View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Helsingin yliopiston digitaalinen arkisto

Ohjelmistokomponenttien yhteentoimivuus

Toni Ruokolainen

Helsinki 30.04.2004

Pro gradu -tutkielma

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta/Osasto — Fakultet/Sektion — Faculty
Matemaattis-luonnontieteellinen

Laitos — Institution — Department
Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Toni Ruokolainen

Työn nimi — Arbetets titel — Title

Ohjelmistokomponenttien yhteentoimivuus

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Pro gradu -tutkielma	30.04.2004	97

Tiivistelmä — Referat — Abstract

Tämän opinnäytetyön tarkoitus on selvittää niitä periaatteita ja tekniikoita, joiden avulla voidaan tarkistaa ohjelmistokomponenttien välinen yhteentoimivuus. Yhteentoimivuuden tarkistamista tarvitaan komponenttien uudelleenkäytön mahdollistamiseksi sekä ohjelmistoarkkitehtuurien suunnittelun yhteydessä.

Ohjelmistokomponentin yhteentoimivuus muiden komponenttien kanssa riippuu sen syntaktisista ja semanttisista ominaisuuksista sekä ulkoisesta käyttäytymisestä. Nämä ominaisuudet määritellään formaalisti tyyppiteoriaa, ontologioita ja prosessialgebroja käyttäen. Komponenttien syntaktinen korvautuvuus määritellään niin kutsuttujen termiautomaattien ja niiden välisten suhteiden avulla. Komponentin semanttiset ominaisuudet määrittelevät yleistä käsistöä käyttäen, mitä komponentti tekee. Semantiikan kuvaukseen käytetään pääasiassa ontologisia käsitteitä ja semanttisissa vertailuissa käytetään avuksi logiikkaa.

Komponentin ulkoinen käyttäytymiskuvaus ilmaisee sen käyttäytymismallin, jota kyseinen komponentti tukee. Käyttäytymiskuvaus on sekä ohje että rajoitus, joka toisaalta antaa mallin oikealliselle vuorovaikutukselle ja toisaalta toimii komponenttien välisen korvautuvuuden tarkistusehtona. Komponentin ulkoinen käyttäytyminen, niin kutsuttu rajapintaprotokolla, kuvataan käyttäen prosessialgebraa. Tässä opinnäytetyössä on prosessialgebraksi valittu π-kalkyyli.

Kun komponenttien ominaisuudet on mallinnettu formaalisti, voidaan komponenttien välinen yhteensopivuus ja korvautuvuus tarkistaa ohjelmallisesti. Tällaista ohjelmistoa voidaan hyödyntää esimerkiksi ohjelmiston suunnitteluprosessissa.

ACM Computing Classification System (CCS):
C.2.4 [Distributed Systems]
D.2.4 [Software/Program Verification]
D.2.12 [Interoperability]

Avainsanat — Nyckelord — Keywords
ohjelmistokomponentti, yhteentoimivuus, formaalit menetelmät, verifiointi, semantiikka, pi-kalkyyli

Säilytyspaikka — Förvaringsställe — Where deposited
Tietojenkäsittelytieteen laitoksen kirjasto, sarjanumero C-2004-

Muita tietoja — övriga uppgifter — Additional information

Sisältö

1 Johdanto **1**

2 Komponenttien yhteentoimivuus **3**

2.1 Yhteensopivuus ja korvautuvuus 4

2.2 Ohjelmistokomponentin tyypitys 5

3 Syntaktinen yhteentoimivuus **8**

3.1 Rajapintarakenteen tyypin määrittely 9

3.1.1 Perustyypit 9

3.1.2 Rakenteiset tyypit 10

3.1.3 Rekursiiviset tyypit 13

3.2 Rajapintarakenteiden korvautuvuus 15

3.2.1 Rakenteellinen isomorfismi 16

3.2.2 Rakenteellinen alityypitys 20

3.3 Rajapintarakenteiden tulkinta termiautomaatteina 21

3.3.1 Termin määritelmä 22

3.3.2 Termiautomaatti 23

3.4 Termien välisen korvautuvuuden tarkistaminen 25

3.4.1 Termien välinen ekvivalenssi 25

3.4.2 Termien välinen alityypitys 27

3.5 Syntaktisen yhteensopivuuden käyttökohteet 31

4 Semanttinen yhteentoimivuus **33**

4.1 Komponentin toimintasemantiikka 33

4.2 Abstraktit tietotyypit 37

4.3 Ontologiat 39

4.4 Toimintasemantiikan määrittelystä ja käyttötavoista 44

5 Protokollatason yhteentoimivuus **47**

5.1 Rajapintaprotokollien yhteensopivuus ja korvautuvuus 47

5.2 Rajapintaprotokollien kuvaaminen prosessialgebralla 49

5.2.1 Yksinkertainen prosessialgebra 51

5.2.2 π-kalkyyli 55

5.3 Mallintarkistus yhteensopivuuden tarkastamisessa 58

5.4 Rajapintaprotokollien korvautuvuus ja prosessien ekvivalenssit 63

5.4.1 Prosessien ekvivalenssi π-kalkyylissä 66

5.4.2 Eräs π-kalkyylin kongruenssi relaatio 68
5.5 Rajapintaprotokollien yhteentoimivuuden tarkistaminen 70
5.5.1 Yhteensopivuuden formalisointi 70
5.5.2 Korvautuvuuden vaatimukset 73
5.5.3 Mietteitä korvautuvuusrelaation muodostamisesta 76

6 Komponenttien yhteentoimivuuden tarkistaminen ohjelmallisesti 78
6.1 Ohjelmiston yleiskuvaus 78
6.2 Komponenttien kuvaaminen ja kuvaamiskielet 79
6.3 TypeMatcher-komponentin toiminta 82
6.4 Tietämysjärjestelmän toiminta 84
6.5 ProcessMatcher-komponentin toiminta 86

7 Yhteenveto 89

Lähteet 91

A OrderedContainer–tietotyyppi

B ServiceInterface-ontologian OWL-kuvaus

C MyBankingService-ontologian OWL-kuvaus

D PiProcess-ontologian OWL-kuvaus

Kuvat

1 Tyyppiä $IntList$ edustava ääretön puu 14
2 Rekursiivisia rajapintarakenteita 14
3 Äärettömän rekursiivisen tyypint $IntList$ äärellinen approksimaatio käyttäen takaisinosoitusta 16
4 Church-Rosser–ominaisuus 19
5 Äärellinen termi t 22
6 Ääretön, säännöllinen termi s 23
7 Esimerkki termiautomaatista 25
8 $match_{plug-in}$ predikaatin toiminnallinen tulkinta 36
9 Nimeämättömän ontologian $\mathcal{O}_{esim}$ rakenne 40
10 Nimetyn ontologian $(\mathcal{O}_{esim}, \mathcal{L}_{esim})$ rakenne 41
11 Eräs ontologioiden jäsennys kerrosrakenteeksi 43
12 RDF-kielen pääresurssien luokkahierarkia 44
13 Esimerkki siirtymäjärjestelmästä 49
14 Pankkiautomaattiesimerkin prosessilausekkeiden LTS-mallit 54
15 Esimerkki nimen viestintään perustuvasta mobiliteetista 55
16 Mallintarkistuksen periaate 59
17 Äärellinen automaatti ja siitä generoituvat suoritusjäljet [CES86] 60
18 Esimerkkejä temporaalilogiikan kaavoista 63
19 Agenttien A ja B siirtymäpuut 64
20 Vahva ja heikko bisilaarisuus 65
21 ServiceInterface-ontologian rakenne 81
22 TypeMatcher-komponentin yleiskuva 83
23 Reasoner-komponentin yleiskuva 84
24 ProcessMatcher-komponentin yleiskuva 86

Taulukot

1 Funktiotyypin evaluointi- ja tyypityssäännöt 10
2 Parityypin evaluointi- ja tyypityssäännöt 11
3 Monikkotyypin evaluointi- ja tyypityssäännöt 12
4 Tyyppien perusluokat ja esimerkit 13
5 Esimerkki komponenttityypistä 13
6 Operaatioden fold ja unfold määritelmät 14
7 $IntList$-termin yksinkertainen lavennus 15
8 $IntList^1$-termin supistaminen 15
9 Tyyppiteorian **Th** aksioomat 17
10 Komponenttien isomorfismin todistaminen rajoitetun tyyppijärjestelmän aksioomilla 19
11 Tyyppiteorian Th reduktiojärjestelmä R_{Th} 20
12 Alityypitysrelaation yleiset ominaisuudet 20
13 Monikkotyypin alityypityssäännöt 21
14 Funktiotyypin alityypityssäännöt 21
15 AC-alityypityksen säännöt $EXPLORE - ARROW$ ja $EXPOLORE - PI$. 28
16 Alityypityksen todistusvelvoitteet muodostava algoritmi 29
17 Joukon U alityypitysrelaation $\leq_{AC}$ kiintopisteen laskeva algoritmi 30
18 Etu-/jälkiehtoihin perustuvat sovituspredikaatit $match_x$ 35
19 $match_{pred}$ tyyppiset yhteensopivuusfunktiot [ZW97] 37
20 ACT-ONE -kielellä määritellyt tietotyypit $bool$ ja $stack$ 38
21 Esimerkki RDF-pohjaisesta ontologian instanssista 43
22 Esimerkki nimetyn ontologian RDF-Schema -muotoisesta osittaisesta määritelmästä 45
23 Esimerkki OWL-rajoituksen käytöstä uuden käsitteen mallintamisessa 45
24 CCS-prosessialgebran siirtymäsäännöt 52
25 Esimerkki prosessialgebran käytöstä 53
26 π-kalkyylin agenttien syntaksi 56
27 π-kalkyylin siirtymäsäännöt 57
28 π-kalkyylin eräs rakenteellinen kongruenssi 58
29 ACTL-temporaalilogiikan kaavoissa käytettävät merkintätavat 61
30 ACTL-logiikan kaavojen syntaksi 62
31 WSDL-kuvauksen yleinen rakenne 80
32 Semanttisten laajennusten liittäminen WSDL-kuvaukseen 82
33 Esimerkki RDF-Schema -kielen alityypityksen määrittelystä JESS-kielellä 85
34 Esimerkki metodin etuehdosta JESS-sääntönä 86

Määritelmät

2.1 Tyypin määritelmä 5
2.2 Ohjelmistokomponentin tyyppi 6
2.3 Alityypityksen turvallisuus 7
3.1 Rakenteellinen isomorfismi 16
3.2 Tyyppien ekvivalenssiteoria T-EQ 18
3.3 Syntaktisen termin määritelmä 22
3.4 Syntaktisen alitermin määritelmä 23
3.5 Termin t termiautomaatti M_t 24
3.6 Termiautomaattien bisimulointi 26
3.7 Termiautomaattien AC-bisimulointi 26
3.8 Termien alityypitys 27
3.9 Termien AC-alityypitys 28
4.1 Semanttinen sovituspredikaatti $match$ 34
4.2 Ontologia 39
4.3 Ontologiaan liittyvä sanasto 40
4.4 Tietämyskanta $\mathcal{KB}$ 41
4.5 Tietämyskantaan liittyvä sanasto $\mathcal{L}^{\mathcal{KB}}$ 42
5.1 Prosessilausekkeiden muodostussäännöt 51
5.2 Siirtymäkaavojen toteutuminen 61
5.3 ACTL-temporaalilogiikan semantiikka 62
5.4 Vahva bisimulaatio 64
5.5 Vahva bisimilaarisuus 65
5.6 Heikko bisimulaatio 65
5.7 Ekvivalenssirelaation kongruenssi 66
5.8 Vahva bisimulointi π-kalkyylissä 67
5.9 π-prosessien ekvivalenssi 67
5.10 Nimien korvauksien rajoitus 68
5.11 Prosessien vahva D-ekvivalenssi 68
5.12 (Vahva) avoin bisimulointi 68
5.13 $\asymp$-simulointi 69
5.14 Agentin onnistunut ja epäonnistunut suoritus 71
5.15 Syötteen välittävä agentti 71
5.16 Synkronisoituvat agentit 71
5.17 Agenttien yhteensopivuusrelaatio 72
5.18 Yhteensopivat agentit 72

5.19 Haarautuva bisimulaatio 75
5.20 Vahva $\mathbf{0} - bisimulointi$ 75

1 Johdanto

Ohjelmistotuotanto on alusta asti elänyt kriisissä: ohjelmistot eivät valmistu ajallaan, projektien budjetit ylittyvät ja tulokseksi saadaan toimimattomia ohjelmistoja. Tätä kriisiä helpottamaan on luotu ilmaisuvoimaisempia ja abstraktiotasoltaan korkeampia ohjelmointikieliä, erilaisia paradigmoja sekä ohjelmistotuotantoprosesseja.

Ohjelmistot tulevat yhä suuremmiksi ja mutkikkaammiksi rakentaa. Lisäksi ohjelmistoilta vaaditaan enemmän. Ohjelmistolta voidaan vaatia, että sitä on mahdollista tarpeen tullen laajentaa ja päivittää, joskus jopa ajonaikaisesti. Lisäksi ohjelmisto voi olla osa suurempaa hajautettua järjestelmää tai sen eri komponentit voivat olla hajautettuna ympäri maailmaa. Saman toiminnallisuuden toteuttavat komponentit voivat olla eri ohjelmointikielillä toteutettuja sekä eri ohjelmistotuotantoryhmien tuotosta. Myös niiden alkuperäiset käyttötarkoitukset voivat erota toisistaan. Tästä syystä ei ole mitenkään selvää, minkälaisia komponentteja ohjelmistoon voidaan valita. Erityisesti avoimissa hajautetuissa järjestelmissä on yksi perustavimmista ongelmista se, mistä voidaan löytää ohjelmiston käyttöön oikeanlainen komponentti.

Viime vuosina ohjelmistokehitys on kulkenut oliokeskeisen ajattelun kautta kohti komponenttipohjaista ohjelmistojen kasausta. Komponenttipohjaisen ohjelmistotuotannon päämääränä on parantaa ohjelmistojen laatua sekä vähentää tuotantoprosessin kustannuksia ja siihen tarvittavaa aikaa hyödyntämällä valmiita, uudelleenkäytettäviä ohjelmistokomponentteja.

Ohjelmistokomponentti on ohjelmistossa käytettävän palvelun abstraktio, joka piilottaa kyseisen palvelun toteutustavan tarkoin määritellyn käyttörajapinnan taakse. Komponenttipohjaiseen ohjelmistotuotantoon liittyy tiukasti käsite *ohjelmistoarkkitehtuuri*, jolla tarkoitetaan järjestelmän komponenttien ja niiden välisisten vuorovaikutuksien kuvausta [LVM95]. Samanlaisten komponenttien tulisi olla korvattavissa keskenään samalla tavoin kuin perinteisillä teollisuuden aloilla on voitu hyödyntää standardiosia jo vuosikymmeniä ja valmis ohjelmisto tulisi voida koostaa eri komponentteja yhdistämällä.

Ohjelmistorakenteiden uudelleenkäyttö voidaan jakaa kahteen pääryhmään: *white-box*– ja *black-box*–uudelleenkäyttö [MN97]. White-box–uudelleenkäyttö on tällä hetkellä yleisempää ja se on perinteinen ohjelmistotuotannossa käytetty malli. Tähän ryhmään kuuluu esimerkiksi ns. “leikkaa-ja-kopioi”–uudelleenkäyttö ja valmiiden luokkien yhdistely suuremmaksi kokonaisuudeksi. White-box–uudelleenkäytössä ongelmana on se, että jotta valmiita ohjelmistorakenteita voitaisiin käyttää, niin niiden toiminnallisuus täytyy tuntea tarkasti. Toteutustapa ja siten toiminnallisuus tarkistetaan käytettävien ohjelmistorakenteiden dokumentoinnista tai lähdekoodista.

Black-box–uudelleenkäyttö on ideaalitapaus, jota tällä hetkellä komponenttipohjaisessa ohjelmistotuotannossa yritetään saavuuttaa. Jotta komponenttia voitaisiin käyttää uudelleen “mustana laatikkona”, tulee sillä olla eksplisiittisesti määritelty toimintatapa, käyttörajapinta sekä vaatimukset toimintaympäristölle (mm. riippuvuudet muista komponenteista). Tässä mallissa ohjelmisto voidaan koota tarkoin määritellyistä komponenteista, jolloin ohjelmiston kokoojan ei tarvitse opetella käytettävien komponenttien toteutustapaa. Tällöin ohjelmisto voisi toimia pelkästään “sovelluslogiikkaliimana” eri standardikomponenttien välillä, joiden toimintapa ja oletukset ympäristön suhteen tunnetaan tarkoin.

Komponenttipohjainen ohjelmistotuotanto lupaa paljon hyvää, mutta se ei ole vielä lunastanut lupauksiaan. Ongelmana on erityisesti niin kutsutut arkkitehtuuriset epäyhteensopivuudet, jotka johtuvat komponenttien implisiittisistä oletuksista ympäristönsä suhteen [GAO95]. Näiden yhteensopimattomuusongelmien vuoksi ohjelmiston tuottajan on vaikea löytää tarkoituksen mukainen komponentti ohjelmistokirjastosta. Lisäksi komponenttien yhteensovittaminen on ongelmallista,

koska komponenttien käyttäytyminen ja oletukset ympäristöstä ovat epätarkasti määriteltyjä.

Komponenttien yhteensopivuudella tarkoitetaan sitä, että komponentit toimivat ennalta määritellyllä tavalla, jos ne yhdistetään toisiinsa [AG97]. Toisin sanoen kaikki viestinnän osapuolet osaavat käsitellä ja tulkita saamansa viestit oikein. Perinteisesti komponentin yhteensopivuus on tarkistettu rajapintamäärittelyn ja sitä kuvaavan dokumentoinnin avulla. Komponentin syntaktinen käyttötapa saadaan rajapinnan rakenteesta, kun taas käyttötarkoitus ja käyttäytyminen, komponentin semantiikka, ilmenee vain selväkielisen dokumentoinnin avulla.

Tämä ei kuitenkaan riitä siinä vaiheessa, kun komponenttien tulisi olla keskenään yhteensopivia ja korvattavissa toisillaan tai kun nämä tarkistukset tulisi tehdä automaattisesti. Syntaktinen tarkastelu komponentin lähdekoodista ei kerro tarpeeksi esimerkiksi komponentin tarkoitetusta käyttötavasta, komponenttien välisistä riippuvuuksista tai niiden välisistä poissulkemisominaisuuksista. Toisaalta selväkielinen dokumentti ei läheskään aina ole tarpeeksi selvä tai yksityiskohtainen, jotta esimerkiksi eri valmistajien komponentit olisivat keskenään yhteistyökykyisiä tai olisivat korvattavissa keskenään [BS00]. Komponenttien välistä yhteistyö- ja korvattavuusanalyysiä varten tarvitaan formaaleja tapoja määritellä komponenttien toimintaa.

Tämän lopputyön tarkoituksena on tutkia komponenttien välistä yhteensopivuutta ja korvautuvuutta sekä erityisesti kuinka nämä ominaisuudet voidaan tarkistaa. Luvussa 2 tutkitaan, mitä tarkoitetaan sellaisilla peruskäsitteillä kuin komponenttien yhteensopivuus, korvautuvuus ja tyypitys. Luvuissa 3 ja 4 käsitellään komponenttien syntaktista ja toimintasemanttista yhteensopivuutta. Luvussa 5 tutustutaan komponentin rajapinnan käyttäytymisen kuvaukseen ja niin kutsuttujen rajapintaprotokollien väliseen yhteentoimivuuteen. Tämän jälkeen luvussa 6 tutkitaan, kuinka voitaisiin toteuttaa ohjelmisto, joka toteuttaa komponenttien välisen yhteentoimivuustarkistuksiin tarvittavan toiminnallisuuden.

2 Komponenttien yhteentoimivuus

Komponenttipohjaisen ohjelmistosuunnittelu rakentuu sille periaatteelle, että pienemmistä ohjelmistokokonaisuuksista, komponenteista, voidaan luoda suurempia kokonaisuuksia. Komponentit ovat ohjelmistorakenteiden abstraktioita. Niiden toteutustapa voi olla esimerkiksi funktion, moduulin tai luokan ilmentymä. Käytettävät komponentit voivat olla eri ohjelmointikielillä toteutettuja ja toisistaan riippumattomien kehittäjien tuottamia. Näistä ja useista muista syistä johtuen komponentit ovat hyvin heterogeenisiä. Heterogeenisyydestä huolimatta olisi suotavaa, että komponentteja voitaisiin käyttää uudelleen erilaisissa yhteyksissä ja saman toiminnallisuuden toteuttavat komponentit olisivat korvattavissa keskenään. Tällöin järjestelmää on helpompi laajentaa ja päivittää yhteentoimivia komponentteja käyttäen.

Jotta komponenttipohjainen ohjelmistonkehitys olisi mielekästä, tulisi ohjelmiston kehittäjän pystyä hakemaan käyttökelpoisia ja yhteistoimintaan kykeneviä komponentteja komponenttikirjastosta ja kehitysympäristön tulisi tukea tätä toiminnallisuutta. Nykyään komponentteja etsitään lähinnä niiden nimen ja dokumentoinnin perusteella esimerkiksi suoraan tiedostojärjestelmästä tai jonkin tietokantajärjestelmän avulla. Kun komponenttikirjaston sisältämien komponenttien lukumäärä kasvaa useisiin tuhansiin, ei tällainen lähestymistapa ole enää mielekäs. Tarvitaan parempia hakutapoja, joissa hakukriteeri kertoo enemmän komponentin ominaisuuksista kuin pelkkä nimi.

Ohjelmistotuotannollisen näkökulman lisäksi komponenttipohjaista lähestymistapaa voidaan käyttää hyväksi hajautettujen järjestelmien toteuttamisessa. Erityisesti niin kutsutuissa avoimissa hajautetuissa järjestelmissä tarvittaisiin jokin apujärjestelmä, jonka avulla yhteentoimivia komponentteja voitaisiin sekä etsiä että hyödyntää tehokkaasti ja joustavasti [BCI+97]. Avoimissa hajautetuissa järjestelmissä komponenttipalvelut sijaitsevat kukin oman hallintoalueensa sisällä. Tämän vuoksi komponenttien yhteentoimivuutta ei voida taata yhtenäiseen kehitys- ja ylläpitosäännöstöön vedoten, koska kullakin palveluntarjoajalla on omat sisäiset intressinsä.

