$ on ekvivalenssirelaatio siirtymäsysteemien välillä.*

Todistus. On osoitettava, että relaatio $\approx_{wbis}$ on refleksiivinen, symmetrinen ja transitiivinen.

Refleksiivisuus tarkoittaa, että $P \approx_{wbis} P$ kaikilla prosesseilla P. Symmetrisyys tarkoittaa, että ehdosta $P \approx_{wbis} Q$ seuraa $Q \approx_{wbis} P$. Sekä refleksiivisuus että symmetrisyys seuraavat suoraan määritelmästä.

On vielä näytettävä, että relaatio on transitiivinen eli ehdoista $P \approx_{wbis} Q$ ja $Q \approx_{wbis} R$ seuraa $P \approx_{wbis} R$. Olkoon $\mathcal{R}$ bisimulaatio P:n ja Q:n välillä, $\mathcal{S}$ bisimulaatio Q:n ja R:n välillä. Muodostetaan prosessiparien joukko $\mathcal{T}$ seuraavasti:

$$\mathcal{T} = \{(P_1, R_1) |\ \exists Q_1 :\ (P_1, Q_1) \in \mathcal{R},\ (Q_1, R_1) \in \mathcal{S}\}.$$

Osoitetaan, että $\mathcal{T}$ on heikko bisimulaatio P:n ja R:n välillä. $\mathcal{T}$:n määritelmän perusteella $(P, R) \in \mathcal{T}$. Olkoon $(P_1, R_1) \in \mathcal{T}$ mielivaltainen ja $P_1 \stackrel{a}{\Longrightarrow} P_2$. Tiedetään, että on olemassa Q_1, jolla $(P_1, Q_1) \in \mathcal{R}$ ja $(Q_1, R_1) \in \mathcal{S}$. Koska $\mathcal{R}$ ja $\mathcal{S}$ ovat heikkoja bisimulaatioita, on olemassa myös prosessit Q_2 ja R_2, joilla $Q_1 \stackrel{a}{\Longrightarrow} Q_2$ ja $(P_2, Q_2) \in \mathcal{R}$ sekä $R_1 \stackrel{a}{\Longrightarrow} R_2$ ja $(Q_2, R_2) \in \mathcal{S}$. Mutta $\mathcal{T}$:n määritelmän nojalla $(P_2, R_2) \in \mathcal{T}$, joten ehto 2 heikon bisimulaation määritelmässä on osoitettu. Ehto 3 osoitetaan samalla tavalla. □

Merkitään $\mathcal{P}$:llä prosessien eli äärellisten siirtymäsysteemien joukkoa. Relaatio $\approx_{wbis}$ määrittelee siis $\mathcal{P}$:hen ekvivalenssirelaation. Matematiikassa joukon ekvivalenssirelaatio määritellään joukon karteesisen tulon itsensä kanssa osajoukkona. Siten tapauksessamme pitäisi olla $\approx_{wbis} \subset \mathcal{P} \times \mathcal{P}$. Voidaankin määritellä, että $\approx_{wbis}$ on maksimaalinen bisimulaatio eli

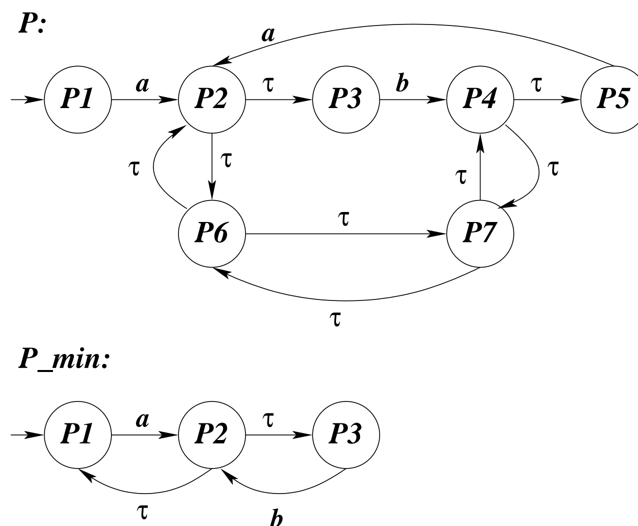
$$\approx_{wbis} = \bigcup \{\mathcal{R} |\ \mathcal{R} \text{ on bisimulaatio}\}.$$

Jos P on prosessi, niin P:n ekvivalenssiluokka $[P]_{\approx_{wbis}}$ on niiden prosessien joukko, jotka ovat heikosti bisimilaariset P:n kanssa. Ekvivalenssiluokat $[P]_{\approx_{wbis}}$ muodostavat $\mathcal{P}$:n

Minimiprosessi

Jos P on siirtymäsysteemi $(S, A, \longrightarrow, s_0)$, niin jokaista tilaa $s \in S$ voidaan myös pitää siirtymäsysteeminä, joka on muuten sama kuin P paitsi että alkutilana on s. Voimme nyt rajoittaa ekvivalenssin $\approx_{wbis}$ joukkoon $S \times S$. Tällöin $\approx_{wbis}$ on ekvivalenssirelaatio joukossa S ja se jakaa S:n ekvivalenssiluokkiin. Voimme nyt muodostaa uuden siirtymäsysteemin, jonka tiloina ovat kyseiset ekvivalenssiluokat. Ekvivalenssiluokasta on siirtymä a:lla johonkin toiseen ekvivalenssiluokkaan, jos luokan jostain tilasta on siirtymä a:lla toisen luokan johonkin tilaan alkuperäisessä siirtymäsysteemissä. Näin saatu siirtymäsysteemi on tilojen suhteen minimaalinen niiden siirtymäsysteemien joukossa, jotka ovat heikosti bisimilaarisia alkuperäisen siirtymäsysteemin kanssa.

Esimerkiksi seuraavat prosessit ovat ekvivalentteja ja jälkimmäinen on minimaalinen.



Edellä olevassa esimerkissä prosessin P tilojen ekvivalenssijoukot ovat $E_1 = \{P1, P5\}$, $E_2 = \{P2, P4, P6, P7\}$ ja $E_3 = \{P3\}$.

Piirtämällä kaikki kaaret mekaanisesti ekvivalenssiluokasta toiseen johtaa helposti tilanteeseen, jossa on paljon turhia kaaria. Kaarten minimointi on monimutkaisempaa kuin tilojen minimointi. Asiaa on käsitelty 90-luvulla Jaana Elorannan väitöskirjassa ja artikkelissa Eloranta, Tienari, Valmari: Essential Transitions To Bisimulation Equivalences, Theoretical Computer Science 179 (1997) 397-419.

Heikko bisimilaarisuus ja rinnakkaisoperaattori

Heikko bisimulaatioekvivalenssi käyttäytyy hyvin rinnakkaisoperaattorin suhteen.

Lause. *Jos* $P \approx_{wbis} Q$, *niin* $P\,|B|\,R \approx_{wbis} Q\,|B|\,R$ *kaikilla toimintojoukoilla* B *ja prosesseilla* R.

Todistus. Olkoon
$$\mathcal{R} = \{(P_1|B|P_3, P_2|B|P_3) \mid P_1 \approx_{wbis} P_2\}.$$
Osoitetaan, että $\mathcal{R}$ on heikko bisimulaatio $P|B|R$:n ja $Q|B|R$:n välillä. Koska $P \approx_{wbis} Q$, pätee $(P|B|R, Q|B|R) \in \mathcal{R}$. Olkoon $P|B|R \overset{a}{\Longrightarrow} P'|B|R'$. Joudutaan tarkastelemaan kahta tapausta.

i) $a \in B$. Nyt $a \neq i$ ja $P \overset{a}{\Longrightarrow} P'$, $R \overset{a}{\Longrightarrow} R'$. Koska $P \approx_{wbis} Q$, on olemassa bisimulaatio $\mathcal{E}$ P:n ja Q:n välillä. Tällöin tiedetään heikon bisimulaation määritelmän nojalla, että on olemassa sellainen Q', että $Q \overset{a}{\Longrightarrow} Q'$ ja $(P', Q') \in \mathcal{E}$. Siis myös $P' \approx_{wbis} Q'$. Suoraan nähdään, että $Q|B|R \overset{a}{\Longrightarrow} Q'|B|R'$. Nyt $\mathcal{R}$:n määritelmän perusteella $(P'|B|R', Q'|B|R') \in \mathcal{R}$.

ii) $a \notin B$. Nyt joko $P \overset{a}{\Longrightarrow} P'$ ja $R \Longrightarrow R'$ tai $P \Longrightarrow P'$ ja $R \overset{a}{\Longrightarrow} R'$. Huomaa, että a voi olla ε. Jos $P \overset{a}{\Longrightarrow} P'$, niin myös $Q \overset{a}{\Longrightarrow} Q'$ ja $P' \approx_{wbis} Q'$, kuten nähtiin edellisessä kohdassa. Siten $Q|B|R \overset{a}{\Longrightarrow} Q'|B|R'$ ja $(P'|B|R', Q'|B|R') \in \mathcal{R}$. Jos taas $R \overset{a}{\Longrightarrow} R'$, $P \Longrightarrow P'$, niin myös tällöin on olemassa Q', jolla $Q \Longrightarrow Q'$ ja $P' \approx_{wbis} Q'$. Edelleen $Q|B|R \overset{a}{\Longrightarrow} Q'|B|R'$. Myös nyt pätee $(P'|B|R', Q'|B|R') \in \mathcal{R}$.

Siis $\mathcal{R}$ täyttää heikon bisimulaation ehdot $P|B|R$:n siirtymien osalta. Samalla tavalla osoitetaan $Q|B|R$:n siirtymien tapaus ja $\mathcal{R}$:n mielivaltaisen alkion siirtymien tapaus.
□

5.5 Heikon bisimulaation laskeminen

Seuraavassa esitettävä algoritmi laskee siirtymäsysteemin tilojen ekvivalenssijoukon. Algoritmia voidaan käyttää myös kahden erillisen prosessin P ja Q ekvivalenssivertailuun. Nimittäin muodostetaan yksi prosessi ottamalla käyttöön uusi alkutila, josta vedetään τ-siirtymät P:n ja Q:n alkutiloihin. Sen jälkeen lasketaan, ovatko tässä prosessissa tilat P ja Q ekvivalentteja. Jos ovat, ovat tietenkin myös vastaavat prosessit ekvivalentteja.

Heikon bisimulaatioekvivalenssin laskemista hallitsee ajallisesti relaatioiden $\overset{a}{\Longrightarrow}$ laskeminen. Yhden relaation $\overset{a}{\Longrightarrow}$ laskeminen on lähes sama asia kuin sen aliverkon transitiivisen sulkeuman laskeminen, joka koostuu pelkästään a-kaarista. Tämähän vie pahimmassa tapauksessa aikaa lähes $\mathcal{O}(n^3)$, missä n on tilojen lukumäärä. Käytännössä on osoittautunut, ettei tässä tapauksessa kannata laskea transitiivisia sulkeumia etukäteen, vaan a-polkuja etsitään tarpeen mukaan. Seuraavassa algoritmissa ei kuitenkaan oteta kantaa, miten relaatiot $\overset{a}{\Longrightarrow}$ määrätään. Esitämme algoritmista yksinkertaisemman version. Monimutkaisempia tietorakenteita soveltamalla

saadaan aikaan periaatteessa hieman tehokkaampi algoritmi, mutta transitiivisten sulkeumien laskeminen syö hyödystä suuren osan pois. Algoritmi perustuu Paigen ja Tarjanin artikkeliin Three partition refinement algorithms, SIAM J. Computing 16 (6), 1987. Fernandezin artikkeli An implementation of an efficient algorithm for bisimulation equivalence, Science of Computer Programming 13: 219-236, 1989, on paras esitys algoritmin sovelluksesta bisimulaatioekvivalenssin laskentaan.

Olkoon $P = (S, A, T, s_0)$ siirtymäsysteemi. Algoritmissa lasketaan tilajoukon S ositus ρ, joka edustaa algoritmin loputtua ekvivalenssiluokkia. Toisin sanoen ekvivalenssiluokassa olevat tilat ovat keskenään ekvivalentteja ja ne voidaan samastaa esimerkiksi minimiverkkoa muodostettaessa. Alussa ρ koostuu yhdestä joukosta, tilajoukosta S. Jos B on S:n osajoukko ja a toiminto, niin algoritmissa käytetään merkintää

$$T_a^{-1}[B] = \{s \in S |\ s \overset{a}{\Longrightarrow} s',\ s' \in B\}.$$

Merkintä $I_{a,B}^{1,2}$ tarkoittaa joukkokokoelmaa

$$\{X\ \cap\ T_a^{-1}[B] |\ X \in I_{a,B}\} \bigcup \{X \setminus T_a^{-1}[B] |\ X \in I_{a,B}\}$$

ja riveillä 4-9 lasketaan operaattorin Φ arvo $\Phi(\rho, \rho)$, joka määritellään seuraavasti. Olkoon $\rho' = \{B_1, \cdots, B_n\}$ perhe S:n osajoukkoja ja ρ S:n ositus. Määritellään

$$\Phi(\rho', \rho) = (\Phi_{B_1} \circ \cdots \circ \Phi_{B_n})(\rho),$$

missä $\Phi_B = \Phi_{a_1,B} \circ \cdots \circ \Phi_{a_n,B}$, jos $a_1, \cdots, a_n$ ovat A:n alkioita, ja

$$\Phi_{a,B}(\rho) = \{X\ \cap\ T_a^{-1}[B] |\ X \in \rho\} \bigcup \{X \setminus T_a^{-1}[B] |\ X \in \rho\}.$$

Algoritmi laskee kuvauksen $\tilde{\Phi}(\rho) = \Phi(\rho, \rho)$ maksimaalisen kiintopisteen ja se osoittautuu olevan maksimaalinen bisimulaatio. Fernandezin esittämä algoritmin oikeellisuustodistus on pitkällinen, mutta samalla elegantti. Se on esitetty myös Aaro Hallikaisen pro gradu -tutkielmassa vuodelta 1994.

Algoritmi: Heikko bisimilaarisuus.

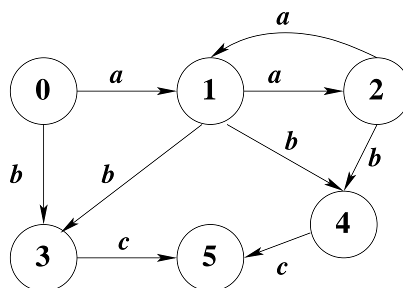
Input: Äärellinen prosessi $P = (S, A, T, s_0)$.

Output: S:n ositus ρ; se edustaa ekvivalentteja tiloja.

Metodi:

1. **begin**
2. $\quad W := \{S\}; \quad \rho := \{S\};$
3. $\quad$ **repeat**
4. $\quad\quad$ choose and remove any $B \in W$;
5. $\quad\quad$ **for** each $a \in ((A \setminus \{\tau\}) \cup \{\epsilon\})$ **do**
6. $\quad\quad\quad I_{a,B} := \{X \in \rho |\ X \cap T_a^{-1}[B] \neq \emptyset,\ X \not\subset T_a^{-1}[B]\};$
7. $\quad\quad\quad I_{a,B}^{1,2} := \{X \cap T_a^{-1}[B] |\ X \in I_{a,B}\} \bigcup \{X \setminus T_a^{-1}[B] |\ X \in I_{a,B}\};$
8. $\quad\quad\quad \rho := (\rho \setminus I_{a,B}) \cup I_{a,B}^{1,2};$
9. $\quad\quad\quad W := (W \setminus I_{a,B}) \cup I_{a,B}^{1,2};$
10. $\quad\quad$ **endfor**;
11. $\quad$ **until** $W = \emptyset$;
12. **end**.

Esimerkki. Sovelletaan algoritmia seuraavaan siirtymäsysteemiin:



Alussa $\rho = \{\{0,1,2,3,4,5\}\}$, $W = \{\{0,1,2,3,4,5\}\}$. Sitten

$$
\begin{aligned}
B &= \{0,1,2,3,4,5\} \\
T_a^{-1}[B] &= \{0,1,2\} \\
I_{a,B} &= \{\{0,1,2,3,4,5\}\} \\
I_{a,B}^{1,2} &= \{\{0,1,2\},\{3,4,5\}\} \\
\rho &= \{\{0,1,2\},\{3,4,5\}\} \\
W &= \{\{0,1,2\},\{3,4,5\}\}
\end{aligned}
$$

$$
\begin{aligned}
T_b^{-1}[B] &= \{0,1,2\} \\
I_{b,B} &= \emptyset \\
I_{b,B}^{1,2} &= \emptyset \\
\rho &= \text{sama} \\
W &= \text{sama}
\end{aligned}
$$

$$
\begin{aligned}
T_c^{-1}[B] &= \{3,4\} \\
I_{c,B} &= \{\{3,4,5\}\} \\
I_{c,B}^{1,2} &= \{\{3,4\},\{5\}\} \\
\rho &= \{\{0,1,2\},\{3,4\},\{5\}\} \\
W &= \{\{0,1,2\},\{3,4\},\{5\}\}
\end{aligned}
$$

$$
\begin{aligned}
T_\epsilon^{-1}[B] &= \{0,1,2,3,4,5\} \\
I_{\epsilon,B} &= \emptyset \\
I_{\epsilon,B}^{1,2} &= \emptyset \\
\rho &= \text{sama} \\
W &= \text{sama}
\end{aligned}
$$

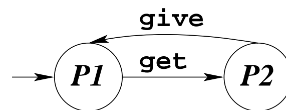
Tämän jälkeen ρ ei enää muutu, mutta algoritmi tutkii siitä huolimatta vielä W:n

joukot jokaisella kolmella toiminnolla ja ϵ:lla. Tehokkaammassa versiossa algoritmi osaa rajata tutkittavia joukkoja paremmin. Huomattakoon, että joukkojen $T_a^{-1}[B]$ laskeminen saattaa τ-siirtymien mukanaollessa vaatia pitkien polkujen tutkimista. Edellä olevassa esimerkissä ei yksinkertaisuuden vuoksi τ-siirtymiä ollut lainkaan.

5.6 Esimerkkejä

5.6.1 AB-protokolla

Tarkastellaan AB-protokollan versiota, jossa on mukana sanomat `get` ja `give`. Tämä protokolla tarjoaa ympäristölle palvelun, joka on yksinkertaisesti kuvattavissa:



Sovelletaan nyt edellä esitettyä heikkoa bisimulaatioekvivalenssia protokollan oikeellisuuden tarkistamiseen. On siis näytettävä, että AB-protokollan yhteistilaverkko on ekvivalentti palveluverkon kanssa. Selvästikään tämä ei pidä paikkaansa, ellei protokollan yhteistilaverkkoa muuteta hieman. Tavallisesti joitakin toimintoja kätketään eli muutetaan τ-siirtymiksi. Tätä varten määrittelemme operaation `hide`:

```
hide a1, a2, ..., an in P
```

muuttaa siirtymäverkkoa P siten, että kaikki toiminnot ai, $i = 1, \cdots, n$, P:ssä korvataan τ:lla.

Nyt AB-protokollan verifiointiprobleema muodostaa lausua seuraavassa muodossa. On osoitettava, että

`hide`	$d0, dd0, d1, dd1, a0, aa0, a1, aa1, st, rt, t$ in `AB-protokolla`
$\approx_{wbis}$	`AB-palvelu`.

AB-protokollan yhteistilaverkko on sen verran suuri, että sen käsittely vaatii ohjelmistoa. Verifiointi ohjelmistoa käyttäen tapahtuu kirjoittamalla sekä itse protokolla että palvelu ohjelmiston tukemalla spesifiointikielellä, kääntämällä spesifikaatiot siirtymäsysteemiksi ja käynnistämällä ohjelma, joka laskee, ovatko siirtymäsysteemit ekvivalentteja. Teemme tämän verifioinnin myöhemmin Lotoksen yhteydessä.

5.6.2 FE-protokolla

Varhaisessa protokollakirjallisuudessa (W. C. Lynch: Reliable full-duplex transmission over half-duplex lines, Comm. ACM, Vol. 11, No. 6, pp 362-372, June 1968)

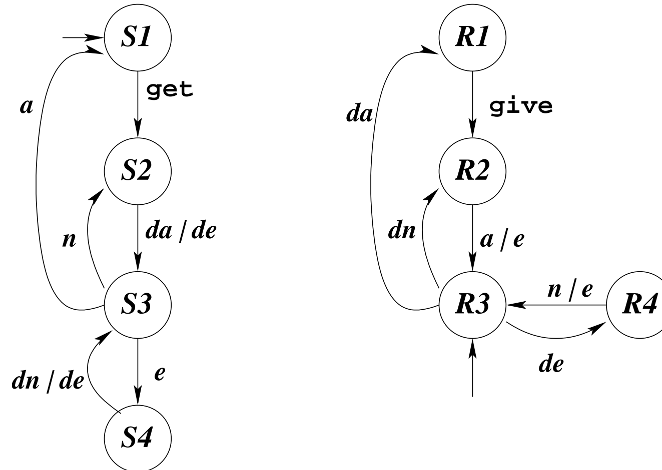
on kuvattu eräs virheellisesti toimiva protokolla, joka on ollut aikanaan todellisessa tuotantokäytössä. Tämän protokollan virhetilanteet esiintyvät niin harvoin, ettei sen puutteellisuutta havaittu ohjelmistoa testattaessa, vaikkakin virheitä sittemmin ilmeni tuotantokäytössä. Tämä virheellinen protokolla on mielenkiintoinen ja opettavainen esimerkki analysoitavaksi.

Virheellinen esimerkkiprotokollamme, lyhyesti FE-protokolla, on symmetrinen yhteystason protokolla. Kaksi partneria S ja R vaihtavat siinä vuorotellen sanomia virhealttiin vuorosuuntaisen kanavan välityksellä. Partnerit lisäävät kuhunkin lähettämäänsä sanomaan kuittausbitin a ("ACK", positiivinen kuittaus) kertoakseen edellisen sanoman saapuneen virheettömänä. Kuittausbitiksi asetetaan n ("NAK", negatiivinen kuittaus), jos edellinen saapuva sanoma oli vääristynyt kanavassa.