Komponentin nimi ei yleisessä tapauksessa kerro mitään sen toiminnallisuudesta tai käyttökelpoisuudesta. Toki voidaan harrastaa yhtenäisiä nimeämistapoja, jotka kuvaavat komponentin toiminnallisuutta, mutta tällainen lähestymistapa ei ole riittävä. Esimerkiksi kahden eri komponenttivalmistajan nimeämiskäytännöt voivat erota toisistaan siinä määrin, että tarvittavan komponentin löytäminen ja uudelleenkäyttö hankaloituu.

Nimeä parempi hakukriteeri on jonkinlainen kuvaus etsittävän komponentin ominaisuuksista. Nämä ominaisuudet voidaan jakaa kahteen ryhmään: syntaktisiin ja semanttisiin ominaisuuksiin. Komponenttien syntaktiset ja semanttiset ominaisuudet määrittelevät komponenttien välisen yhteentoimivuuden. Syntaktisista ominaisuuksista selviää miten komponenttia tulee käyttää. Komponentin käyttötapa määritellään perinteisesti rajapintakuvauksen avulla, joka määrittelee muun muassa metodien syöte- ja paluuarvojen tyypit.

Useimmissa tapauksissa komponentin rajapinnan ja varsinaisen toteutuksen tulisi olla siinä mielessä erilliset, että julkinen rajapinta paljastaa vain komponentin toteuttamat palvelut (esimerkiksi metodikutsut), mutta ei komponentin varsinaista toimintatapaa. Tällä lähestymistavalla saavutetaan se etu, että komponentin toiminnallisuuden päivitys tai korvaaminen muilla tekniikoilla ei näy asiakkaalle päin. Komponentin rajapinnan kuvauksen tulisi viitata mahdollisimman vähän varsinaiseen toteutustekniikkaan, jotta komponenttia voitaisiin uudelleenkäyttää tehokkaasti.

Semanttiset ominaisuudet kuvaavat kuinka komponentti käyttäytyy. Semanttisiin ominaisuuksiin kuuluvat metodien toimintasemantiikka, kommunikoinnissa käytettävien viestien tulkinnat sekä komponentin ulkoisen käyttäytymisen kuvaus. Metodien toimintasemantiikka antaa kuvauksen siitä, minkälaisessa tilanteessa yksittäistä metodia voidaan kutsua ja mitkä ovat metodikutsun seu-

raukset sekä vaikutukset. Viestien semantiikan määritteleminen mahdollistaa sen, että voidaan erottaa toisistaan komponentit, jotka käsittelevät maksumääräyksiä niistä komponenteista, jotka käsittelevät koiria ja kissoja. Ulkoisen käyttäytymisen kuvauksilla voidaan erottaa ja identifioida komponentteja niiden käyttäytymisen perusteella. Esimerkiksi komponentti, joka hyväksyy kaksi epäonnistunutta sisäänkirjautumista ennen lopettamista, voidaan erottaa komponentista, jonka mukaan käyttäjä voi yrittää sisäänkirjautumista niin kauan kunnes onnistuu.

Tässä luvussa käsitellään yleisellä tasolla komponenttien yhteentoimivuutta. Luvussa 2.1 käsitellään yhteensopivuuden ja korvautuvuuden käsitteitä ohjelmistokomponenttien näkökulmasta. Tämän jälkeen luvussa 2.2 annetaan ohjelmistokomponentin määritelmä sekä käsitellään hieman perinteistä tyyppiteoriaa, jotta yhteentoimivuustarkistuksia voidaan määritellä tyyppiteoreettisin keinoin.

2.1 Yhteensopivuus ja korvautuvuus

Komponenttien yhteentoimivuudella tarkoitetaan toisiinsa yhdistettyjen komponenttien kykyä kommunikoida ja toimia yhdessä oikein [AG97]. Jotta komponenttipohjaisesta ohjelmistotuotantoprosessista saataisiin kaikki hyöty irti, tulisi komponenttien kyetä yhteistyöhön riippumatta toteutuskielestä, rajapinnasta tai laitealustasta [Kon93].

Yhteentoimivuudella on sekä staattisia että dynaamisia piirteitä [Weg96]. Staattiset vaatimukset sisältävät rajapintojen rakenteiden yhteentoimivuuden sekä komponenttien väliset riippuvuudet. Staattinen yhteentoimivuus voidaan tarkistaa tyyppitarkistuksella esimerkiksi ohjelmiston toteutuksen aikana. Dynaamisia vaatimuksia ovat muun muassa vaatimukset viestintäprotokollasta ja komponenttien käytöksestä, toisin sanoen vaatimukset komponentin ajonaikaisesta käyttäytymisestä.

Staattisten vaatimuksien tuomat ongelmat ovat tällä hetkellä hyvin ymmärrettyjä ja niiden ratkaisemiseksi on kehitetty väliohjelmistoalustoja, komponenttipohjaisia ohjelmointikehitysympäristöjä ja näiden tueksi erilaisia rajapinnankuvauskieliä (*Interface Definition Language*) eli IDL:iä. Näiden tekniikoiden avulla heterogeenisen ympäristön luomat epäyhteensopivuudet eri tyyppien välillä voidaan kiertää [BS00]. Muun muassa CORBA-väliohjelmistoalustalle ja Microsoftin COM-arkkitehtuurille on määritelty omat IDL-kielensä. Nämä mahdollistavat sen, että kyseisillä alustoilla eri komponentteja ei tarvitse luoda samaa ohjelmointikieltä tai -paradigmaa käyttäen, koska kyseiset IDL-kuvaukset on käännetty usean eri ohjelmointikielen käyttöön. Esimerkiksi CORBA-arkkitehtuurissa IDL-kuvaukset voidaan kääntää mm. C++-, C-, Smalltalk- ja Ada-kielille [OHE97].

Komponenttien dynaamisiin vaatimuksiin eli niiden ajonaikaiseen käyttäytymiseen perustuvaa yhteentoimivuutta ei sen sijaan tunneta vielä yhtä hyvin. Dynaamiset vaatimukset ovatkin ongelma, johon komponenttipohjaisessa ohjelmistotuotannossa ja järjestelmissä ei olla vielä pystytty tyhjentävästi vastaamaan. Käyttökelpoisia komponentteja on vaikea löytää ja komponenttien yhteensovittaminen on hankalaa. Ongelmat johtuvat uudelleenkäytettävien komponenttien implisiittisesti määritellyistä vaatimuksista ja oletuksista ympäröivän järjestelmän rakenteen suhteen [GAO95].

Komponenttien yhteentoimivuuden ensimmäinen aspekti on komponenttien keskinäinen *korvautuvuus* (*substitutability*). Korvautuvuudella tarkoitetaan sitä, että keskenään vaihdettavat komponentit käyttäytyvät ulkoisesti samalla tavalla. Huomioitavaa on, että kyseisten komponenttien ei tarvitse olla täydellisiä kopioita toisistaan, riittää vain että korvauksen jälkeen muut komponentit voivat käyttää järjestelmää samalla tavalla kuin ennenkin [LW94].

Yhteentoimivuuden toinen vaatimus on komponenttien välinen *yhteensopivuus* (*compatibility*).

Yhteensopivuudella tarkoitetaan sitä, että komponentit ymmärtävät toisiaan ja kykenevät ennalta määritellyn kaltaiseen yhteistyöhön. Komponentit ymmärtävät toisiaan, jos ne pystyvät hyödyntämään toistensa kutsurajapintoja oikealla tavalla. Kutsurajapintojen oikea käyttötapa tarkoittaa muun muassa palveluiden (metodien) oikeaa kutsujärjestystä ja valideja syötteitä. Yhteensopivuustarkistuksiin voi myös sisältyä esimerkiksi liiketoiminnallisia tarkistuksia, esimerkiksi erilaisten toiminnallisten ja lainopillisten sopimuksien yhteensovittamista [QMT$^+$02].

Yhteensopivuudesta voidaankin oikeastaan puhua vain viestien välityksen yhteydessä ja yhteensopivuus on luonteeltaan dynaamisempaa kuin korvautuvuus, joka voidaan usein tarkistaa esimerkiksi ohjelmiston käännösaikana. Yhteensopivuuteen tutustutaankin tarkemmin luvussa 5, jossa tarkastellaan protokollatason yhteentoimivuutta.

Yhteensopivuus ja korvautuvuus muodostavat yhteentoimivuuden kaksi ulottuvuutta, jotka eivät kuitenkaan ole täysin erillisiä tai riippumattomia käsitteitä. Jotta komponentti A voidaan korvata komponentilla B, tulee niiden olla korvattavissa keskenään ja B:n pitää olla yhteensopiva kaikkien niiden komponenttien kanssa, joiden kanssa A oli yhteensopiva. Toisaalta myös yhteensopivuustarkasteluissa tarvitaan korvautuvuustarkistuksia.

2.2 Ohjelmistokomponentin tyypitys

Tyypin käsite tulee matematiikan joukko-opista, jossa tyyppejä tarvitaan rajaamaan objektien välisiä vuorovaikutuskeinoja ja sulkemaan pois “liian suuriin” joukkoihin liittyviä ongelmia, kuten esimerkiksi Russellin paradoksi [Leh00]. Tyypillä tarkoitetaan nimettyä joukkoa objekteja, joiden käyttäytyminen on jollain tasolla yhteneväistä [CW85]. Tämä on niin kutsuttu klassinen tulkinta tyypeille. Muut tyypin tulkintavat ovat “tyypit algebroina” ja “tyypit teorioina” [CFP99].

Esimerkiksi kokonaislukujen tyyppi on joukko alkioita, jotka käyttäytyvät samoin alkioihin kohdistuvien operaatioiden, kuten yhteen-, vähennys- ja jakolaskun, suhteen. Ohjelmointikielen yhteydessä tyypillä tarkoitetaan joukkoa objekteja, joilla on yhteinen struktuuri ja jotka käyttäytyvät samoin niihin kohdistuvien operaatioiden yhteydessä [AC93]. Tässä opinnäytetyössä käytetään tyypille tulkintaa, joka on annettu määritelmässä 2.1.

Määritelmä 2.1 (Tyypin määritelmä)
Tyypillä tarkoitetaan joukkoa objekteja, joiden rakenne on “tarvittavalla tarkkuustasolla” yhteneväinen ja jotka ulkoisesti käyttäytyvät “samalla tavalla” niihin kohdistuvien operaatioiden suhteen kaikissa konteksteissa.

Määritelmä 2.1 on tarkoituksella jätetty hieman avoimeksi tässä vaiheessa. Seuraavissa luvuissa kuvataan tarkemmin, mitä on “tarvittava tarkkuustaso” ja mitä tarkoittaa se, että objektit käyttäytyvät ulkoisesti “samalla tavalla ... kaikissa konteksteissa”. Yksittäisellä tyypillä ei välttämättä ole yksikäsitteistä edustajaa, vaan tiettyä tyyppiä edustavat objektit voivat esiintyä eri muodoissa. Objektien kuuluminen tiettyyn tyyppiin määritellään sopivaa ekvivalenssirelaatiota käyttäen.

Tyypityksen avulla olioita, joilla on havaittavissa oleva rakenne ja käyttäytyminen, voidaan jakaa ekvivalenssiluokkiin. Erityisesti komponentit voidaan jakaa ekvivalenssiluokkiin komponenttien tyyppien mukaan, kunhan komponenttien tyypitykselle annetaan sopivat perusteet. Yhteentoimivuuden tarkistaminen onkin komponenttien välistä tyyppitarkistusta.

Tässä opinnäytetyössä ohjelmistokomponentti tulkitaan ohjelmiston toiminnallisuuden abstraktioksi, jolla on eksplisiittinen kuvaus eli rajapinta. Rajapinta määrittelee komponentin tyypin ja komponenttien välisen yhteentoimivuuden tarkistaminen on niiden komponenttityyppien välisten suhteiden tarkastelemista tietyn tyyppijärjestelmän sääntöjen sisällä.

Komponentin rajapinta sisältää joukon metodien ja attribuuttien kuvauksia, tyypityskontekstin sekä niin kutsutun rajapintaprotokollan kuvauksen. Metodien ja attribuuttien kuvaukset sitovat tietyn tyypin kuhunkin metodin tai attribuutin nimeen. Tyyppi voi olla joko yksinkertainen ja jakamaton perustyyppi tai jollakin tyyppikonstruktorilla muodostettu rakenteinen tyyppi. Metodien ja attribuuttien kuvausta kutsutaan rajapinnan rakenteeksi (*signature*).

Rajapintaprotokolla kuvaa komponentin ulkoisen käyttäytymisen ottamatta kantaa komponentin toteutusyksityiskohtiin. Ulkoisen käyttäytymisen kuvaus määrittelee lailliset järjestykset metodikutsuille ja viestien kommunikoinnille. Siinä missä perinteiset tyypit määrittelevät tietynlaisen staattisen tyypitysrakenteen, määrittelee rajapintaprotokolla dynaamisen käyttäytymistyypin.

Ohjelmistokomponentilla viitataan tästä lähtien nimen omaan ohjelmistokomponentin rajapinnan kuvaukseen, komponentin tyyppiin, joka koostuu rakenteellisesta kuvauksesta (*signature*), tyyppimäärittelyistä sekä rajapintaprotokollan kuvauksesta. Ohjelmistokomponentin määritelmä on annettu määritelmässä 2.2.

Määritelmä 2.2 (Ohjelmistokomponentin tyyppi)
Ohjelmistokomponentin tyyppi eli rajapinta $\mathcal{C} = (\Sigma, \mathcal{P}, \mathcal{T})$ *koostuu rajapinnan rakenteesta* Σ*, rajapintaprotokollasta* $\mathcal{P}$*, sekä tyypityskontekstista* $\mathcal{T}$*.*

Rajapinnan rakenne Σ on joukko tyyppimäärityksiä $(a : T)$, "*a on tyyppiä T*", missä a on metodin tai attribuutin nimi ja T jokin tyyppi. Rajapinnan rakenteen tyypitystä käsitellään tarkemmin luvussa 3.

Rajapintaprotokolla $\mathcal{P}$ on formaali prosessikuvaus, joka kuvaa komponentin ulkoisen käyttäytymisen. Ulkoisen käyttäytymisen kuvaamiseen käytetään prosessialgebraa, jonka periaatteisiin tutustutaan luvussa 5.

Tyypityskonteksti $\mathcal{T}$ määrittelee perustyypit sekä niiden väliset ekvivalenssi- ja alityypityssuhteet. Tyypityskonteksti antaa tulkinnan käytettäville tyypeille ja niiden yhdisteille. Tyypityskonteksti on siis avainasemassa, kun halutaan selvittää komponentin toiminnan semantiikka. Komponentin semanttisten ominaisuuksien tarkistusta käsitellään luvussa 4.

Alityypitys on yhteentoiminnan kannalta tärkeä käsite, joka tulee esiin erityisesti komponenttien välistä korvautuvuutta tarkasteltaessa. Alityypille voidaan antaa joukko-opillinen tulkinta: jos tyyppi σ' on tyypin σ alityyppi (merkitään $\sigma' <: \sigma$), niin silloin tyyppiin σ' kuuluvat objektit muodostavat jonkin tyypin σ osajoukon [CW85]. Täten kaikki tyypin σ' alkiot ovat myös joukon σ alkioita, toisin sanoen joukon σ' alkioiden operationaalinen käyttäytyminen on yhteneväistä joukon σ alkioiden käyttäytymisen kanssa.

Alityyppirelaatiot voidaan jakaa kolmeen ryhmään: joukko-opilliseen sisältyvyyssuhteeseen, arvojoukkojen koersioon ja rajapintojen laajennokseen [Gil01]. Joukko-opilliseen sisältyvyyssuhteeseen perustuvassa alityypityksessä $\sigma' <: \sigma$, jos ja vain jos σ':n arvoalue kuuluu σ:n arvoalueeseen. Arvojoukkojen koersioon perustuvassa alityyppirelaatiossa tyyppi σ' on tyypin σ alityyppi, jos ja vain jos jokainen σ':n arvo voidaan muuntaa arvoksi σ:n arvojoukkoon [Gil01]. Koersiossa käytettävä muunnos on yleensä luonteeltaan semanttista ja koersiomuunnokset annetaan yleensä staattisesti, esimerkiksi ohjelman käännöksen yhteydessä [CW85]. Koersioon perustuvaa alityypityssuhdetta käytetään oliopohjaisissa ohjelmointikielissä, joissa alityyppioliota voidaan laajentaa uusilla attribuuteilla. Rajapintojen laajennokseen perustuvassa alityypityksessä alityypin tulee sisältää vähintääkin kaikki samat palvelut kuin ylityyppinsäkin.

Alityypityksen operationaalinen tulkinta on se, että jos $\sigma' <: \sigma$, niin tyyppiä σ' edustavia objekteja voidaan käyttää missä tahansa kontekstissa, jossa voidaan käyttää σ-tyyppisiä objekteja [CFP99].

Alityypitys on esijärjestysrelaatio eli refleksiivinen ja transitiivinen [CFP99].

Olkoon alityypityksen tulkinta mikä tahansa, vaaditaan siltä ominaisuus, joka takaa tiettyjen ominaisuuksien säilymisen korvaamisen yhteydessä. Toisin sanoen, tyypin ja sen alityypin ulkoisen käyttäytymisen tulee olla yhteneväistä [LW94]. Tämä ominaisuus on formalisoitu määritelmässä 2.3 [LW94].

Määritelmä 2.3 (Alityypityksen turvallisuus)
Olkoon S ja T kaksi sellaista tyyppiä, että $S <: T$. Olkoon lisäksi $x : S$ ja $y : T$ tyyppejä S ja T edustavia objekteja (komponentteja) ja $\phi : \{x_i\}_{i \in I} \rightarrow \{true, false\}$ on totuusarvoinen funktio, joka kuvaa, onko jokin objektin x ulkoinen ominaisuus eli attribuutti i voimassa. Tällöin $\phi(x_i) = \phi(y_i)$ tulee olla voimassa kaikilla $0 < i < n$, missä n on objektin y ulkoisten ominaisuuksien lukumäärä.

Määritelmä 2.3 sitoo alityyppirelaation koskemaan rakenteen lisäksi objektin käyttäytymiseen ja semantiikkaan. Määritelmässä mainitut objektin tarkistettavissa olevat ominaisuudet saadaan objektin spesifikaatiosta, joka sisältää tarvittavat käyttäytymisen ja semantiikan kuvaukset [LW94].

Ohjelmistokomponenttien yhteistoiminta perustuu komponenttityyppien ominaisuuksiin. Yhteentoimivuuden takaamiseksi onkin luotava sopiva tyyppijärjestelmä, joka formalisoi yhteensopivuuden ja korvautuvuuden käsitteet siten, että tarvittavat ominaisuudet ovat voimassa yhteentoiminnan aikana. Korvautuvuus määritellään kaikille rajapinnan osioille: rakenteelliselle kuvaukselle joukossa Σ, tyypityskontekstin struktuurissa $\mathcal{T}$ ja rajapintaprotokollille $\mathcal{P}$. Yhteentoimivuus on sen sijaan määritelty vain rajapintaprotokollille.

Seuraavissa luvuissa tullaan käsittelemään komponenttityyppien yhteensopivuuden ja korvautuvuuden käsitteitä eri tasoilla. Sekä rakenteelliselle kuvaukselle, tyypityskontekstille, että rajapintaprotokollille määritellään tarvittavat relaatiot ja käsitteet, joiden avulla komponenttien välinen yhteentoimivuus voidaan tarkistaa.

3 Syntaktinen yhteentoimivuus

Syntaktinen yhteentoimivuus tarkoittaa ohjelmistokomponenttien korvautuvuutta käyttörajapinnan syntaktisen rakenteen tasolla. Yhteentoimivuuden käsitteen sisällä joudutaan rajoittumaan pelkästään syntaktisten rakenteiden keskinäiseen korvautuvuuteen, koska ilmaisua "*tyypit A ja B ovat syntaktisella tasolla keskenään yhteensopivia*" ei voida mitenkään järkevällä tavalla ilmaista ilman lisäinformaatiota tyyppien välisistä vuorovaikutussuhteista.

Syntaktisella tasolla komponentin todistettavissa olevat ominaisuudet rajoittuvat vain sen rajapinnan määrittelyn syntaktiseen rakenteeseen, esimerkiksi metodien ja muuttujien nimiin ja niiden järjestykseen. Syntaktisella yhteentoimivuudella tarkoitetaan siis komponenttien syntaktisten rakenteiden keskinäistä korvautuvuutta.

Rajapintarakenteiden (*signature*) korvautuvuus voidaan tarkistaa joko termien nimiä tai termejä edustavia syntaktisia rakenteita vertailemalla. Perinteisissä ohjelmointikielissä vertailut tehdään yleensä nimiin perustuen. Rakenteiden vertailussa tarkistetaan, ovatko termeistä muodostuvat puut tai verkot isomorfisia tyyppijärjestelmässä määriteltyjen aksioomien vallitessa.

Rajapinnan syntaktisen rakenteen tarkastelu on tärkeää tarkistettaessa komponenttien välistä korvautuvuutta, koska rajapinnan syntaktinen rakenne määrittelee komponentin teknisen käyttötavan. Jos palvelurajapinnan syntaktinen rakenne muuttuu, niin palvelua käyttävien asiakkaiden tulee sopeutua jollain tavalla tähän muutokseen. Esimerkiksi CORBA-arkkitehtuurissa rajapinnan syntaktinen rakenne voidaan tarkistaa komponentista ajonaikaisesti käyttämällä DII-tekniikkaa (*Dynamic Invocation Interface*) [OHE97].

Rajapinnan syntaktisen rakenteen muutoksiin varautuminen tekee ohjelmistoista kuitenkin monimutkaisempia ja siten herkempiä syntaktisille tai semanttisille virheille. Tämän takia palvelukomponenttia korvattaessa pyritään rajapinnan syntaktinen rakenne säilyttämään entisellään, jos suinkin vain mahdollista. Aina rajapintarakenteen säilyttäminen ei ole mahdollista, esimerkiksi silloin, kun palvelun rajapintaa täytyy laajentaa uudella toiminnallisuudella. Tällöin tarvitaan korvautuvuusrelaatioita, jotka identifioivat kaksi syntaktista rajapintakuvausta, vaikka niiden rakenteet eivät olekaan täysin identtisiä.

Komponenttien keskinäisen korvautuvuuden tarkistaminen syntaktisella tasolla perustuu niiden rajapintojen syntaktisen rakenteen vertailuun. Vertailut jakautuvat kahteen ryhmään: eksaktiin vertailuun ja lievennettyihin (*relaxed*) vertailuihin. Eksaktin vertailun tulos on tosi jos ja vain jos kahden eri komponentin syntaktiset rakenteet ovat identtiset. Lievennetyissä vertailuissa käytetään vertailuehtoja, jotka käyttävät hyväkseen esimerkiksi alityypitysrelaatiota ja niiden sääntöjä. Erityisesti komponentteja päivitettäessä tai etsittäessä sopivan palvelun toteuttavaa komponenttia ovat lievennetyt vertailuehdot käyttökelpoisempia kuin eksakti vertailu.

Tässä luvussa tarkastellaan rajapintarakenteiden korvautuvuutta. Aliluvussa 3.1 tutustutaan rajapinnan rakenteen kuvaamiseen tarvittaviin käsitteisiin ja niihin liittyviin teorioihin. Rajapintarakenteen kuvaaminen perustuu perinteisen tyyppiteorian käsitteisiin. Rajapinnan rakenteen kuvaus muodostetaan yhdistelemällä perustyyppejä tarvittavin tyyppikonstruktorein. Perustyyppien määrittelyyn tutustutaan tarkemmin luvussa 4, jossa käsitellään komponentin semanttisia ominaisuuksia. Tyyppimäärittelyiden jälkeen käsitellään aliluvussa 3.2 rajapintarakenteiden välistä korvautuvuutta. Aliluvussa 3.3 annetaan rajapinnan rakenteelle tulkinta niin kutsuttuna termiautomaattina sekä määritellään termiautomaattien välisille korvautuvuustarkistuksille soveltuvat relaatiot. Tämän jälkeen luvussa 3.4 kuvataan pääpiirteissään rajapintarakenteiden korvautuvuuden tarkistamiseen tarvittavien algoritmien toimintaperiaatteet. Lopuksi luvussa 3.5 kuvaillaan käyttökohteita ja -tapoja, joihin tässä luvussa kuvattuja menetelmiä voidaan soveltaa.

3.1 Rajapintarakenteen tyypin määrittely

Komponenttityypin $C = (\Sigma, \mathcal{P}, \mathcal{T})$ rajapinnan rakenne Σ määritellään soveltaen tyypitettyyn λ-kalkyyliin perustuvia käsitteitä ja menetelmiä [Pie02]. Rajapinnan rakenteen määrittelyyn on käytössä joukko perustyyppejä, joiden tulkinta on määritelty tyypityskontekstissa $\mathcal{T}$. Perustyypeistä voidaan tyyppikonstruktoreja käyttäen koostaa monimutkaisempia tyyppejä, esimerkiksi funktioita tai listoja.

Tyypitetyissä kielissä voidaan erotella toisistaan kolme erilaista tyyppien luokkaa: *perustyypit*, *rakenteiset tyypit* ja *rekursiiviset tyypit*. Tyypitetyn kielen perustyyppejä voivat olla esimerkiksi perinteisistä ohjelmointikielistä tutut $Bool$-, Int-, $Real$-tyypit.

Edellä mainittujen tyyppien lisäksi on käytössä myös *tyyppimuuttujat*, jotka toimivat nimettyinä paikanpitiminä mille tahansa tyypeille. Tyyppimuuttujaan voidaan siis tarpeen mukaan sijoittaa mikä tahansa muu tyyppi. Tyyppimuuttujaa merkitään yleisesti kreikkalaisilla kirjaimilla $(\alpha, \beta, \ldots)$. C-kielisissä esimerkeissä tyyppimuuttujan esittelemiseen käytetään tässä tutkielmassa avainsanaa “Any”. Tyyppimuuttujia ei tulla kuitenkaan komponenttien rajapintojen kuvauksissa käyttämään.

Tyypitetyn kielen syntaksi ja semantiikka määritellään päättelysäännöillä, joiden avulla kielen termeihin voidaan sitoa yksikäsitteinen tyyppi. Päättelysäännöt muodostavat kielen *tyyppijärjestelmän* [CFP99]. Tyyppijärjestelmän avulla voidaan tarkistaa, onko termi oikein tyypitetty ja ovatko kaksi tyypitetyn kielen termiä samaa tyyppiä. Termit ovat samaa tyyppiä, jos ne tyyppijärjestelmän sääntöjä käyttämällä redusoituvat “samaan” normaalimuotoon [CFP99]. Termi on normaalimuodossaan, jos ja vain jos termiin ei voida soveltaa enää yhtään redusointisääntöä [DJ90]. Normaalimuotojen ekvivalenssin ei tarvitse välttämättä olla identiteettikuvaus, vaan kuten tullaan myöhemmin huomaamaan, voidaan normaalimuotoisten termien välisen identiteettikuvauksen sijaan käyttää joustavampia termien välisiä relaatioita.

3.1.1 Perustyypit

Perustyypit ovat tyyppijärjestelmän jakamattomia primitiivejä. Perustyypit voidaan määritellä tyypityskontekstissa esimerkiksi eksplisiittisenä säännöstönä, joka luettelee tyyppien nimet ja niiden ominaisuudet (“Int on vähintään 16-bittinen kokonaisluku”). Tällaista lähestymistapaa suositaan useissa proseduraalisissa kielissä, kuten C-kielessä. Myös tyyppien väliset ekvivalenssit ja alityypitykset annetaan tässä tapauksessa yleensä eksplisiittisinä sääntöinä. Esimerkiksi C++ -ohjelmointikielessä *class Child : public Parent ;* määrittelee eksplisiittisesti, että luokka *Child* on luokan *Parent* aliluokka.

Toinen tapa perustyyppien määrittelyyn on käyttää joko abstrakteja tietotyyppejä tai ontologioita [EM85, UG96]. Abstraktien tietotyyppien teoriaa soveltamalla voidaan tyyppien välinen ekvivalenssi todistaa matemaattisesti. Abstrakteilla tietotyypeillä ei voida määritellä muita kuin puhtaasti matemaattisia tyyppejä, kuten listoja tai puita. Tämän vuoksi ne eivät sovellu välttämättä reaalimaailman käsitteiden kuvaamiseen. Abstrakteihin tietotyyppeihin tutustutaan luvussa 4.2.

Ontologioiden avulla voidaan kuvata jaettuja käsitemalleja, jotka määrittelevät käsitteille tulkintoja sekä käsitteiden välisiä suhteita [UG96]. Ontologioilla voidaan määritellä käsitteitä miltä tahansa sovellusalueelta eli se ei ole abstraktien tietotyyppien tavoin rajoittunut pelkästään matemaattisten käsitteiden määrityksiin. Ontologioihin liittyy aina jossain vaiheessa tulkintaa tai jollain tasolla tehty yhteinen päätös peruskäsitteistä, joten ontologioiden avulla määritellyt käsitteet eivät ole matemaattisesti yhtä “todistettavissa” kuin abstraktit tietotyypit. Ontologiat perustuvat kuiten-

kin matemaattisen logiikan säännöille, joten käsitteiden välisiä ekvivalenssi- ja alityypityssuhteita voidaan johtaa logiikan sääntöjä noudattamalla. Ontologioita käsitellään luvussa 4.3.

3.1.2 Rakenteiset tyypit

Perustyyppejä voidaan yhdistää rakenteisiksi tyypeiksi *tyyppikonstruktoreilla*, joita on kolme: funktiotyyppi, karteesinen tulo ja tyyppien erillinen yhdiste [CW85]. Tyyppien karteesiset tulot ja erilliset yhdisteet voivat olla joko nimeämättömiä tai nimettyjä. Nimettyä karteesista tuloa kutsutaan yleisesti tietueeksi (*record*) ja vastaavasti nimettyä yhdistettä variantiksi [CW85, Pie02].

Rakenteisten tyyppien tulkinta ja tyypitys määritellään evaluointi- ja tyypityssääntöjä käyttämällä. Evaluointisäännöt, jotka ovat muotoa $t \to t'$, määrittelevät sääntöjä, joiden mukaan termi t muuntuu sääntöä soveltamalla termiksi t. Jos termiin ei voida soveltaa mitään evaluointisääntöä, niin termi on normaalimuodossaan. Normaalimuotoista termiä kutsutaan myös arvoksi [Pie02]. Tyypityssäännöt määrittelevät, mikä on tietyn muotoisen termin tyyppi. Tyypityssäännöt ovat muotoa $\mathcal{T} \vdash t : T$, joka tarkoittaa sitä, että tyypityskonteksissa $\mathcal{T}$ termi t on tyyppiä T. Tyypityskonteksti tulkitaan tämän luvun tarkasteluissa joukoksi tyypityksiä $x : T$. Yhteentoimivuustarkaskeluissa ollaan kiinnostuneita erityisesti tyypityskontekstin määrittelemistä tyypityssäännöistä, ei niinkään evaluointisäännöistä.

Tyyppi $A \to B$ kuvaa funktiota, joka ottaa syötteekseen tyyppiä A olevan arvon ja palauttaa tyyppiä B edustavan arvon. Funktiotyypin evaluointi- ja tyypityssäännöissä käytetään hyväksi λ-kalkyylin abstraktion (*abstraction*) ja sovelluksen (*application*) käsitteitä. Abstraktio $\lambda x : T_1.t_2$ on λ-kalkyylin termi, joka kuvaa jonkin arvon abstrahoimista lausekkeesta t_2 muuttujaksi x [Pie02]. Abstrahointi vastaa siis ohjelmointikielen funktion tai metodin käsitettä, jossa funktiorungossa käytettävä arvo on abstrahoitu funktion syöteparametriksi x. Esimerkiksi C-kielen funktio *int plusOne(int i) { return i+1; }* voitaisiin ilmaista λ-kalkyylin abstraktiona $\lambda x : Int.(x + 1)$.

Funktiotyypin evaluointi- ja tyypityssäännöt on annettu taulukossa 1. Tämän opinnäytetyön kannalta evaluointisääntöjen esittely on käytännössä turhaa, koska komponenttien yhteentoimivuuden tarkastuksessa ollaan kiinnostuneita termien rakenteellisista kuvauksista (tyypitys) eikä niinkään termien tulkinnan määrittelyistä (evaluointi). Evaluointisäännöt ovat kuitenkin tärkeä osa perinteistä tyyppiteoriaa, joten niitä esitellään jatkossa tyypityssääntöjen rinnalla.

Evaluointisäännöt		Tyypityssäännöt	
$\dfrac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2}$	(E-APP1)	$\dfrac{x : T \in \mathcal{T}}{\mathcal{T} \vdash x : T}$	(T-VAR)
$\dfrac{t_2 \to t_2'}{v_1\ t_2 \to v_1\ t_2'}$	(E-APP2)	$\dfrac{\mathcal{T}, x : T_1 \vdash t_2 : T_2}{\mathcal{T} \vdash \lambda x : T_1.t_2 : T_1 \to T_2}$	(T-ABS)
$(\lambda x : T_{11}.t_{12})v_2 \to [x \to v_2]t_{12}$	(E-APPABS)	$\dfrac{\mathcal{T} \vdash t_1 : T_{11} \to T_{12} \quad \mathcal{T} \vdash t_2 : T_{11}}{\mathcal{T} \vdash t_1\ t_2 : T_{12}}$	(T-APP)

Taulukko 1: Funktiotyypin evaluointi- ja tyypityssäännöt

Funktiotyypin evaluointisäännöt *E-APP1* ja *E-APP2* määrittelevät, kuinka abstraktion muuttujan tai rungon evaluointi vaikuttavat abstraktion tulkintaan. Sääntö *E-APPABS* määrittelee tulkinnan abstraktion sovellukselle: kun abstraktiota sovelletaan, niin kaikki abstraktion rungossa t_{12} esiin-

tyvät muuttujat x korvataan arvolla v_2. Tämä vastaa intuitiota ohjelmointikielten funktion tai metodin käytöstä.

Sovellus $t_1\ t_2$ kuvaa perinteisen ohjelmointikielen kaltaista funktiokutsua, missä t_1 edustaa funktion runkoa ja t_2 siihen sovellettavaa termiä. Jos oletetaan, että on määritelty kokonaislukutyyppi Int, sille yhteenlaskuoperaatio $+$ ja t on abstraktio $\lambda x : Int.(x + 1)$, niin sovellus $t\ 2$ evaluoituu, ensin sääntöä *E-APPABS* ja sitten kokonaislukujen yhteenlaskua soveltaen, seuraavasti: $t\ 2 \rightarrow (2 + 1) \rightarrow 3$.

Tyypityssääntö *T-VAR* ilmaisee, että jos tyypityskontekstissa $\mathcal{T}$ on määritelty tyypitys $x : T$, niin aina voidaan päätellä, että $\mathcal{T} \vdash x : T$. Sääntö *T-ABS* määrittelee sen, että jos abstraktion runko on tyyppiä T_2 ja abstraktion parametri on tyyppiä T_1, niin abstraktion tyyppi on $T_1 \rightarrow T_2$. Tyypityssäännön *T-APP* mukaan sovelluksen tyyppi tulkitaan abstraktiotyypin $T_1 \rightarrow T_2$ paluuarvon tyypiksi T_2.

Jos jonkin metodin f syöte on tyyppiä a ja paluuarvo tyyppiä b, niin tällöin f on tyyppiä $a \rightarrow b$, merkitään $f : (a \rightarrow b)$. Sulkeita käytetään selventämään tyypitysten kirjoittamista.

Karteesinen tulo $(A \times B)$ määrittelee tyyppiparin, jonka mahdolliset arvot ovat arvojen tulojoukosta $A \times B$. Karteesinen tulo voidaan tarpeen mukaan tulkita joko järjestetyksi tai järjestämättömäksi. Tulotyyppi määritellään käyttämällä notaatiota $\{a, b\}$, missä a ja b ovat joitain termejä. Karteesisen tulotyypin (järjestetyn parin) evaluointi- ja tyypityssäännöt on annettu taulukossa 2 [Pie02]. Evaluointisäännöissä käytetään merkintää t kuvaamaan jotakin termiä ja merkintää v kuvaamaan jotakin arvoa, siten ollen termiä, joka ei enää redusoidu yksinkertaisempaan muotoon.

Evaluointisäännöt		Tyypityssäännöt	
$\{v_1, v_2\}.1 \rightarrow v_1$	(E-PAIRBETA1)		
$\{v_1, v_2\}.2 \rightarrow v_2$	(E-PAIRBETA2)		
$\dfrac{t_1 \rightarrow t_1'}{t_1.1 \rightarrow t_1'.1}$	(E-PROJ1)	$\dfrac{\mathcal{T} \vdash t_1 : T_1 \quad \mathcal{T} \vdash t_2 : T_2}{\mathcal{T} \vdash \{t_1, t_2\} : T_1 \times T_2}$	(T-PAIR)
$\dfrac{t_1 \rightarrow t_1'}{t_1.2 \rightarrow t_1'.2}$	(E-PROJ2)	$\dfrac{\mathcal{T} \vdash t : T_{11} \times T_{12}}{\mathcal{T} \vdash t_1.1 : T_{11}}$	(T-PROJ1)
$\dfrac{t_1 \rightarrow t_1'}{\{t_1, t_2\} \rightarrow \{t_1', t_2\}}$	(E-PAIR1)	$\dfrac{\mathcal{T} \vdash t : T_{11} \times T_{12}}{\mathcal{T} \vdash t_1.2 : T_{12}}$	(T-PROJ2)
$\dfrac{t_2 \rightarrow t_2'}{\{t_1, t_2\} \rightarrow \{t_1, t_2'\}}$	(E-PAIR2)		

Taulukko 2: Parityypin evaluointi- ja tyypityssäännöt

Evaluointisäännöt *E-PAIRBETA1* ja *E-PAIRBETA2* määrittelevät parityypille kaksi operaatiota, joilla parista voidaan eristää joko sen ensimmäinen tai toinen alkio. Tyypityssäännöt *T-PAIR1* ja *T-PAIR2* määrittelevät, mitä tyyppiä näiden operaatioiden tulostermit edustavat. Evaluointisäännöt *E-PROJ* ja *E-PAIR* kuvaavat kuinka parityyppi käyttäytyy siinä tapauksessa, kun jompaa kumpaa sen sisältämistä termeistä evaluoidaan.

Parityyppi voidaan yleistää monikoksi, joka voi myös olla järjestetty tai järjestämätön. Monikko voi sisältää nolla tai useampia alkioita. Monikkoon liittyvät evaluointi- ja tyypityssäännöt ovat analogisia parityypin vastaavien sääntöjen kanssa ja ne on annettu taulukossa 3.

Monikkotyypin määrittelyissä käytetään merkintöjä $\{t_i^{i\in 1..n}\}$ ja $\{T_i^{i\in 1..n}\}$ kuvaamaan monikkomuotoisia termejä ja tyyppejä. Merkinnällä $\{t_i^{i\in 1..n}\}$ tarkoitetaan monikkoa, joka sisältää n termiä. Säännön *T-TUPLE* mukaan tämän termin tyyppi on $\{T_i^{i\in 1..n}\}$, jos kaikille i on voimassa $\mathcal{T} \vdash t_i : T_i$ eli termi t_i on tyyppiä T_i tyypityskontekstissa $\mathcal{T}$.

Evaluointisäännöt

$$\{v_i^{i\in 1..n}\}.j \to v_j \qquad \text{(E-PROJTUPLE)}$$

$$\frac{t_1 \to t_1'}{t_1.i \to t_1'.i} \qquad \text{(E-PROJ)}$$

$$\frac{t_j \to t_j'}{\{v_i^{i\in 1..j-1}, t_j, t_k^{k\in j+1..n}\} \to \{v_i^{i\in 1..j-1}, t_j', t_k^{k\in j+1..n}\}} \qquad \text{(E-TUPLE)}$$

Tyypityssäännöt

$$\frac{\forall i \quad \mathcal{T} \vdash t_i : T_i}{\mathcal{T} \vdash \{t_i^{i\in i..n}\} : \{T_i^{i\in 1..n}\}} \qquad \text{(T-TUPLE)}$$

$$\frac{\mathcal{T} \vdash t_1 : \{T_i^{i\in 1..n}\}}{\mathcal{T} \vdash t_1.j : T_j} \qquad \text{(T-PROJ)}$$

Taulukko 3: Monikkotyypin evaluointi- ja tyypityssäännöt

Jos jokin monikko s on tyyppiä $a : A \times b : B \times c : C$, niin merkitään $s : (a : A \times b : B \times c : C)$ tai $s : \{a : A, b : B, c : C\}$.

Monikot (ja parit) voivat olla myös nimettyjä, jolloin monikon termeihin voidaan viitata ennalta määritellyillä nimillä [Pie02]. Nimetystä karteesisesta tulosta voidaan hakea arvo nimen perusteella seuraavan säännön avulla: jos $\{l_i : v_i^{i\in 1...n}\}$ on nimetty karteesinen tulo, jossa on n alkiota ja l_j on j:nnen alkion nimi, niin $\{l_i : v_i^{i\in 1...n}\}.l_j$ evaluoituu tyypiksi v_j. Jos esimerkiksi $A = \{a : int, b : long\}$, niin termi $A.a$ on tyyppiä int ja termi $A.b$ on tyyppiä $long$ [Pie02, sivu 129]. Tähän sinänsä erittäin käytännölliseen monikkotyypin varianttiin ei tässä tutkielmassa tutustuta tarkemmin. Esimerkiksi C-kielen $struct$ määreet ovat tällaisia nimettyjä monikkoja.

Erillinen yhdiste $(A + B)$ edustaa yhdistelmätyyppiä, joko voi olla joko tyyppiä A tai tyyppiä B. Ohjelmointikielissä summatyyppiä tarvitaan edustamaan tyyppimääritelmiä, joissa lopullinen tyypitys voidaan valita tarpeen mukaan. Jos esimerkiksi puun *Tree* solmu voi olla joko kokonaisluku tai puun juuri, niin tälläinen solmu on tyyppiä *Node : (Int + Tree)*. Summatyypin lisääminen tyyppijärjestelmään tuo mukanaan teoreettisia ongelmia, mutta onneksi summatyyppiä ei yleensä tarvita komponenttien rajapintakuvauksissa. Tämän vuoksi ei summatyypin evaluointi- tai tyypityssääntöjä käsitellä tässä opinnäytetyössä.

Taulukossa 4 on eriteltynä edellä esitellyt tyyppiluokat sekä niitä vastaavat C-kielen rakenteet.

Komponentin rajapintarakenteen kuvaamiseen käytetään perustyyppejä, tulotyyppiä ja funktio-

Tyyppiluokka	λ-kalkyyli	Pseudo-C
Perustyypit	esim. a : bool, b : int	bool a; int b;
Tyyppimuuttujat	α	Any a;
(Nimetty) karteesinen tulo	(a : bool × b : int)	struct{ int a; int b; }
(Nimetty) erillinen yhdiste	(a: bool + b : int)	union{ bool a; int b; }
Funktio	f : (int → bool)	bool f(int);

Taulukko 4: Tyyppien perusluokat ja esimerkit

tyyppiä. Komponenttien sisältämät termit ovat siis muotoa $t = b \in \mathcal{B} \mid t \times t \mid t \rightarrow t$, missä $\mathcal{B}$ on tyypityskontekstissa $\mathcal{T}$ määriteltyjen perustyyppien joukko. Komponentti itsessään on sisältämiensä termien karteesinen tulo. Moniparametristen funktioiden syöteparametrit ilmaistaan yhtenä karteesisena tulona. Täten esimerkiksi C-kielen funktio $int\ max(int\ a, int\ b)$ on tyyppiä $max : (a : int \times b : int) \rightarrow int$.