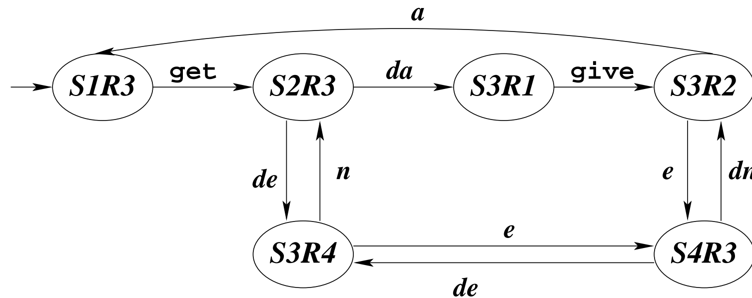
Kummankin partnerin lähetyslogiikka on seuraava: Jos edellinen saapuva sanoma sisälsi negatiivisen kuittauksen tai se osoittautui vääristyneeksi (jolloin vastaliikenteen NAK on ehkä tuhoutunut), käsillä oleva sanoma lähetetään uudelleen. Muutoin lähetetään seuraava sanoma. Molemmissa tapauksissa saapuvan liikenteen edellisen sanoman asianmukainen kuittausbitti, ACK tai NAK, lisätään lähtevään sanomaan.

Protokollan vastaanottologiikkaa ei anneta eksplisiittisesti Lynchin artikkelissa, jossa lähinnä vain osoitetaan mahdottomaksi suunnitella toimintavarmaa vastaanottologiikkaa edellä esitetylle lähetysmekanismille. Eräs mahdollinen vastaanottologiikka, jonka otamme käyttöön FE-protokollassamme, on seuraava: Kun sanoma saapuu virheettömänä ja sisältää ACK:in se toimitetaan kohdeprosessille. Jos sanoma sisältää NAK:in, sanoma hylätään (saapuvan sanoman arvellaan olevan uudelleenlähetys).

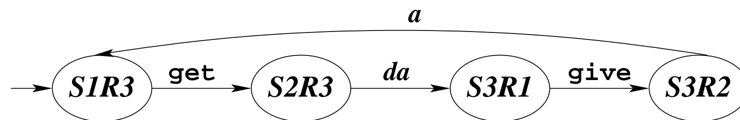
Mallinnamme aluksi protokollan yksinkertaistetussa muodossa. Jos protokolla toimii virheellisesti siinä muodossa, se toimii virheellisesti täydellisenäkin. Jos taas yksinkertaistettu versio toimii oikein, on tarpeen tutkia vielä täydellinen versiokin. Yksinkertaistetussa versiossa kommunikointi on synkronista, erillisiä kanavia ei käytetä. Datan välitys tapahtuu pelkästään S:ltä R:lle. Kun datasanomaan d liitetään vastaliikenteen ACK tai NAK, saadaan datasanoma da tai dn. R lähettää vain kuittauksen a tai n. Sanomanimet de ja e tarkoittavat vääristynyttä data- ja kuittaussanomaa. Siirtymäverkot S ja R on annettu alla:



Yhteistilaverkko on seuraavassa kuvassa:



Siitä nähdään protokollan perussykli



joka kuvaa sanomanvälitystä linjan toimiessa virheettömästi. Yksittäisestä linjavirheestä protokolla toipuu pienen sivureitin kautta:

$$S2R3 \xrightarrow{de} S3R4 \xrightarrow{n} S2R3$$

tai

$$S3R2 \xrightarrow{e} S4R3 \xrightarrow{dn} S3R2$$

Kahden peräkkäisen sanoman vääristyminen linjalla voi aiheuttaa virhetoiminnon FE-protokollassa. Tämän näemme seuraavassa mahdollisessa yhteistilaverkon reitissä:

$$S1R3 \xrightarrow{\mathsf{get}} S2R3 \xrightarrow{de} S3R4 \xrightarrow{e} S4R3 \xrightarrow{dn} S3R2 \xrightarrow{a} S1R3 \xrightarrow{\mathsf{get}} S2R3 \xrightarrow{da} S3R1 \xrightarrow{\mathsf{give}} S3R2$$

Tässä toimintareitissä otetaan kaksi sanomaa lähetettäviksi, mutta vain jälkimmäinen toimitetaan perille. Protokolla voi siis hukata sanoman. Seuraava yhteistilareitti paljastaa puolestaan tilanteen, jossa protokolla toimittaa sanoman perille kahteen kertaan:

$$S1R3 \xrightarrow{\mathsf{get}} S2R3 \xrightarrow{da} S3R1 \xrightarrow{\mathsf{give}} S3R2 \xrightarrow{e} S4R3 \xrightarrow{de} S3R4 \xrightarrow{n} S2R3 \xrightarrow{da} S3R1 \xrightarrow{\mathsf{give}} S3R2.$$

Edellä suoritettu analysointi perustui yhteistilaverkon yksityiskohtaiseen analyysiin. Virheellinen toiminta saadaan kuitenkin selville myös automaattisesti. Sitä varten kuvataan protokollan palvelu, joka tässä tilanteessa sattuu olemaan sama kuin AB-protokollassa. Ohjelmiston avulla voidaan osoittaa, että

$$\texttt{hide}\ da, de, a, n, e\ \texttt{in}\ \mathrm{FE-protokolla}\ \ \not\approx_{wbis}\ \ \mathrm{FE-palvelu}.$$

Tämä nähdään helposti myös manuaalisesti. Yritetään muodostaa heikkoa bisimulaatiota:

$$\mathcal{R} = \{(S1R3, P1), (S2R3, P2), (S3R2, P2), (S1R3, P2)\cdots\}.$$

Nyt tilasta $S1R3$ voi tapahtua `take`, mutta tilassa $P2$ vain `give` on mahdollista. Siten heikkoa bisimulaatiota ei voida muodostaa, joten prosessit eivät ole ekvivalentteja.

5.7 Johtopäätöksiä ja ongelmia

Siirtymäsysteemiformalismia on mahdollista kehittää edelleen. Jos menetelmää aiotaan soveltaa käytännössä, on kiinnitettävä huomiota seuraaviin kohtiin:

1. Miten kuvataan siirtymäsysteemi muuten kuin piirroksen avulla.
2. Miten kuvataan usean siirtymäsysteemin eli prosessin rinnakkainen toiminta. Tässä yhteydessä on otettava kantaa seuraaviin kohtiin:
   - miten tulkitaan prosessien rinnakkainen toiminta, ts. miten mallinnetaan aikaa rinnakkaisen systeemin taustalla;
   - onko prosessien kommunikointi synkronista vai asynkronista;
   - sallitaanko monisynkronointi.
3. Miten otetaan tietosisältö huomioon tapahtumissa.
4. Siirtymävaihtoehdot saattavat riippua sanomien sisällöstä.
5. Reaaliaikasovelluksissa ja muissakin tilanteissa tarvitaan aikarajoja. Miten nämä ilmaistaan formalismissa.
6. Siirtymillä voi käytännössä olla hyvin erilaiset todennäköisyydet (esim. tn(virhe) $<< 1$). Onko järkevää olettaa kuvauksissa siirtymävaihtoehdot aina yhtä todennäköisiksi.

7. Tavallisissa siirtymäsysteemeissä synkronointipisteet (sanomat) ovat etukäteen tiedossa ja niiden käyttö on kiinnitetty. Kuitenkin esimerkiksi meklarijärjestelmissä prosessi pyytää tietoa meklarilta, joka välittää prosessille palvelua antavan prosessin palveluportin. Tällaisessa tilanteessa palveluportti eli synkronointipiste ei ole etukäteen kiinnitetty, vaan se voi vaihdella varsin dynaamisesti suorituksen aikana. Miten tällaiset tilanteet käsitellään siirtymäsysteemeissä.

Edellä mainituista tavoitteista on kaikki toteutetu muodossa tai toisessa. Erityisesti kohtiin 1-4 on saatu melko laajasti hyväksytyt ratkaisut.

Siirtymäsysteemejä kuvataan nykyisin prosessialgebrojen avulla. Näitä ovat mm. Milnerin CCS (1980, 1989), Hoaren CSP (1985) ja Bergstran ja Klopin ACP (1984). Näissä on ratkaistu kohdat 1-2 karkeasti ottaen samaan tapaan:

- Siirtymäsysteemi kuvataan algebrallisella lausekkeella.
- Rekursion avulla saadaan ilmaistua silmukat. Rekursion avulla voidaan luoda myös äärettömiä siirtymäsysteemeitä ja käynnistää dynaamisesti suorituksen aikana uusia prosesseja.
- Prosessien kommunikointi on synkronista, ts. prosessi ei voi suorittaa synkronointitoimintoa a ennenkuin toinen (tai toiset) on valmis sen suorittamaan. Tällöin a suoritetaan yhtäaikaa. Epäsynkroninen kommunikointi saadaan aikaan ottamalla järjestelmään mukaan erillinen kanavaprosessi kuten AB-protokollassa.
- Useasta prosessista koostuvan systeemin toimintaa kuvataan myös siirtymäsysteemillä. Rinnakkaisuutta mallinnetaan lomituksen avulla:
  - Tapahtumat ovat atomaarisia.
  - Jos tapahtumat a ja b suoritetaan rinnan, niin ajatellaan, että joko a tapahtuu ennen b:tä tai b ennen a:ta. Siten meillä on suoritusvaihtoehdot ab ja ba, jotka esiintyvät myös koko systeemiä kuvaavassa verkossa.
  - Koska rinnakkaisuutta kuvataan lomituksen avulla, syntyy hyvin paljon vaihtoehtoja ja siten hyvin suuria siirtymäsysteemejä (abc, acb, cab, bac, bca, cba).
- CCS:ssä vain kaksi prosessia voi synkronoitua, CSP:ssä sallitaan monisynkronointi, samoin ACP:ssä.

Prosessialgebroissa on myös mukana tiedonvälitysehdot siirtymille. Sen sijaan perinteellisissä algebroissa ei ole otettu huomioon reaaliaikasovelluksia eikä todennäköisyyksiä. Näitä ominaisuuksia on myöhemmin lisätty siirtymäsysteemeihin ja tutkimus jatkuu edelleen.

Lotos on sukua CCS:lle ja CSP:lle. Siinä on

- synkroninen kommunikointi,

- monisynkronointi,
- lomitussemantiikka.

Lotoksessa prosesseja voidaan yhdistää rinnan yleisemmin kuin mainituissa prosessialgebroissa. Tästä on sekä hyötyä että haittaa. Esimerkiksi mainituissa prosessialgebroissa voidaan todistaa hyödyllisiä algebrallisia ominaisuuksia rinnakkaisoperaattoreille. Lotoksesta nämä lait puuttuvat (esim. assosiatiivisuus). Siirtymäsysteemien yhteydessä käsitelty rinnakkaisoperaattori on juuri Lotoksen operaattori.

Voimakkaimmin Lotos poikkeaa muista tietotyyppien määrittelyssä. Tietotyyppien kuvausmekanismi perustuu abstraktien tietotyyppien algebralliseen spesifiointiin. Algebrallinen spesifiointi kehittyi vuodesta 1970 alkaen läheisesti denotaatiosemantiikan kanssa. Algebrallisessa spesifioinnissa datatyyppien ja operaatioiden merkitys nousee niiden määrittelemästä algebrasta (termialgebra, initiaalisemantiikka). Se on erittäin voimakas tekniikka, jonka avulla voidaan määritellä esimerkiksi luonnolliset luvut tyhjästä. Toisaalta se vaikuttaa hankalalta ja monimutkaiselta aluksi. Siksi Lotoksen laajennuksessa E-Lotoksessa onkin tehty mahdolliseksi määritellä tietotyyppejä enemmän tavanomaisia ohjelmointikieliä muistuttavalla tavalla.

Lotos on suunniteltu käytäntöön. Se on laajempi kuin teoreettiset kielet CCS, CSP ja ACP. Tämän vuoksi se saattaa tuntua monimutkaiselta ja on tietty houkutus käyttää vain osaa siitä (ns. Basic Lotos). Tämä olisi kuitenkin virhe. Täysi Lotos oikein käytettynä ei ole sen monimutkaisempi kuin tavalliset ohjelmointikieletkään. Kuvauksista tulee täyden Lotoksen avulla tiiviimpiä kuin perus-Lotoksen avulla. Tällä kurssilla rajoitumme pääasiassa kuitenkin perus-Lotokseen.

Lotoksen laajennukseen (E-Lotos) on otettu mukaan myös aikakäsite. Aika-Lotosta ei kuitenkaan käsitellä tällä kurssilla. Erityistä tarvetta olisi myös toteuttaa kohta 7 eli dynaaminen synkronointi. Siihen on olemassa elegantti ratkaisu, joka pohjautuu CCS:ään. Tämä CCS:n laajennos tunnetaan nimellä π-kalkyyli. Sitä on yritetty sovittaa myös Lotokseen, mutta toistaiseksi yritykset eivät ole johtaneet yleisesti hyväksyttyihin ratkaisuihin.

Luku 6

Perus-Lotos

6.1 Johdanto

Selvennetään aluksi terminologiaa. Olkoon siirtymäsysteemissä siirtymä $S_i \stackrel{a}{\longrightarrow} S_j$. Symbolia a kutsutaan usein *toiminnaksi* (action). Kun otetaan huomioon, missä kohdassa toiminto esiintyy, puhutaan *tapahtumasta* (event). Jos esimerkiksi prosessi suorittaa tapahtumat *abcaade*, niin suorituksessa esiintyy toiminto a ja kolme erilaista tapahtumaa, joissa on mukana a. Tapahtumien erotteleminen toisistaan saattaa tuottaa ongelmia. Eräs mahdollisuus on käyttää alaindeksejä. Siten suorituksessa *abcaade* on tapahtumat a_1, a_2 ja a_3.

Lotoksessa toiminto jaetaan kahteen osaan, *porttinimeen* ja *tieto-osaan*. Esimerkiksi täydessä Lotoksessa a voisi olla $g!2?x$: Boolean, missä g on porttinimi, !2 ilmaisee, että g:ssä tapahtuu synkronoinnin yhteydessä luvun 2 lähetys toiselle prosessille (tai toisille prosesseille), ja $?x$: Boolean tarkoittaa, että synkronoinnin yhteydessä otetaan vastaan Boolen arvo muuttujaan x. Perus-Lotoksessa ei käytetä tieto-osaa lainkaan. Siten ei ole oleellista, puhutaanko toiminnosta vai portista.

Hieman yksinkertaistaen voidaan sanoa, että perus-Lotos on algebrallinen tapa kuvata äärellisiä tai äärettömiä siirtymäsysteemejä. Perus-Lotoksen esikuvina ovat olleet Milnerin CCS ja Hoaren CSP. Perus-Lotoksessa on

- kaksi valmista Lotos-lauseketta eli -prosessia,
- prosessien kutsumekanismi,
- kaksi valmiiksi määriteltyä toimintoa:
  - i eli *näkymätön* tai *sisäinen* toiminto,
  - δ eli *onnistunut lopetus*,
- 9 operaattoria.

Spesifioija voi käyttää näkymätöntä toimintoa i vapaasti. Sen sijaan onnistunut lopetus ei ole spesifioijan käytössä, vaan se tulee käyttöön tiettyjen operaattoreiden

semantiikan määrittelyssä. Operaattoreiden, sisäisen toiminnon ja spesifioijan määräämien toimintojen avulla konstruoidaan kaikki prosessit. Käymme seuraavassa läpi operaattorit yksityiskohtaisesti.

6.2 Valmiit prosessit

Lotoksessa on kaksi ennalta määriteltyä prosessia, **stop** ja **exit**. **Stop** on täysin pysähtynyt prosessi, joka ei tee mitään. **Exit** on onnistunut lopetus. Se ilmoittaa lopettamisesta suorittamalla toiminnon δ, jonka jälkeen se pysähtyy.

Prosessien toiminta määritellään täsmällisesti *siirtymäsäännöillä*, jotka antavat prosesseille tai operaattoreille *operationaalisen semantiikan*. Siirtymäsäännössä on kolme osaa:

- lauseke (prosessi) P,
- toiminto a, johon P voi välittömästi osallistua,
- uusi lauseke Q, joksi P muuttuu toiminnon a suorituksen jälkeen.

Tällöin käytetään merkintää $P \xrightarrow{a} Q$.

Lausekkeeseen **stop** ei liity siirtymäsääntöjä. Siten mikään siirtymä ei ole mahdollinen siitä.

Prosessin **exit** semantiikka määritellään säännöllä

$$\mathbf{exit} \xrightarrow{\delta} \mathbf{stop}.$$

6.3 Etutoiminto (action prefix)

Ensimmäinen operaattori on etutoiminto eli action prefix. Sen symbolina on puolipiste ';'. Jos a on toiminto ja P Lotos-prosessi, niin etutoiminnon avulla muodostetaan uusi prosessi $a; P$. Se suorittaa toiminnon a ja jatkaa sitten kuten P. Siirtymäsääntö on yksinkertainen:

$$a; P \xrightarrow{a} P.$$

Ei ole sallittua kirjoittaa $P; Q$ tai $\delta; P$, missä P ja Q ovat Lotos-prosesseja.

Voimme jo spesifioida yksinkertaisia siirtymäsysteemejä Lotoksella.

Esimerkki 1. Olkoon siirtymäsysteemi seuraava:

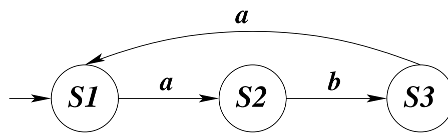
$$\longrightarrow S1 \xrightarrow{a} S2 \xrightarrow{b} S3 \xrightarrow{c} S4.$$

Sitä vastaa Lotos-prosessi

```
process P[a,b,c] := a; b; c; stop endproc
```

Siis prosessin esittelyssä on varattu sana **process**, jonka jälkeen annetaan prosessin nimi ja hakasuluissa käytettävät porttinimet. Sijoitusmerkin jälkeen kirjoitetaan prosessia vastaava Lotos-lauseke ja lopuksi varattu sana **endproc**.

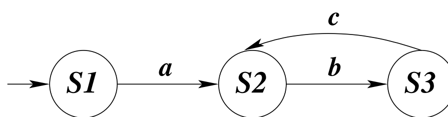
Esimerkki 2. Silmukka voidaan toteuttaa rekursiivisesti. Olkoon siirtymäsysteemi seuraavanlainen:



Vastaava prosessi on

```
process Q[a,b] := a; b; a; Q[a,b] endproc
```

Esimerkki 3. Siirtymäsysteemien suoraviivainen koodaus Lotos-kielelle on hieman kömpelöä kuten seuraava esimerkki osoittaa.




```
  process R[a,b,c] := a; S[b,c]
where
  process S[b,c] := b;c; S[b,c] endproc
  endproc
```

Yleensä ei kannatakaan kirjoittaa ensin siirtymäsysteemiä ja sen jälkeen sitä vastaavaa Lotos-prosessia, vaan parempi on suunnitella prosessi heti Lotokseen sopivaksi. Tähän Lotos tarjoaa hyviä mahdollisuuksia, joihin tutustutaan matkan varrella.

6.4 Kätkentä (hiding)

Kätkentä muuttaa toimintoja suorituksen aikana näkymättömiksi. Operaattorin syntaksi on seuraava:

$$\text{hide } a_1, a_2, \cdots, a_n \text{ in } P$$

Tällainen prosessi toimii kuten P, mutta tekee toiminnon i, kun P tekee jonkin toiminnoista a_i. Siirtymäsäännöt ovat helposti ymmärrettäviä:

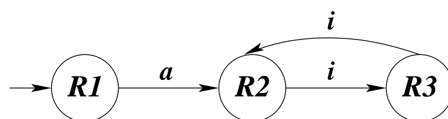
- Jos $P \xrightarrow{a} Q$ ja $a \in \{a_1, \cdots, a_n\}$, niin

$$\text{hide } a_1, \cdots, a_n \text{ in } P \xrightarrow{i} \text{hide } a_1, \cdots, a_n \text{ in } Q.$$

- Jos $P \xrightarrow{a} Q$ ja $a \notin \{a_1, \cdots, a_n\}$, niin

$$\text{hide } a_1, \cdots, a_n \text{ in } P \xrightarrow{a} \text{hide } a_1, \cdots, a_n \text{ in } Q.$$

Esimerkki. Jos R on kuten edellisen kohdan esimerkissä 3, niin prosessia hide b, c in R vastaava siirtymäsysteemi on



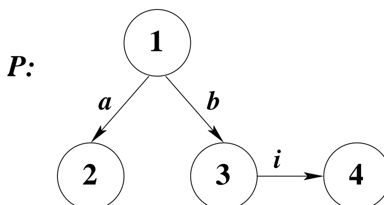
6.5 Valinta (choice)

Valintaoperaattorin symbolina on '[]', esimerkiksi P [] Q. Valintaoperaattorin avulla kokoonpantu prosessi on alkutilassa epädeterministinen. Ensimmäinen tapahtuma voi olla joko P:n tai Q:n. Sen jälkeen jatketaan P:n tai Q:n mukaisesti riippuen ensimmäisestä tapahtumasta. Siirtymäsäännöillä sama asia on ilmaistaan seuraavasti:

- Jos $P \xrightarrow{a} P'$, niin $P\ [\,]\ Q \xrightarrow{a} P'$.
- Jos $Q \xrightarrow{a} Q'$, niin $P\ [\,]\ Q \xrightarrow{a} Q'$.

Nyt voimme jo kuvailla minkälaisia siirtymäsysteejä tahansa.

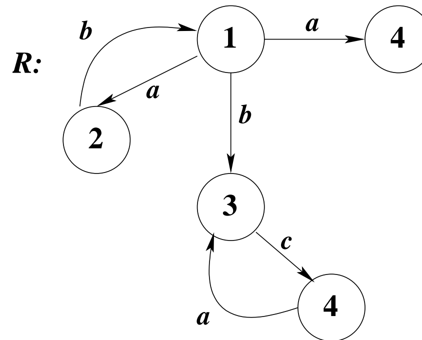
Esimerkki 1. Olkoon tilasiirtymäsysteemi seuraava:



Sitä vastaa prosessi

```
process P[a,b] := (a; stop) [] (b; i; stop) endproc
```

Esimerkki 2. Siirtymäsysteeminä on nyt



Vastaava Lotos-prosessi on

```
process R[a,b,c] := (a; b; R[a,b,c]) []
                    (b; S[a,c])       []
                    (a; stop)
where

    process S[a,c] := c; a; S[a,c] endproc
endproc
```

Esitämme vielä, miten lauseke muuttuu, kun suoritetaan tapahtumat b, c, a, c:

```
(a; b; R[a,b,c]) [] (b; S[a,c]) [] (a; stop) --b-->
 c; a; S[a,c]   --c-->
 a; S[a,c]      --a-->
 c; a; S[a,c]   --c-->
 a; S[a,c]
```

6.6 Rinnakkaisoperaattori

Tähän mennessä määriteltyjen operaattoreiden avulla voidaan konstruoida vain sarjallisesti toimivia prosesseja. Rinnakkaisoperaattorin (parallel composition) avulla voidaan prosessit panna toimimaan rinnakkain. Jos P ja Q ovat prosesseja, niin merkintä

$$P \ \ |[a_1, \cdots, a_n]| \ \ Q$$

tarkoittaa seuraavaa:

- P ja Q etenevät rinnakkain.
- Tapahtumat $a_1, \cdots, a_n$ ja δ ovat mahdollisia vain, jos sekä P että Q ovat niihin valmiita ja suorittavat ne yhtäaikaa ($a_1, \cdots, a_n$ ovat synkronointitoimintoja tai -portteja).
- P ja Q suorittavat muut tapahtumat toisistaan riippumatta.