Komponentti sisältää joko yhden tai useamman termin, joka voi olla mikä tahansa edellä esitellyistä tyypeistä paitsi summatyyppi. Taulukossa 5 on pseudokielinen esimerkki erään komponentin $Example1$ määrittelystä. Kyseisen komponentin tyyppi on muotoa $Example1 : ((a : int) \times (b : double) \times (fun1 : (void \rightarrow int)) \times (fun2 : ((x : int) \times (y : int)) \rightarrow int))$.

```
component Example1 {
  int a;
  double b;
  int fun1();
  int fun2(int x,int y);
};
```

Taulukko 5: Esimerkki komponenttityypistä

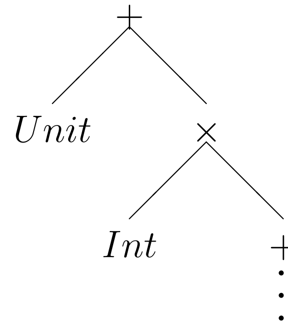
3.1.3 Rekursiiviset tyypit

Rekursiivisia tyyppejä käytetään ohjelmointikielissä määrittelemään potentiaalisesti äärettömiä tyyppejä, kuten listoja tai puita. Rekursiiviset tyypit määritellään rekursio-operaattorilla μ, jonka yleinen muoto on $A = \mu t.\alpha$. Kyseinen operaattori määrittelee rekursiivisen tyypin A, jossa rekursiomuuttuja t esiintyy termissä α. Esimerkiksi $IntList = \mu t.Unit + (Int \times t)$ määrittelee äärettömän Int-listan [AC93]. Rekursiiviset tyypit, kuten muutkin rakenteiset tyypit esitetään yleensä puiden tai verkkojen avulla. Kuvassa 1 on kuvattuna tyyppiä $IntList$ edustava puu.

Rekursiiviset tyypit tulevat vastaan myös komponenttien rajapintarakenteiden kuvauksissa. Rekursiivista käyttäytymistä rajapintarakenteissa esiintyy silloin, kun rajapintakuvaus viittaa itseensä tai kun kaksi rajapintakuvausta viittaavaat toisiinsa ristikkäin. Ensimmäisessä tapauksessa rajapinta on itserekursiivinen kun taas jälkimmäisessä tapauksessa rajapinnat ovat keskenään rekursiivisia.

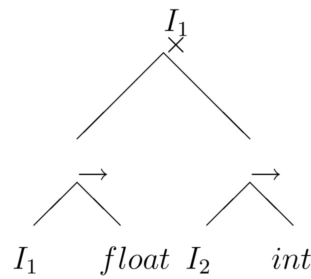
Kuvassa 2 on kaksi rajapintarakenteen pseudokielistä määrittelyä, I_1 ja I_2, jotka ovat sekä itserekursiivisia että keskenään rekursiivisia. Kuvausten alla on rajapintojen rakenteet puumuodossa. Funktiotyyppinen termi on kaksihaarainen alipuu, jonka vasen haara edustaa funktion syötettä ja oikea haara paluuarvoa.

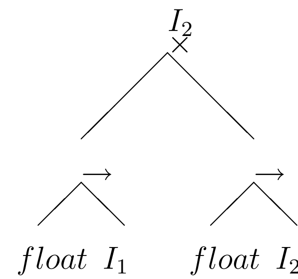
$$IntList = \mu t(Unit + Int \times t)$$



Kuva 1: Tyyppiä $IntList$ edustava ääretön puu

$$\begin{array}{l} component\ I_1 \{ \\ float\ m_1(I_1\ a); \\ int\ m_2(I_2\ a); \\ \} \end{array} \qquad \begin{array}{l} component\ I_2 \{ \\ I_1\ m_3(float\ a); \\ I_2\ m_4(float\ a); \\ \} \end{array}$$





Kuva 2: Rekursiivisia rajapintarakenteita

Rekursiivinen tyyppi $\mu t.\alpha$ voidaan laventaa ($unfold$) siten, että rekursiomuuttujan t paikalle sijoitetaan rekursiivinen tyyppi $\mu t.\alpha$. Lavennusoperaatiolle on myös "vastaoperaatio" eli supistus ($fold$), jonka avulla rekursion "häntää" voidaan supistaa. Operaatioden $fold$ ja $unfold$ keskinäinen suhde on määritelty taulukossa 6 [AC93]. Taulukossa 6 merkintä, joka on muotoa $M : \alpha$ kuvaa rekursiivista termiä, jonka tyyppi on α.

supistus **fold**	$M : [\mu t.\alpha/t]\alpha \Rightarrow (fold_{\mu t.\alpha} M) : \mu t.\alpha$
lavennus **unfold**	$M : \mu t.\alpha \Rightarrow (unfold_{\mu t.\alpha} M) : [\mu t.\alpha/t]\alpha$

Taulukko 6: Operaatioden fold ja unfold määritelmät

Taulukon 6 määritelmien mukaan rekursiivisen termin M supistamisen tuloksena saadaan tyyppiä $\mu t.\alpha$ oleva termi. Lavennussäännön mukaan rekursiivisen termin $M : \mu t.\alpha$ laventaminen palauttaa termin, jonka tyyppi on $[\mu t.\alpha/t]\alpha$. Yllä esitellyn $IntList$-tyypin yksinkertainen lavennus (suoritettu yksi $unfold$-operaatio) on kuvattu taulukossa 7. Kuvauksessa on käytetty lyhennysmerkintää a kuvaamaan $IntList$-tyypin lauseketta $\mu t.Unit + (Int \times t)$. Vastaavasti $unfold[IntList]$-operaation tulostyypin $IntList^1$ supistaminen $fold$-operaatiolla on kuvattu taulukossa 8.

$unfold[IntList] =$ $[\mu t.a/t]Unit + (Int \times (\mu t.Unit + (Int \times t))) =$ $[IntList/t]Unit + (Int \times (\mu t.Unit + (Int \times t))) =$ $Unit + (Int \times (Unit + (Int \times IntList))) = IntList^1$

Taulukko 7: $IntList$-termin yksinkertainen lavennus

$fold[IntList^1] =$ $fold[Unit + (Int \times (Unit + (Int \times IntList)))] =$ $fold[Unit + (Int \times (\mu t.Unit + (Int \times t)))] =$ $\mu t.Unit + (Int \times t) = IntList.$

Taulukko 8: $IntList^1$-termin supistaminen

Kaksi rekursiivista tyyppiä ovat ekvivalentit, jos niiden äärettömät lavennukset ovat ekvivalentteja [AC93]. Äärettömyys on kuitenkin hankala käsite, koska äärettömiä rakenteita ei voida suoraan vertailla toisiinsa. Tämän takia rekursiivisten tyyppien vertailussa täytyykin käyttää sopivia äärellisiä approksimointimenetelmiä. Sopivien äärellisten approksimointisääntöjen muodostaminen ei kuitenkaan ole täysin triviaalia.

Ääretön rekursiivinen tyyppi α voidaan karkeasti approksimoida siten, että lavennusoperaatioiden tulosta vastaava puu (verkko) katkaistaan joltain syvyydeltä n. Tällöin saadaan tulokseksi rekursiivisen tyypin α äärellinen lavennus α^n [AC93]. Äärellisten lavennusten avulla voidaan rekursiivisten tyyppien ekvivalenssille antaa tulkinta muodossa $\alpha \approx \beta \; jos \; \forall n : \alpha^n \approx \beta^n$.

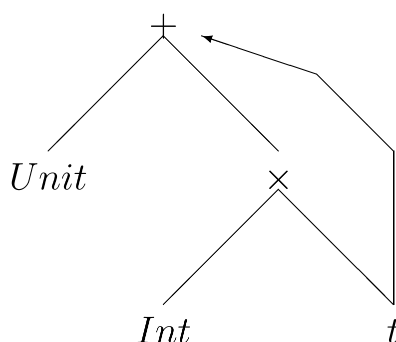
Toinen approksimointitapa on kuvata äärellinen rekursiivinen tyyppi verkkona. Rekursiivinen tyyppi $T = \mu t.\alpha$ kuvataan verkkona, jossa jokaisesta α:n rekursiomuuttujasta t on yksisuuntainen kaari sitä vastaavaan rekursio-operaattoriin μt (*back-pointer*) [AC93]. Tämä erityisesti ohjelmointikielissä varsin luonnollinen lähestymistapa tuottaa vain approksimaation äärettömästä rekursiivisesta tyypistä: joissain tapauksissa rekursion syvyys voi myös määritellä solmujen ja sitä vastaavien tyyppien tulkinnan. Esimerkiksi järjestetyssä listassa ei voida olettaa, että kaikki sen alkiot ovat saman arvoisia. Kuvassa 3 on kuvattuna rekursiivinen tyyppi $IntList$ käyttäen takaisinosoitusta.

Rekursiivisilla tyypeillä, äärellisillä deterministisillä automaateilla ja säännöllisillä kielillä on suora yhteys toisiinsa, koska rekursiivisten tyyppien ja äärellisten determinististen automaattien avulla voidaan määritellä kaikki säännölliset kielet [HMU01]. Rekursiivista tyyppiä T kuvaava verkko voidaankin tulkita äärelliseksi automaatiksi M_T. Automaatin M_T avulla voidaan tarkistaa, kuuluuko tyyppiä σ esittävä lause automaatin M_T hyväksymään kieleen. Jos $\sigma \in L(M_T)$, niin σ on eräs rekursiivisen tyypin T instanssi. Lisäksi automaattien M_S ja M_T väliltä voidaan tarkistaa mielenkiintoisia ominaisuuksia. Jos $L(M_T) \subseteq L(M_S)$ ja $L(M_S) \subseteq L(M_T)$ niin silloin T ja S edustavat ekvivalentteja rekursiivisia tyyppejä. Rekursiivisten tyyppien yhteydessä automaatteja kutsutaan yleensä termiautomaatiksi (*term automata*). Rekursiivisten tyyppien tulkintaan äärellisinä termiautomaatteina tutustutaan tarkemmin kappaleessa 3.3.

3.2 Rajapintarakenteiden korvautuvuus

Rajapintarakenteiden välinen korvautuvuus voidaan todistaa siten, että kahden rajapinnan rakenteen välille löydetään sopiva korvautuvuusrelaatio. Korvautuvuusrelaatio on kuvaus yhdeltä raja-

$$Intlist = \mu t(Unit + Int \times t)$$



Kuva 3: Äärettömän rekursiivisen tyypint $IntList$ äärellinen approksimaatio käyttäen takaisinosoitusta

pinnan rakenteelta toiselle, tavallaan "käännösfunktio", jonka tehtävänä on ilmoittaa, onko rajapintojen rakenteelliset struktuurit tarvittavalla tasolla samankaltaisia. Kuten jo aikaisemmissa luvuissa oh vihjattu, voidaan korvautuvuusrelaatiot jakaa kahteen pääryhmään: eksakteihin ja löyhennettyihin ehtoihin. Tässä aliluvussa tutustutaan kumpaankin näistä pääryhmistä. Eksakti korvautuvuusrelaatio palautuu rakenteellisen isomorfismin tarkistukseen, kun taas löyhennetty korvautuvuusrelaatio palautuu niin kutsuttuun rakenteelliseen alityypitykseen. Rakenteellisen isomorfismin käsitteeseen ja siihen liittyviin teorioihin tutustutaan luvussa 3.2.1. Rakenteellista alityypitystä käsitellään luvussa 3.2.2.

3.2.1 Rakenteellinen isomorfismi

Tyyppien τ ja τ' ekvivalenssilla tarkoitetaan niitä edustavien termien muodostamien puiden (verkkojen) välistä isomorfismia. Termien nimiä ei oteta huomioon, vaan tarkistuksessa keskitytään pelkästään puiden (verkkojen) rakenteeseen. Rakenteiden välinen isomorfismi tarkoittaa, että kahden puun (tai verkon) välillä on olemassa jokin bijektiivinen kuvaus. Termien välisen rakenteellisen isomorfismin määritelmä on annettu määritelmässä 3.1.

Määritelmä 3.1 (Rakenteellinen isomorfismi)
Kaksi termiä ovat isomorfisia, jos niiden välillä on jokin bijektiivinen (muunto)kuvaus siten, että tyyppien rakenteet voidaan muuntaa toisikseen ilman informaation häviämistä [FCB02].

Määritelmässä 3.1 informaation häviämättömyydellä tarkoitetaan sitä, että termeissä käytettävät primitiiviarvot, kuten esimerkiksi kokonaisluvut, kuvautuvat ekvivalenteiksi arvoiksi. Määritelmässä 3.1 mainittu muuntokuvaus voi olla luonteeltaan myös semanttinen, jolloin informaation häviämättömyys on subjektiivista ja primitiivitermien välinen muuntokuvaus on eksplisiittisesti annettu.

Yksinkertaisin isomorfismi on termien välinen identiteettikuvaus, $f : x \rightarrow x$. Kuten jo tämän luvun alustuksessa todettiin, identiteettikuvaukseen perustuva isomorfismi ei käytännössä ole kovinkaan mielenkiintoinen tai käyttökelpoinen korvautuvuusrelaatio. Isomorfismien käyttö termien välisissä korvautuvuustarkistuksissa muuttuu käytännölliseksi, kun isomorfismi sidotaan tyyppijärjestelmän sääntöihin. Käytettävän isomorfismin ominaisuudet määräytyvät mm. sen mukaan,

mitä tyyppijärjestelmää tai ohjelmointikieltä käytetään [CPR04]. Lisäksi valintaa rajaavat teoreettiset ja käytännölliset rajoitteet. Jotkin käytännön kannalta mielenkiintoisista isomorfismeista ovat ratkeamattomia, kuten esimerkiksi erimuotoisten tyyppien alityypitys (*non-structural subtyping*) [KR03b].

Komponenttien syntaktisten rakenteiden sovituksissa joudutaan käyttämään usein menetelmiä, jotka perustuvat johonkin tyyppiteoriaan ja siihen liittyvään termien redusointijärjestelmään. Redusointijärjestelmän avulla termien muodostamat rakenteet voidaan saattaa normaalimuotoon, jos sellainen on olemassa. Tämän jälkeen termien normaalimuotoisia rakenteita (yleensä puu- tai verkkomuodossa) voidaan verrata keskenään sopivaa menetelmää käyttäen. Sopivan redusointijärjestelmän olemassaolo ja soveltuvuus eivät kuitenkaan ole itsestäänselvyyksiä, vaan niiden eksistenssi ja ominaisuudet, tärkeimpänä laskettavuus, riippuvat suoraan käytettävästä tyyppijärjestelmästä.

Tyyppiteorioiden perusteet ja ominaisuudet voidaan palauttaa matemaattiseen kategoriateoriaan [Pie02]. Kategoriateorian avulla voidaan formaalilla tavalla tutkia erilaisten matemaattisten struktuureiden abstrakteja ominaisuuksia. Matemaattisia struktuureja, jotka muodostavat omia kategorioitaan ovat esimerkiksi joukot, relaatiot verkot, automaatit ja tyypit [Gog91].

Ohjelmointikielet ja kategoriateoria käsittelevät molemmat abstrakteja objekteja ja morfismeja niiden välillä. Molemmat ovat "teorioita funktioista" [AL91]. Käytettävän tyyppiteorian laskettavuus ja todistusvoima perustuu nimenomaan siihen, minkälaiseen kategoriaan se kuuluu. Tyypitetyn λ-kalkyylin muoto, jossa rajoitutaan vain tulo- ja funktiotyyppeihin, kuuluu niin kutsuttujen suljettujen karteesisten kategorioiden perheeseen, joka puolestaan takaa sen, että kyseisessä tyyppiteoriassa on olemassa äärellisesti aksiomatisoituva redusointijärjestelmä [BCL92].

Taulukossa 9 luetellut aksioomat ja tyypeille määritelty ekvivalenssi $\equiv$ määrittelevät tyyppiteorian Th, joka kuuluu niin kutsuttuihin suljettuihin karteesisiin kategorioihin [BCL92]. Tyyppijärjestelmää on rajoitettu siten, että se sisältää vain tulo- ja funktiotyypin. Tyyppiteoria, joka sisältää lisäksi summatyypin $(\tau + \tau')$ sekä nollatyypin $(\mathbf{0})$, muodostaa bikarteesisen kategorian, joka ei ole äärellisesti aksiomatisoitavissa [FCB02]. Tämä johtuu siitä, että kyseisen kategorian ominaisuudet voidaan rinnastaa luonnollisten lukujen ominaisuuksiin, joka on myös bikarteesinen kategoria. Luonnollisten lukujen ekvivalenssiteoria ei ole äärellisesti aksiomatisoitavissa. Vaikka summatyypit olisivatkin erittäin käyttökelpoisia komponenttien yhteentoimivuuden tarkistamisessa, ei niitä käsitellä tämän opinnäytteen yhteydessä. Itse asiassa, vastausta siihen, onko tyyppijärjestelmän $\tau = \mathcal{T}|\mathbf{1}|\tau_1 \times \tau_2|\tau_1 \to \tau_2|\mathbf{0}|\tau_1 + \tau_2$ ekvivalenssirelaatio ratkeava, ei vielä tiedetä [FCB02].

Tyyppiteoria **Th** koostuu tyyppien ekvivalenssiteoriasta $T - EQ$ sekä seuraavista aksioomista			
(1)	$A \times \mathbf{1}$	$= A$	Vakiotyypin tulo
(2)	$A \times B$	$= B \times A$	Tulon kommutatiivisuus
(3)	$A \times (B \times C)$	$= (A \times B) \times C$	Tulon assosiatiivisuus
(4)	$(A \times B) \to C$	$= A \to (B \to C)$	Curry
(5)	$A \to (B \times C)$	$= (A \to B) \times (A \to C)$	Funktion distributiivisuus
(6)	$A \to \mathbf{1}$	$= \mathbf{1}$	Vakiotyyppi funktion paluuarvona
(7)	$\mathbf{1} \to A$	$= A$	Vakiotyyppi funktion syötteenä

Taulukko 9: Tyyppiteorian **Th** aksioomat

Tyyppien ekvivalenssiteorian intuitio on, että jos kaksi tyyppiä ovat ekvivalentteja, niin silloin näitä tyyppejä edustavat mielivaltaiset termit ovat myös ekvivalentteja [Pie02, sivu 441]. Tyyppien ekvivalenssiteoria on annettu määritelmässä 3.2. Määritelmässä 3.2 käytetään merkintää $\equiv$ merkitsemään tyyppien välistä ekvivalenssia.

Määritelmä 3.2 (Tyyppien ekvivalenssiteoria T-EQ)

$$\frac{\mathcal{T} \vdash t : S \quad S \equiv T}{\mathcal{T} \vdash t : T} \quad \textit{(T-EQ)}$$

Rakenteisten, rekursiivisten tyyppien välinen ekvivalenssi $\equiv$ voidaan määritellä usealla eri tavalla. Ekvilessimääritelmiä on muodostettu käyttäen esimerkiksi monotonisten funktioiden kiintopisteteoriaa tai äärellisten automaattien (bi)simulointiteoriaa [PZ00]. Tässä opinnäytetyössä rakenteisten tyyppien ekvivalenssi (isomorfismi) määritellään niin kutsuttujen termiautomaattien väliseksi bisimuloinniksi. Tämän ekvivalenssin muodostamiseen ja määrittelyyn tutustutaan aliluvussa 3.3.

Tyyppien välinen ekvivalenssi $\equiv$ käyttää hyväkseen perustyyppien välistä ekvivalenssia. Perustyyppien välinen ekvivalenssi $\equiv_b$ on määritelty tyypityskontekstissa $\mathcal{T}$ ja on siten joko abstraktien tietotyyppien tai ontologisten käsitteiden välinen ekvivalenssirelaatio. Perustyyppien välisen ekvivalenssin määrittelytapoihin tutustutaan luvuissa 4.2 ja 4.3.

Tyyppiteorian Th säännöt muodostavat käytännön kannalta mielenkiintoisen ja rajoitetun tyyppijärjestelmän, joissa tyyppien välinen isomorfismi voidaan todistettavasti ratkaista [BCL92]. Käytettävän λ-kalkyylin kieliopin termit muodostetaan seuraavien sääntöjen mukaan: $\tau = \mathbf{1} \mid b \in \mathcal{T} \mid \tau \rightarrow \tau \mid \tau \times \tau$ [BCL92, ZGC03]. Edellä esitellyn kieliopin muodostavat siten vakiotyyppi $\mathbf{1}$ (*unit type*), jonka ainoa alkio on tyhjä joukko $()$, perustyyppien joukko $\mathcal{T}$, sekä funktio- ja tulotyypit. Vakiotyyppi $\mathbf{1}$ voidaan rinnastaa esimerkiksi C-ohjelmointikielen *void*-tyyppiin [Pie02].

Tulon kommutatiivisuus mahdollistaa sen, että rakenteellisten tyyppien, kuten komponenttien, vertailussa tulotyyppien termien järjestyksellä ei ole väliä. Tulon assosiatiivisuuden nojalla voidaan komponentit tunnistaa isomorfisiksi, vaikka ne olisivat ryhmitelty eri tavalla. Funktion distributiivisuutta ja Curry-sääntöä voidaan hyödyntää tilanteissa, joissa toinen komponenteista tarjoaa saman palvelun joukkona osapalveluita ja toinen yhtenä koosteisena palveluna. Taulukossa 10 on esimerkki kahden eri komponentin isomorfismin todistamisesta rajoitetun tyypitetyn lambdakalkyylin aksioomia noudattaen.

Tyyppien isomorfismin todistaminen perustuu yleensä niin kutsutun *redusointijärjestelmän* käyttämiseen. Redusointijärjestelmä määrittelee tyyppiteorian aksioomille transitiivisen redusointirelaation [BCL92]. Relaatiossa “$>$” tarkoittaa sitä, että lausekkeen vasen puoli on “monimutkaisempi” kuin lausekkeen oikea puoli. Toinen tapa ilmoittaa relaatio $A > B$ on sanoa, että tyyppi A redusoituu tyypiksi B.

Jotta redusointijärjestelmä R olisi teorian E päättelyjärjestelmä, niin sen tulee olla äärellinen, päättyvä, Church-Rosser ja välttävä sekä riittävä teorialle E [DJ90]. Redusointijärjestelmä on äärellinen, jos siinä on äärellinen määrä redusointisääntöjä. Redusointijärjestelmä on päättyvä (*terminating*), jos kaikki redusoinnit päättyvät joskus. Redusointijärjestelmä R on riittävä ja välttävä teorian E suhteen, jos kaikki redusointijärjestelmän tuottamat termit ovat teorian E termejä ja toisaalta, jos kaikki teorian E termit ovat redusointijärjestelmässä R esiintyviä termejä.

Church-Rosser -ominaisuus tarkoittaa sitä, että jos termi A voidaan redusoida sekä termeiksi A_1 ja A_2, niin on olemassa jokin redusoitu muoto B siten, että sekä A_1 että A_2 redusoituvat muotoon B [DJ90]. Church-Rosser -ominaisuus takaa sen, että jos termin A redusointi päättyy, niin sen tuloksena on yksikäsitteinen normaalimuoto $nf(A)$. Church-Rosser -ominaisuus ei siis itsessään takaa, että redusointi päättyy tai että termillä on normaalimuoto.

Church-Rosser -ominaisuus on kuvattuna kuvassa 4. Kuvassa termi A redusoituu jotakin redusointijärjestelmän sääntöä käyttäen (merkintänä $*$) sekä termeiksi $A1$ että $A2$. Termeihin $A1$ ja $A2$ voidaan puolestaan soveltaa taas kahta vaihtoehtoista redusointisääntöä. Näistä säännöistä vain toiset

```
component A {                component B {
 int Fun1(int x);             int fun1(char a,int i);
 char Fun2(char c);           char fun2(char a);
 struct myStruct {            int j;
  int i;                      char a;
  char c;                    }
 }
 int Fun3(char c);
}
```

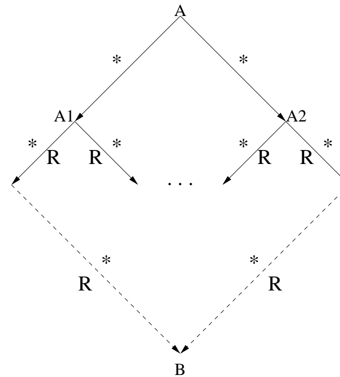
$A : ((int \to int) \times (char \to char) \times (int \times char) \times (int \to char))$
$B : ((int \to (char \times int)) \times (char \to char) \times int \times char)$
Käsitellään komponenttia B.

Välitulos $*$: $\frac{int\to(char\times int)}{(int\to char)\times(int\to int)}(3)$

$$\cfrac{\cfrac{\cfrac{\cfrac{(int\to(char\times int))\times(char\to char)\times int\times char}{(int\times char\times(int\to(char\times int)))\times(char\to char)}(1)}{(int\times char)\times(int\to(char\times int))\times(char\to char)}(2)}{(int\times char)\times(int\to char)\times(int\to int)\times(char\to char)}(*)}{((int\to int)\times(char\to char)\times(int\times char)\times(int\to char)):A}$$

Taulukko 10: Komponenttien isomorfismin todistaminen rajoitetun tyyppijärjestelmän aksioomilla

päätyvät loppujen lopuksi yhteiseen muotoon B.



Kuva 4: Church-Rosser–ominaisuus

Taulukossa 11 on tyyppiteorian Th redusointijärjestelmä R. Redusointijärjestelmää kutsutaan usein myös tyyppien tai termien muunnossysteemiksi (*type rewriting / term rewriting system*) riippuen siitä, onko kyseessä tyyppiteoria tai logiikan redusointijärjestelmä [EM85, BCL92].

Redusointijärjestelmän tarkoituksena on muuttaa tyypit *normaalimuotoon*. Tyyppi on normaalimuodossa, jos ja vain jos sitä ei enää voida redusoida yksinkertaisempaan muotoon relaation $>$ suhteen [BCL92]. Tyypin S normaalimuotoa merkitään $nf(S)$. Redusointitiojärjestelmä R_{Th} takaa, että kaikilla tyypeillä on yksikäsitteinen normaalimuoto $nf(S)$ reduktiojärjestelmässä R_{Th}. Normaalimuodolla $nf(S)$ on joko tyyppiä T tai tyyppiä $S_1 \times \ldots S_n$, missä S_i ei sisällä tyyppiä T tai tulotyyppiä $\times$ [BCL92].

Redusointijärjestelmä R_{Th} muodostaa tyyppiteorian Th päättelyjärjestelmän, eli sen avulla voidaan päättää ovatko kaksi tyyppiä ekvivalentteja. Redusointijärjestelmä R_{Th} on todistettavasti

1)	$A \times T > A$
1')	$T \times A > A$
3)	$A \times (B \times C) > (A \times B) \times C$
4)	$(A \times B) \to C > A \to (B \to C)$
5)	$A \to (B \times C) > (A \to B) \times (A \to C)$
6)	$A \to T > T$
7)	$T \to A > A$

Taulukko 11: Tyyppiteorian Th reduktiojärjestelmä R_{Th}

konfluentti ja vahvasti normalisoituva [BCL92]. Konfluenttisuus ja vahva normalisoituvuus takaavat sen, että termin A redusointi muunnossysteemiä R_{Th} käyttäen päättyy aina normaalimuotoon $nf(A)$.

Tyyppien ekvivalenssitodistusten algoritminen laskenta etenee yksinkertaistaen siten, että ensin tarkistettavat tyypit A ja B muutetaan normaalimuotoihinsa $nf(A)$ ja $nf(B)$. Tämän jälkeen normaalimuotoja vertaillaan rekursiivisesti ekvivalenssirelaatiota $\equiv$ käyttäen. Kaksi termiä ovat isomorfisia, jos ne edustavat samaa muotoa ($\times$ tai $\to$) ja jos niiden operandit (alitermit) ovat isomorfisia [ZGC03].

3.2.2 Rakenteellinen alityypitys

Rakenteellisella alityypityksellä tarkoitetaan kahden rajapintatyypin A ja B välistä rakenteellista suhdetta. Tämä suhde ei ole välttämättä ekvivalenssi, vaan se on osittainen bijektiivinen kuvaus rajapintarakenteesta toiselle. Ennen rakenteellisen alityypityksen tarkempaa määrittelyä pitää alityypitysrelaation ominaisuudet määritellä formaalisti. Alityypitysrelaatiota, jossa tyyppi A on B:n alityyppi, merkitään $A <: B$. Alityypityksen yleiset ominaisuudet on määritelty taulukossa 12 [Pie02, sivut 182–183].

$\dfrac{\mathcal{T} \vdash t : S \quad S <: T}{\mathcal{T} \vdash t : T}$	(T-SUBS)
$S <: S$	(S-REFL)
$\dfrac{S <: U \quad U <: T}{S <: T}$	(S-TRANS)

Taulukko 12: Alityypitysrelaation yleiset ominaisuudet

Sääntö T-SUBS tarkoittaa sitä, että jos t on tyyppiä S, ja S on tyypin T alityyppi kontekstissa $\mathcal{T}$, niin silloin t on myös tyyppiä T. S-REFL ilmaisee alityyppirelaation refleksisyyden, joka on voimassa kaikissa konteksteissa. Sääntö S-TRANS määrittelee alityypityksen transitiiviseksi relaatioksi: jos S on U:n alityyppi ja U on T:n alityyppi, niin S on myös T:n alityyppi.

Yllä mainittujen yleisten alityypityssääntöjen lisäksi tarvitsemme säännöt rakenteisten tyyppikonstruktorien alityypitysrelaatiolle. Monikkotyyppien alityypityksen säännöt on määritelty taulukossa 13 [Pie02, sivut 183–184].

Säännön $S - RcdWidth$ mukaan alityypin tulee sisältää vähintään samat alkiot kuin ylityyppin-

$$\{l_i : T_i^{i \in 1...n+k}\} <: \{l_i : T_i^{i \in 1...n}\} \qquad \text{(S-RcdWidth)}$$

$$\frac{\forall i\, S_i <: T_i}{\{l_i : S_i^{i \in 1...n}\} <: \{l_i : T_i^{i \in 1...n}\}} \qquad \text{(S-RcdDepth)}$$

$$\frac{\{k_j : S_j^{j \in 1...n}\}\ on\ \{l_i : T_i^{i \in 1...n}\} : n\ permutaatio}{\{k_j : S_j^{j \in 1...n}\} <: \{l_i : T_i^{i \in 1...n}\}} \qquad \text{(S-RcdPerm)}$$

Taulukko 13: Monikkotyypin alityypityssäännöt

sä, mutta alityyppi saa sisältää myös ylimääräisiä alkioita. Tämä on luonnollinen, yleisesti ohjelmointikielissä käytetty tulkinta alityypitykselle, jossa alityyppi voi sisältää laajemman toiminnallisuuden. Säännön $S - RcdDepth$ mukaan alityypin alkiot ovat ylityypin vastaavien alkioiden alityyppejä. Lopuksi säännöllä $S - RcdPerm$ mahdollistetaan se, että monikkotyyppien alityyppirelaatiota tarkastellessa termin alkioiden järjestyksellä ei ole väliä.

Yllä esiteltyjä sääntöjä käyttämällä voidaan luoda joustavia ehtoja kahden eri rakenteellisen tyypin korvautuvuudelle. Tässä vaiheessa tulee jo huomioida, että sääntöjen $S - RcdDepth$ ja $S - RcdPerm$ käyttäminen voivat johtaa alityyppitarkistuksessa hyvin syviin $(S - RcdDepth)$ ja / tai leveisiin $(S - RcdPerm)$ hakupuihin tyyppitarkistuksen yhteydessä. Tämän takia alityyppitarkistuksessa tuleekin näiden sääntöjen käyttöä jollakin tapaa rajoittaa, jotta tarkistusavaruus ei tulisi liian suureksi tai saataisi hyödyttömiä tuloksia.

Funktiotyyppien alityypitysrelaatio määräytyy säännön $S - Arrow$ mukaan, joka on annettu taulukossa 14 [Pie02, sivu 184].

$$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \qquad \text{(S-Arrow)}$$

Taulukko 14: Funktiotyypin alityypityssäännöt

Funktioiden alityypitysrelaatio on epäsymmetrinen syöte- ja tulosjoukkojen suhteen: funktion S, joka on funktion T alityyppi, syötejoukko sisältää T:n syötejoukon, kun taas S:n tulosjoukko sisältyy T:n tulosjoukkoon. Tämä on juuri se ominaisuus, mitä intuitiivisesti alityyppifunktiolta odotetaankin: kaikki ylityypin syötteet kelpaavat alityypin syötteeksi ja toisaalta alityyppifunktion tulokset kuuluvat ylityypin tuottamien tulosten joukkoon.

3.3 Rajapintarakenteiden tulkinta termiautomaatteina

Tyyppien (rekursiivisten tai ei-rekursiivisten) välisen isomorfismin tarkistamiseksi tyyppien T ja S normaalimuotoisista termeistä muodostetaan niin kutsuttu *termiautomaatti* (*term automata*). Näiden automaattien avulla tarkistetaan ovatko tyypit isomorfisia. Tyyppien välinen isomorfismi voi olla joko ekvivalenssi- tai alityypitysrelaatio, joita vastaavat relaatiot automaateille ovat *bisimulaatio-* ja *simulaatioekvivalenssi* [Par81, Mil89a, JPZ02, CPR04]. Tässä aliluvussa annetaan tyyppijärjestelmän termeille tulkinta termiautomaatteina

3.3.1 Termin määritelmä

Rekursiivisten tyyppien isomorfismin tarkistamisessa tyypit tulkitaan termiautomaateiksi. Itse termi tulkitaan funktioksi, joka kuvaa syntaktisen puun numeroidut polut aakkoston alkioiksi. Aakkoston Σ termi määritellään osittaisfunktion avulla, joka kuvaa luonnollisten lukujen äärellisen jonon aakkostoon Σ. Osittaisfunktiolla tarkoitetaan sellaista kuvausta, joka ei ole välttämättä määritelty kaikille lähtöjoukon alkioille ja joka kuvaa lähtöjoukon alkion korkeintaan yhdelle maalijoukon alkiolle.

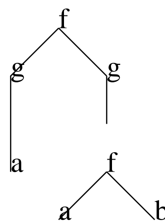
Aakkoston Σ termit koostuvat perustyypeistä $\mathcal{T}$, tulo- ja funktiotyypeistä sekä tarvittaessa rekursio-operaattorista μ. Aakkoston Σ termit on täten muotoa $t = b \in \mathcal{B} \mid t \times t \mid t \to t \mid \mu l.t$. Ariteetilla tarkoitetaan monikkotermin alkioiden lukumäärää. Joukko tyyppejä, joiden ariteetti on n merkitään Σ_n. Esimerkiksi tyypin $(a \times b)$ ariteetti on kaksi eli se kuuluu joukkoon Σ_2. Luonnollisten lukujen joukkoa merkitään symbolilla ω ja ω^* merkitsee äärellisten lukujonojen joukkoa.

Aakkoston Σ termi on osittaisfunktio $t : \omega^* \to \Sigma$ jonka lähtöjoukko $\mathcal{D}(t)$ on epätyhjä, koska se sisältää ainakin tyhjän merkkijonon ϵ. Lisäksi lähtöjoukko $\mathcal{D}(t)$ on prefiksisuljettu, toisin sanoen jos $\alpha = \omega_1\omega_2 \dots \omega_n \in \mathcal{D}(t)$ niin silloin myös kaikki α:n prefiksit eli etuliitteet $\{\epsilon, \omega_1, \omega_1\omega_2, \dots, \omega_1 \dots \omega_{n-1})$ kuuluvat joukkoon $\mathcal{D}(t)$. Osittaisfunktiolle t on voimassa: jos $t(\alpha) \in \Sigma_n$, niin $\{i \mid \alpha i \in \mathcal{D}(t)\} = \{0, 1 \dots, n-1\}$, missä Σ_n sisältää kaikki aakkoston termit, joiden asteluku (ariteetti) on n. Tämä ehto määrittelee termin $\alpha \in \Sigma_n$ siirtymien numeroinnin välille $\{0, 1, \dots, n\}$. Siirtymät numeroidaan vasemmalta oikealle numerojärjestyksessä.

Määritelmä 3.3 (Syntaktisen termin määritelmä)
Termi t on osittaisfunktio $t : \omega^* \to \Sigma$*, joka kuvaa luonnollisten lukujen merkkijonon aakkoston* Σ *alkioille. Sillä on seuraavat ominaisuudet:*

- *osittaisfunktion t lähtöjoukko* $\mathcal{D}(t)$ *on epätyhjä, sillä se sisältää ainakin tyhjän merkkijonon* ϵ.
- *jos termin t asteluku on* n*, niin sillä on* n *seuraajaa, joihin siirtymät on numeroitu* $\{0 \dots n-1\}$.
- *t on prefiksisuljettu*

Elementti $\alpha \in \omega^*$ on tyyppipuun lehti, jos $\alpha \in \mathcal{D}(t)$, mutta α ei ole minkään muun termin prefiksi, eli $t(\alpha) \in \Sigma_0$. Esimerkiksi, jos $\Sigma = \{f, g, a, b\}$, missä f, g, a, b:n ariteetit ovat vastaavasti $2, 1, 0, 0$, niin termi $t : f(g(a), g(f(a, b)))$ on kuvan 5 muotoinen puu.



Kuva 5: Äärellinen termi t

Termin t lehtiä edustavat merkkijonot $00, 100, 101$, joille $t(00) = t(100) = a$ ja $t(101) = b$. Termin t lähtöjoukko $\mathcal{D}(t) = \{\epsilon, 0, 1, 00, 10, 100, 101\}$. Tyypin t solmut ovat $t(\epsilon) = t(10) = f$ ja $t(0) = t(1) = g$ [KPS93].

Kaikki syntaktiset termit voidaan kuvata puun muodossa. Termillä on alipuun kanssa analoginen käsite alitermi, joka määritellään termin käsitteen avulla.

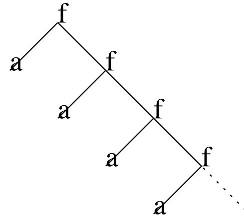
Määritelmä 3.4 (Syntaktisen alitermin määritelmä)
Määritellään osittaisfunktio $t \downarrow \alpha : w^* \rightarrow \Sigma$ siten, että $(t \downarrow \alpha)(\beta) = t(\alpha\beta)$. Jos t on termi ja $\alpha \in \omega^*$, niin funktiota $t \downarrow \alpha : \omega^* \rightarrow \Sigma$ kutsutaan alitermiksi kohdassa α, siten että $(t \downarrow \alpha)(\beta) = t(\alpha\beta)$.

Osittaisfunktio $t \downarrow \alpha$ kuvaa lukujonon $(\alpha\beta) \in \omega^*$ suffiksin β aakkoston Σ alkioille $t(\alpha\beta) \in \Sigma$ kaikille $\alpha\beta \in \mathcal{D}(t)$, kun α on on jonkin joukon $\mathcal{D}(t)$ lukujonon mikä tahansa prefiksi. Osittaisfunktio $t \downarrow \alpha$ on termi, koska t on prefiksisuljettu. Lisäksi osittaisfunktion lähtöjoukko $\mathcal{D}_\alpha$ on epätyhjä, sillä se sisältää ainakin tyhjän lukujonon ϵ. Osittaisfunktiota $t \downarrow \alpha$ kutsutaan termin t alitermiksi kohdassa α.

Jos siis alitermiä $t \downarrow \alpha$ merkitään t':lla, niin silloin $t'(\beta) = t(\alpha\beta)$. Kuvassa 5 t:n alitermit ovat f, g, a, b. Jos määritellään $t' = t \downarrow 1$, niin silloin $t' \downarrow (01) = b$. Tällöin t' on termin t:n alitermi (ensimmäinen solmu juuresta oikealle, g) ja sen alitermi $t' \downarrow (01)$ on alitermi b.

Termiä t kutsutaan *äärelliseksi*, jos sen lähtöjoukko $\mathcal{D}(t)$ on äärellinen. Äärellisten termien joukolle käytetään merkintää T_F. Kuvan 5 esittämä termi on äärellinen, koska sen lähtöjoukko $\mathcal{D}(t) = \{\epsilon, 0, 1, 00, 10, 100, 101\}$ on äärellinen [KPS93].

Termi t on *säännöllinen*, jos sillä on äärellinen määrä eri alitermejä, eli jos joukko $\{t \downarrow \alpha | \alpha \in \omega^*\}$ on äärellinen [KPS93]. Säännöllisten termien joukolle käytetään merkintää T_R. Kuvassa 6 on esimerkki säännöllisestä termistä s. Termin s alitermien joukko $\{t \downarrow \alpha | \alpha \in \omega^*\}$ koostuu vain kahdesta alkiosta, nimittäin a:sta ja s:stä. Huomioitavaa on, että s:n lähtöjoukko on kuitenkin ääretön säännöllinen joukko $1^* + 1^*0$, jossa $s(1^n0) = a$ ja $s(1^n) = f$ kaikille $n \geq 0$ [KPS93]. Edellä mainittujen määritelmien mukaan kaikki äärelliset termit ovat säännöllisiä termejä eli $T_F \subseteq T_R$.



Kuva 6: Ääretön, säännöllinen termi s

Äärelliset ja säännölliset termit (kieliopit) muodostavat niiden rakenteisten tyyppien joukon, jotka ovat esitettävissä äärellisen termiautomaatin avulla. Äärelliset termit muodostavat epärekursiivisten tyyppien joukon ja säännölliset termit muodostavat rekursiivisten tyyppien joukon. Kahden rekursiivisen tyypin α ja β isomorfismin tarkistaminen voidaan siis redusoida kahden äärellisen termiautomaatin isomorfismiin [AC93].

3.3.2 Termiautomaatti

Jokaisella äärellisen aakkoston Σ säännöllisellä termillä t on olemassa esitys äärellisenä automaattina, jota kutsutaan termiautomaatiksi [KPS93]. Aakkoston Σ äärellistilallinen termiautomaatti M on monikko $M = (Q, \Sigma, q_0, \delta, \ell)$, jonka määritelmä on annettu määritelmässä 3.5.

Määritelmä 3.5 (Termin t termiautomaatti M_t)
Termiä t kuvaava äärellinen automaatti M_t on monikko $(Q, \Sigma, q_0, \delta, \ell)$, missä

1. $Q = \{t \downarrow \alpha : \alpha \in \omega^*, \mathcal{D}(t \downarrow \alpha) \neq 0\}$ *on äärellinen tilojen joukko*
2. Σ *on aakkosto*
3. $q_0 \in Q, q_0 = t = t \downarrow \epsilon$ *on aloitustila*
4. $\delta : Q \times \omega \to Q$ *on siirtymäfunktio (osittaisfunktio), ω on luonnollisten lukujen joukko. $\delta(q, i)$ on määritelty jos ja vain jos $\alpha \in \omega^*$, $t(\alpha) = q$ ja $\alpha i \in \mathcal{D}(t)$.*
5. $\ell : Q \to \Sigma$ *on nimeämisfunktio, siten että kaikille $q \in Q$, jos $\ell(q) \in \Sigma_n$, niin $\{i | \delta(q, i)$ on määritelty $\} = \{0, 1, \ldots, n-1\}$*

Automaatin tilat Q ovat siis joukko osittaisfunktioita $t \downarrow \alpha$, $q_0 = q_0$ on termin t:n juuri. Nimeämisfunktio ℓ kuvaa tilan q sitä vastaavaan alitermiin $(t \downarrow \alpha)(\epsilon)$, eli se nimeää tilan q aakkoston Σ alkiolla. Siirtymisfunktio $\delta(q, i)$ on kuvaus alitermistä q sen i:lle seuraaja-alitermille $q \downarrow i$.