Semantiikka saadaan seuraavista siirtymäsäännöissä. Niissä oletetaan, että $a_i \neq i, \delta$, $i = 1, \cdots, n$.

- Jos $P \xrightarrow{a} P'$ ja $a \notin \{\delta, a_1, \cdots, a_n\}$, niin

$$P \ |[a_1, \cdots, a_n]| \ Q \xrightarrow{a} P' \ |[a_1, \cdots, a_n]| \ Q.$$

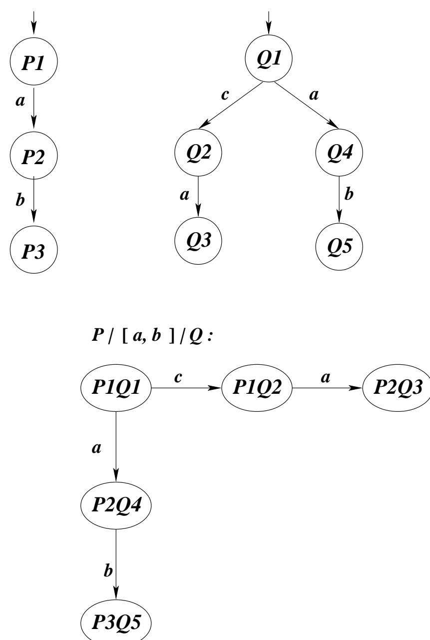
- Jos $Q \xrightarrow{a} Q'$ ja $a \notin \{\delta, a_1, \cdots, a_n\}$, niin

$$P \ |[a_1, \cdots, a_n]| \ Q \xrightarrow{a} P \ |[a_1, \cdots, a_n]| \ Q'.$$

- Jos $P \xrightarrow{a} P'$, $Q \xrightarrow{a} Q'$ ja $a \in \{\delta, a_1, \cdots, a_n\}$, niin

$$P \ |[a_1, \cdots, a_n]| \ Q \xrightarrow{a} P' \ |[a_1, \cdots, a_n]| \ Q'.$$

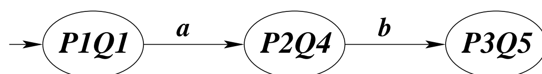
Siirtymäsääntöjen nojalla myös $P \ |[a_1, \cdots, a_n]| \ Q$ voidaan tulkita siirtymäsysteemiksi, ns. P:n ja Q:n *yhteistilaverkoksi* (global state graph, reachability graph). Seuraava esimerkki havainnollistaa, mistä on kysymys:



Jos $[a_1, \cdots, a_n]$ käsittää kaikki P:n ja Q:n tapahtumat ($\neq i$), niin voidaan käyttää merkintää

$$P \ || \ Q.$$

Jos P ja Q ovat kuten edellisessä esimerkissä, niin $P \ || \ Q$ käsittää vain siirtymät:

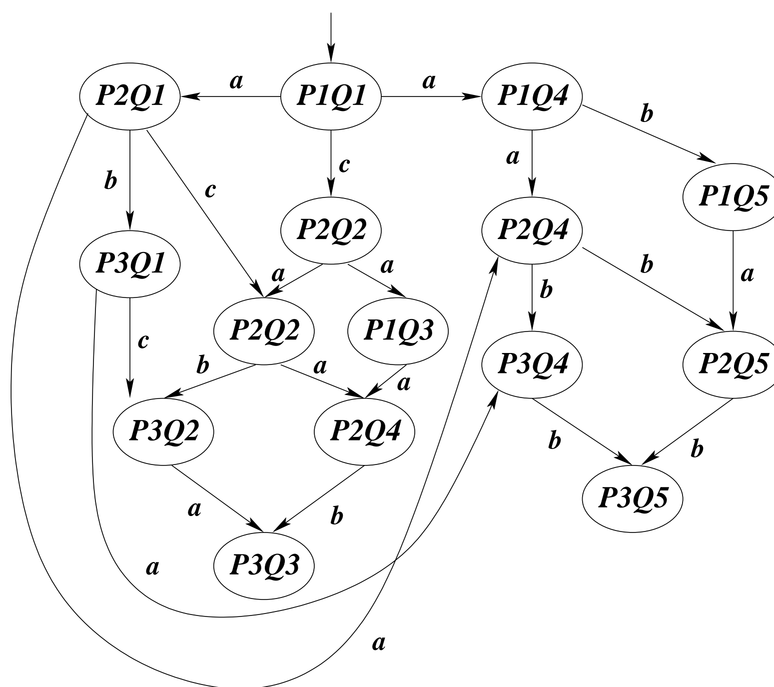


Q ei voi nyt suorittaa c:tä, koska c kuuluu synkronointijoukkoon, mutta P ei sisällä sitä.

Jos taas synkronointijoukko on tyhjä, niin voidaan käyttää merkintää

$$P \ ||| \ Q.$$

Tätä kutsutaan *täydelliseksi lomitukseksi*, koska P ja Q voivat suorittaa kaikki tapahtumansa toisistaan riippumatta. Seurauksena yhteistilaverkko kasvaa usein ekspontiaalisesti. Edellisten prosessien tapauksessa täydellinen lomitus tuottaa seuraavan verkon:



Edellisistä esimerkeistä nähdään, miten rinnakkaisuutta mallinnetaan. Ensinnäkin prosessit etenevät toisistaan riippumatta *lomittain* (lomitussemantiikka). Synkronointitapahtumassa kumpikin prosessi suorittaa tapahtuman yhtäaikaa. Toisen on odotettava, että toinen on valmis tapahtumaan. Synkronointi ja tietojen välitys sen yhteydessä perustuu siis *tiukkaan kytkentään* (tight coupling). Se sopii perustaksi

joihinkin sovelluksiin, esimerkiksi sellaisiin, joissa prosessit ovat samassa koneessa tai prosessin todella todella täytyy odottaa partnerinsa mukaantuloa.

Usein kuitenkin hajautetuissa sovelluksissa käytetään asynkronista tiedonvälitystä eli *löyhää kytkentää* (loose coupling). Tällaisissa tilanteissa on mallinnettava kanava omana prosessinaan. Prosessi luovuttaa sanoman kanavalle tiukan kytkennän pohjalta ja vastaanottaja saa sanoman kanavalta samalla tavalla. Kuitenkin lähettäjän ja vastaanottajan välillä toimii löyhä kytkentä, koska sanoma voi jäädä kanavaan joksikin aikaa lähettäjän jatkaessa toimintaansa.

On paljon keskulteltu myös lomituksen sopivuudesta rinnakkaisuuden mallintamiseen. On esitetty useita muita tapoja, joissa rinnakkaisuutta kuvataan yksityiskohtaisemmin. Tällöin on kysymys ns. *todellisen rinnakkaisuuden* malleista (true concurrency). Esimerkiksi Lotokselle on annettu todellisen rinnakkaisuuden mukainen semantiikka, mutta se on melko monimutkainen. Yleinen mielipide lienee, että lomitussemantiikka on useimpiin tilanteisiin riittävä. Se on myös algoritmien kannalta yksinkertainen.

6.7 Peräkkäisyys

Peräkkäisoperaattorin (sequential composition) merkintä on '>>'. Prosessi $P >> Q$ toimii kuten P, kunnes P lopettaa onnistuneesti, ja jatkaa sitten kuten Q. P:n lopetus ilmenee näkymättömänä tapahtumana. Siirtymäsäännöt ovat taas yksinkertaiset:

- Jos $P \xrightarrow{a} P'$ ja $a \neq \delta$, niin
$$P \;>>\; Q \xrightarrow{a} P' \;>>\; Q.$$
- Jos $P \xrightarrow{\delta} P'$, niin
$$P \;>>\; Q \xrightarrow{i} Q.$$

Esimerkki.

$$\begin{array}{r} a;\mathbf{exit} \;>>\; b;\mathbf{exit} \;\xrightarrow{a} \\ \mathbf{exit} \;>>\; b;\mathbf{exit} \;\xrightarrow{i} \\ b;\mathbf{exit} \;\xrightarrow{b} \\ \mathbf{exit} \;\xrightarrow{\delta} \\ \mathbf{stop} \quad\; \end{array}$$

6.8 Keskeytys

Keskeytyksen merkintä on P [> Q. Sen tulkinta on, että systeemi toimii aluksi kuten P. Milloin tahansa ennenkuin P on lopettanut onnistuneesti, Q voi aloittaa,

jolloin P:n toiminta loppuu. Q ei voi enää aloittaa, jos P on lopettanut onnistuneesti. Siirtymäsäännöt ovat seuraavat:

- Jos $P \xrightarrow{a} P'$ ja $a \neq \delta$, niin
$$P \,[>\, Q \xrightarrow{a} P' \,[>\, Q.$$
- Jos $P \xrightarrow{\delta} P'$, niin
$$P \,[>\, Q \xrightarrow{\delta} P'.$$
- Jos $Q \xrightarrow{a} Q'$, niin
$$P \,[>\, Q \xrightarrow{a} Q'.$$

Esimerkki.

$$\begin{array}{ccc}
a;\ b;\ \mathbf{exit}\ [>\ r;\ \mathbf{stop} & \xrightarrow{r} & \mathbf{stop} \\
\downarrow a & & \\
b;\ \mathbf{exit}\ [>\ r;\ \mathbf{stop} & \xrightarrow{r} & \mathbf{stop} \\
\downarrow b & & \\
\mathbf{exit}\ [>\ r;\ \mathbf{stop} & \xrightarrow{r} & \mathbf{stop} \\
\downarrow \delta & & \\
\mathbf{stop} & &
\end{array}$$

6.9 Operaattoreiden sitovuus

Voimakkaimmin sitoo ';'. Sen jälkeen tulevat järjestyksessä valinta, rinnakkaisoperaattorit, keskeytys, peräkkäisyys ja kätkentä. Esimerkiksi lauseke

$$\mathbf{hide}\ a\ \mathbf{in}\ a; P\ [\,]\ Q\ >>\ R \,||\, S\ [>\ T$$

on sama kuin

$$\mathbf{hide}\ a\ \mathbf{in}\ (((a;P)[\,]Q) >> ((R \,||\, S)[>\ T)).$$

Samantasoiset operaattorit sitovat oikealta. Siten esimerkiksi

$$P \,|[a]|\, Q \,|[b]|\, R \;=\; P \,|[a]|\, (Q \,|[b]|\, R).$$

6.10 Prosessin kutsu

Prosessin kutsu on periaatteessa samanlainen kuin proseduurin kutsu tavallisissa ohjelmointikielissä, mutta Lotoksessa tämäkin on määritelty formaalisti.

Tarvitsemme apuoperaattorin, *uudelleen nimeämisen*. Sille käytetään merkintää

$$[b_1/a_1, b_2/a_2, \cdots, b_n/a_n].$$

Se tarkoittaa, että prosessissa portit $a_1, \cdots, a_n$ korvataan porteilla $b_1, \cdots, b_n$. Uudelleen nimentä ei ole Lotos-kieleen kuuluva operaattori, vaan se on määrittelytekninen apuväline, jota tarvitaan prosessin kutsun määrittelyssä. Operaattorin semantiikka määritellään formaalisti seuraavilla siirtymäsäännöillä.

- Jos $P \xrightarrow{a} P'$ ja $a = a_i \in \{a_1, \cdots, a_n\}$, niin
$$P[b_1/a_1, \cdots, b_n/a_n] \xrightarrow{b_i} P'[b_1/a_1, \cdots, b_n/a_n].$$
- Jos $P \xrightarrow{a} P'$ ja $a \notin \{a_1, \cdots, a_n\}$, niin
$$P[b_1/a_1, \cdots, b_n/a_n] \xrightarrow{a} P'[b_1/a_1, \cdots, b_n/a_n].$$

Uudelleen nimeämisoperaattorin avulla voidaan kirjoittaa prosessin kutsun määrittelevät siirtymäsäännöt. Oletetaan, että prosessi P on esitelty kaavan

$$\textbf{process } P[a_1, \cdots, a_n] := KL \textbf{ endproc}$$

avulla, missä KL on jokin Lotoksen käyttäytymislauseke (eli yksinkertaisesti prosessi). Jos nyt on olemassa siirtymä

$$KL[b_1/a_1, \cdots, b_n/a_n] \xrightarrow{a} Q$$

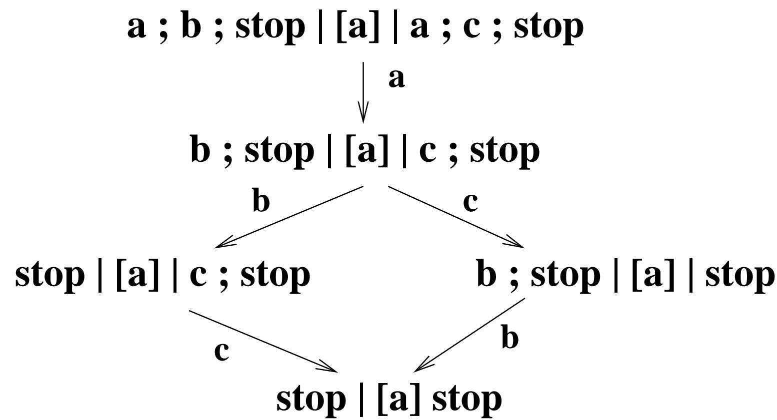
niin silloin on olemassa myös siirtymä

$$P[b_1, \cdots, b_n] \xrightarrow{a} Q.$$

Tämä sääntö määrittelee prosessin kutsun. Useimmissa tapauksissa prosessin kutsu käyttäytyy luonnollisella tavalla. Joskus on kuitenkin oltava huolellinen. Nimittäin edellä mainittujen sääntöjen perusteella uudelleen nimentä tehdään dynaamisesti tapahtuman suorituksen yhteydessä eikä staattisesti prosessin aloituksessa. Tarkastellaan esimerkiksi seuraavaa prosessia:

```
process P[a,b,c] := a; b; stop |[a]| a;c; stop endproc.
```

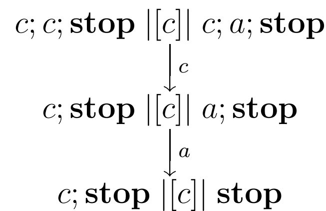
Se on siirtymäsysteeminä muotoa



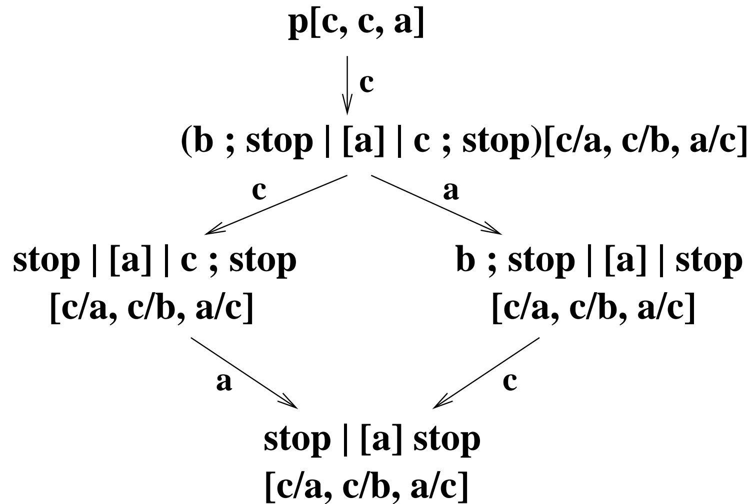
Tarkastellaan kutsua $P[c, c, a]$. Jos uudelleen nimentä suoritettaisiin staattisesti, P muuttuisi lausekkeeksi

```
c; c; stop |[c]| c; a; stop
```

jota vastaisi siirtymäsysteemi $LTS2$:



Lotoksen dynaaminen uudelleen nimentä tarkoittaa, että nimetään siirtymäsysteemissä $LTS1$ tapahtumat uudestaan, mutta kaaria ei muuteta. Seuraavassa kaaviossa näkyy, miten uudelleen nimentä suoritetaan askel kerrallaan:



6.11 Exit ja noexit

Prosessin parametrilistan perään kirjoitetaan tavallisesti määre `exit` tai `noexit`. `Exit` ilmaisee, että prosessiin voidaan liittää toinen prosessi peräkkäisoperaattorilla '>>'. Siis prosessissa on jossakin kohden haara, joka päättyy `exit`-prosessiin. `Noexit` taas ilmaisee, ettei `exit` haaraa ole eli prosessiin ei voida liittää toista prosessia peräkkäisoperaattorilla.

Esimerkki.

```
process P[a,b]: exit := (a; exit) [] b; stop endproc
process Q[c,d]: noexit := (c; c; d; stop) [] d; c; stop endproc
```

6.12 Esimerkkejä

6.12.1 Tuottaja-kuluttaja-spesifikaatio

Kirjoitamme pienen sovelluksen täydellisesti Peruslotoksella. Järjestelmässä on tuottajaprosessi, joka lähettää kuluttajaprosessille kaksi sanomaa kanavan välityksellä. Kanava voi kadottaa toisen tai molemmat sanomat. Sanomien järjestys ei kuitenkaan kanavassa muutu. Kuluttuja ei joudu hämminkiin, vaikka sanomia katoaisikin kanavassa, vaan kaikki tilanteet on otettu huomioon.

```
specification Producer_Consumer[pc1, pc2, cc1, cc2]: exit
behavior
    (Producer[pc1, pc2] ||| Consumer[cc1, cc2])
    ||
```

```
    Channel[pc1, pc2, cc1, cc2]

where

  process Producer[pc1, pc2]: exit :=
    pc1; pc2; exit
  endproc

  process Consumer[cc1, cc2]: exit :=
    cc1;
      (
        cc2;
          exit
      []
        exit
      )
    []
      cc2;
        exit
    []
      exit
  endproc

  process Channel[pc1, pc2, cc1, cc2]: exit :=
    pc1;
    (
      pc2;
        cc1;
          exit
    []
      cc1;
        pc2;
          exit
    []
      i;
        pc2;
          exit
    )
    >>
    (
      cc2;
        exit
    []
      i;
```

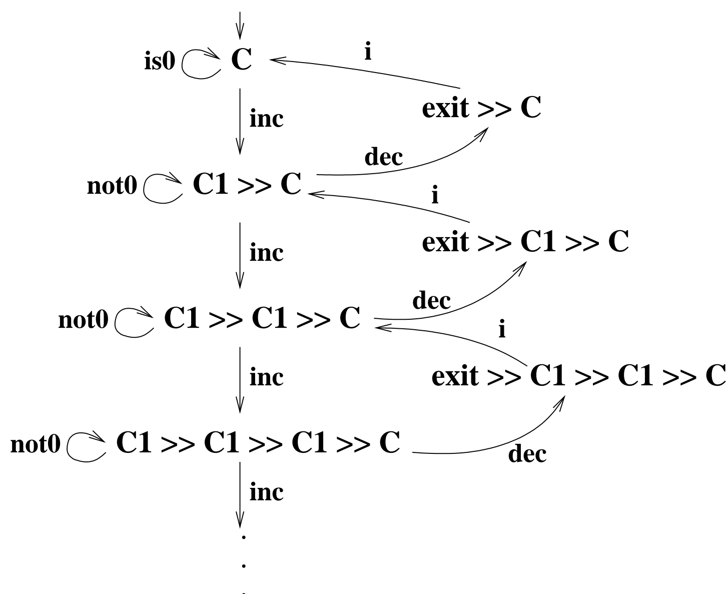
```
      exit
    )
  endproc
endspec
```

6.12.2 Laskuri

Seuraava esimerkki on prosessista, josta generoituu ääretön siirtymäsysteemi.

```
process Counter[is0, not0, inc, dec]: noexit :=
  (inc; C1[not0, inc, dec]
        >>
   Counter[is0, not0, inc, dec])
   []
   (is0; Counter[is0, not0, inc,dec])
where
  process C1[not0, inc, dec]: exit :=
    (dec; exit)
    []
    (inc; C1[not0, inc, dec] >> C1[not0, inc, dec])
    []
    (not0; C1[not0, inc, dec])
  endproc
endproc
```

Siirtymäsysteeminä prosessi on muotoa:



6.12.3 Asiakas-palvelin -systeemi

Kirjoitetaan luvussa 2.2 kuvattu asiakas-palvelin -systeemi Peruslotoksella. Samalla nähdään, miten prosesseissa olevia parametreja voidaan käyttää hyväksi. Oletetaan, että järjestelmässä on kolme asiakasta. Riittää spesifioida kaksi asiakasta, joista toinen kuvaa aloittaja-asiakasta ja toinen muita. Puskureita riittää kuvata vain yksi. Parametreja vaihtamalla saadaan sitten koko systeemi kuvattua näiden ja palvelimen avulla.

```
specification Client_Server_System[t1,t2,t3]: noexit
behavior

 hide cs1, cs2, cs3, sb1, sb2, sb3, bc1, bc2, bc3 in
   Server[cs1, cs2, cs3, sb1, sb2, sb3]
        |[cs1, cs2, cs3, sb1, sb2, sb3]|
   ( (Client1[bc1, cs1, t1, t2] |[bc1]| Buffer[sb1, bc1])
          |[t1, t2]|
     ( (Client[bc2, cs2, t2, t3] |[bc2]| Buffer[sb2, bc2])
          |[t3]|
       (Client[bc3, cs3, t3, t1] |[bc3]| Buffer[sb3, bc3])
     )
   )

where

process Client1[bc,cs,t1, t2]: noexit :=

   cs; (bc; t2; t1; Client1[bc, cs, t1, t2] []
        t2; bc; t1; Client1[bc, cs, t1, t2] )
endproc


process Client[bc, cs, t1, t2]: noexit :=
    t1; cs; (bc; t2; Client[bc, cs, t1, t2] []
             t2; bc; Client[bc, cs, t1,t2] )
endproc


process Server[cs1, cs2, cs3, sb1, sb2, sb3]: noexit :=

   cs1; sb1; Server[cs1, cs2, cs3, sb1, sb2, sb3]
     []
   cs2; sb2; Server[cs1, cs2, cs3, sb1, sb2, sb3]
```

```
     []
   cs3; sb3; Server[cs1, cs2, cs3, sb1, sb2, sb3]
endproc

process Buffer[sb, bc] : noexit :=

   sb; bc; Buffer[sb, bc]
endproc

endspec
```

6.13 Ohjelmistot

Yhteistilaverkkojen muodostaminen ja heikon bisimulaatioekvivalenssin laskeminen vaativat vähänkään realistisissa tapauksissa tietokoneen apua. Tällä kurssilla käytämme pääasiassa ranskalaista Caesar/Aldebaran-ohjelmistoa, mutta jonkin verran myös suomalaista ARA-ohjelmistoa.