Siirtymäfunktion δ avulla voidaan määritellä induktiivinen siirtymäfunktio $\hat{\delta} : Q \times \omega^* \to Q$ seuraavasti:

$$\hat{\delta}(q, \epsilon) = q$$
$$\hat{\delta}(q, \alpha i) = \delta(\hat{\delta}(q, \alpha), i)$$

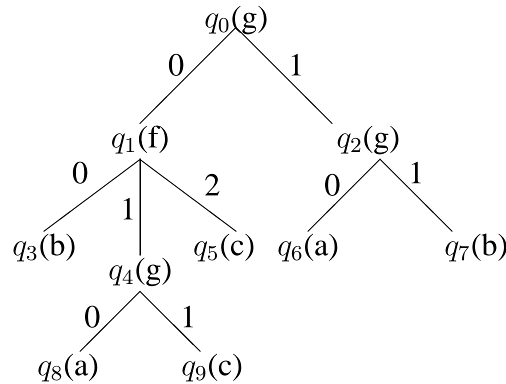
Termi t voidaan siirtymäfunktiota $\hat{\delta}$ käyttäen ilmaista muodossa $\lambda\alpha.\ell(\hat{\delta}(q, \alpha))$. Funktiota $\hat{\delta}$ käyttäen voidaan mistä tahansa alitermistä q siirtyä polkua α eteenpäin, kun funktiolla δ siirryttiin aina seuraavaan tilaan i siirtymää pitkin. Funktion tulkinta on se, että $\lambda\alpha.\ell(\hat{\delta}(q, \alpha))$ ottaa syötteekseen tilan q sekä lukujonon $\alpha = \omega_1\omega_2 \ldots \omega_n$ ja käy sitten läpi automaattia M_t yksi siirtymä (α:n luku ω_i) kerrallaan. Jos syötteen α siirtymät ($q \xrightarrow{\omega_1} q_i \xrightarrow{\omega_2} \ldots \xrightarrow{\omega_n} q_n$) vievät automaatin tilaan q_n, niin silloin yllä mainittu funktio palauttaa termin (q_n). Jos jokin siirtymä ω_i syötteessä α ei ole määritelty siirtymäfunktiolle $\delta(q_{i-1}, \omega_i)$, niin silloin funktio ei ole määritelty [KPS93].

Kuvassa 7 on kuvattu äärellisen termin $t = g(f(b, g(a, c), c), g(a, b))$ automaatti
$M_t = \{\{q_0, q_1, \ldots, q_9\}, \{a, b, c, f, g\}, q_0, \delta, \ell\}$, missä $\{a, b, c\} \in \Sigma_0$, $f \in \Sigma_3$ ja $g \in \Sigma_2$. Kuvassa siirtymäfunktio δ ja nimeämisfunktio ℓ ovat seuraavanlaiset:

$$\begin{array}{lllll}
\delta(q_0, 0) = q_1 & \delta(q_0, 1) = q_2 & \delta(q_1, 0) = q_3 & \delta(q_1, 1) = q_4 & \delta(q_1, 2) = q_5 \\
\delta(q_2, 0) = q_5 & \delta(q_2, 1) = q_6 & \delta(q_4, 0) = q_8 & \delta(q_4, 1) = q_9 & \\
\ell(q_0) = g & \ell(q_1) = f & \ell(q_2) = g & \ell(q_3) = b & \ell(q_4) = g \\
\ell(q_5) = c & \ell(q_6) = a & \ell(q_7) = b & \ell(q_8) = a & \ell(q_9) = c
\end{array}$$

Jos M on termiautomaatti, niin silloin sen esittämä termi $t_M = \lambda\alpha.\ell(\hat{\delta}(q, \alpha))$ [KPS93]. Jos on olemassa sellainen termiautomaatti M, että $t = t_M$, niin termin t sanotaan olevan esitettävissä. Esitettävissä olevien termien, säännöllisten termien ja rekursiivisten tyyppien välillä on yhteys siten, että kun t on jokin termi, niin seuraavat ehdot ovat keskenään ekvivalentteja [KPS93, JPZ02]:

1. t on säännöllinen
2. t on esitettävissä
3. t voidaan määritellä äärellisellä joukolla sääntöjä,
   jotka sisältävät μ-operaattoreita.



Kuva 7: Esimerkki termiautomaatista

3.4 Termien välisen korvautuvuuden tarkistaminen

Edellisessä aliluvussa kuvattiin, kuinka rakenteisilla tyypeillä muodostettavat termit voidaan tulkita termiautomaateiksi. Tässä aliluvussa kuvataan, kuinka termiautomaattien välisiä bisimulointi- ja simulointiekvivalensseja käyttäen voidaan tarkistaa termien välinen rakenteellinen ekvivalenssi ja alityypitys.

3.4.1 Termien välinen ekvivalenssi

Äärellisten automaattien samankaltaisuus määritellään niiden hyväksymien siirtymien mukaan. Jos kaksi eri automaattia hyväksyvät täsmälleen samat siirtymät samassa järjestyksessä, on luontevaa olettaa, että ne ovat jollain tasolla ekvivalentteja. Tässä aliluvussa tutustutaan tarkemmin siihen, miten tämä samankaltaisuus formalisoidaan ja minkälaisilla algoritmeilla termiautomaattien samankaltaisuus voidaan tarkistaa.

Automaattien välisistä relaatioista tarkastelemme erityisesti simulaatio- ja bisimulaatioekvivalensseja, jotka soveltuvat hyvin termiautomaattien suhteiden tarkistamiseen. Äärellisten automaattien bisimulaatioekvivalenssin tarkistamiseksi on kehitetty tehokkaita algoritmeja, jotka perustuvat Robert Paigen ja Robert Tarjanin kehittämään ositusalgoritmiin [PT87]. Ositusalgoritmia voidaan voidaan soveltaa myös termiautomaattien tapauksessa.

Simulaatioekvivalenssi voidaan nimensä mukaisesti tulkita siten, että automaatti A voi simuloida automaattia B, vaikka ne eivät välttämättä olekaan täysin samankaltaisia. Bisimulaatio on tiukempi ehto, joka voidaan tulkita siten että automaatti A voi simuloida automaattia B, mutta automaatti B voi myös simuloida automaattia. Bisimilaariset automaatit ovat täysin ekvivalentteja (ulkoiselta) käytökseltään, toisin sanoen, ne hyväksyvät täsmälleen samat kielet.

Tarkastelemme nyt kahta äärellistä termiautomaattia $M =< Q, \Sigma, q_0, \delta, \ell >$ ja $M' =< Q', \Sigma, q'_0, \delta', \ell' >$, missä Q ja Q' ovat termiautomaattien tilat, q_0 ja q'_0 niiden alkutilat, δ ja δ' niiden siirtymäfunktiot ja ℓ, ℓ' tilojen merkintäfunktiot.

Otetaan käyttöön kaksi lyhennysmerkintää funktiotyypille ja tulotyypille. Jos tyypille σ on voimassa $\sigma(\epsilon) =\rightarrow$ ja sen alipuut kohdissa 0 ja 1 ovat τ_1 ja τ_2, niin merkitään $\tau_1 \rightarrow \tau_2$. Jos $\sigma(\epsilon) = \Pi^n$ eli σ on tulotyyppi, jossa n termiä ja sen alipuut ovat $\tau_i, i \in \{0 \ldots n-1\}$, niin merkitään $\Pi_0^{n-1}\tau_i$ [JPZ02].

Automaatti M bisimuloi automaattia M' ($M \cong M'$), jos on olemassa bisimulointirelaatio $\mathcal{R} \subseteq Q \times Q'$ siten, että se toteuttaa määritelmän 3.6 ehdot [JPZ02]. Jos $M_t \cong M'_t$, niin silloin tyypit t ja t' ovat ekvivalentit.

Määritelmä 3.6 (Termiautomaattien bisimulointi)
Termiautomaattien $M =< Q, \Sigma, q_0, \delta, \ell >$ *ja* $M' =< Q', \Sigma, q'_0, \delta', \ell' >$ *bisimulointi on relaatio* $\mathcal{R}$ *siten, että kun* $\sigma \in Q$ *ja* $\tau \in Q'$*, niin*

- *jos* $(\sigma, \tau) \in \mathcal{R}$*, niin* $\sigma(\epsilon) = \tau(\epsilon)$
- *jos* $(\sigma_1 \rightarrow \sigma_2, \tau_1 \rightarrow \tau_2) \in \mathcal{R}$*, niin* $(\sigma_1, \tau_1) \in \mathcal{R}$ *ja* $(\sigma_2, \tau_2) \in \mathcal{R}$
- *jos* $(\Pi_{i=0}^{n-1}\sigma_i, \Pi_{i=0}^{n-1}\tau_i) \in \mathcal{R}$*, niin* $\forall i \in \{0 \ldots n-1\} : (\sigma_i, \tau_i) \in \mathcal{R}$.

Bisimulaatiot ovat suljettuja yhdisteen suhteen, joten on olemassa suurin bisimulaatio $\mathcal{R} = \bigcup\{R : R\ on\ bisimulaatio\}$. Kaksi tyyppiä τ_1 ja τ_2 ovat bisimilaariset (merkitään $\tau_1 \cong \tau_2$), jos ja vain jos on olemassa bisimulointi $\mathcal{R}$ siten että $(\tau_1, \tau_2) \in \mathcal{R}$ [JPZ02].

Termiautomaattien bisimulaatiorelaatio $\cong$ identifioi kaksi termiä, jos ja vain jos ne ovat täysin ekvivalentit. Termien täydellinen ekvivalenssi on liian vahva ehto. Käyttökelpoisempi bisimulointiekvivalenssi saadaan, kun tulotyyppien Π^n sallitaan olevan assosiatiivinen ja kommutatiivinen. Termiautomaattien assosiatiivinen ja kommutatiivinen bisimulointi, merkitään $\cong_{AC}$, on annettu määritelmässä 3.7.

Määritelmä 3.7 (Termiautomaattien AC-bisimulointi)
Termiautomaattien $M =< Q, \Sigma, q_0, \delta, \ell >$ *ja* $M' =< Q', \Sigma, q'_0, \delta', \ell' >$ *AC-bisimulointi on relaatio* $\mathcal{R}$ *siten, että kun* $\sigma \in Q$ *ja* $\tau \in Q'$*, niin*

- *jos* $(\sigma, \tau) \in \mathcal{R}$*, niin* $\sigma(\epsilon) = \tau(\epsilon)$
- *jos* $(\sigma_1 \rightarrow \sigma_2, \tau_1 \rightarrow \tau_2) \in \mathcal{R}$*, niin* $(\sigma_1, \tau_1) \in \mathcal{R}$ *ja* $(\sigma_2, \tau_2) \in \mathcal{R}$
- *jos* $(\Pi_{i=0}^{n-1}\sigma_i, \Pi_{i=0}^{n-1}\tau_i) \in \mathcal{R}$*, niin on olemassa bijektio* $b : N \rightarrow N$*, siten että* $\forall i \in \{0 \ldots n-1\} : (\sigma_i, \tau_{b(i)}) \in \mathcal{R}$.

Myös AC-bisimuloinnit on suljetty yhdisteen suhteen, joten on olemassa suurin AC-bisimulaatio $\mathcal{R} = \bigcup\{R : R\ on\ AC - bisimulaatio\}$. Tyyppien AC-ekvivalenssi $=_{AC}$ on suurin AC-bisimulaatio. Tyypit τ_1 ja τ_2 ovat ekvivalentit, jos ja vain jos on olemassa AC-bisimulointi $\mathcal{R}$ siten että $(\tau_1, \tau_2) \in \mathcal{R}$.

Bisimulointi voidaan tulkita verkon $Q \cup Q'$ karkeimman, vakaan osituksen laskemiseksi, missä Q ja Q' ovat kahden automaatin tilajoukot. Voidaan todistaa, että seuraavat ominaisuudet ovat keskenään ekvivalentteja [JPZ02]:

1. $(\tau_1, \tau_2) \in \mathcal{R}$ jos ja vain jos on olemassa refleksiivinen bisimulointi C (termi)automaattien $A_1 =< Q_1, \Sigma, q_{01}, \delta_1, \ell_1 >$ ja $A_2 =< Q_2, \Sigma, q_{02}, \delta_2, \ell_2 >$ välillä siten, että $(q_{01}, q_{02}) \in C$
2. (termi)automaattien A_1 ja A_2 välillä on refleksiivinen bisimulointi C siten, että $(q_{01}, q_{02}) \in C$, jos ja vain jos on olemassa vakaa joukon P ositus S siten, että q_{01} ja q_{02} kuuluvat samaan S:n osioon

Tyyppien τ_1 ja τ_2 ekvivalenssin tarkistaminen voidaan siis palauttaa verkon $Q_1 \cup Q_2$ ekvivalenssiosituksen laskemiseen. Tyyppien AC-ekvivalenssi voidaan tarkistaa $\mathcal{O}(n \log n)$ ajassa käyttämällä algoritmia, joka soveltaa Paigen ja Tarjanin ositusalgoritmia, kun n on tyyppejä τ_1 ja τ_2 termiautomaattien A_1 ja A_2 koko [JPZ02]. Paigen ja Tarjanin ositusalgoritmissa joukon U alkiot jaetaan ekvivalenssiluokkiin jonkin alkioiden välisen relaation E suhteen, kun on annettu U ja sen alustava ositus P. Ositusalgoritmi on siis muotoa $Refine(U, E, P) \to P$.

Kun ositusalgoritmia käytetään tyyppien AC-ekvivalenssin tarkistamiseen, tehdään tyypeistä ensin termiautomaatit A_1 ja A_2. Termiautomaatteista muodostetaan yhdisteautomaatti $A = A_1 \cup A_2$, siten että $Q = Q_1 \cup Q_2$, $\delta : Q \times \omega \to Q$, missä $\delta = \delta_1 \oplus \delta_2$ ja $\ell : Q \to \Sigma$, missä $\ell = \ell_1 \oplus \ell_2$, kun $\oplus$ on kahden funktion erillinen yhdiste [JPZ02]. Tämän jälkeen yhdisteautomaatin tilat jaetaan ekvivalenssiluokkiin käyttäen ositusalgoritmia. Ositusalgoritmille annetaan syötteenä tilojen joukko Q, termien bisimulointirelaatio $\mathcal{R}$ sekä alustava ositus $P = (Q \times Q)$. Ositusalgoritmin $Refine(U, E, P)$ tuloksena on karkein, eli vähiten ekvivalenssiluokkia sisältävä, ositus P, jossa solmut q_1 ja q_2 kuuluvat samaan osioon jos ja vain jos ne ovat ekvivalentit relaation E suhteen [PT87].

3.4.2 Termien välinen alityypitys

Rekursiivisten tyyppien AC-ekvivalenssi samastaa kaksi tyyppiä τ_1 ja τ_2 toisiinsa, jos ja vain jos niillä on täsmälleen sama rakenne kun alitermien keskinäisellä järjestyksellä ei ole väliä. Tämä ei ole kuitenkaan tarpeeksi joustava relaatio silloin, jos halutaan esimerkiksi identifioida ylityyppi ja sen rakenteellinen alityyppi keskenään. Alityypin ja ylityypin välinen identifiointi on erittäin hyödyllistä, koska tällöin komponentteja voidaan laajentaa rakenteellisesti ilman, että niiden välinen syntaktinen korvautuvuus rikkoontuisi. Jos esimerkiksi $\tau_1 =_{AC} \tau_2$, niin tällöin $\tau_1 \neq_{AC} (\tau_2 \times \alpha)$, mistä jo huomataan, että AC-ekvivalenssi ei ole tarpeeksi joustava komponenttien välisen yhteentoimivuuden kriteeri, kun komponenttien syntaktinen rakenne on altis laajentamiselle. Tarvitaan siis syntaktisten rakenteiden välinen relaatio, joka mahdollistaa tyyppien toiminnallisuuden laajentamisen.

Joustavamman vertailurelaation kehittämiseksi otetaan käyttöön termien alityypitys, joka mahdollistaa tyyppirakenteiden laajentamisen. Vertailurelaation perustana on termien alityypitys, joka on määritelmän 3.8 mukainen [CPR04].

Määritelmä 3.8 (Termien alityypitys)
Olkoon $\leq_0$ seuraavien sääntöjen mukainen järjestysrelaatio, kun s on mielivaltainen termi:

$$\bot \leq_0 s \quad s \leq_0 \top \quad \to \leq_0 \to \quad \frac{n \geq m}{\Pi^n \leq_0 \Pi^m}$$

Tällöin binäärinen relaatio $\mathcal{R} \subseteq \Sigma \times \Sigma$ on $\leq$-simulointi jos ja vain jos seuraavat säännöt ovat voimassa:

$$\frac{\tau_1 \; \mathcal{R} \; \tau_2}{\tau_1(\epsilon) \leq_0 \tau_2(\epsilon)} \; SUB-TOP \qquad \frac{\tau_1 \to \tau_2 \; \mathcal{R} \; \tau_1' \to \tau_2'}{\tau_1' \; \mathcal{R} \; \tau_1 \quad \tau_2 \; \mathcal{R} \; \tau_2'} \; SUB-ARROW$$

$$\frac{\Pi_{i=0}^{n-1} \tau_i \; \mathcal{R} \; \Pi_{i=0}^{m-1} \tau_i'}{(\tau_i \; \mathcal{R} \; \tau_i')^{i \in \{0,\ldots,m-1\}}} \; SUB-PI$$

Termien alityypitys $\leq$ on suurin $\leq$-simulaatio $\bigcup(\mathcal{R} : \mathcal{R}\, on \; \leq -simulaatio)$.

Kuten rekursiivisten tyyppien ekvivalenssin tapauksessa, voidaan alityypityksellekin määritellä versio, joka tulkitsee tulotyypit assosiatiiviseksi ja kommutatiiviseksi. Tämän niin kutsutun AC-alityypityksen säännöt ovat määritelmässä 3.9 [CPR04].

Määritelmä 3.9 (Termien AC-alityypitys)
Olkoon $\mathcal{R} \subseteq \Sigma \times \Sigma$ ja $\mathcal{R}' \subseteq \Sigma \times \Sigma$ binäärisiä relaatioita. $\mathcal{R} \subseteq \Sigma \times \Sigma$ on $\leq_{AC}$-simulaatio olettaen relaatio $\mathcal{R}'$, jos ja vain jos seuraavat säännöt ovat voimassa, kun b on bijektio luonnollisten lukujen joukosta $\{0 \ldots m-1\}$ joukkoon $\{0 \ldots n-1\}$:

$$\frac{\tau_1 \; \mathcal{R} \; \tau_2}{\tau_1(\epsilon) \leq_0 \tau_2(\epsilon)} \; SUBAC - TOP \qquad \frac{\tau_1 \to \tau_2 \; \mathcal{R} \; \tau_1' \to \tau_2'}{\tau_1' \; (\mathcal{R} \cup \mathcal{R}') \; \tau_1 \quad \tau_2 \; (\mathcal{R} \cup \mathcal{R}') \; \tau_2'} \; SUBAC - ARROW$$

$$\frac{\Pi_{i=0}^{n-1} \tau_i \; \mathcal{R} \; \Pi_{i=0}^{m-1} \tau_i'}{(\tau_{b(i)} \; (\mathcal{R} \cup \mathcal{R}') \; \tau_i')^{i \in \{0,\ldots,m-1\}}} \; SUBAC - PI$$

$\mathcal{R}$ on $\leq_{AC}$-simulaatio jos ja vain jos se on $\leq_{AC}$-simulaatio ja $\mathcal{R}'$ on tyhjä joukko. Termien AC-alityypitys $\leq_{AC}$ on suurin $\leq_{AC}$-simulaatio.

Termien AC-simulaation toiminta voidaan tulkita siten, että tyypit τ_1 ja τ_2 ovat AC-similaarisia, jos ne ovat suhteessa $\mathcal{R}$, kun oletetaan tyyppien välille relaatio $\mathcal{R}'$. Tyypit τ_1 ja τ_2 ovat siis AC-alityypityssuhteessa, jos ja vain jos relaatio $\mathcal{R}$ on validi alityypitysrelaatio ilman mitään oletuksia. Relaatio $\mathcal{R}'$ tavallaan edustaa todistusvelvoitteita, jotka tulee todistaa valideiksi, jotta tyyppien välinen AC-alityypitys olisi voimassa. Tällaista todistustekniikkaa kutsutaan englannin kielellä *up-to*-tekniikaksi, kuten esimerkiksi "*bisimulation up-to bisimilarity*" ja "*bisimulation up-to context*" [Mil89b, SW01].

Voidaan todistaa, että relaatio $\leq_{AC}$ on esijärjestys eli se on refleksiivinen ja transitiivinen [CPR04]. Relaatio $\leq_{AC}$ ei kuitenkaan ole symmetrinen ja koska ositusalgoritmi laskee joukon ekvivalenssiosituksen, ei $\leq_{AC}$-relaation laskemiseen voida soveltaa Paigen ja Tarjanin ositusalgoritmia. Tämän vuoksi AC-alityypityksen tarkistaminen on vaativampaa.

Tyyppiparin (τ, τ') sanotaan olevan validi, jos $\tau \leq_{AC} \tau'$ ja epävalidi muuten. AC-alityypityksen tarkistuksessa yritetään antaa siis vastaus sille, onko $p_0 = (\tau, \tau')$ validi. AC-alityypityksen tarkistus tehdään kahdessa vaiheessa. Ensimmäisessä vaiheessa lasketaan alityypitysvelvoitteiden joukko U, jonka alkiot ovat tyyppien pareja (τ, τ') [CPR04]. Joukko U on pienin joukko, joka sisältää parin p_0 ja joka on suljettu taulukon 15 sääntöjen mukaan.

$$\frac{(\tau_1 \to \tau_2, \tau_1' \to \tau_2') \in U}{(\tau_1', \tau_1) \in U \quad (\tau_2, \tau_2') \in U} \; EXPLORE - ARROW$$

$$\frac{(\Pi_{i=0}^{n-1} \tau_i, \Pi_{j=0}^{m-1} \tau' j) \in U}{((\tau_i, \tau_j') \in U)^{i \in \{0,\ldots,n-1\}, j \in \{0,\ldots,m-1\}}} \; EXPLORE - PI$$

Taulukko 15: AC-alityypityksen säännöt $EXPLORE - ARROW$ ja $EXPOLORE - PI$

Säännöllä $EXPLORE - ARROW$ ilmaistaan, että jotta funktiotyyppi $\tau_1 \to \tau_2$ voidaan todistaa olevan funktiotyypin $\tau_1' \to \tau_2'$ alityyppi, niin täytyy todistaa että tyyppiparit (τ_1', τ_1) ja (τ_2, τ_2') ovat valideja. Tämä vastaa perinteisen tyyppiteorian tulkintaa funktiotyyppien alityypityksestä, jossa funktion syötteiden tulee olla niin sanotusti kontravariantteja alityypitysoperaation suhteen.

Sääntö $EXPLORE - PI$ ilmaisee, että jos halutaan todistaa, että tulotyyppi τ_1' on tulotyypin τ_1 alityyppi, niin todistusvelvoitteisiin lisätään tulotyyppien alkioiden kaikki parit $(\tau_i \times \tau_j')$. Tyyppiparin $p_0 = (\tau, \tau')$ alityypityksen todistusvelvotteiden joukon U laskemiseen tarvittava algoritmi on kuvattu taulukossa 16 [CPR04].

Taulukon 16 algoritmi saa syötteekseen tyyppiparin p_0. Algoritmi laskee sääntöjä $EXPLORE -$

Require: Tyyppipari $p_0 = (\tau, \tau')$
Ensure: Alityypitysvelvoitteiden joukko U

```
1: U := ∅, W := {p0}
2: while W ≠ ∅ do
3:     poista pari p joukosta W
4:     if p ∈ U then
5:         continue
6:     else
7:         lisää p joukkoon U
8:         if p on muotoa (τ1 → τ2, τ'1 → τ'2) then
9:             lisää (τ'1, τ1) ja (τ2, τ'2) joukkoon W
10:        end if
11:        if p on muotoa (Π_{i=0}^{n-1} τi, Π_{j=0}^{m-1} τ'j) then
12:            lisää kaikki parit (τi, τ'j), i ∈ {0, ..., n−1}, j ∈ {0, ..., m−1} joukkoon W
13:        end if
14:    end if
15: end while
```

Taulukko 16: Alityypityksen todistusvelvoitteet muodostava algoritmi

$ARROW$ (rivit 8–9) ja $EXPLORE - PI$ (rivit 11–12) soveltaen todistusvelvoitteiden joukon U, jota käytetään alityypitystarkistuksen toisessa vaiheessa. Kun algoritmi on lopettanut suorituksensa, sisältää joukko U kaikki parit (τ, τ') joiden tulee olla valideja, jotta p_0 olisi validi [CPR04].

Alityypitysvelvoitteiden laskemisen jälkeen siirrytään toiseen vaiheeseen, jossa joukosta U lasketaan suurin alijoukko S, jonka kaikki tyyppiparit p_i ovat valideja. Jos $p_0 = (\tau, \tau')$ kuuluu joukkoon S algoritmin lopetettua toimintansa, niin $\tau \leq_{AC} \tau'$. Joukon S muodostaminen aloitetaan täydestä joukosta U, josta joka kierroksella poistetaan ne epävalidit parit. Algoritmi lopettaa suorituksensa, kun relaation $\leq_{AC}$ kiintopiste joukossa U on saavutettu. Kiintopisteen $\leq_{AC}$ joukosta U lasketaan taulukossa 17 kuvatun algoritmin mukaisesti [CPR04].

Taulukossa 17 kuvattu algoritmi saa syötteekseen todistusvelvoitteiden joukon U. Rivillä 1 suoritetaan algoritmin käsittelemien joukkojen alustus, jossa validien parien joukko S ja epävalidien parien joukko F alustetaan tyhjiksi joukoiksi. Joukossa W pidetään yllä niiden parien joukkoa, joiden alityypitys tulee tarkistaa. Rivillä 1 joukoksi W asetetaan kaikkien todistusvelvoitteiden joukko U. Algoritmin oikeellinen toiminta perustuu siihen, että se ylläpitää kahta invarianttia: (W, S, F) on joukon U ositus ja S on $\leq_{AC}$-simulointi olettaen, että W:n parit ovat valideja [CPR04]. Alustusten jälkeen invariantit ovat triviaalisti voimassa, koska S ja F ovat tyhjiä joukkoja ja $W = U$.

Algoritmi ylläpitää edellä mainittuja invariantteja ja siirtää joukosta W alkioita joko joukkoon S tai F, riippuen siitä, onko alkio validi vai epävalidi. Riveillä 4–5 käsitellää perustapaus, jossa tyyppien välinen alityypitysrelaatio on suoraan havaittavissa. Tämä tarkoittaa sitä, että käytettävässä tyyppijärjestelmässä on voimassa perustyyppien välinen alityyppisuhde $(\tau \leq_{TS} \tau')$. Tällöin alkio $p = (\tau, \tau')$ voidaan siirtää validien parien joukkoon S.

Riveillä 6–11 alkio p sisältää kaksi funktiotyyppiä $\tau_1 \rightarrow \tau_2$ ja $\tau'_1 \rightarrow \tau'_2$. Alkio p lisätään validien alkioiden joukkoon S, jos epävalidien parien joukko F ei sisällä kumpaakaan pareista (τ'_1, τ_1) tai (τ_2, τ'_2). Tämän operaation tulkinta on juuri se, että tyyppipari p tulkitaan validiksi, olettaen että kaikki tyyppiparit joukossa W ovat valideja. Jos jompikumpi pareista (τ'_1, τ_1) tai (τ_2, τ'_2) tiedetään

Require: Joukko U tyyppipareja
Ensure: Alityypitysrelaation $\leq_{AC}$ kiintopiste S

1: $W := U, S := F := \emptyset$
2: **while** $W \neq \emptyset$ **do**
3: poista pari $p = (\tau, \tau')$ joukosta W
4: **if** p on muotoa $(\bot, \tau')$ tai $(\tau, \top)$ tai $(\tau \leq_{TS} \tau')$ **then**
5: lisää p joukkoon S
6: **else if** p on muotoa $(\tau_1 \to \tau_2, \tau'_1 \to \tau'_2)$ **then**
7: **if** $(\tau'_1, \tau_1) \notin F$ ja $(\tau_2, \tau'_2) \notin F$ **then**
8: lisää p joukkoon S
9: **else**
10: lisää p joukkoon F ja kaikki $q \in pred(p)$ joukosta S joukkoon W
11: **end if**
12: **else if** p on muotoa $(\Pi_{i=0}^{n-1}\tau_i, \Pi_{j=0}^{m-1}\tau'_j)$ **then**
13: **if** on olemassa bijektio $b : \{0, \ldots, m-1\} \to \{0, \ldots, n-1\}$, siten että $\forall j \in 0, \ldots, m-1, (\tau_{b(j)}, \tau'_j) \notin F$ pätee **then**
14: lisää p joukkoon S
15: **else**
16: lisää p joukkoon F ja kaikki $q \in pred(p)$ joukosta S joukkoon W
17: **end if**
18: **else**
19: lisää p joukkoon F ja kaikki $q \in pred(p)$ joukosta S joukkoon W
20: **end if**
21: **end while**

Taulukko 17: Joukon U alityypitysrelaation $\leq_{AC}$ kiintopisteen laskeva algoritmi

epävalidiksi, niin alkio p lisätään joukkoon F ja kaikki alkion p vanhemmat $q \in pred(p)$ siirretään joukosta S uudelleen validoitaviksi joukkoon W. Alkio q on alkion p vanhempi, jos ja vain jos q on joko tulo- tai funktiotyyppiä ja p esiintyy q:ta vastaavan tyypin säännön $EXPLORE-ARROW$ tai $EXPLORE-PI$ johtopäätöksessä.

Riveillä 12–17 käsitellään tapaus, missä p on tyyppipari $(\Pi_{i=0}^{n-1}\tau_i, \Pi_{j=0}^{m-1}\tau'_j)$. Tällöin alkio p tulkitaan validiksi, olettaen että joukon W alkiot ovat valideja, jos ja vain jos löytyy jokin injektiivinen kuvaus $b : \{0, \ldots, m-1\} \to \{0, \ldots, n-1\}$, siten että (τ_i, τ'_j) ei ole epävalidien parien joukossa F. Toisin sanoen kaikille tyypeille τ'_j täytyy löytää erilliset τ_i siten, että (τ_i, τ'_j) on validi.

Kuvauksen b etsimiseen tulotyyppien alkioiden välille käytetään maksimaalisen sovituksen laskemista kaksiosaisessa verkossa (U, V, E) (*maximum matching of bipartite graph*), kun $U = \tau_i$, $V = \tau'_j$ ja $E = (\tau_i \times \tau_j), i \in \{0, \ldots, n-1\}, j \in \{0, \ldots, m-1\}$. Kyseisen algoritmin aikavaatimus on luokkaa $\mathcal{O}(n^{5/2})$ [HK73]. Kaksiosaisen verkon $G = (U, V, E)$ kaarien joukko $M \subseteq E$ on sovitus, jos mikään verkon G solmu $s \in U \cup V$ ei liity useampaan kuin yhteen kaareen $e \in E$. Sovitus $M \subseteq E$ on maksimaalinen, jos kaikille sovituksille $M' \subseteq E$ on voimasssa $|M'| \leq |M|$. Tyyppiparin $(\Pi_{i=0}^{n-1}\tau_i, \Pi_{j=0}^{m-1}\tau'_j)$ validiksi todistamisessa etsitään verkolle $G = (U, V, E)$, $U = \tau_i$, $V = \tau'_j$, $E = (\tau_i \times \tau'_j)$ sellainen maksimaalinen sovitus $M \subseteq E$, että $(\tau, \tau') \in M$ jos ja vain $(\tau, \tau') \notin F$. Jos löydetään sovitus M, siten että $|M| = m$, niin p on validi. Muuten p ei ole validi ja se siirretään joukkoon F.

Rivillä 19 alkio p merkitään invalidiksi, jos se ei ole täyttänyt mitään edellisistä testeistä. Tämä tarkoittaa sitä, että p on esimerkiksi muotoa $(\Pi_{i=0}^{n-1}\tau_i, \tau'_1 \to \tau'_2)$, jolle ei ole määritetty alityypitys-

relaatiota.

Kun algoritmi lopettaa toimintansa, on joukko W tyhjä ja (S, F) on joukon U ositus. Jos $p_0 \notin F$, niin pari p_0 on validi [CPR04]. Algoritmin aikavaativuutta dominoi bijektion b löytäminen rivillä 13, jonka aikavaatimus on $\mathcal{O}(d^{5/2})$, missä $d = max(n, m)$. Lisäksi algoritmin pääsilmukka suoritetaan korkeintaan $size(U)$ kertaa, missä $size(U)$ on todistusvelvoitteiden joukon koko. Täten algoritmin pahimman tapauksen aikavaativuudeksi saadaan $\mathcal{O}(size(U)d^{5/2})$ [CPR04].

3.5 Syntaktisen yhteensopivuuden käyttökohteet

Ohjelmistokomponenttien syntaktisiin ominaisuuksiin perustuvia yhteensopivuustarkistuksia voidaan hyödyntää esimerkiksi uudelleenkäytössä ja komponenttikirjastojen selailussa, indeksöinnissä ja analyysissä [ZW95]. Sen avulla ohjelmiston kehittäjä voi esimerkiksi jakaa suuren komponenttikirjaston ekvivalenssiluokkiin syntaktisten ominaisuuksien mukaan tai tehdä siihen suoria hakuja. Syntaktista yhteensopivuuden tarkistusta voidaan käyttää suotimena: suuresta määrästä komponentteja rajataan ensin haulla pienempi potentiaalinen joukko, josta sitten voidaan hakea haluttu komponentti. Uudelleenkäytettäviä komponentteja on helpompi etsiä syntaktisten ominaisuuksien perusteella, varsinkin jos komponenteilla ei ole yhtenäistä tai tunnettua nimeämiskäytäntöä.

A. Moormann Zaremskin ja J.M Wingin tutkimusryhmä toteutti syntaktiseen yhteensopivuuteen perustuvan komponenttikirjaston hallintajärjestelmän [ZW95]. Komponenttikirjasto koostui 1451 SML-kielen funktiosta, jotka oli koostettu kolmesta erillisestä funktiokirjastosta. Järjestelmä toimi vain funktioiden tasolla, mutta menetelmä on suoraan yleistettävissä myös ympäristöihin, jotka sisältävät suurempia ohjelmistokomponentteja kuten moduuleita. Empiirisessä tutkimuksessa käytettiin järjestelmää uudelleenkäytettävien funktioiden etsimiseen, funktiokirjaston analysointiin, selailuun, suodattamiseen ja indeksöintiin.

Uudelleenkäytettävien funktioiden etsimisessä saadaan syntaktiseen yhteensopivuuteen perustuvassa haussa helposti liikaa vaihtoehtoja, varsinkin jos yhteensopivuusehtoja löyhennetään liikaa. Esimerkiksi ehto $\alpha \rightarrow \alpha$ uncurry-, -muuttujankorvaus ja uudelleenjärjestyslöyhennyksin palauttaisi komponenttikirjastosta kaikki mahdolliset funktiot. Liian tiukat yhteensopivuusehdot eivät nekään tuota aina haluttuja tuloksia.

Syntaktinen yhteensopivuus soveltuu komponenttikirjaston analysointiin ja erilaisten indeksien ja ekvivalenssiluokkien muodostamiseen. Analysoinnissa voidaan esimerkiksi hakea niiden funktioiden lukumäärä, jotka ottavat vastaan kaksi int-parametria ja palauttavat yhden int-muotoisen luvun. Suuri komponenttikirjasto on helpommin selailtavissa, jos käyttäjällä on käytettävissän työkalu, joka lajittelee komponentit luokkahierarkiaan niiden syntaksin perusteella ja antaa tämän luokkahierarkian selailuun sopivan, mahdollisesti graafisen, työkalun. Tällöin ohjelmiston kehittäjä voisi helposti kulkea luokkahierarkiassa eteenpäin esimerkiksi yleisemmältä parametritasolta spesifisempään ja siten löytää käyttökelpoisia komponentteja.

Tässä luvussa kuvaillut menetelmiä voidaan käyttää myös helpottamaan ohjelmistosuunnittelun työtä muutenkin, kuin pelkästään suoranaisen etsimisen muodossa. Korvautuvuustesteistä, jotka käyttivät assosiatiivis-kommutatiivista ekvivalenssia tai alityypitystä, saadaan “sivutuotteena” kahden komponenttikuvauksen välinen muuntokuvaus (bijektio b, katso esimerkiksi määritelmä 3.7). Tätä muuntokuvausta voidaan käyttää esimerkiksi komponenttiadapterien (puoli-) automaattiseen tuottamiseen.

Komponenttien syntaktiset kuvaukset saadaan suhteellisen helposti käyttöön, koska komponen-

tin toteutusvaiheessa rajapinta joka tapauksessa määritellään käytettävän ohjelmointikielen syntaksin mukaan. Rajapinnan syntaktinen kuvaus voidaan tämän jälkeen hakea lähdekoodista joko kääntäjän tukemana lisäominaisuutena [ZW95] tai jälkikäteen erillisellä työkalulla. Tämän jälkeen kuvaukset komponenttien rajapinnoista voidaan laittaa johonkin tietojenhallintajärjestelmään, esimerkiksi relaatiotietokantaan. Syntaktisen yhteensopivuuden tarkistus on sidottu käytettävän ohjelmointikielen tai muun sopivan ontologian käsitteisiin, muuten ei pystyttäisi tarkistamaan tyyppien, erityisesti perustyyppien kuten *int* ja *bool*, yhteensopivuutta. Jotta komponentteja voitaisiin käyttää ohjelmointikieliriippumattomasti, tulisikin rajapintakuvaus tallentaa mahdollisimman universaalissa muodossa, joka ei ole riippuvainen ohjelmointikielestä. Tähän käyttöön sopivat esimerkiksi rajapinnankuvauskielet, IDL:t, joiden avulla rajapinnan syntaksi voidaan määritellä ohjelmointikielestä riippumatta.

Syntaktisen yhteensopivuuden suurin puute on se, että komponentin rajapinnan tyyppi ei kerro juuri mitään sen toiminnallisuudesta. Jotta komponenttikirjastosta voitaisiin löytää uudelleenkäytettäviä komponentteja, tulisi hakumääreessä voida määrittelä myös komponentin toivottu toimintatapa. Kun hakumääreeseen sisällytetään kuvaus komponentin semantiikasta, voidaan syntaktisesti yhteensopivien komponenttien joukosta valita vain ne, jotka toimivat halutulla tavalla.

Syntaktisen yhteensopivuuden tarkistaminen on kevyempi operaatio kuin komponentin semantiikan tulkitseminen ja sen pohjalta tehtävä yhteensopivuustarkistus. Komponenttien hakuoperaatiossa voitaisiinkin käyttää syntaktista yhteensopivuutta esisuotimena, joka suodattaa pois ne komponentit, jotka eivät ole syntaktisten epäyhteensopivuuksien takia käyttökelpoisia. Tämän jälkeen tähän pienempään komponenttien osajoukkoon voitaisiin kohdistaa semanttisiin ominaisuuksiin perustuva haku. Komponenttien semanttisiin ominaisuuksiin tutustutaan luvussa 4.

4 Semanttinen yhteentoimivuus

Komponenttien syntaktinen yhteensopivuus on nykyään melko hyvin ymmärretty ja hallittu ongelma. Tuotantokäytössä on ollut jo useita vuosia komponenttialustoja, jotka perustuvat syntaktisesti yhteensopivien komponenttien hyväksikäyttöön. Tällaisia komponenttialustoja ovat muun muassa OMG:n CORBA, Sun Microsystemsin kehittämä EJB ja Microsoftin DCOM. Syntaktinen yhteensopivuus on kuitenkin riittämätön peruste komponenttien onnistuneelle yhteistyölle.

Syntaktisesti yhteensopivat komponentit voivat esimerkiksi olettaa toimivansa erilaisessa ympäristössä, jolloin näiden komponenttien yhteensovittaminen on joko hyvin hankalaa tai jopa täysin mahdotonta [BS00]. Jos esimerkiksi yhteistoiminta sujuu teknisellä tasolla oikein, mutta toinen komponenteista tulkitsee tietyn kokonaisluvun euroiksi ja toinen drakhmoiksi, ei lopputulos ole varmaankaan sellainen, mitä toivottiin.

Komponenttien semanttisella yhteensopivuudella tarkoitetaan sitä, että komponenttien toimintatavat ja toiminnan tarkoitukset ovat yhteensopivia. Tässä luvussa käsitellään komponenttien semantiikan kuvaamista. Komponenttien semantiikka määritellään sekä antamalla semanttiset tulkinnat komponentin metodeille sekä käytettäville perustyypeille.

Komponentin semanttiset kuvaukset määrittelevät operaatioiden toimintasemantiikan ja käytettävien peruskäsitteiden tulkinnat. Operaatioiden toimintasemantiikkaa kuvataan etu- ja jälkiehdoilla ja peruskäsitteiden tulkinta määritellään käyttäen joko niin kutsuttuja abstrakteja tieotyyppejä tai ontologioita.

Komponentin $\mathcal{C} = (\Sigma, \mathcal{P}, \mathcal{T})$ semantiikka määritellään tyypityskontekstissa $\mathcal{T}$. Tyypityskontekstin semanttiset määrittelyt voidaan jakaa kahteen osaan: toimintasemantiikkaan, joka määrittelee tulkinnat komponentin yksittäisille funktioille ja metodeille sekä käsitteiden semantiikka, joka määrittelee perustyyppien tulkinnan ja perustyyppien väliset tyypityssuhteet.

Komponentin toimintojen (metodien) semantiikan kuvaamiseen käytetään perinteistä etu- ja jälkiehtoihin perustuvaa kuvausta. Etuehdot määrittelevät metodin suorittamiselle välttämättömät kriteerit ja jälkiehdot määrittelevät metodin suorituksen jälkeisen maailman tilan. Komponentin toimintojen semantiikan kuvaamista käsitellään luvussa 4.1.

Komponenttien välisessä kommunikoinnissa ja metodikutsuissa käytettävän tietosisällön pitää myös olla semanttiselta sisällöltään yhteensopivaa. Tietosisältöjen semantiikkaa voidaan kuvata joko algebrallisesti tai ontologisesti. Algebralliset kuvaukset soveltuvat hyvin erilaisten tietorakenteiden, kuten listojen ja puiden, kuvaamiseen, koska niillä itsellään on algebrallinen luonne. Algebrallisiin tietotyyppeihin tutustutaan luvussa 4.2.

Ontologiset kuvaukset sopivat paremmin tilanteisiin, joissa tietyn käsitteen merkitys on sidottu johonkin viitekehykseen ja siinä esiintyviin suhteisiin. Ontologioilla voidaan kuvata merkityksiä tietorakenteille, jotka eivät ole matemaattisia luonteeltaan. Ontologian avulla voidaan esimerkiksi ilmaista, että "*Y on X:n sisko jos ja vain jos Y on nainen ja X:llä ja Y:llä on samat vanhemmat*". Ontologiat perustuvat perusmerkitysten ja niiden välisten suhteiden määrittelyihin. Ontologioiden perusteita ja käyttötapoja käsitellään luvussa 4.3.

4.1 Komponentin toimintasemantiikka

Kahden komponentin toimintatavat voivat erota toisistaan paljonkin, vaikka niiden syntaktiset rajapintakuvaukset olisivat täsmälleen samanlaiset. Ensinnäkin, komponentin tarjoamien funktioiden tarkoituksissa voi olla eroja. Esimerkiksi funktiotyyppi $Int \to Int$ sopii sekä yhteen-,

vähennys- ja jakolaskuun sekä äärettömään määrään muita funktioita, joiden tarkoitukset voivat olla täysin erilaiset. Toiseksi, vaikka kahdella funktiolla tyyppiä $Int \rightarrow Int$ sattuisikin olemaan sama tarkoitus, voi niillä olla erilaiset oletukset toiminnan oikeellisille edellytyksille. Toinen funktioista voi esimerkiksi olettaa, että Int-tyyppinen syöte on aina positiivinen, kun taas toinen ei rajoita syötteen arvoja millään tavalla.