6.13.1 ARA

ARA-ohjelmisto on VTT:ssä Antti Valmarin johdolla 1990-luvulla kehitetty ohjelmisto, joka generoi Lotos-kuvauksesta yhteistilaverkon, minimoi ja vertailee yhteistilaverkkoja sekä näyttää verkkoja graafisesti. ARA-ohjelmistolle voi antaa syötteeksi yhden Lotos-prosessin koko spesifikaation sijasta. Käytettävä ekvivalenssi on ns. CFFD-ekvivalenssi, bisimulaatioekvivalenssia ei ARA:ssa ole toteutettu. Täten kurssilla ohjelmistoa voi käyttää lähinnä yhteistilaverkon generointiin. ARA:n hyvä puoli on se siinä, että sen saa myös omalle PC:lle. Katso ohjeita www-sivulta `http://www.cs.tut.fi/ohj/VARG/VARG.html`.

ARA käsittelee pääasiassa perus-Lotosta. Siinä on tosin myös laajennoksia, jotka mahdollistavat yksinkertaisten tyyppien (kokonaisluvut, totuusarvot) käyttämisen datasanomissa, mutta varsinaisesti se ei tue täyttä Lotosta.

ARA:n käyttöliittymä on rivipohjainen, unix-tyyppinen. Tärkeimmät komennot ovat seuraavassa. Komennoista on huomattava, että ne eroavat sen suhteen, käytetäänkö PC:n Windows:ta vai Solariksen Unixia. Unixissa komentojen perään liitetään pääte -u tai -ux. Seuraavassa luettelossa komennot ovat Windows-muodossa:

- `arassg kuvaus.lot kuvaus.lts` muodostaa tiedostossa `kuvaus.lot` olevasta Lotos-kuvauksesta yhteistilaverkon, joka sijoitetaan tiedostoon `kuvaus.lts`.
- `araillu kuvaus.lts` näyttää muodostuneen verkon graafisesti näytöllä.
- `aranor kuvaus.lts kuvaus.nlts` minimoi verkon CFFD-ekvivalenssin suhteen. Minimoitu verkko tulostetaan tiedostoon `kuvaus.nlts`.

- `aracomp kuvaus1.lts kuvaus2.lts` vertailee, ovatko kaksi kaksi verkkoa ekvivalentteja CFFD:n kannalta.

6.13.2 Aldebaran

Caesar/Aldebaran on Ranskasta lähtöisin oleva täyden Lotoksen ohjelmisto, jota on kehitetty laajasti myös Kanadassa ja Espanjassa. Sen käyttöliittymä on graafinen ja helppokäyttöinen. Hieman vaikeampaa on laatia oikeanmuotoiset tiedostot täyden Lotoksen datan käsittelyyn, mutta tällä kurssilla tulemme toimeen perustyypeillä, joita vastaavat tiedostot löytyvät ohjelmiston yhteydessä olevista esimerkeistä.

Ohjelmisto on laaja ja se tuntee monia ekvivalensseja ja esitysmuotoja. Tällä kurssilla keskitymme Aldebaranin ohjelmiin ja esitysmuotoihin. Ohjelmisto toimii Linuxissa koneessa *melkinkari*. Ohessa on tarvittavat määritykset tiedostoon `.profile`:

```
CADP=/opt/cadp ; export CADP
export PATH=$PATH:/opt/cadp/com
export PATH=$PATH:/opt/cadp/bin.iX86
```

Graafinen käyttöliittymä käynnistyy komennolla `xeuca`. Tyypillisesti verifiointi etenee siten, että ensin generoidaan Lotos-kuvauksista siirtymäsysteemit. Tässä kannattaa valita esitysmuodoksi Aldebaran. Siirtymäsysteemi on `aut`-tarkenteisessa tiedossa. Tästä saa tiedot valitsemalla ikkunasta kohdan `information` (klikkaamalla ensin kyseistä tiedostoa). Siirtymäverkkoa voi minimoida (`reduce`) tai sitä voi verrata toiseen. Valitse jälleen sopiva versio useasta mahdollisesta (tyypillisesti Aldebaran ja Observational Equivalence = heikko bisimulaatioekvivalenssi).

Aldebaran generoi yhteistilaverkon huomattavasti nopeammin kuin ARA. Työläin kohta on minimointi tai vertailu, joka saattaa kestää minuutteja, vaikka yhteistilaverkko on generoitu sekunneissa.

Ongelmana verifioinnissa on, että kirjoitusvirheet Lotos-koodissa eivät välttämättä näy syntaksin tarkastuksen yhteydessä. Esimerkiksi joku portti saattaa puuttua rinnakkaisoperaattorin porttilistasta. Siten ensimmäisiä ilmoituksia lukkiutumista ei aina kannata uskoa, varsinkin jos yhteistilaverkko on yllättävän suuri tai pieni.

6.14 AB-protokolla

Näytetään tyypillinen tilanne verifioinnissa AB-protokollan yhteydessä. Tavoitteena on verifioida jälkimmäinen AB-protokollan versio, siis se, jossa käytetään kanavaa ja sanomia `get` ja `give`. Koodataan kohdassa 4.2 esitetty protokolla kanavineen perus-Lotoksella:

```
specification AB[get, give]:
  noexit

behavior

hide d0, d1, dd0, dd1, a0, a1, aa0, aa1, st, rt, t in

 ( (Sender[get, d0, d1, aa0, aa1, st,rt,t]
    |[t,st,rt]|
  Timer[st,t,rt])

    |||

  Receiver[give, dd0, dd1, a0, a1]
 )
    |[d0,d1,dd0,dd1,a0,a1,aa0,aa1]|

  Channel[d0,dd0,d1,dd1,a0,aa0,a1,aa1]

where

process Sender[get,d0,d1,aa0,aa1,st,rt,t]: noexit :=

  get; Transmit[d0, aa0, aa1, st,rt,t] >>
  get; Transmit[d1,aa1, aa0, st,rt,t] >>
  Sender[get,d0,d1,aa0,aa1,st,rt,t]

  where

  process Transmit[d0, aa0, aa1, st,rt,t]: exit :=

    aa0; Transmit[d0, aa0, aa1, st, rt, t]
         []
    d0; st; (t; Transmit[d0, aa0, aa1, st,rt,t]
             []
            aa0; rt; exit
           )
  endproc
endproc


process Timer[st,t,rt] :noexit :=

   st;(t;Timer[st,t,rt] [] rt; Timer[st,t,rt])
```

```
endproc


process Receiver[give, dd0, dd1, a0, a1]: noexit :=

   dd0; give; Ack0[give, dd0, dd1, a0, a1]
      []
   dd1; Ack1[give, dd0, dd1, a0, a1]

   where


         process Ack1[give, dd0, dd1, a0, a1]: noexit :=

            dd1; Ack1[give, dd0, dd1, a0, a1]
               []
            a1; Receiver[give, dd0, dd1, a0, a1]
         endproc



   process Ack0[give, dd0, dd1, a0, a1]: noexit :=

      dd0; Ack0[give, dd0, dd1, a0, a1]
        []
      a0; (dd0; Ack0[give, dd0, dd1, a0, a1]
            []
           dd1; give; Ack1[give, dd0, dd1, a0, a1]
          )


   endproc

endproc



process Channel[d0,dd0, d1, dd1, a0, aa0, a1, aa1] : noexit :=

   d0; (i; Channel[d0,dd0, d1, dd1, a0, aa0, a1, aa1]
         []
        i; dd0; Channel[d0,dd0, d1, dd1, a0, aa0, a1, aa1]
       )
   []
```

```
    d1; (i; Channel[d0,dd0, d1, dd1, a0, aa0, a1, aa1]
           []
          i; dd1; Channel[d0,dd0, d1, dd1, a0, aa0, a1, aa1]
         )
    []
    a0; (i; Channel[d0,dd0, d1, dd1, a0, aa0, a1, aa1]
           []
          i; aa0; Channel[d0,dd0, d1, dd1, a0, aa0, a1, aa1]
         )
    []
    a1; (i; Channel[d0,dd0, d1, dd1, a0, aa0, a1, aa1]
           []
          i; aa1; Channel[d0,dd0, d1, dd1, a0, aa0, a1, aa1]
         )
endproc

endspec
```

Yhteistilaverkko muodostuu nyt 91 tilasta ja 177 siirtymästä, joista 156 on τ-siirtymiä. Verkko ei sisällä lukkiumia. Nyt verkkoa voidaan verrata AB-protokollan tuottamaan palveluun. Tehdään tämä siten, että minimoidaan yhteistilaverkko heikon bisimulaatioekvivalenssin suhteen. Tuloksena syntyy verkko, joka on täsmälleen sama kuin palvelu. Siten on osoitettu, että AB-protokolla toimii oikein.

Luku 7

Datan käsittely

Lotoksessa prosessit voivat lähettää toisilleen erityyppistä tietoa. Vastaanottavan prosessin siirtymä voi riippua saapuneen sanoman tiedon laadusta. Prosessit voivat käsitellä dataa Lotoksessa kuten normaaleissakin ohjelmointikielissä. Oleellinen ero Lotoksen ja tavanomaisten ohjelmontikielten välillä on se, että Lotoksessa tietotyypit määritellään algebrallisesti ja tietotyyppejä voi käyttää ainoastaan tässä formalislismissa esiteltyjen operaatioiden kautta. Tällä kurssilla ei omia tietotyyppejä määritellä, mutta tavallisimpia tietotyyppejä käytetään valmiina. Valmiit esittelyt saadaan esimerkiksi Caesar/Aldebaran-ohjelmiston esimerkeistä.

7.1 Datan lähetys ja vastaanotto

Täydessä Lotoksessa toiminto koostuu porttinimestä ja yhdestä tai useammasta lähetys- tai vastaanottotapahtumasta. Esimerkiksi seuraavassa lausekkeessa prosessi lähettää portin g kautta kokonaisluvun ja ottaa samalla vastaan totuusarvon:

```
g !2 ?x:Boolean
```

Portti g on se, jota aikaisemmin on kutsuttu toiminnoksi. Täydessä Lotoksessa voi portti esiintyä yksinään kuten Peruslotoksessa, mutta useimmiten siihen todella liittyy yksi tai useampi lähetys- tai vastaanottotapahtuma.

Ensimmäinen kysymys, joka täytyy ratkaista lähetyksiä käytettäessä, on synkronointi: milloin kaksi prosessia voivat synkronoitua samassa portissa, jos kummassakin porttiin liittyy myös lähetystä ja vastaanottoa. Tarkastellaan ensin tilannetta, jossa kaksi prosessia on synkronoitumassa portissa g ja kummassakin on lisäksi yksi lähetys tai vastaanotto. Lotos-standardi määrittelee tällöin synkronointiehdot seuraavasti:

- Jos kaksi lähettävää tapahtumaa $g!E$ ja $g!D$ ovat synkronoitumassa, synkronointi voi tapahtua, jos E ja D ovat samaa tyyppiä ja value(E) =value(D). Kysymyksessä on *arvon sovitus* (value matching).

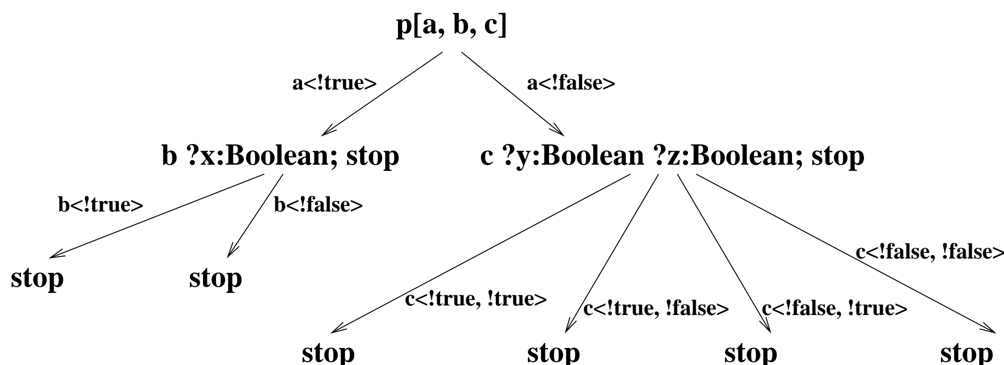
- Jos toinen tapahtumista on lähettävä, $g!E$, ja toinen vastaanottava, $g?x : T$, niin synkronointi tapahtuu, jos E:n arvo on tyyppiä T. Kysymyksessä on *arvon välitys* (value passing).
- Jos molemmat tapahtumista ovat vastaanottavia, $g?x : T$ ja $g?y : U$, niin synkronointi voi tapahtua, jos $T = U$. Synkronoinnin yhteydessä generoidaan arvo joukosta T, joka tulee sekä x:n että y arvoksi. Kysymyksessä on *arvon generointi* (value generation).

Yllä mainitut säännöt tulevat käyttöön rinnakkaisoperaattorin yhteydessä. Seuraavista esimerkeistä käy selville, miten siirtymäsysteemi esitetään, kun dataa on mukana.

Esimerkki 1. Olkoon P seuraava prosessi:

```
P[a,b,c] := a !true; b ?x:Boolean; stop
               []
            a !false; c ?y:Boolean ?z:Boolean; stop
endproc
```

Siitä generoituu seuraava siirtymäsysteemi.



Siirtymäsysteemin muodostuksessa on huomattava seuraavat seikat. Siirtymät ovat muotoa $g < e_1 e_2, \cdots, e_n >$, missä e_i on lähetystapahtuma. Jos alkuperäisessä prosessissa on pelkkä vastaanotto ilman, että samalla tapahtuu mitään synkronointia lähettävän prosessin kanssa, siirtymäsysteemin tilasta lähtee lähetyssiirtymä jokaista mahdollista vastaanotettavaa arvoa kohti. Tämä vastaa tilannetta, että jokin kyseisistä arvoista voi tulla prosessille todellisen suorituksen aikana. Mikäli tyypin arvoalue on ääretön (esim. kokonaisluvut), lähtee tilasta niinmuodoin ääretön määrä siirtymiä. Tämä on otettava huomioon, kun prosesseja käsitellään ohjelmistojen avulla.

Tarkastellaan nyt vuorostaan alussa mainittuja kolmea sääntöä synkronoinnin yhteydessä. Näitä eri tilanteita voidaan soveltaa seuraavissa tapauksissa.

Ensimmäisen vaihtoehdon mukaista (arvon sovitus) tapausta voidaan käyttää haarautumisessa. Esimerkiksi seuraava prosessi ottaa vastaa totuusarvon ja haarautuu sen mukaan, onko arvo tosi vai epätosi.

```
process P[g, g1,g2] :=
   g!true; Q[g1]
    []
   g!false; R[g2]
endproc
```

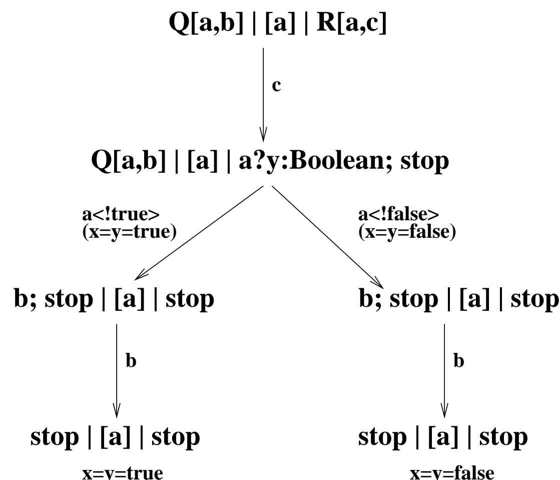
Tämä on lyhyempi tapa, kuin ottaa arvo vastaan ensin johonkin muuttujaan, $g?x : Bool$, ja käyttämällä sen jälkeen vahteja (guards; näemme myöhemmin, miten vahteja käytetään).

Toinen vaihtoehto on luonnollinen lähetys- ja vastaanottotilanne. Kolmas vaihtoehto on hieman erikoinen. Sitä voidaan käyttää arvon generoimiseen. Siirtymäverkossa on piirrettävä kuitenkin kaari jokaista arvoa kohti eli on otettava huomioon kaikki generointivaihtoehdot. Tämä vastaa tapausta, että prosessissa on vastaanotto ilman synkronointia.

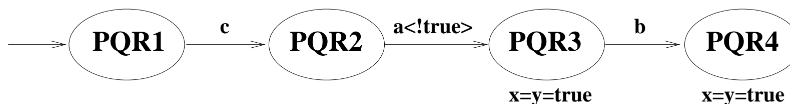
Esimerkki 2.

```
P[a] := a!true; P[a]
Q[a,b] := a?x:Boolean; b; stop
R[a,c] := c; a?y:Boolean; stop
S[a,b,c] := P[a] |[a]| (Q[a,b] |[a]| R[a,c])
```

Muodostetaan ensin Q:n ja R:n yhteistilaverkko synkronointitapahtuman ollessa a:



Huomattakoon yhteistilaverkossa haarautuminen tapahtuman a yhteydessä. Ensimmäinen haara vastaa tapausta, että generoitu arvo on true, toinen tapausta, että generoitu arvo on false. P tarjoaa arvoa true. Yhdistettäessä P:tä jo muodostettuun yhteistilaverkkoon valikoituu yhteistilaverkosta se haara, joka vastaa tätä arvoa:



7.2 Parametrit aloituksessa ja lopetuksessa

Lotoksessa prosessiin voidaan liittää porttilistan lisäksi parametrilista. Esimerkiksi seuraavassa prosessissa on portit a, b ja c sekä parametrina luonnollinen luku n ja totuusarvo *bit*. Prosessi lähettää totuusarvon ja luonnollisen luvun portista a. Se ottaa vastaan uuden totuusarvon ja uuden luvun porteista b ja c. Sen jälkeen prosessi alkaa alusta, mutta uusilla parametrien arvoilla.

```
process P[a,b,c](bit:Bool, n:Nat):noexit :=
   a !n !bit;
      b ?x:Bool;
         c ?y:Nat;
            P[a,b,c](x,y)
endproc
```

Kun prosessi lopettaa onnistuneesti, prosessi voi samalla lähettää arvoja jatkavalle prosessille. Lähetettävät arvot kirjoitetaan prosessiin **exit**,

$$\text{exit}(E_1, \cdots, E_n).$$

Yllä oleva käyttäytymislauseke tarjoaa tapahtumat

$$\delta!E_1 \cdots !E_n.$$

Tässä $E_1, \cdots, E_n$ ovat lausekkeita, joiden arvon prosessi lähettää eteenpäin lopettaessaan onnistuneesti.

Seuraavassa määritellään kuluttajaprosessi, joka ottaa vastaan yhden sanoman, jossa on mielivaltaisen luonnollisen luvun lisäksi parillinen järjestysluku ja lopettaa onnistuneesti lähettämällä samalla kyseisen järjestysluvun eteenpäin:

```
process Consumer[a]: exit(Nat) :=
   a ?Msg:Nat ?Seq:Nat [(Seq mod 2) = 0];
      exit(Seq)
endproc
```

Prosessissa on käytetty sanoman vastaanoton yhteydessä loogista ehtoa, jonka täytyy toteutua ennen kuin synkronointi voi tapahtua.

Lopussa lähetettävien arvojen tulee olla sopusoinnussa alun exit-määreiden kanssa. Jos prosesseja yhdistetään rinnakkaisoperaattorilla, niiden täytyy onnistuneesti lopettaessaan synkronoitua portissa δ. Toisin sanoen kaikkien prosessien tulee tarjota **exit**, joissa on sama määrä parametreja ja parametrien tulee sopia yhteen aikaisemmin esitettyjen synkronointisääntöjen mukaisesti.

Parametria, joka voi saada minkä tahansa tyypin T mukaisen arvon, merkitään `any T`. Esimerkiksi jos $P1$ ja $P2$ on yhdistetty rinnakkaisoperaattorilla ja

- $P1$ lopettaa `exit(any Boolean, 1)`,
- $P2$ lopettaa `exit(true, any Nat)`,

niin lopettaessaan

- $P1$ tarjoaa δ?dummy : Bool!1,
- $P2$ tarjoaa δ!true?dummy : Nat.

Nämä ovat sopusoinnussa ja ympäristöön systeemi tarjoaa tapahtuman

$$\delta\text{!true!1.}$$

Esimerkki 3. Laaditaan prosessi, joka ottaa vastaan järjestysnumerolla varustettuja sanomia ja laskee sekä parittomien että parillisten järsjestysnumeroiden lukumäärän. Lopettaessaan prosessi lähettää eteenpäin sanomien kokonaismäärän, joka voi olla nolla. Lopetus voi tulla milloin tahansa.

```
process Consumer[a](Odd_Num_Msg, Even_Num_Msg: Nat):exit(Nat,Nat):=
   a ?Msg: Nat ?n: Nat [(n mod 2) = 0];
      Consumer[a](Odd_Num_Msg, Even_Num_Mgs+1)
  []
   a ?Msg: Nat ?n: Nat [(n mod 2) <> 0];
      Consumer[a](Odd_Num_Msg+1, Even_Num_Msg)
  []
   exit(any Nat, Odd_Num_Msg + Even_Num_Msg)
endproc
```

□

Prosesseja peräkkäisoperaattorilla yhdistettäessä parametrien arvot voidaan ottaa huomioon **accept**-lauseella. Se on muotoa

$$P_1 \;\; >> \;\; \textbf{accept} \;\; x_1 : T_1, \cdots, x_n : T_n \;\; \textbf{in} \;\; P_2$$

Jos P_1 lopettaa onnistuneesti exitillä, seuraavaksi suoritetaan P_2. P_2:n muuttujat $x_1, \cdots, x_n$ saavat alkuarvoikseen exitin yhteydessä eteenpäin välitetyt arvot, joiden täytyy olla sopusoinnussa tyyppien T_i kanssa.

Laaditaan esimerkkinä tuottajaprosessi, joka generoi arvon ja lähettää sen järjestysnumeron kera tai lopettaa ilmoittaen samalla lähetettyjen sanomien lukumäärän.

```
process Producer[b](n: Nat): exit(Nat,Nat) :=
   (Generate_Ele >>
   accept Msg: Nat in
      b !Msg !n;
         Producer[b](n+1)
   )
  []
   exit(n, any Nat)

where

   process Generate_Ele: exit(Nat) :=
      exit(any Nat)
   endproc
endproc
```

7.3 Vahdit

Olemme jo käyttäneet ehtolausekkeita rajaamaan vastaanottotapahtumaa kuten esimerkiksi kuluttajan tapauksessa:

```
a ?Msg:Nat ?Seq:Nat [(Seq mod 2) = 0];
```

Ehdon täytyy olla tosi, ennen kuin synkronointi voi onnistua. Tällainen ehto voidaan kirjoittaa myös ensimmäiseksi ja silloin puhutaan *vahdista*. Esimerkkinä yksinkertaisesta vahteja käyttävästä rakenteesta olkoon seuraava:

```
     [x=2 or x=3] -> g !x !0; stop
   []
     [x < 2] -> g!x !1; stop
```

Kirjoitamme vielä kuluttajan vahtien avulla:

```
process Consumer[a] (Odd_Num_Msg, Even_Num_Msg: Nat):exit(Nat,Nat):=
   a ?Msg: Nat ?n: Nat;
   (
     [(n mod 2) = 0] -> Consumer[a](Odd_Num_Msg, Even_Num_Msg+1)
    []
     [(n mod 2) <> 0] -> Consumer[a](Odd_Num_Msg+1, Even_Num_Msg)
   )
   []
   exit(any Nat, Odd_Num_Msg + Even_Num_Msg)
endproc
```

7.4 Yleistetty valinta

Yleistetty valinta on muotoa

```
choice X:T [] B
```

joka on yhtäpitävä seuraavan lausekkeen kanssa:

$$[t_1/X]B\ [\,]\ [t_2/X]B\ [\,]\ \cdots\ [\,]\ [t_n/X]B.$$

Tässä $t_1, t_2, \cdots, t_n$ ovat kaikki tyypin T mukaiset arvot ja merkintä $[t_i/X]B$ tarkoittaa, että X:n esiintymät B:ssä on korvattu t_i:llä.

Äärettömät arvojoukot ovat periaatteessa sallittuja. Käytännössäkin niitä voi esiintyä kuten seuraavassa lausekkeessa:

```
choice x:Nat []
  [(x mod 2) = 0] -> g !x; stop
```

Usea muuttuja voi saada arvon yhdessä choice-rakenteessa:

$$\text{choice } X_1 : T_1, \cdots, X_n : T_n\ [\,] P(X_1, \cdots, X_n).$$

Myös porttien välinen valinta on mahdollinen:

$$\text{choice } a \text{ in } [a_1, \cdots, a_n]\ [\,]\ P[a](\cdots).$$

Yllä olevan lausekkeen perusteella valituksi tulee jokin kutsuista

$$P[a_1], \cdots, P[a_n].$$

Laajempana esimerkkinä olkoon järjestelyn spesifiointi Lotoksella. Nyt ei määritellä mitään järjestelyalgoritmia, vaan ainoastaan yleisellä tasolla lukujen järjestäminen suuruusjärjestykseen. Tarvitsemme tyypin `NatList`, joka edustaa luonnollisten lukujen listaa. Lisäksi käytämme operaattoreita `IsPermuted` ja `IsOrdered`. Edellistä käytetään muodossa

```
IsPermuted(SortedList, UnsortedList)
```

ja se palauttaa arvon tosi, jos jälkimmäinen lista on edellisen permutaatio. Jälkimmäinen operaatio testaa, onko argumenttina annettu lista järjestyksessä vai ei.

```
input ?UndortedList: NatList;
   choice SortedList: NatList []
       [IsPermuted(SortedList, UnsortedList) and
        IsOrdered(SortedList)]    ->
      output !SortedList
```

Lopuksi huomattakoon, että choice-rakennetta voidaan käyttää usealla tasolla:

```
g ?x: Nat;
   choice y: Nat[]
      [y > x] -> (choice w,z: Nat []
                     [w*z=y] -> g!z; exit
                     []
                     [prime(y)] -> i; g !y; exit
                 )
```

7.5 Sijoitus

Käyttäytymislausekkeessa muuttujille voidaan antaa arvoja LET-lauseella:

```
gate ?x: Nat;
   (let ext_val = 3*((x+3)*x-2)**(x*s),
        int_val = (x*x)+2 in
       gate !ext_val !int_val;
          stop
       []
       gate ?y: Nat !int_val;
          stop
       []
       gate !ext_val ?y:Nat;
          stop
   )
```

7.6 Par-lause

Par-lause on muotoa

```
par <gateDeclarations> <parallelOp>
```

Sitä käytetään liittämään rinnakkain yhteen useita prosesseja vaihtelemalla yhtä tai useampaa porttia. Esimerkiksi:

```
par g1 in [a1, a2, a3] parop,
    B[g1, h1]
```

missä B[g1, h1] on käyttäytymislauseke ja parop on rinnakkaisoperaattori kuten esimerkiksi || tai |||. Vastaava käyttäytymislauseke kirjoitettuna rinnakkaisoperaattorien avulla näyttäisi seuraavalta:

```
B[a1, h1] parop B[a2, h1] parop B[a3, h1]
```

7.7 Katsaus tietotyyppeihin

Lotoksessa on mahdollista määritellä omia tietotyyppejä. Lotokseen on valittu tietotyyppien algebrallinen kuvausmenetelmä, joka perustuu ACT ONE -kieleen. Tällainen menetelmä on formaali ja siinä vältetään kaikki tietotyyppien toteutukseen liittyvät yksityiskohdat. Tietotyypit ovat abstrakteja ja data näiden abstraktien tietotyyppien alkioita. Esimerkiksi taulukon määrittely C-kielessä ottaa jo jossain määrin kantaa taulukon toteutukseen ja tällaista halutaan välttää Lotoksessa.

7.7.1 Totuusarvot abstraktina tietotyyppinä

Esitetään tärkeimmät käsitteet esimerkin avulla. Tarkoituksena on määritellä totuusarvot algebrallisesti Lotoksen syntaksin mukaisesti. Seuraavassa tämä määrittely:

```
type Boolean is

   sorts Bool

   opns
      true, false:         -> Bool
      not:            Bool -> Bool

   eqns
      ofsort Bool
         not(true) = false;
         not(false) = true;
endtype (* Boolean *)
```

- Aluksi ilmoitetaan tietotyypin nimi ja sen jälkeen alkaa sen kuvaus.
- Kohdassa *sorts* (lajit) annetaan niiden tyyppien nimet, joista tietotyyppi rakennetaan. Tyyppi on itse asiassa melko pitkälle sama kuin joukko.
- Tietotyyppiä käsitellään operaatioiden avulla. Lotos-kuvauksessa operaatiot (opns) esitellään lajien jälkeen. Esittelyssä mainitaan operaation lähtöjoukko ja maalijoukko. Lähtöjoukko voi olla karteesinen tulo $T_1 \times T_2 \times \cdots \times T_n$ matemaattisessa mielessä, mutta Lotoksessa (ja ACT ONE -kielessä) käytetään merkintää $T_1, T_2, \cdots, T_n$. Vakiot tulkitaan operaatioiksi, joilla ei ole argumentteja. Siten vakioille annetaan vain maalijoukko.
- Lopuksi kuvataan operaatioiden semantiikka yhtälöiden avulla. Juuri tässä on menetelmän vaikeus sekä suunnittelijan että algoritmisten menetelmien kan-

nalta. Varattu sana *ofsorts* ja joukon nimi ilmaisee, mitä arvoja yhtälöissä käsitellään. Tässä kohdassa voidaan luetella useita yhtälöryhmiä.

Algebrallisessa spesifioinnissa kohtia sorts ja opns kutsutaan *signatuuriksi* (signature). Kun annetaan konkreettiset joukot ja signatuurin mukaiset konkreettiset operaatiot, jotka täyttävät yhtälöiden asettamat ehdot, saadaan eräs spesifikaatiota vastaava *algebra*. Voi olla useita algebroja, jotka toteuttavat signatuurin ja yhtälöt. On vielä määriteltävä, mitkä kaikista algebroista todella edustavat haluttua tietotyyppiä.

Huomautus. Matematiikassa algebralla tarkoitetaan rengasta, johon on lisäksi määritelty moduulistruktuuri. Siten algebrassa on kaksi laskutoimitusta, jotka käyttäytyvät kuten yhteen- ja kertolasku. Lisäksi algebran alkioita voidaan kertoa toisen renkaan alkioilla. Algebrallisessa spesifioinnissa algebralla tarkoitetaan puolestaan ns. *universaalialgebran rakenteita*. Tällaisessa rakenteessa voi olla mielivaltaisen monta operaatiota eikä operaatioiden tarvitse täyttää mitään ennalta annettuja ehtoja. Operaatioissa argumentit voivat olla yhdestä tai useammasta joukosta.

7.7.2 Termialgebra

Esitellään vielä pari esimerkkiä. Määritellään ensin luonnolliset luvut:

```
type NaturalNumber is

   sorts Nat

   opns
      0:        -> Nat
      Succ: Nat -> Nat
endtype
```

Toisena esimerkkinä on pinon määrittely. Pinoon viedään totuusarvoja, jotka on määritelty jo aikaisemmin.

```
type BooleanStack is Boolean

   sorts Stack

   opns
      empty:             -> Stack
      push:  Bool, Stack -> Stack
      top:         Stack -> Bool
      pop:         Stack -> Stack
      IsEmpty:     Stack -> Bool
```

```
eqns
   for all b: Bool, s:Stack
      ofsort Bool
         top(push(b,s)) =b;
         IsEmpty(empty) = true;
         IsEmpty(push(b,s)) = false;
      ofsort Stack
         pop(push(b,s)) = s;
endtype
```

Algebrallisen tietotyypin määrittelystä saadaan luonnollisella tavalla eräs algebra, ns. *termialgebra*. Se muodostetaan soveltamalla vakioihin operaatioita mielivaltaisen monta kertaa. Esimerkiksi luonnollisten lukujen termialgebra koostuu alkioista

```
0, Succ(0), Succ(Succ(0)), Succ(Succ(Succ(0))), jne .
```

Jos tietotyypin määrittelyssä käytetään yhtälöitä, eivät kaikki tällä tavoin saadut alkiot ole erillisiä. Yhtälöt määrittelevät termien välille ekvivalenssirelaation ja termialgebraan valitaankin tällöin termien ekvivalenssijoukot yksittäisten termien sijasta. Luonnollisten lukujen tapauksessa yhtälöitä ei ole, joten jokainen termi on oma ekvivalenssiluokkansa. Boolen arvojen tapauksessa termejä on äärettömän monta eli termien joukko on

$$\{\mathrm{not}^n(\mathrm{true}) |\, n \geq 1\} \cup \{\mathrm{not}^n(\mathrm{false}) |\, n \geq 1\} \cup \{\mathrm{true}, \mathrm{false}\}.$$

Yhtälöiden nojalla ekvivalenssiluokkia on kuitenkin vain kaksi, joten termialgebra koostuu luokista [true] ja [false].

Pinon tapauksessa erilaisia ekvivalenssiluokkia edustavat termit

```
empty, push(true, empty), push(false, empty),
push(true, push(true, empty)), push(false, push(true, empty)),
push(true, push(false, empty)), push(false, push(false, empty)),
jne
```

Tietotyypin algebrallisesta spesifikaatiosta saadaan siis suoraan eräs algebra, *tekijätermialgebra* (quotient term algebra), joka koostuu termien ekvivalenssiluokista. Algebrallisen spesifikaation semantiikan mukaan tämä edustaa "oikeaa"algebraa. Muut algebrat ovat myös semantiikan mukaisia, jos ne ovat isomorfisia tekijätermialgebran kanssa. Emme kuitenkaan määrittele tässä yhteydessä isomorfiaa tarkemmin. Riittää vain todeta, että se on tavallisen rengas- tai algebraisomorfismin yleistys universaalialgebraan.

7.8 Tietotyypin määrittely käytännössä

Algebrallinen spesifiointi on sen verran yleinen menetelmä, ettei sitä ole mahdollista hallita täysin algoritmisesti. Voi esimerkiksi sattua, ettei ole mekaanista menetel-

mää tarkistaa, ovatko kaksi termiä ekvivalentteja vai eivät. Tämän vuoksi Lotos-spesifikaatiota kirjoitettaessa onkin kiinnitettävä huomiota siihen, että tietotyyppiä voidaan käsitellä algoritmisesti. Seuraavassa on muutamia ohjeita, joita tulee noudattaa.

Lotos-ohjelmistoissa tietotyypin yhtälöt tulkitaan *muunnossysteemiksi* (rewriting system) siten, että yhtälön vasemman puolen mukainen lauseke voidaan muuttaa (sieventää) oikean puolen mukaiseksi lausekkeeksi. Esimerkiksi tyypin Boolean tapauksessa muunnossäännöt ovat

```
not(true) -> false,   not(false) -> true.
```

Siten lauseketta `not(not(not(true)))` voidaan yllä olevien sääntöjen nojalla sieventää mekaanisesti:

```
not(not(not(true)))->not(not(false))->not(true)->false.
```

Huomattakoon, että periaatteessa algebrallinen spesifiointi ei edellytä muunnossysteemitulkintaa edellä mainitussa mielessä, vaan yhtälöitä voidaan soveltaa vasemmalta oikealle ja oikealta vasemmalle. Jos kuitenkin kuvausta aiotaan analysoida Lotos-ohjelmistojen avulla, muunnossysteemitulkinta edellä esitetyssä mielessä on otettava huomioon. Tyypillisesti siis ohjelmisto toimii seuraavasti:

- Mielivaltainen termi (tietotyypin alkio) luodaan soveltamalla sallittuja operaatioita jossain järjestyksessä vakioon tai vakioihin.
- Termillä tulee olla yksikäsitteinen ns. *normaalimuoto*, joka saadaan termistä soveltamalla käypiä muunnossääntöjä kunnes termi ei enää sievene.
- Muunnos voidaan kohdistaa mihin tahansa kohtaan termissä. Tässä yhteydessä tarvitaan algoritmista termin muodon tunnistamista (pattern matching).
- Yhtälöiden oikealla puolella saa esiintyä vain sellaisia muuttujia, jotka esiintyvät myös vasemmalla puolella.
- Yhtälöt täytyy kirjoittaa niin, että oikea puoli on yksinkertaisempi kuin vasen puoli.
- Yhtälöt täytyy kirjoittaa niin, että normaalimuoto on riippumaton muunnossääntöjen soveltamisjärjestyksestä (*konfluenssiperiaate tai -ominaisuus*).
- Yhtälöt täytyy kirjoittaa niin, että mikä tahansa termi voidaan evaluoida eli saattaa normaalimuotoon.
- Erityisesti yhtälöt eivät saa sisältää silmukoita kuten $x + y = y + x$.

Yllä olevia sääntöjä noudatettaessa ohjelmisto pystyy tarkistamaan esimerkiksi kahden termin identtisyyden muuttamalla termit normaalimuotoon ja tutkimal-

la, ovatko normaalimuodot samoja. Yhtälöiden kirjoittamisessa on aiheellista ottaa huomioon vielä seuraavaa:

- Valitaan ensin operaatiot, jotka esittävät perusalkioita.
- Kirjoitetaan yhtälöt jokaiselle muulle operaatiolle.
- Kirjoitetaan yhtälöt niin, että vasemmalla puolella on muu operaatio ja oikealla perusalkioita tuottava operaatio.
- Jo määriteltyä operaatiota voidaan käyttää myös oikealla.

Tietoliikenneprotokollien spesifioinnissa ei yleensä tarvita monimutkaisia tietorakenteita, vaan luvut, Boolen arvot, erilaiset vakiot, listat ja tietueet ovat riittäviä. Nämä on helppo spesifioida algebrallisesti.

Esimerkki: Lista. Määritellään lista Lispin tyyliin. Käytössä on viisi operaatiota **NIL, CONS, CAR, CDR** ja **ATOM**. Listan alkiot ovat luonnollisia lukuja.

```
type NATURAL_LIST is NATURAL, BOOLEAN
   sorts LIST
   opns NIL: -> LIST
        CONS: NAT, LIST -> LIST
        CAR:  LIST -> NAT
        CDR:  LIST -> LIST
        ATOM: LIST -> BOOL
   eqns forall N:NAT, L:LIST
      ofsort LIST
         CAR(CONS(N,L)) = N;
         CDR(CONS(N,L)) = L;
      ofsort BOOL
         ATOM(NIL) = true;
         ATOM(CONS(N,L)) = false
endtype
```

7.8.1 Tietotyypit CADP-ohjelmistossa

CÆSAR/Aldebaran-ohjelmistossa tietotyyppien käsittely vaatii hieman ylimääräisiä toimenpiteitä. Tietotyypin perusalkiot eli vakiot on erikseen osoitettava ohjelmistolle kirjoittamalla vakion perään kommentti (* constructor *). Toinen lisäys koskee tietotyyppien esikäsittelyä. CADP-ohjelmisto kääntää tietotyyppien algebrallisen spesifikaation C-kieliseksi. Tämä tapahtuu automaattisesti, mutta käyttäjän on tietyissä tapauksissa ilmoitettava rajoituksista ohjelmistolle. Rajoitukset kirjoitetaan t-tiedostoon. Tämän jälkeen tietotyypit käsitellään ja muodostuu h-tiedosto. Valitettavasti dokumenteissa ei näytä olevan tarkkoja ohjeita t-tiedostojen käytöstä. Paras on katsoa demoesimerkkejä, joita on runsaasti hakemistossa /opt/cadp/demos koneessa *melkinkari*.

Luku 8

Esimerkkejä

8.1 AB-protokolla täydellä Lotoksella

Ohessa on CAPD-ohjelmiston demoissa oleva AB-protokolla. Aluksi on määriteltävä alternoiva bitti tietotyyppinä, koska sitä ei nyt haluta käyttää sanoman nimessä, vaan parametrina. Tämä vaatii seuraavat määrittelyt tiedostossa BITALT.lib:

```
library
   BOOLEAN, NATURAL
endlib

type BIT is
   sorts BIT
   opns 0 (*! constructor *),
        1 (*! constructor *) : -> BIT
        not : BIT -> BIT
   eqns
      forall X, Y:BIT
      ofsort BIT
         not (0) = 1;
         not (1) = 0;
endtype

type MESSAGES is NATURAL renamedby
   sortnames MSG for NAT
endtype
```

Bitin määrittelyssä käytetään valmiita määrittelyjä BOOLEAN ja NATURAL, jotka tulevat cadp-ohjelmiston mukana. Havainnollisuuden vuoksi katsotaan, miten nämä tyypit on määritelty:

```
type Boolean is
   sorts
      Bool (*! implementedby ADT_BOOL comparedby ADT_CMP_BOOL
               enumeratedby ADT_ENUM_BOOL printedby ADT_PRINT_BOOL *)
   opns
      false     (*! implementedby ADT_FALSE constructor *),
      true      (*! implementedby ADT_TRUE constructor *) : -> Bool
      not       (*! implementedby ADT_NOT *) : Bool -> Bool
      _and_     (*! implementedby ADT_AND *),
      _or_      (*! implementedby ADT_OR *),
      _xor_     (*! implementedby ADT_XOR *),
      _implies_ (*! implementedby ADT_IMPLIES *),
      _iff_     (*! implementedby ADT_IFF *),
      _eq_      (*! implementedby ADT_EQ_BOOL *),
      _ne_      (*! implementedby ADT_NE_BOOL *) : Bool, Bool  -> Bool
   eqns
      forall x, y : Bool
      ofsort Bool
         not (true)   =  false;
         not (false)  =  true;
         x and true   =  x;
         x and false  =  false;
         x or true    =  true;
         x or false   =  x;
         x xor y      =  (x and not (y)) or (y and not (x));
         x implies y  =  y or not (x);
         x iff y      =  (x implies y) and (y implies x);
         x eq y       =  x iff y;
         x ne y       =  x xor y;
endtype


type Natural is Boolean
   sorts Nat (*! implementedby ADT_NAT comparedby ADT_CMP_NAT
                 enumeratedby ADT_ENUM_NAT printedby ADT_PRINT_NAT *)
   opns 0 (*! implementedby ADT_N0 constructor *),
        1 (*! implementedby ADT_N1 *),
        2 (*! implementedby ADT_N2 *),
        3 (*! implementedby ADT_N3 *),
        4 (*! implementedby ADT_N4 *),
        5 (*! implementedby ADT_N5 *),
        6 (*! implementedby ADT_N6 *),
        7 (*! implementedby ADT_N7 *),
```

```
        8 (*! implementedby ADT_N8 *),
        9 (*! implementedby ADT_N9 *) : -> Nat
        Succ (*! implementedby ADT_SUCC constructor *) : Nat -> Nat
        _+_  (*! implementedby ADT_PLUS *),
        _*_  (*! implementedby ADT_MULT *),
        _**_ (*! implementedby ADT_POWER *),
        _-_  (*! implementedby ADT_MINUS *),
        _div_ (*! implementedby ADT_DIV *),
_mod_ (*! implementedby ADT_MOD *) : Nat, Nat -> Nat
        _eq_ (*! implementedby ADT_EQ_NAT *),
        _ne_ (*! implementedby ADT_NE_NAT *),
        _lt_ (*! implementedby ADT_LT_NAT *),
        _le_ (*! implementedby ADT_LE_NAT *),
        _gt_ (*! implementedby ADT_GT_NAT *),
        _ge_ (*! implementedby ADT_GE_NAT *),
_==_ (*! implementedby ADT_EQ_BIS_NAT *),
        _<>_ (*! implementedby ADT_NE_BIS_NAT *),
_<_ (*! implementedby ADT_LT_BIS_NAT *),
_<=_ (*! implementedby ADT_LE_BIS_NAT *),
_>_ (*! implementedby ADT_GT_BIS_NAT *),
_>=_ (*! implementedby ADT_GE_BIS_NAT *) : Nat, Nat -> Bool
        min (*! implementedby ADT_MIN *),
        max (*! implementedby ADT_MAX *),
        gcd (*! implementedby ADT_GCD *),
        scm (*! implementedby ADT_SCM *) : Nat, Nat -> Nat
   eqns
      forall m, n : Nat
      ofsort Nat
         1 = Succ (0);
         2 = Succ (1);
         3 = Succ (2);
         4 = Succ (3);
         5 = Succ (4);
         6 = Succ (5);
         7 = Succ (6);
         8 = Succ (7);
         9 = Succ (8);
      ofsort Nat
         m + 0 = m;
         m + Succ(n) = Succ(m) + n;
      ofsort Nat
         m * 0 = 0;
         m * Succ(n) = m + (m * n);
      ofsort Nat
```

```
    m ** 0 = Succ(0);
    m ** Succ(n) = m * (m ** n)
ofsort Nat
    m - 0 = m;
    Succ (m) - Succ (n) = m - n;
ofsort Nat
    n ne 0, m lt n => m div n = 0;
    n ne 0, m ge n => m div n = 1 + ((m - n) div n);
ofsort Nat
    n ne 0, m lt n => m mod n = m;
    n ne 0, m ge n => m mod n = ((m - n) mod n);
ofsort Bool
    0 eq 0                   = true;
    0 eq Succ (n)            = false;
    Succ (m) eq 0            = false;
    Succ (m) eq Succ (n)     = m eq n;
ofsort Bool
    m ne n                   = not (m eq n);
ofsort Bool
    0 lt 0                   = false;
    0 lt Succ (n)            = true;
    Succ (n) lt 0            = false;
    Succ (m) lt Succ (n)     = m lt n;
ofsort Bool
    m le n                   = (m lt n) or (m eq n);
ofsort Bool
    m ge n                   = not (m lt n);
ofsort Bool
    m gt n                   = not (m le n);
ofsort Bool
    m == n                    = m eq n;
    m <> n                    = m ne n;
    m <= n                    = m le n;
    m < n                     = m lt n;
    m > n                     = m gt n;
    m >= n                    = m ge n;
ofsort Nat
    m le n => min (m, n) = m;
    m gt n => min (m, n) = n;
ofsort Nat
    m ge n => max (m, n) = m;
    m lt n => max (m, n) = n;
ofsort Nat
    m eq n, m ne 0 => gcd (m, n) = m;
```

```
         m lt n, m ne 0 => gcd (m, n) = gcd (m, n - m);
         m gt n, n ne 0 => gcd (m, n) = gcd (m - n, n);
      ofsort Nat
         scm (m, n) = (m * n) div gcd (m, n);
endtype
```

Tyyppi BOOLEAN tuntuu yksinkertaiselta. Sen sijaan tyyppi NATURAL saattaa aiheuttaa hämmennystä aluksi. Luonnolliset luvut esitetään muodossa `succ(succ(succ(...(0)...)`; toisin sanoen luvuilla ei ole samaa esitystä kuin kymmenjärjestelmän luvuilla. Poikkeuksena ovat luvut $0, \cdots, 9$, jotka voidaan esittää normaalisti.

Cadp-ohjelmiston tietotyyppien käsittelijä (Caesar-adt) tarvitsee tietoja rajoituksista, joilla supistetaan mahdollisten alkioiden joukkoa. Tässä tapauksessa rajoitukset liittyvät sanomien dataosaan, jotka ovat luonnollisia lukuja väliltä 0 ja 4. Rajoitukset kirjoitetaan t-tiedostoon, tässä tapauksessa tiedostoon BITALT.t:

```
#define CAESAR_ADT_EXPERT_T 4.4

/* This file restricts the message numbers to the integer range 0..4 */

#define ADT_ENUM_NAT(CAESAR_ADT_0) for ((CAESAR_ADT_0) = 0;
(CAESAR_ADT_0) < 5; ++(CAESAR_ADT_0))
```

Kuten huomataan, määrittely on varsin kryptista. Kun lisäksi dokumentti ei tällä kohtaa ole kovin selkeä, lienee parasta vain matkia esimerkkijä t-tiedostojen kohdalla.

Tietotyypit vaativat siis huomattavan osan koko määrittelystä, sillä varsinaisen protokollan kuvaus ei ole kovin pitkä:

```
specification ALTERNATING_BIT_PROTOCOL [PUT, GET] : noexit

library BITALT endlib

behaviour

   hide SDT, RDT, RDTe, RACK, SACK, SACKe in
      (
         (
```

```
        TRANSMITTER [PUT, SDT, SACK, SACKe] (0 of BIT)
        |||
        RECEIVER [GET, RDT, RDTe, RACK] (0 of BIT)
        )
     |[SDT, RDT, RDTe, RACK, SACK, SACKe]|
        (
        MEDIUM1 [SDT, RDT, RDTe]
        |||
        MEDIUM2 [RACK, SACK, SACKe]
        )
     )

where

(*----------------------------------------------------------------------*)

   process TRANSMITTER [PUT, SDT, SACK, SACKe] (B:BIT) : noexit :=
      PUT ?M:MSG;            (* acquisition d'un message *)
         TRANSMIT [PUT, SDT, SACK, SACKe] (B, M)
   where
      process TRANSMIT [PUT, SDT, SACK, SACKe] (B:BIT, M:MSG) : noexit :=
         SDT !M !B;          (* emission du message *)
            (
            SACK !B;         (* bit de controle correct *)
               TRANSMITTER [PUT, SDT, SACK, SACKe] (not (B))
            []
            SACK !(not (B)); (* bit de controle incorrect => reemission *)
               TRANSMIT [PUT, SDT, SACK, SACKe] (B, M)
            []
            SACKe;           (* indication de perte => reemission *)
               TRANSMIT [PUT, SDT, SACK, SACKe] (B, M)
            []
            i;               (* timeout => reemission *)
               TRANSMIT [PUT, SDT, SACK, SACKe] (B, M)
            )
      endproc
   endproc

(*----------------------------------------------------------------------*)

   process RECEIVER [GET, RDT, RDTe, RACK] (B:BIT) : noexit :=
      RDT ?M:MSG !B;         (* bit de controle correct *)
         GET !M;             (* livraison du message *)
            RACK !B;         (* envoi d'un acquittement correct *)
```

```
            RECEIVER [GET, RDT, RDTe, RACK] (not (B))
    []
    RDT ?M:MSG !(not (B));  (* bit de controle incorrect => *)
       RACK !(not (B));     (* envoi d'un acquittement incorrect *)
          RECEIVER [GET, RDT, RDTe, RACK] (B)
    []
    RDTe;                   (* indication de perte => *)
       RACK !(not (B));     (* envoi d'un acquittement incorrect *)
          RECEIVER [GET, RDT, RDTe, RACK] (B)
    []
    i;                      (* timeout => *)
       RACK !(not (B));     (* envoi d'un acquittement incorrect *)
          RECEIVER [GET, RDT, RDTe, RACK] (B)
  endproc

(*----------------------------------------------------------------------------*)

  process MEDIUM1 [SDT, RDT, RDTe] : noexit :=
    SDT ?M:MSG ?B:BIT;   (* reception d'un message *)
       (
       RDT !M !B;        (* transmission correcte *)
          MEDIUM1 [SDT, RDT, RDTe]
       []
       RDTe;             (* perte avec indication *)
          MEDIUM1 [SDT, RDT, RDTe]
       []
       i;                (* perte silencieuse *)
          MEDIUM1 [SDT, RDT, RDTe]
       )
  endproc

(*----------------------------------------------------------------------------*)

  process MEDIUM2 [RACK, SACK, SACKe] : noexit :=
    RACK ?B:BIT;   (* reception d'un acquittement *)
       (
       SACK !B;    (* transmission correcte *)
          MEDIUM2 [RACK, SACK, SACKe]
       []
       SACKe;      (* perte avec indication *)
          MEDIUM2 [RACK, SACK, SACKe]
       []
       i;          (* perte silencieuse *)
          MEDIUM2 [RACK, SACK, SACKe]
```

```
        )
   endproc

endspec
```

8.2 Poissulkemisalgoritmeja

Toisena esimerkkinä katsomme toisen tyyppisiä sovelluksia. Kysymyksessä eivät enää ole tietoliikenneprotokollat vaan hajautetut algoritmit, joista keskitymme keskinäiseen poissulkemiseen. Aluksi näytämme pari virheellistä ratkaisua, joiden virheet paljastuvat Lotos-kuvauksen analyysissä. Lopuksi analysoimme toimivaa ratkaisua. Kaikki ratkaisut ovat puhtaasti ohjelmallisia, laitteistopiirteitä (semaforit, keskeytykset) ei käytetä.

8.2.1 Ratkaisu 1

Aluksi yritämme ratkaista poissulkemisen seuraavalla algoritmilla:

```
var DOOR: (OPEN, CLOSED); (* jaettu muuttuja *)

  DOOR := OPEN;

prosessi 1                                   prosessi 2

repeat                                   repeat
 (* jatka testaamista *)                  (* jatka testaamista *)
until DOOR = OPEN;                       UNTIL DOOR = OPEN;

DOOR := CLOSED;                          DOOR := CLOSED;
  siirry kriittiselle alueelle;            siirry kriittiselle alueelle;
  poistu kriittiseltä alueelta;           poistu kriittiseltä alueelta;
DOOR := OPEN;                            DOOR := OPEN;
```

Hajautettu algoritmi käyttää jaettua muuttujaa DOOR, joka voi saada vain kaksi arvoa. Jos muuttujan arvo on OPEN, prosessi voi siirtyä kriittiselle alueelle, muuten ei.

Ratkaisu ei selvästikään toimi, sillä prosessit saattavat lukea jaetun muuttujan arvon yhtäaikaa, jolloin molemmat siirtyvät samaan aikaan kriittiselle alueelle. Mallinnetaan nyt algoritmi Lotoksella ja todetaan tuo virhe.

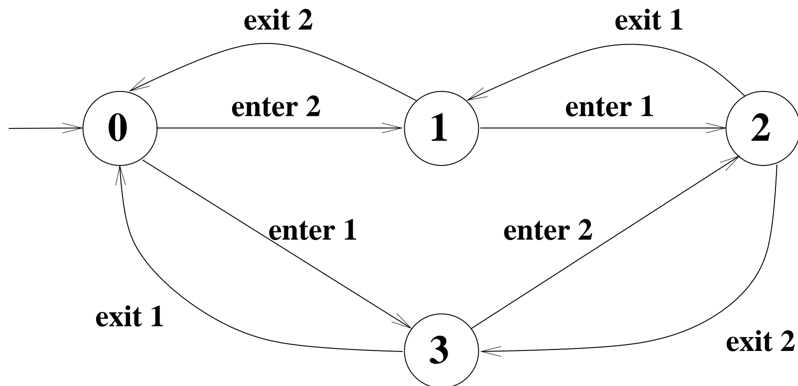
Ongelmana on jaetun muuttujan käyttö. Lotoksessa prosessit eivät voi välittää tietoa jaetun muuttujan kautta, joten se on mallinnettava prosessina. Vastatkoon muuttujaa *Door* prosessi $DOOR$. $DOOR$ kommukoi ulkomaailman, siis prosessien 1 ja 2, kanssa portin D kautta. Muuttujan arvo on prosessin $DOOR$ parametrin VAL_D arvo. Jos prosessi haluaa lukea arvon, se synkronoituu portissa D sanomalla `D !READ ?VAL:Bool`. Synkronoinnin jälkeen muuttuja VAL saa arvokseen muuttujan VAL_D arvon ja $DOOR$ palaa alkutilaansa muuttumatta mitään. Jos sen sijaan prosessi haluaa muuttaa $DOOR$in arvoa, se synkronoituu sanomalla `D !WRITE !VAL` ja vastaavasti $DOOR$ synkronoituu sanomalla `D !WRITE ?VAL:Bool`. Tämän jälkeen prosessilla on $DOOR$in arvo muuttujassaan VAL ja $DOOR$ käynnistää itsensä uudestaan alusta, mutta tällä kertaa parametrilla VAL.

```
specification mutex1[enter1, exit1, enter2, exit2]: noexit


library
  BOOLEAN
endlib


type COMMAND is
  sorts COMMAND
  opns
     READ (*! constructor *),
     WRITE (*! constructor *) : ->COMMAND
endtype



behavior


hide NCS0, NCS1, D in

  (
    P[NCS0, D, enter1, exit1]
       |||
    P[NCS1, D, enter2, exit2]
  )
   |[D]|
   DOOR[D](true)


where
```

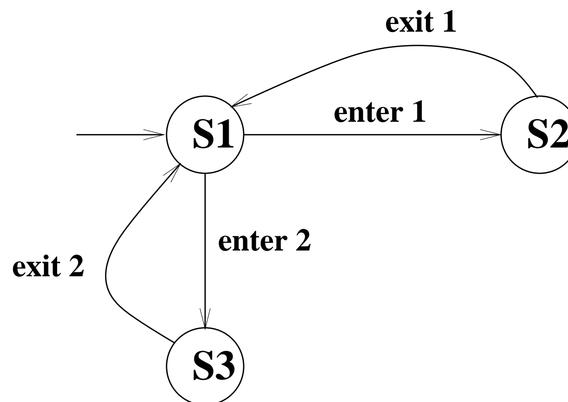
```
  process P[NCS, D, entern, exitn]:noexit :=

    NCS;
      P_AUX[NCS, D, entern, exitn]
  endproc


  process P_AUX[NCS, D, entern, exitn]:noexit :=

      D !READ ?VAL_D:Bool;

       ([VAL_D] ->
           D !WRITE !false;
             entern;
               exitn;
                 D !WRITE !true;
                  P[NCS, D, entern, exitn]


       []

        [not(VAL_D)]->
          P_AUX[NCS, D,entern, exitn]
       )
  endproc


  process DOOR[D](VAL_D:Bool):noexit :=

    D !READ !VAL_D;
      DOOR[D](VAL_D)
     []
    D !WRITE ?VAL:Bool;
     DOOR[D](VAL)
  endproc

endspec
```

Kun systeemistä muodostetaan yhteistilaverkko, se on melko suuri käsittäen 94 tilaa ja 188 siirtymää. Se voidaan minimoida suoritusjälkiekvivalenssin suhteen, jol-

loin saadaan verkko:



Siitä nähdään, että kumpikin prosessi voi siirtyä kriittiselle alueelle yhtäaikaa. Siten algoritmi on virheellinen. Tämä voitaisiin todeta myös automaattisesti kuvaamalla poissulkemisen vaatimus seuraavana "palveluna":



Selvästikään algoritmin verkko ei ole suoritusjälkiekvivalentti vaatimuksen kanssa

8.2.2 Ratkaisu 2

Seuraavassa versiossa ei käytetä jaettua muuttujaa, vaan kummallakin prosessilla on oma muuttujansa. Toinen prosessi voi kuitenkin lukea toisen prosessin muuttujan arvon.

Algoritmi on seuraava:

```
var PROCESS1, PROCESS2 : (INSIDE, OUTSIDE);
```

```
PROCESS1 := OUTSIDE;

(*process 1*)

  PROCESS1 := INSIDE;
  repeat
  until PROCESS2 = OUTSIDE;
  enter1;
  exit1;
  PROCESS1 := OUTSIDE;
```

```
PROCESS2 := OUTSIDE;

(*process 2*)

  PROCESS2 := INSIDE;
  repeat
  until PROCESS1 = OUTSIDE;
  enter2;
  exit2;
  PROCESS2 := OUTSIDE;
```

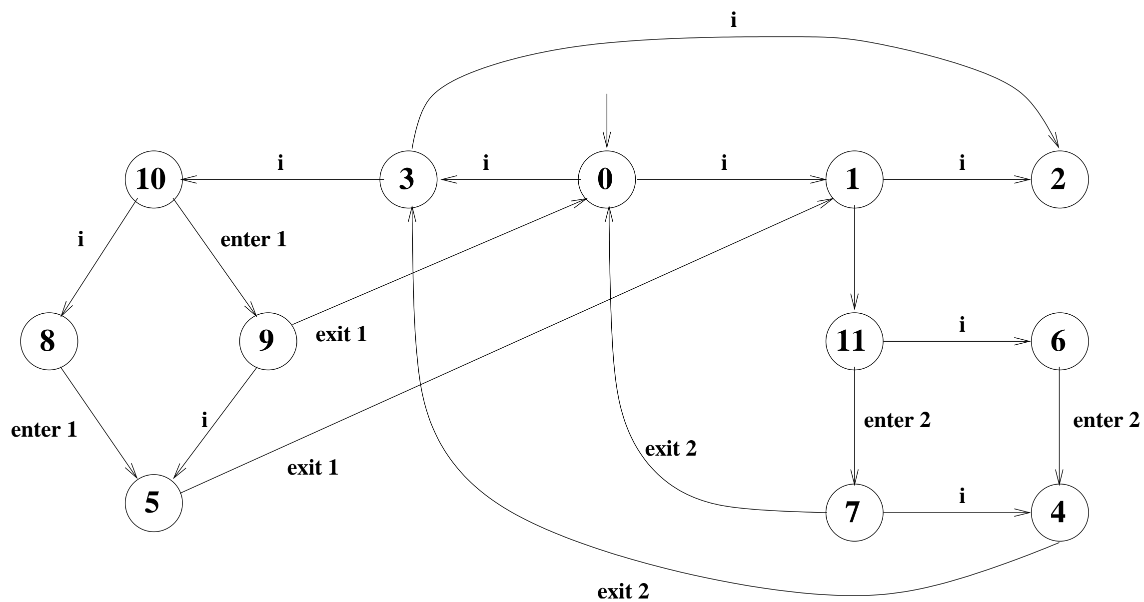
Tämä ratkaisu puolestaan johtaa lukkiumaan, jos kumpikin prosessi tekee yhtäaikaa sijoituksen PROCESS := INSIDE. Mallinnetaan taas algoritmi Lotoksella ja katsotaan, miten virhe ilmenee. Ongelmana on nyt lokaalien muuttujien mallinnus. Lokaali muuttuja voisi olla prosessin parametrina. Toinen prosessi voi lukea sen jonkin portin kautta ja prosessi itse voi muuttaa parametrin arvoa let-lauseella. Ongelmana tässä lähestymistavassa on se, että toisen prosessin pitäisi kyetä lukemaan lokaali muuttuja milloin tahansa. Siksi on helpompaa mallintaa lokaali muuttuja taas samaan tapaan prosessina kuin kuin jaettu muuttuja. Nyt vain toinen prosessi voi muuttaa lokaalin muuttujan arvoa, mutta kumpikin voi lukea arvon. Täten kommunikoinnissa lokaalin muuttujan kanssa tarvitaan kaksi porttia.

```
specification mutex2[enter1, exit1, enter2, exit2]: noexit

library
  BOOLEAN
endlib


type COMMAND is
  sorts COMMAND
  opns
    READ (*! constructor *),
    WRITE (*! constructor *) : -> COMMAND
endtype
```

```
behavior

  hide NCS1, NCS2, G1, G2 in

  (P[NCS1, enter1, exit1, G1, G2]
      |||
   P[NCS2, enter2, exit2, G2, G1]
  )

   |[G1, G2]|

  (
   PROC[G1](false)
     |||
   PROC[G2](false)
  )


where


  process P[NCS, entern, exitn, G1, G2]:noexit :=

    NCS;
     G1 !WRITE !true;
      P_AUX[NCS, entern, exitn, G1, G2]
  endproc


  process P_AUX[NCS, entern, exitn, G1, G2]:noexit :=

    G2 !READ ?VAL_P2:Bool;
      (
       [VAL_P2] -> P_AUX[NCS, entern, exitn, G1, G2]
         []
       [not(VAL_P2)] ->
           entern;
             exitn;
               G1 !WRITE ! false;
                P[NCS, entern, exitn, G1, G2]
      )

   endproc
```

```
  process PROC[G](VAL_PROC:Bool):noexit :=

     G !READ !VAL_PROC;
       PROC[G](VAL_PROC)
      []
     G !WRITE ?VAL:Bool;
       PROC[G](VAL)
  endproc

 endspec
```

Kun yhteistilaverkko muodostetaan, siinä ei näy suoraan lukkiumia, sillä prosessit voivat jatkaa lokaalin muuttajan arvon testaamista, vaikka arvo ei muutukaan. Suoritusjälkiekvivalenssi ei myöskään paljasta mitään, sillä se ei säilytä lukkiumia. Kokeillaan heikkoa bisimulaatioekvivalenssia. Se antaa seuraavan verkon:



Nytpä lukkiuma näkyykin, sillä bisimulaatioekvivalenssi muuttaa elävät lukkiumat, joissa on vain sisäistä laskentaa, tavallisiksi lukkiumiksi.

8.2.3 Dekkerin algoritmi

Ensimmäinen toimiva ohjelmallinen ratkaisu poissulkemisongelmaan on Dekkerin keksimä. Ratkaisu perustuu kahden edellisen yrityksen yhdistämiseen. Siten proses-

seilla on omat muuttujansa, joita tosin toinen prosessi voi lukea. Lisäksi algoritmissa käytetään jaettua muuttujaa. Koodi on seuraavassa:

```
var PROCESS1, PROCESS2: (INSIDE, OUTSIDE);
    TURN: 1..2; (* jaettu muuttuja *)

TURN :=1;
PROCESS1 := OUTSIDE;                          PROCESS2 := OUTSIDE;

(* prosessi 1 *)                              (* prosessi 2 *)

PROCESS1 := INSIDE;                           PROCESS2 := INSIDE;
if PROCESS2 = INSIDE then                     if PROCESS1 = INSIDE then
   if TURN = 2 then                              if TURN = 1 then
      PROCESS1 := OUTSIDE;                          PROCESS2 := OUTSIDE;
      repeat until TURN = 1;                        repeat until TURN = 2;
      PROCESS1 := INSIDE;                           PROCESS2 := INSIDE;
   end if;                                       end if;
   repeat                                        repeat
   until PROCESS2 = OUTSIDE;                     until PROCESS1 = OUTSIDE;
end if;                                       end if;

critical section;                             critical section;

TURN := 2;                                    TURN := 1;
PROCESS1 := OUTSIDE;                          PROCESS2 := OUTSIDE;
```

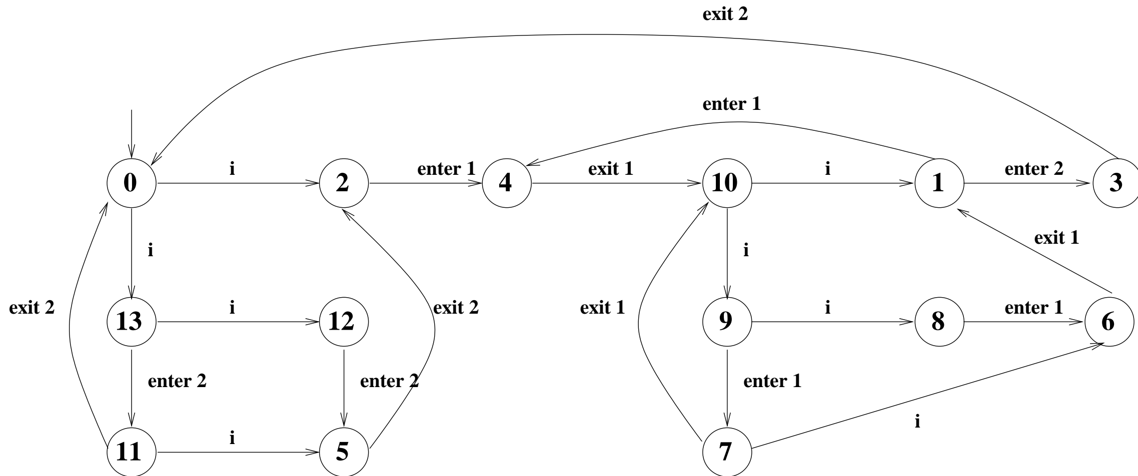
Algoritmin Lotos-kuvaus noudattaa edellä sovellettuja periaatteita. Jaettu muuttuja esitetään prosessina. Muuttujat `PROCESS1` ja `PROCESS2` voidaan esittää useammalla tavalla. Sovelletaan tässä käytäntöä, että nuokin muuttujat esitetään prosesseina. Annetaan niille kuitenkin uudet nimet `FLAG[F0]` ja `FLAG[F1]`. Lotos-koodi näyttää nyt seuraavalta:

```
specification DEKKER [NCS0, CS0, NCS1, CS1, F0, F1, T] : noexit

library
   BOOLEAN,
   NATURAL
endlib

type COMMAND is
   sorts COMMAND
   opns
```

```
      READ  (*! constructor *),
      WRITE (*! constructor *) : -> COMMAND
endtype

behaviour

    hide NCS0, NCS1, F0, F1, T in
      (
      P [NCS0, enter1, exit1, F0, F1, T] (0)
      |||
      P [NCS1, enter2, exit2, F1, F0, T] (1)
      )
   |[F0, F1, T]|
      (
      FLAG [F0] (false)
      |||
      FLAG [F1] (false)
      |||
      TURN [T] (0)
      )

where

   process P [NCS, entern, exitn, FJ, FI, T] (J : Nat) : noexit :=
      NCS; (* non critical section *)
         FJ !WRITE !true;
            P_AUX_1 [NCS, entern, exitn, FJ, FI, T] (J)
   endproc

   process P_AUX_1 [NCS, entern, exitn, FJ, FI, T] (J : Nat) : noexit :=
      FI !READ ?VAL_FI:Bool;
         (
         [VAL_FI] ->
            T !READ ?VAL_T:Nat;
               (
               [VAL_T <> J] ->
                  FJ !WRITE !false;
                     P_AUX_2 [NCS, entern, exitn, FJ, FI, T] (J)
               []
               [VAL_T == J] ->
                  P_AUX_1 [NCS, entern, exitn, FJ, FI, T] (J)
               )
         []
         [not (VAL_FI)] ->
```

```
            entern; (* critical section *)
              exitn;
               T !WRITE !(J + 1) mod 2;
                  FJ !WRITE !false;
                     P [NCS, entern, exitn, FJ, FI, T] (J)
         )
   endproc

   process P_AUX_2 [NCS, entern, exitn, FJ, FI, T] (J : Nat) : noexit :=
      T !READ ?VAL_T:Nat;
         (
         [VAL_T <> J] ->
            P_AUX_2 [NCS, entern, exitn, FJ, FI, T] (J)
         []
         [VAL_T == J] ->
            FJ !WRITE !true;
               P_AUX_1 [NCS, entern, exitn, FJ, FI, T] (J)
         )
   endproc

   process FLAG [F] (VAL_FLAG : BOOL) : noexit :=
      F !READ !VAL_FLAG;
         FLAG [F] (VAL_FLAG)
      []
      F !WRITE ?VAL:BOOL;
         FLAG [F] (VAL)
   endproc

   process TURN [T] (VAL_TURN : NAT) : noexit :=
      T !READ !VAL_TURN;
         TURN [T] (VAL_TURN)
      []
      T !WRITE ?VAL:NAT;
         TURN [T] (VAL)
   endproc

endspec
```

Yhteistilaverkko on nyt melko suuri, 1138 tilaa ja 2276 siirtymää. Kun se minimoidaan havaintoekvivalenssin suhteen, saadaan seuraava verkko:



Huomataan, ettei lukkiumia esiinny. Verkko on lisäksi suoritusjälkiekvivalenssi palvelun kanssa, joten algoritmi toimii oikein. Sen sijaan verkko ei ole heikosti bisimilaarinen palvelun kanssa, mutta tämä ei olekaan vaatimus. On siis tilanteesta riippuvaista, minkälaisia ekvivalensseja käytetään ja missä laajuudessa.

Luku 9

Aikalogiikkoja

9.1 Johdanto

Tässä viimeisessä luvussa esitellään temporaalilogiikan eli aikalogiikan käyttöä hajautettujen järjestelmien verifioinnissa. Aikalogiikka sopii erityisen hyvin elävyysominaisuuksien osoittamiseen. Toinen tärkeä hyöty on, että aikalogiikan avulla voidaan osoittaa yksittäisen ominaisuuden pitävän paikkansa systeemissä. Aina ei ole mahdollista tai järkevää spesifioida systeemiä yksityiskohtaisesti ja osoittaa totaalista oikeellisuutta. Temporaaligiikka sallii ominaisuuksien abstraktin määrittelemisen ilman, että mukaan tuodaan kaikkia yksityiskohtia. Ekvivalenssipohjaisessa verifioinnissa spesifikaatiot tahtovat olla liian konkreettisia.

Erilaisia aikalogiikkoja on useita. Perustavin jako jakaa logiikat kahteen luokkaan: *lineaarisen ajan* ja *haarautuvan ajan* logiikkoihin. Lisäongelmia tälle kurssille tuo se seikka, että temporaalilogiikan puolella lähtökohtana ei yleensä ole ollut siirtymäsysteemiformalismi siinä mielessä kuin sitä on käsitelty tällä kurssillla. Aikalogiikat perustuvat sen sijaan ns. *Kripken struktuureihin.* Kripken struktuuri on kyllä siirtymäsysteemi, mutta siirtymiin ei liity tapahtumia. Sen sijaan tiloilla oletetaan olevan ominaisuuksia, joita voidaan analysoida temporaalilogiikan kaavojen avulla. Näin ollen joudumme esittelemään uuden formalismin prosessien kuvaukselle. Kripken struktuurien ja siirtymäsysteemien välillä on kuitenkin täydellinen vastaavuus, joten siirtymäsysteemit eivät ole periaatteellinen este temporaalilogiikan käytölle.

Käsittelemme tässä luvussa Kripken struktuurit, niiden ja siirtymäsysteemien välisen yhteyden, lineaarisen ajan logiikan LTL ja haarautuvan ajan logiikan CTL perusteet. Siirtymäsysteemejä varten on kehitetty aikalogiikkoja muistuttavia logiikkoja. ACTL on CTL:n suora muunnos siirtymäsysteemeihin. Toinen tärkeä formalismi siirtymäsysteemeissä on μ-kalkyyli, joka on myös lähellä CTL:ää. Emme kuitenkaan käsittele näitä tällä kurssilla. Tavoitteena on antaa yleiskuva aikalogiikasta niin, että siihen kykenee tarvittaessa perehtymään omin voimin. Samalla luku toimii johdantona valinnaiselle kurssille Automaattinen verifiointi.

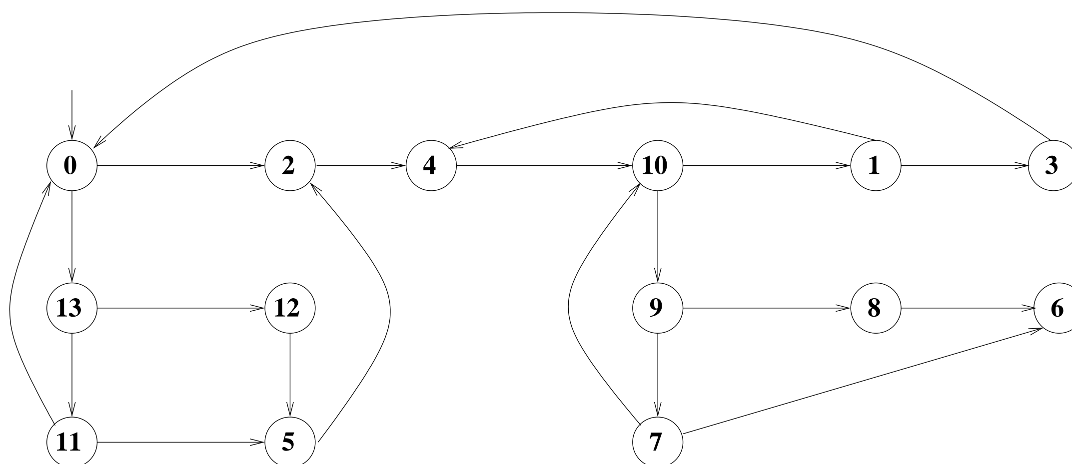
9.2 Kripken struktuurit

Perinteisesti temporaalilogiikka käsittelee mallinnusformalismeja, joissa prosessit esitetään verkkoina niin, että tiloissa on informaatiota. Siirtymät ovat sen sijaan ilman tapahtumia. Jotta aikalogiikkoja voitaisiin määritellä ja käsitellä formaalisti, myös yleisessä, siirtymäsysteemejä vastaavassa mallissa on jotenkin määriteltävä tilojen data. Se tehdään seuraavasti Kripken struktuurien avulla.

Määritelmä 5 *Kripken struktuuri on rakenne* $\mathcal{K} = (S, AP, L, \longrightarrow, 0_S)$, *missä*

- S *on tilojen joukko;*
- AP *on äärellinen epätyhjä atomisten propositiolauseiden joukko;*
- $L : S \longrightarrow \mathcal{P}(AP)$ *on funktio, joka liittää jokaiseen tilaan ne propositiot, jotka ovat tosia kyseisessä tilassa;*
- $\longrightarrow \subset S \times S$ *on siirtymäfunktio; alkiota* $(r, s) \in \longrightarrow$ *kutsutaan siirtymäksi ja sitä merkitään* $r \longrightarrow s$*;*
- 0_S *on alkutilajoukko.*

Esimerkkinä esitetään Dekkerin algoritmista aikaisemmin johdettu 14:n tilan siirtymäverkko Kripken struktuurina siten, että tiloissa on oleellinen tieto poissulkemisesta. Verkkona struktuuri saadaan suoraan siirtymäverkosta:



Otetaan käyttöön kaksi atomista propositiota, p_1 ja p_2. Propositio p_1 ilmaisee, että prosessi 1 on kriittisellä alueella ja p_2 ilmaisee, että prosessi 2 on kriittisellä alueella. Alkuperäisen kuvion perusteella p_1 on tosi tiloissa 4, 6 ja 7. Vastaavasti p_2 on tosi tiloissa 3, 5 ja 11. Muissa tiloissa p_1 ja p_2 ovat epätosia. Nämä ehdot määräävät joukon AP:n ja kuvauksen L. Keskinäisen poissulkemisen ehto voidaan nyt ilmaista logiikan lauseella "missään tilassa ei päde, että p_1 ja p_2 olisivat tosia yhtäaikaa".

Esimerkissä Kripken struktuuri muodostettiin ad hoc -periaatteella. Voidaan osoittaa, että muunnos siirtymäsysteemistä Kripken struktuuriin on aina mahdollista.

Lause 2 *Jokainen siirtymäsysteemi voidaan muuntaa (ekvivalentiksi) Kripken struktuuriksi.*

Todistus. Olkoon $LTS = (S, A, T, s_0)$ siirtymäsysteemi. Muodostetaan Kripken struktuuri $\mathcal{K} = (S', AP, L, \longrightarrow, 0_S)$ seuraavasti. Tilajoukko $S' = S \times A$. Tilojen (s_1, a_1) ja (s_2, a_2) välillä on siirtymä $\mathcal{K}$:ssa, $(s_1, a_1) \longrightarrow (s_2, a_2)$, joss on olemassa tila $s \in S$, jolla

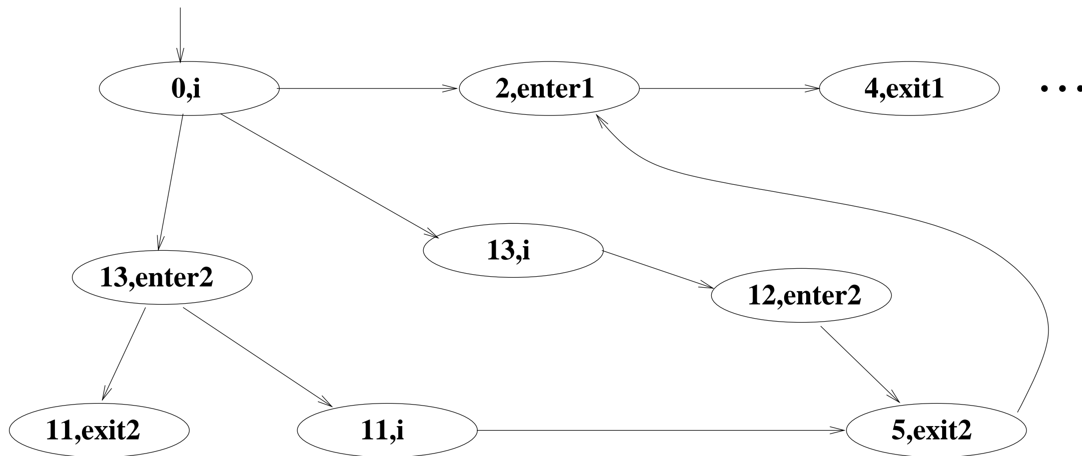
$$s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2}$$

LTS:ssä. Tämä määrittelee siirtymärelaation $\longrightarrow$ $\mathcal{K}$:ssa. Alkutilaksi $\mathcal{K}$:ssa tulee joukko

$$0_S = \{(s_0, a_0) \in S' \mid s_0 \xrightarrow{a_0} \ LTS\text{: ss}\}\,.$$

Olkoon η muuttuja, joka voi saada arvoikseen tapahtuman. Propositiolauseet ovat muotoa $\eta = a$, missä $a \in A$. Lauseen tulkinta on, että jos $\eta = a$ on tosi jossain $\mathcal{K}$:n tilassa, niin järjestelmä voi seuraavaksi suorittaa a:n siinä tilassa. Asetetaan nyt $L : S' \longrightarrow \mathcal{P}(AP)$ kaavalla $(s, a) \mapsto (\eta = a)$. $\Box$

Edellä Dekkerin algoritmin Kripken struktuuri muodostettiin tilanteen mukaan manuaalisesti. Jos sama tehdään lauseen konstruktion avulla, saadaan seuraava Kripken struktuuri (joka on piirretty vain puoliksi).



Toisinkin päin muunnos on mahdollinen.

Lause 3 *Jokainen Kripken struktuuri voidaan muuntaa vastaavaksi siirtymäsysteemiksi.*

Todistus. Harjoitustehtävä. Vihje: Siirtymien tapahtumiksi täytyy valita propositiojoukkoja. $\Box$

9.3 Lineaarisen ajan logiikka LTL

Lineaarisen ajan logiikka oli ensimmäinen aikalogiikkaformalismi, jota ruvettiin soveltamaan tietojenkäsittelyyn. Sen tärkeimpiä kehittäjiä olivat Manna ja Pnueli.

LTL:n lähtökohtana on kaksi operaattoria, X ja U. Jos p on propositio, niin Xp tarkoittaa, että seuraavan kerran eli seuraavassa tilassa p pätee. Kaava pUq puolestaan väittää, että p pätee siihen kunnes q pätee. Formaalisti LTL:n kaavat määritellään seuraavasti:

Määritelmä 6 *Olkoon AP atomisten propositioiden joukko. LTL:n* kaava *määritellään induktiivisesti:*

1. *Jos* $\phi \in AP \cup \{\top, \bot\}$, *niin* ϕ *on kaava.*
2. *Jos* ϕ *ja* ψ *ovat kaavoja, niin myös* $(\neg\phi)$, $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \rightarrow \psi)$ *ja* $(\phi \equiv \psi)$ *ovat kaavoja.*
3. *Jos* ϕ *ja* ψ *ovat kaavoja, niin myös* $X\phi$ *ja* $\phi U \psi$ *ovat.*

Kaavojen semantiikka määritellään formaalisti Kripken struktuurin $\mathcal{K}$ ja sen polun avulla. Polku π Kripken struktuurissa on äärellinen tai ääretön tilajono $s_0 s_1 \cdots s_n$ tai $s_0 s_1 \cdots$. Tila s_0 on aina Kripken struktuurin alkutila ja perättäisten tilojen välillä on siirtymä. Polku saa olla äärellinen vain, jos se päättyy tilaan, josta ei enää ole siirtymiä eteenpäin. Toisinaan vaaditaan, että kaikista tiloista täytyy olla siirtymiä, joten vain äärettömät polut tulevat kysymykseen. Käytännössä sopimuksilla ei ole suurta merkitystä. Jos π on polku $s_0 s_1 \cdots$, niin π^k tarkoittaa polun osaa $s_k s_{k+1} \cdots$.

LTL:n kaavan totuus määritellään ensin jonkin polun π suhteen. Jos ϕ on kaava ja se on tosi π:n suhteen, merkitään $\pi \models \phi$. Seuraavassa kaavan totuus polun suhteen määritellään täsmällisesti. Vakioiden $\top$ ja $\bot$ kohdalla tulkinta on, että edellinen on tosi jokaisessa tilassa, jälkimmäinen on epätosi jokaisessa tilassa.

Määritelmä 7 *Kaavan* ϕ *totuus polulla* $\pi = s_0 s_1 \cdots$ *määritellään seuraavasti:*

- *Jos* $\phi \in A \cup \{\bot, \top\}$ *on atominen propositio tai vakio, niin* $\pi \models \phi$ *joss* $s_0 \models \phi$ *(eli* $\phi \in L(s_0)$ *tai* ϕ *on* $\top$*).*
- $\pi \models \phi_1 \vee \phi_2$ *joss* $\pi \models \phi_1$ *tai* $\pi \models \phi_2$.
- $\pi \models \phi_1 \wedge \phi_2$ *joss* $\phi \models \phi_1$ *ja* $\pi \models \phi_2$.
- $\pi \models \neg\phi$ *joss* $\pi \not\models \phi$.
- $\pi \models X\phi$ *joss* π^1 *on olemassa ja* $\pi^1 \models \phi$.
- $\pi \models \phi_1 U \phi_2$ *joss* $\pi \models \phi_2$ *tai on olemassa sellainen* $k > 0$, *että* π^k *on määritelty,* $\pi^k \models \phi_2$ *ja kaikilla* i, $0 \leq i < k$, $\pi^i \models \phi_1$.

Tila s toteuttaa kaavan ϕ Kripken struktuurissa $\mathcal{K}$, $\mathcal{K}, s \models \phi$, jos ja vain jos jokaisella s:stä alkavalla polulla π $\mathcal{K}$:ssa pätee $\pi \models \phi$.

Kaava ϕ on tosi struktuurissa $\mathcal{K}$, $\mathcal{K} \models \phi$, joss $\mathcal{K}$:n jokaisella polulla $\pi \models \phi$.

Operaattorit F, G ja R

Tarkastellaan kaavaa $\top U\phi$. Koska $\top$ on tosi jokaisessa tilassa, $\top U\phi$ on tosi polulla π joss $\pi^k \models \phi$, jollakin $k \geq 0$. Siis $\top U\phi$ on tosi, jos ϕ on tosi jossain vaiheessa tulevaisuudessa. Kaavalle käytetään merkintää $F\phi$.

Kaava $G\phi$ puolestaan sanoo, että ϕ on tosi jokaisessa polun tilassa. Se voitaisiin ilmaista myös F:n avulla muodossa $\neg F\neg\phi$.

Kaava $\phi R\psi$ ilmaisee, että äärellisen askelmäärän jälkeen ϕ pätee ja sitä ennen ψ pätee jokaisessa tilassa mukaan lukien se tila, jossa ϕ pätee.

Tyypillisiä kaavoja

Seuraavassa on tyypillisiä LTL:n kaavoja ja niiden havainnollinen tulkinta.

1. Keskinäinen poissulkeminen:
$$G\neg(\text{critical}_1 \wedge \text{critical}_2)$$
2. Korkeintaan yksi pyyntö kuitataan:
$$\bigwedge_{i<j} G\neg(\text{ack}_i \wedge \text{ack}_j)$$
3. Elävyysominaisuus, jonka mukaan vuoroni on äärettömän usein:
$$GF\,\text{myturn}$$
4. Elävyysominaisuus, että *try*-alueelle pääseminen johtaa lopulta kriittiselle alueelle:
$$G(\text{try} \rightarrow F\text{critical})$$
5. Alustuksen jälkeen systeemi pysyy alustettuna:
$$FG\,\text{initialized}$$

Aikalogiikan avulla ei ole helppo lausua kaikkia ominaisuuksia. Esimerkiksi annettakoon seuraava hissin ominaisuus.

Sillä aikavälillä, kun hissiä pyydetään kerrokseen ja se avaa ovensa tuossa kerroksessa, hissi voi saapua (ohittaa) tuohon kerrokseen korkeintaan kaksi kertaa:

$$\begin{array}{lll} G(& (& \text{call} \wedge F\text{open}) \rightarrow \\ & & ((\neg\text{atfloor} \wedge \neg\text{open})U \\ & & (\text{open} \vee ((\text{atfloor} \wedge \text{open})U \\ & & (\text{open} \vee ((\neg\text{atfloor} \wedge \neg\text{open})U \\ & & (\text{open} \vee ((\text{atfloor} \wedge \neg\text{open})U \\ & & (\text{open} \vee ((\neg\text{atfloor} \wedge \text{open})))))))))) \end{array}$$

9.4 Haarautuvan ajan logiikka CTL

Toisena logiikkaformalismina esitellään vielä CTL, *Computational Tree Logic*. Se lisää neljä uutta operaattoria operaattorivalikoimaan:

- EX (existential),
- AX (all),
- $E(\cdot U \cdot)$,
- $A(\cdot U \cdot)$.

Määritelmä 8 *CTL:n* kaava *määritellään induktiivisesti.*

1. *Jos* $\phi \in AP \cup \{\top, \bot\}$, *niin* ϕ *on kaava.*
2. *Jos* ϕ *ja* ψ *ovat kaavoja, niin ovat myös* $(\neg\phi)$, $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \rightarrow \psi)$ *ja* $(\phi \leftrightarrow \psi)$.
3. *Jos* ϕ *ja* ψ *ovat kaavoja, niin ovat myös* $EX\phi$, $AX\phi$, $E(\phi U \psi)$ *ja* $A(\phi U \psi)$.

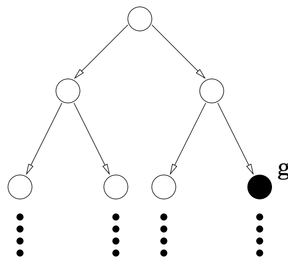
Kaavojen totuus määritellään Kripken struktuurin M ja sen tilan suhteen seuraavasti.

- Jos $\phi \in AP \cup \{\bot, \top\}$, niin $M, s_0 \models \phi$ joss $s_0 \models \phi$.
- $M, s_0 \models \phi \vee \psi$, joss $M, s_0 \models \phi$ tai $M, s_0 \models \psi$.
- $M, s_0 \models \phi \wedge \psi$, joss $M, s_0 \models \phi$ ja $M, s_0 \models \psi$.
- $M, s_0 \models \neg\phi$, joss $M, s_0 \not\models \phi$.
- $M, s_0 \models EX\phi$, joss on olemassa $s' \in S$, jolla $s_0 \rightarrow s'$ ja $M, s' \models \phi$.
- $M, s_0 \models AX\phi$, joss kaikilla $s' \in S$ pätee: jos $s_0 \rightarrow s'$, niin $M, s' \models \phi$.
- $M, s_0 \models E(\phi U \psi)$, joss
  - $M, s_0 \models \psi$ tai
  - on olemassa polku $\pi = s_0 \longrightarrow s_1 \longrightarrow \cdots \longrightarrow s_n \longrightarrow \cdots$, jolle löytyy sellainen $k > 0$, että $M, s_K \models \psi$ ja kaikilla i, $0 \leq i < k$, $M, s_i \models \phi$.
- $M, s_0 \models A(\phi U \psi)$, joss
  - $M, s_0 \models \psi$ tai
  - kaikille äärettömille poluille $\pi = s_0 \longrightarrow s_1 \longrightarrow \cdots \longrightarrow s_n \longrightarrow \cdots$ löytyy sellainen $k > 0$, että $M, s_K \models \psi$ ja kaikilla i, $0 \leq i < k$, $M, s_i \models \phi$; tai
  - kaikille äärellisille poluille $\pi = s_0 \longrightarrow s_1 \longrightarrow \cdots \longrightarrow s_n$ pätee $M, s_i \models \phi$ kaikilla i, $0 \leq i \leq n$.

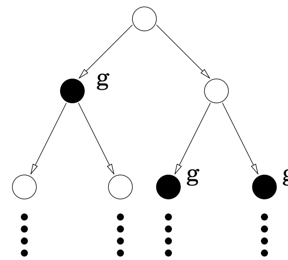
Tavallisesti kaavoissa käytetään seuraavia operaattoreita, jotka voidaan ilmaista edellisten avulla:

1. AX: kaikilla poluilla seuraavaksi pätee.
2. EX: on olemassa polku, jolla seuraavaksi pätee.
3. AF: kaikilla poluilla pätee jossain vaiheessa.
4. EF: jollakin polulla pätee jossain vaiheessa.
5. AG: kaikilla poluilla pätee aina.
6. EG: jollakin polulla pätee aina.
7. AU eli $A[f \cup g]$: kaikilla poluilla pätee f kunnes g pätee.
8. EU eli $E[f \cup g]$: jollakin polulla pätee f kunnes g pätee.
9. AR eli $A(fRg)$: kaikilla poluilla pätee g siihen asti kunnes f pätee mukaan lukien se tila, jossa f pätee ensimmäisen kerran; f:n ei kuitenkaan tarvitse päteä missään tilassa.
10. ER eli $E(fRg)$: jollakin polulla pätee g siihen asti kunnes f pätee mukaan lukien se tila, jossa f pätee ensimmäisen kerran; f:n ei kuitenkaan tarvitse päteä missään tilassa.

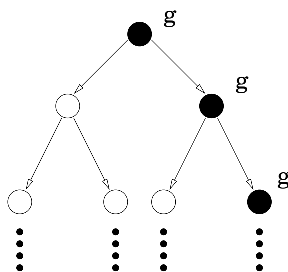
Neljä tavallisinta operaattoria kuvioina:



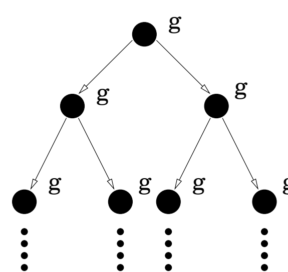
$M, s_0 \models \mathbf{EF}\, g$



$M, s_0 \models \mathbf{AF}\, g$



$M, s_0 \models \mathbf{EG}\, g$



$M, s_0 \models \mathbf{AG}\, g$

Tyypillisiä CTL:n kaavoja ovat mm:

1. $EF(Start \wedge \neg Ready)$: On mahdollista saavuttaa tila,jossa *Start* pätee, mutta *Ready* ei päde.
2. $AG(Req \longrightarrow AF Ack)$: Jos tehdään pyyntö, se lopulta kuitataan.
3. $AG(AF\ DeviceEnabled)$: Propositio *DeviceEnabled* pätee äärettömän usein jokaisella laskentapolulla.
4. $AG(EF\ Restart)$: Jokaisesta tilasta on mahdollista saavuttaa *Restart*-tila.

Kumpikaan logiikoista CTL ja LTL ei sisälly toiseen. Esimerkiksi LTL ei voi ilmaista polun olemassaoloa, mutta CTL voi kuten kaavalla $EX\phi$. Toisaalta CTL ei voi ilmaista elävyysrajoituksia kuten LTL:n kaavaa

$$GF(\nu = tick) \longrightarrow GF(\nu = \beta).$$

Sen tähden onkin muodostettu näiden logiikkojen laajennos CTL*, joka sisältää sekä CTL:n että LTL:n. Sitä ei kuitenkaan käsitellä tällä kurssilla.

9.5 Mallintarkistus

Mallintarkistus (model checking) tarkoittaa tekniikkaa, jonka avulla systeemistä rakennetaan äärellinen malli ja testataan, että haluttu ominaisuus pätee mallissa. Temporaalilogiikan yhteydessä tämä merkitsee, että logiikan avulla ilmaistaan haluttu ominaisuus, systeemistä generoidaan Kripken struktuuri ja testataan, päteekö logiikan kaava struktuurissa vai ei. Tarvitaan siis kaksi ohjelmistokomponenttia: Ensin ohjelmisto, joka jäsentää jollain spesifiointikielellä kirjoitetun kuvauksen ja generoi siitä Kripken struktuurin. Sen jälkeen jäsennetään temporaalilogiikan kaava ja testataan, onko se tosi struktuurissa. Seuraavassa katsomme, minkälainen algoritmi ratkaisee totuusongelman edellyttäen, että logiikan kaava on oikein muodostettu.

Kaavan totuuden selvittäminen LTL:ssä ja CTL:ssä eroaa oleellisesti toisistaan. LTL:ssä totuuden selvittäminen on hankalampaa. Vaativuudeltaan se on pahimmassa tapauksessa PSPACE-täydellistä, joskin käytännössä se on yleensä helpompaa. Sen sijaan CTL:n kaavan totuuden selvittäminen on polynomista. Sen tähden katsomme CTL:n mallintarkistusalgoritmin ja jätämme LTL:n algoritmit jatkokursseille. Seuraava CTL:n algoritmien kuvaus on teoksesta Clarke, Grumberg, Peled, *Model Checking*.

Olkoon $\mathcal{K} = (S, R, L, O_{s_0})$ Kripken struktuuri ja f CTL:n kaava. Algoritmi tulee toimimaan siten, että se liittää jokaiseen tilaan s tiedon siitä, mitkä f:n alikaavat ovat tosia tuossa tilassa. Merkitään tuota alikaavojen joukkoa tilassa s symbolilla $label(s)$. Aluksi $label(s)$ on $L(s)$. Sen jälkeen algoritmi kiertää silmukassa siten, että i. kierroksella kaikki f:n alikaavat, joissa on $i - 1$ sisäkkäistä operaattoria, tulevat arvotetuksi. Lopussa pätee ehto $M, s \models F$ joss $f \in label(s)$.

Mikä tahansa CTL:n kaava voidaan ilmaista operaattoreiden $\neg$, $\vee$, EX, EU ja EG avulla. Siten mallintarkistusalgoritmin tarvitsee käsitellä vain kuutta tapausta f, $\neg f_1$, $f_1 \vee f_2$, $EX\ f_1$, $E[f_1 U f_2]$ ja $EG\ f_1$, missä f on atominen ja f_1 ja f_2 on puolestaan saatu käyttämällä edellä mainittuja viittä operaattoria. Edellytyksenä tietysti on, että CTL:n mielivaltainen kaava muunnetaan ensin yhteen kuudesta tapauksesta. Tämäkin voidaan tehdä algoritmisesti, mutta siihen ei puututa tässä luvussa.

Jos kaava on muotoa $\neg f$, merkataan ne tilat, joissa f on epätosi eli joita f ei ole merkannut. Kaavalle $f_1 \vee f_2$ merkataan ne tilat, jotka joko f_1 tai f_2 on merkannut. Kaavalle $EX\ f$ merkataan ne tilat, joiden jokin edeltäjä on merkattu f:llä.

Kaava $g = E[f_1 U f_2]$ vaatii hieman enemmän käsittelyä. Ensin etsitään kaikki ne tilat, jotka f_2 merkkaa. Sen jälkeen liikutaan kaaria taaksepäin lähtien f_2:n merkkaamista tiloista ja tutkitaan, onko f_1 merkinnyt kyseisen tilan. Jos f_1 on merkinnyt tilan, g merkkaa myös saman tilan. Näin jatketaan polulla taaksepäin kunnes tullaan tilaan, jota f_1 ei ole merkinnyt. Seuraavassa on algoritmi, joka tekee yllä kuvatun etsinnän.

```
procedure CheckEU(f1, f2)
    T := {s | f2 ∈ label(s)};
    for all s ∈ T do label(s) := label(s) ∪ { E[f1 U f2] };
    while  T ≠ ∅ do
        choose s ∈ T;
        T := T \ {s};
        for all t such that R(t, s) do
            if E[f1 U f2] ∉ label(t) and f1 ∈ label(t) then
                label(t) := label(t) ∪ { E[f1 U f2] };
                T := T ∪ {t};
            end if;
        end for all;
    end while;
end procedure;
```

Tapaus $g = EG\ f$ on hieman hankalampi. Sen ratkaisu perustuu vahvasti yhtenäisiin komponentteihin. Olkoon $\mathcal{K}$ Kripken struktuuri. Muodostetaan $\mathcal{K}$:sta $\mathcal{K}'$ jättämällä pois kaikki ne tilat, joissa f ei päde. Tiloihin liittyvät kaaret ja todet propositiot jätetään myös pois.

Propositio 1 *$\mathcal{K}$, $s \models EG\ f$, joss seuraavat kaksi ehtoa ovat voimassa:*

1. *$s \in S'$.*
2. *On olemassa polku $\mathcal{K}'$:ssa, joka johtaa s:stä johonkin epätriviaalin vahvasti yhtenäisen komponentin solmuun t. (Komponentti on epätriviaali, jos se sisältää enemmän kuin yhden solmun.)*

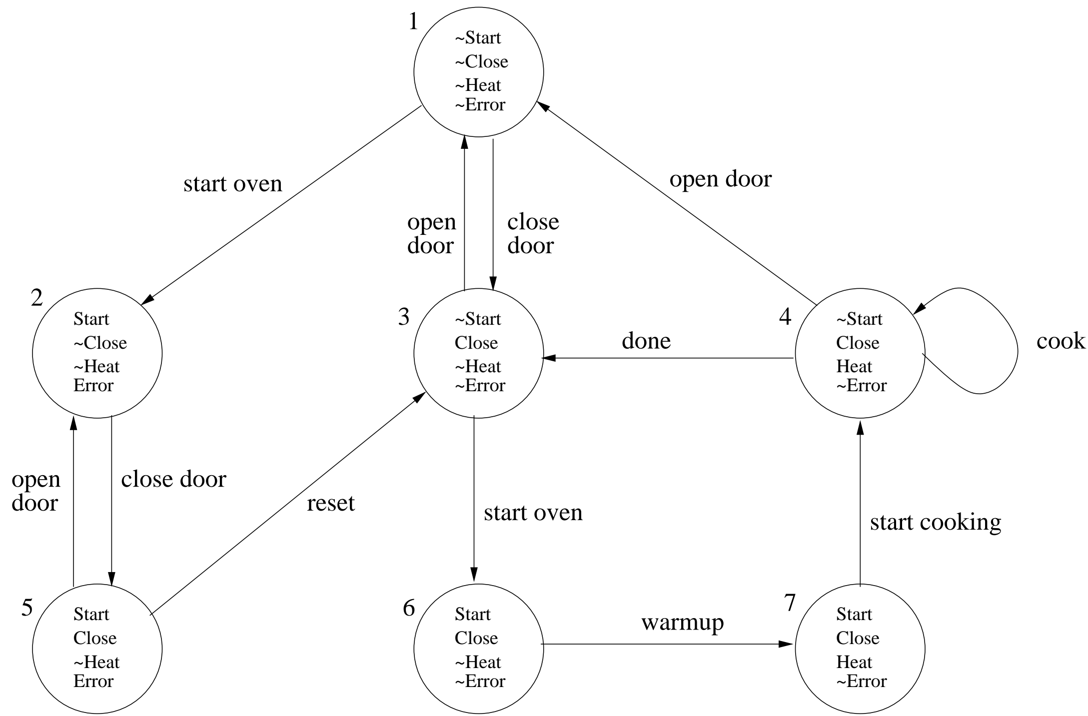
Todistus. Oletetaan, että $\mathcal{K}, s \models EG\ f$. Selvästi $s \in S'$. Olkoon π ääretön polku, joka alkaa s:stä ja jonka jokaisessa solmussa f pätee. Koska $\mathcal{K}$ on äärellinen, on mahdollista kirjoittaa π muotoon $\pi = \pi_0\pi_1$, missä π_0 on äärellinen π:n alkuosa ja π_1 π:n sellainen ääretön loppuosa, että jokainen π_1:n tila esiintyy polulla äärettömän usein. Näin π_0:n tilat kuuluvat verkkoon $\mathcal{K}'$. Olkoon C π_1:n tilajoukko. Selvästi myös C kuuluu verkkoon $\mathcal{K}'$. Osoitetaan nyt, että minkä tahansa kahden C:n tilan välillä on polku. Olkoon $s_1, s_2 \in C$. Olkoon $s_1^{(i)}$ jokin kohta polulla π_1, jossa s_1 esiintyy. Jossain etäämpänä esiintyy puolestaan s_2 π_1:n oletusten perusteella. Siis s_1:n ja s_2:n välillä on äärellinen polku, jonka kaikki tilat ovat C:ssä. Siis C on vahvasti yhtenäinen komponentti tai sen osa.

Oletetaan, että ehdot (1) ja (2) ovat voimassa. Olkoon π_0 polku s:stä t:hen. Olkoon π_1 äärellinen, vähintään yhden mittainen polku t:stä t:hen. Tällainen π_1 löytyy, koska $t \in C$ ja C on vahvasti yhtenäinen. Jokainen tila äärettömällä polulla $\pi = \pi_0\pi_1^\omega$ toteuttaa f:n. Koska π on polku $\mathcal{K}$:ssa alkaen s:stä, pätee $M, s \models EG\ f$. □

Algoritmi kaavan $g = EG\ f$ tarkistukseen saadaan suoraan propositiosta. Algoritmissa joudutaan ensin laskemaan vahvasti yhtenäiset komponentit. Tämä voidaan tehdä lineaarisessa ajassa Tarjanin algoritmilla. Se esitetään useimmissa algoritmi- ja tietorakennekirjoissa. Jos tämä algoritmi oletetaan tunnetuksi, saadaan $EG\ f$:n laskeva algoritmi seuraavasti.

```
procedure CheckEG(f1)
    S' := {s | f1 ∈ label(s)};
    SCC := {C | C is a nontrivial SCC of S' };
    T := ⋃_{C∈SCC} {s | s ∈ C};
    for all s ∈ T do label(s) := label(s) ∪ { EG f1};
    while T ≠ ∅ do
        choose s ∈ T;
        T := T \ {s};
        for all t such that t ∈ S' and R(t, s) do
            if EG f1 ∉ label(t) then
                label(t) := label(t) ∪ { EG f1 };
                T := T ∪ {t};
            end if;
        end for all;
    end while;
end procedure;
```

Tarkastellaan esimerkkinä seuraavaa mikroaaltouunin spesifikaatiota.



Olkoon tarkistettava kaava $AG(Start \longrightarrow AF \text{ Heat})$. Kaava väittää, että kaikissa suorituksissa mikroaaltouunin käynnistäminen johtaa joissain vaiheessa siihen, että uuni on lämmin. Kaava voidaan kirjoittaa algoritmin vaatimaan muotoon

$$\neg EF(\text{Start} \wedge EG \neg \text{Heat}).$$

Määrätään ensin tilat, joissa vallitsee ehto **Start** tai ¬**Heat**:

$$\begin{array}{rcl} \text{Start} & : & \{2,5,6,7\} \\ \neg\text{Heat} & : & \{1,2,3,5,6\} \end{array}$$

Seuraavaksi määrätään vahvasti yhtenäiset komponentit, joiden tiloissa vallitsee ¬**Heat**. Saadaan yksi komponentti $\{1,2,3,5\}$. Komponentin jokainen tila merkataan kaavalla EG ¬Heat. Toisaalta algoritmin while-silmukka ei tuo uusia tiloja merkattavaksi.

Muistetaan, että $EF\, f = E[\text{true} U\, f]$. Kun symbolilla $S(f)$ merkitään niitä tiloja, joissa f on tosi, saadaan

$$S(EF(\text{Start} \wedge EG \neg \text{Heat})) = S(E[\text{true}\, U\, (\text{Start} \wedge EG \neg \text{Heat})]).$$

Siten joukon $S(EF(\text{Start} \wedge EG \neg \text{Heat}))$ tilojen laskeminen aloitetaan määrittelemällä

ne tilat, joissa Start $\wedge$ $EG\neg$Heat on tosi. Saadaan joukko

$$T = \{2, 5\}.$$

Tämän jälkeen kuljetaan tiloista 2 ja 5 taaksepäin ja merkataan jokainen löydetty tila kaavalla EF(Start $\wedge$ $EG\neg$Heat). Löydetään tilat

$$1, 2, 3, 4, 5, 6, 7.$$

Kaavan negaation laskeminen sujuu ottamalla komplementti kaikkien tilojen suhteen. Siis

$$S(\neg EF(\text{Start} \wedge EG\neg\text{Heat})) = \emptyset.$$

Koska Kripken struktuurin alkutila ei kuulu tähän joukkoon, struktuuri ei toteuta kaavaa.

Syy kaavan pätemättömyyteen löytyy mikroaaltouunin erikoisista käyttötavoista. Jos uuni aloitetaan ja sen jälkeen jatkuvasti vain suljetaan ja avataan ovea, ei lämmitys pääse käyntiin. Samoin jos aloitetaan ja suljetaan ovi, mutta sen jälkeen resetoidaan ja avataan ovi sekä jatketaan samassa syklissä, ei lämmitys pääse alkuun. Tällaiset suorituspolut liittyvät käsitteeseen *reiluus*. Voidaan väittää, ettei alituinen uunin ovien avaaminen ja sulkeminen ole reilua uunin käyttöä. Samanlainen epäreilu tilanne vallitsisi tiedonsiirrossa, jos kanava aina hukkaisi tai vääristäisi sanoman eikä koskaan päästäisi oikeaa sanomaa läpi.

Reiluus voidaan ottaa huomioon mallintarkistuksessa melko yksinkertaisella tavalla. Riittää määritellä sopiva tilajoukko F ja vaatia, että reilussa, äärettömän pitkässä suorituksessa jokainen F:n tila esiintyy äärettömän monta kertaa. Mallintarkistuksessa voidaan tämän jälkeen rajoittua polkuihin, jotka ovat reiluja määritellyn tilajoukon suhteen. Esimerkiksi mikroaaltouunin toiminta saadaan pakotettua reiluksi vaatimalla, että ehdon (ja siis ehtoa vastaavien tilojen) **Start** $\wedge$ **Close** $\wedge$ $\neg$**Error** tulee esiintyä äärettömän usein. Siis ehto takaa, että uunin käytössä joudutaan pääsykliin. Tällaisilla poluilla mallintarkistusesimerkin kaava pätee. CTL:n mallintarkistusalgoritmeja voidaan helposti muuttaa ottamaan huomioon reiluusehdot. Nämä modifikaatiot jäävät kuitenkin jatkokursseille.