Kun etsitään sopivaa palveluntarjoajaa tiettyyn tehtävään, ongelmaksi muodostuu sellaisen komponentin löytäminen, joka sopivan syntaktisen rajapinnan lisäksi myös toimii halutulla tavalla. Suurista komponenttikirjastoista voidaan löytää syntaksisesti yhteensopivia komponentteja, jotka eivät kuitenkaan ole keskenään toimintatavaltaan samankaltaisia [ZW97]. Tämän vuoksi komponenttikirjastosta pitäisi pystyä komponentteja erottelemaan toisistaan myös toimintasemantiikan tasolla. Komponentin toimintasemantiikka voidaan kuvailla (käytännön sanelemien rajojen sisällä) logiikan kaavoilla. Loogisten kaavojen ja teoreemantodistajien avulla voidaan komponenteista erottaa semanttisesta eri tavoin käyttäytyvät yksilöt.

Prosessin toimintasemantiikka kuvataan *etu-* ja *jälkiehdoilla*, jotka ovat predikaattilogiikan kaavoja [ZW97]. Etu- ja jälkiehdot P, Q sekä prosessi C muodostavat niin kutsutun Hoaren kolmikon, $\{P\}\ C\ \{Q\}$, joka ilmaisee että jos prosessi C suoritetaan ympäristössä, joka täyttää ehdot P ja C terminoi, niin silloin ympäristö täyttää ehdot Q [Cou90]. Hoaren kolmikkoja käytetään niin kutsutussa Hoaren logiikassa, joka kehitettiin erityisesti sarjallisten ohjelmien ominaisuuksien todistamiseen.

Etuehdot kuvaavat ympäristön tilaa ennen laskennan suoritusta. Esimerkiksi pinotietotyyppiin c kohdistuvan järjestysoperaation etuehtona voisi olla se, että pino ei ole tyhjä: $not(isEmpty(c))$. Jälkiehto kuvaa ympäristön tilaa laskennan jälkeen ja järjestysoperaation yhteydessä se tarkoittaa, että kaikkien alkioiden tulee olla järjestyksessä: $\forall(i,j), 0 < i < j < sizeOf(c) :\ c[i] \leq c[j]$.

Jotta toimintasemantiikan tarkistaminen olisi mahdollista, tulee tarkistajalla ja tarkistusympäristöllä olla käytössään yhteinen sanasto. Edellisessä järjestetyn pinon esimerkissä tulee predikaattien $sizeOf$ ja $isEmpty()$ olla tiedossa, jonka lisäksi alkioille tulee olla määriteltynä vertailuoperaattori $< ja \leq$ sekä pinolle määriteltynä indeksointioperaattori $[]$.

Sanastoon kuuluvat tiedot käytettävistä tyypeistä ja perusoperaatioista. Edellä mainitussa pinoesimerkissä esiintyvät ehdot ovat järkeviä vain, jos sekä tarkastajalla ja ympäristöllä on yhteinen tulkinta siitä, mitä tarkoittavat pino ja siihen kohdistuvat operaatiot kuten $isEmpty, sizeof$ ja indeksointioperaattori $[]$. Komponenttien semanttisen yhteensopivuuden määrittely voidaankin jakaa kahteen osaan: alimmalla tasolla määritellään tietotyypit ja niiden operaatiot ja näiden määritelmien pohjautuen määritellään komponenttien toimintasemantiikan kuvaukset.

Seuraavissa tarkasteluissa, jotka koskevat komponentin tarjoamien funktioiden semantiikkaa, oletetaan, että on annettu hyvin määritelty sanasto Σ, jota käyttäen toimintojen semanttiset kuvaukset voidaan määritellä.

Funktion spesifikaatio P sisältää toimintakuvauksen sekä etuehdot P_{pre} ja jälkiehdot P_{post} [ZW97]. Jos funktion etuehdot P_{pre} ovat voimassa funktiota kutsuttaessa, niin määritelmä takaa että funktiokutsun jälkeen ympäristö on tilassa P_{post}. Jos P_{pre} on epätosi, ei laskennan vaikutusta ympäristön tilaan voida taata. Esimerkiksi jakolaskualgoritmin f spesifikaatio $f(x) = \frac{1}{x}, f_{pre} : x \neq 0$ ei takaa oikeaa toiminnallisuutta jos $x = 0$.

Yleinen muoto kahden funktiospesifikaatioiden P ja Q yhteensopivuuden tarkistavalle sovituspredikaatille $match$ on annettu määritelmässä 4.1 [ZW97].

Määritelmä 4.1 (Semanttinen sovituspredikaatti $match$)

$$\begin{aligned} match_{pre/post}(P,Q) &= (Q_{pre}\ \mathcal{R}_1\ P_{pre}) \wedge (\hat{P}\ \mathcal{R}_2\ Q_{post}) \\ match_{pred}(P,Q) &= (P_{pred}\ \mathcal{R}\ Q_{pred}) \end{aligned}$$

missä $\mathcal{R}_1$, $\mathcal{R}_2$ on joko ekvivalenssi- tai implikaatiorelaatio, $\mathcal{R}$ voi olla näiden lisäksi myös käänteinen implikaatio ($\Leftarrow$) ja $\hat{P}$ on joko P_{post} tai $P_{pre} \wedge P_{post}$.

Etu- ja jälkiehtoihin perustuvaa sovituspredikaattia $match_{pre/post}$ käytetään silloin, kun etu- ja jälkiehdot ovat tiedossa. Yhteensopivuustarkastus $match_{pred}$ on käyttökelpoinen, jos funktioita P ja Q halutaan tai joudutaan tarkastelemaan loogisina kokonaisuuksina. Lisäksi $match_{pred}$-predikaattia voidaan käyttää myös silloin, kun funktiolle ei voida erikseen määritellä etu- ja jälkiehtoja, vaan niiden välillä on esimerkiksi tiukasti looginen riippuvuussuhde. $match_{pre/post}$ on tiukempi ehto funktioiden yhteensopivuudelle kuin $match_{pred}$ [ZW97].

Taulukossa 18 on lueteltuna mahdolliset yhdistelmät relaatioille $\mathcal{R}_1, \mathcal{R}_2$ ja predikaatille $\hat{P}$ sekä annettu etu-/jälkiehtoihin perustuvalle sovituspredikaatille $match_x$ kuvaava nimi [ZW97]. Predikaateissa $match_{plug-in-post}$ ja $match_{guarded-post}$ esiintyy symboli $*$, joka tarkoittaa sitä, että kyseisessä tapauksessa etuehtoja ei tarkasteta.

Predikaattisymboli	$\mathcal{R}_1$	$\mathcal{R}_2$	$\hat{P}$
$match_{E-pre/post}$	$\Leftrightarrow$	$\Leftrightarrow$	P_{post}
$match_{plug-in}$	$\Rightarrow$	$\Rightarrow$	P_{post}
$match_{plug-in-post}$	$*$	$\Rightarrow$	P_{post}
$match_{guarded-plug-in}$	$\Rightarrow$	$\Rightarrow$	$P_{pre} \wedge P_{post}$
$match_{guarded-post}$	$*$	$\Rightarrow$	$P_{pre} \wedge P_{post}$

Taulukko 18: Etu-/jälkiehtoihin perustuvat sovituspredikaatit $match_x$

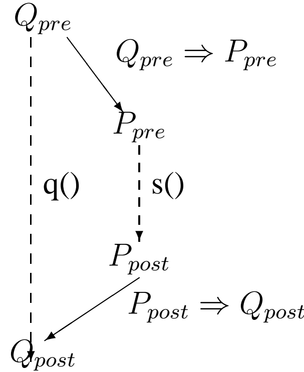
Taulukosta 18 nähdään, että esimerkiksi $match_{plug-in}$-tyyppisen sovituksen auki kirjoitettu predikaattifunktio on $match_{plug-in}(P,Q) = (Q_{pre} \Rightarrow P_{pre}) \wedge (P_{post} \Rightarrow Q_{post})$. Eksaktin yhteensopivuuden predikaattifunktio $match_{E-pre/post}$ on tosi, jos ja vain jos kahden funktiospesifikaation P ja Q etu- ja jälkiehdot ovat loogisesti ekvivalentit.

Vaatimus ekvivalenssista on liian tiukka vaatimus erityisesti silloin, kun etsitään sellaisia funktioita S, joilla voidaan korvata spesifikaation Q toteuttava funktio. Lisäksi looginen ekvivalenssi ei anna tilaa ohjelmistotuotannossa tarvittavalle perinnälle, joka yleensä ilmenee toiminnallisuuden ja siten semantiikan laajentamisena tai erikoistamisena. Taulukon 18 lopuissa predikaattifunktioissa ei etu- ja jälkiehtojen ekvivalenssia tästä syystä vaadita.

Erityisen mielenkiintoinen taulukossa 18 esiintyvistä sovitinpredikaateista on $match_{plug-in}$, joka ilmaisee niin kutsutun käyttäytymisen alityypitysrelaation (*behavioural subtyping*) [LW94]. Funktiot, jotka ovat käyttäytymisensä puolesta tämänlaisessa alityypitysrelaatiossa, voidaan korvata keskenään vaikka niiden etu- ja jälkiehdot eivät olekaan täysin ekvivalentit.

Sovituspredikaatin $match_{plug-in}$ intuitiivinen tulkinta on sellainen, että funktiota P voidaan kutsua aina, jos funktiota Q:kin voidaan kutsua. Toisaalta, kun P on onnistuneesti lopettanut suorituksensa, se jättää ympäristön tilaan, jossa ainakin kaikki Q:n vaatimat jälkiehdot ovat voimassa. Jos $match_{plug-in}(P,Q)$ on tosi, niin funktio Q voidaan korvata funktiolla P mutta ei välttämättä toisin päin. $Q_{pre} \Rightarrow P_{pre}$ takaa, että Q:n vaatimukset ovat P:n vaatimusten osajoukko eli jos Q:n

vaatimukset pätevät niin myös P:n vaatimukset pätevät. $P_{post} \Rightarrow Q_{post}$ takaa sen että P:n suoritus palauttaa ympäristön tilaan joka on on myös Q:n suorituksen hyväksyttävä lopputila. Kuvassa 8 on kuvattuna $match_{plug-in}$-yhteensopivuuden toiminnallinen tulkinta, jossa P on funktion $p()$ spesifikaatio ja Q funktion $q()$ spesifikaatio.



Kuva 8: $match_{plug-in}$ predikaatin toiminnallinen tulkinta

Joissain tapauksissa funktioiden etuehdot S_{pre}, Q_{pre} voidaan olettaa merkityksettömiksi yhteensopivuuden kannalta. Tällöin halutaan löytää yhteensopivia funktioita, jotka päätyvät samaan tilaan riippumatta niiden vaatimuksista. Yksinkertaisin esimerkki funktioista, joille ei yleensä määritellä etuehtoja, ovat niin kutsutut aksessorifunktiot, jotka vain palauttavat informaatiota. Taulukossa 18 etuehtojen jättäminen pois yhteensopivuustarkistuksesta on merkitty tähdellä (*).

Alityypityksen mahdollistavan $plugin$-sovitinpredikaattien lisäksi on taulukossa 18 määritelty kaksi versiota sovitinpredikaateista, jotka ilmaisevat niin kutsutun invariantin ominaisuuden. Joidenkin funktioiden toiminnallisuus perustuu siihen, että ne ylläpitävät *invarianttia*. Invariantti on (ympäristön) ominaisuus, joka ei saa muuttua laskentojen välillä. Esimerkiksi järjestetylle jonolle invariantti voisi olla ominaisuus joka ylläpitää jonon alkioiden nousevaa järjestystä: $A = \forall i, j : 0 \leq i < j < q.length() : q[i] \leq q[j]$. Tämän invariantin tulee olla voimassa aina, toisin sanoen kaikille q:lle määritetyille funktioille etu- ja jälkiehdossa tulee ominaisuuden A olla voimassa. Invariantteja varten funktioiden yhteensopivuuspredikaateille on määritelty $guarded$-variaatiot $match_{guarded-plug-in}$ ja $match_{guarded-post}$.

Funktioiden toimintasemantiikkoja voidaan vertailla myös kaavojen $P_{pred} = P_{pre} \Rightarrow P_{post}$ ja $Q_{pred} = Q_{pre} \Rightarrow Q_{post}$ välisenä loogisena relaationa $match_{pred} = P_{pred} \mathcal{R}\ Q_{pred}$. Tarvittaessa P_{pred} tai Q_{pred} voidaan määritellä myös konjunktioksi $P_{pred} = P_{pre} \wedge P_{post}$, joka on implikaatiota tiukempi ehto. Konjunktiomuodolle on käyttöä erityisesti silloin kun etuehto tulkitaan jonkin toiminnan vahdiksi (*guard*) [ZW97].

Jos $\mathcal{R}$ on looginen ekvivalenssirelaatio, niin funktio $match_{pred}$ on tosi, jos ja vain jos vertailtavien funktioiden S ja Q toimintasemantiikka on täysin ekvivalenttia. Tällöin funktiot ovat keskenään korvattavissa toisillaan toimintasemantiikan tasolla. Kun $\mathcal{R}$ on looginen implikaatio $\Rightarrow$, palauttaa $match_{pred}$ arvon tosi, jos funktion Q spesifikaatio Q_{pred} on toimintasemantiikaltaan *yleisempi* kuin P:n spesifikaatio P_{pred}. Tällöin voidaan sanoa, että P on toimintasemanttisesti tarkasteltuna Q:n alityyppi ja siten funktio Q on korvattavissa funktiolla P. Jos relaatio $\mathcal{R}$ on käänteinen implikaatio $\Leftarrow$, niin $match_{pred}$ on tosi, jos P on Q:n implementaatio eli jos Q on toimintasemantiikaltaan P:n alityyppi. Taulukossa 19 on lueteltuna kaikki $match_{pred}$ tyyppiset yhteensopivuusfunktiot.

Funktioiden ja metodien toimintasemantiikkaan perustuvaa yhteensopivuutta voidaan käyttää hy-

Predikaattisymboli	$\mathcal{R}$
$match_{E-pred}$	$\Leftrightarrow$
$match_{gen-pred}$	$\Rightarrow$
$match_{spcl-pred}$	$\Leftarrow$

Taulukko 19: $match_{pred}$ tyyppiset yhteensopivuusfunktiot [ZW97]

väksi ohjelmistotuotannon aikana ja rajatussa määrin myös ajon aikaisesti. Ohjelmistotuotannossa tässä luvussa kuvattuja toimintoja tukevaa työkalua voidaan käyttää sopivien ohjelmistokomponenttien etsintään komponenttikirjastosta. Erityisesti $guarded - plugin$-tyyppiset haut ovat hyödyllisiä, koska ne formaalilla tavalla määrittelevät komponenttien toimintasemanttisen alityyppirelaation [ZW97]. Komponenttikirjastosta voidaan $guarded - plugin$-tyyppisellä haulla hakea komponentteja, joilla voidaan korvata tietyn spesifikaation Q toteuttava komponentti.

Toimintasemantiikan tarkistaminen suoritetaan loogisia teoreemoja todistamalla. Teoreeman todistaminen on hyvin raskas prosessi eikä yleisessä tapauksessa ole algoritmisesti ratkeava ongelma (esimerkiksi luonnollisten lukujen algebran epätäydellisyys). Tietotyyppien yhteydessä on usein luonnollista määritellä äärettömiä tyyppejä, kuten listoja. Tällaisten tietotyyppien tarkistaminen voi vaatia teoreemantodistajalta hyvin monimutkaista ja raskasta heuristiikkaa.

Yleinen teoreemantodistaja vaatiikin joissain tapauksissa apua käyttäjältä. Näiden syiden takia toimintasemantiikan ajonaikainen tarkistaminen ei useimmissa tapauksissa ole järkevää, varsinkaan silloin jos järjestelmällä on jonkinlaisia ajoitusvaatimuksia. Toimintasemantiikan algoritminen tarkistaminen onnistuu vain erikoistapauksissa, jotka voivat esimerkiksi kieltää äärettömät tyypit tai rajoittavat spesifioinnissa käytettävää algebraa. Komponentin toimintasemantiikan tarkistaminen tulisikin suorittaa vain mahdollisimman pienelle joukolle potentiaalisia komponentteja. Etsintäjoukkoa voidaan rajoittaa esimerkiksi suorittamalla komponenttien esikarsinta syntaktisen yhteensopivuuden perusteella [ZW97, ZW95].

4.2 Abstraktit tietotyypit

Eri ohjelmistokomponentit voivat olla syntaktisesti yhteensopivia, vaikka ne eivät sitä olisikaan toiminnaltaan. Jos esimerkiksi etsitään palvelua, joka lisää kokonaisluvun jonoon, niin halutaan palvelu, jonka syntaktinen rakenne on muotoa $Int \rightarrow List$. Tämä ei tietenkään ole riittävä ehto oikean palvelun löytämiseksi, sillä pelkästään syntaksin perusteella ei pystytä päättelemään, onko $List$ jono- tai pino-tyyppinen lista vai onko se lista ollenkaan. Tietotyyppien yhteensopivuuden tarkistamiseksi perustyyppien ominaisuudet ja tyypityssuhteet tulee pystyä määrittelemään ja tarkistamaan formaalisti.

Tietotyypit ja niihin kohdistuvat operaatiot voidaan määritellä *abstraktien tietotyyppien* (ADT, *Abstract Data Type*) avulla. ADT:t perustuvat matemaattisen universaalialgebran teorioihin, jotka tutkivat algebrallisten rakenteiden ominaisuuksia. Tietotyypin määrittely eli algebra muodostetaan käyttäytymisen abstrahoinnin avulla [EM85]. Tietotyypin algebrallinen kuvaus sisältää vakioiden ja operaattoreiden symbolit sekä operaattoreiden sekä operaattoreiden aksioomat [EM85].

Taulukossa 20 on määriteltynä tietotyyppi *bool* ja pinotietotyyppi $stack$ Ehrigin kehittämän ACT-ONE -kielen syntaksilla ilmaistuna [EM85]. Abstraktin tietotyypin määrittely alkaa luettelemalla käytettävien tietotyyppien nimet kohdassa $sorts$. Operaatioiden tyypit annetaan $opns$-määreen

jälkeen. Operaatioiden toimintasemantiikka eli algebran aksioomat määritellään $eqns$-osiossa ekvivalenssilausekkeina.

$bool =$		
	$sorts:$	$bool$
	$opns:$	$TRUE, FALSE \to bool$
		$NOT : bool \to bool$
		$AND : bool\ bool \to bool$
	$eqns:$	$b \in bool$
		$NOT(TRUE) = FALSE$
		$NOT(NOT(b)) = b$
		$b\ AND\ TRUE = b$
		$b\ AND\ FALSE = FALSE$

$stack =$		
	$sorts:$	$alphabet$
		$stack$
	$opns:$	$K_1, \ldots, K_N :\to alphabet$
		$EMPTY :\to stack$
		$ERROR :\to alphabet$
		$PUSH : alphabet\ stack$
		$\to stack$
		$POP : stack \to stack$
		$TOP : stack \to alphabet$
	$eqns:$	$x \in alphabet, s \in stack$
		$POP(PUSH(x, s)) = s$
		$TOP(PUSH(x, s)) = x$
		$POP(EMPTY) = EMPTY$
		$TOP(EMPTY) = ERROR$

Taulukko 20: ACT-ONE -kielellä määritellyt tietotyypit $bool$ ja $stack$

Abstraktien tietotyyppien avulla ohjelmointikielissä käytettävät tietotyypit ja niihin kohdistuvat operaatiot voidaan palauttaa matemaattisen algebran teorioihin. Tällä saavutetaan se etu, että tietotyyppien ja operaatioiden yhteensopivuus saadaan formaalisti tarkistettua, eikä tietyn ympäristön kaikilla toimijoilla tarvitse olla samaa nimeämiskäytäntöä eri tietotyyppien suhteen.

ADT:n avulla voidaan esimerkiksi kaksi erinimistä tietotyyppiä $Stack$ ja $MyStackX$ todeta samoiksi tietorakenteiksi, jos niiden abstraktien tietotyyppien määrittelemät algebrat ovat symboleiden nimeämistä vaille isomorfisia [EM85]. Abstraktien tietotyyppien avulla voidaan kuvata yksinkertaisten tietotyyppien kuten kokonaislukujen, rationaalilukujen ja totuusarvojen lisäksi myös monimutkaisempia tietorakenteita kuten pinoja, jonoja, tekstijonoja, joukkoja, puita ja verkkoja [EM85].

Abstraktien tietotyyppien käyttöön liittyy kaksi suurta ongelmaa. Ensimmäinen on teoreettinen ja se koskee abstraktien tietotyyppien ja yleisemmin, algebrojen laskettavuutta: on olemassa useita, täysin luonnollisia algebroja, joissa termien ekvivalenssia ei voida todistaa, koska niitä ei voida saattaa normaalimuotoon [DJ90]. Esimerkiksi luonnollisten lukujen joukossa määriteltyjä lausekkeita ei voida todistaa ekvivalenteiksi äärellisellä aksiomatisoinnilla.

Toinen, hieman käytännöllisempi ongelma abstraktien tietotyyppien käytössä liittyy niiden määrittelemiseen ja käyttöön. ADT:n määritteleminen on vaikeaa ja niiden käyttäminen vaatii tiettyä matemaattista kypsyyttä. Ei voida yleisesti olettaa, että kaikki ihmiset osaisivat määritellä tarvittavat matemaattiset ominaisuudet tarvittavalle tietotyypille. Toiseksi, suurin osa käytännön elämässä tarvittavista tiedoista eivät ole matemaattisia luonteeltaan, vaan ne ovat ennemminkin symboleita tietyille ihmisten välisille vuorovaikutussuhteille ja niissä tarvittavalle tiedonvaihdolle. On hyvin hankalaa pelkästään matemaattista sanastoa käyttämällä erottaa toisistaan käsitteet “Aamutähti” ja “Iltatähti”, jotka molemmat viittaavaat samaan Venus-planeettaan. Tällaisten asioiden viittaamiseen ja käsittelemiseen tarvitaan käsitteitä, jotka ovat vähemmän formaaleita ja joihin voidaan

liittää tietynlaista, yleisesti hyväksyttyä tulkintaa.

4.3 Ontologiat

Ontologioiden avulla voidaan määritellä käsitteitä ja niiden välisiä suhteita kuvaavia sanastoja. Abstrakteista tietotyypeistä poiketen voidaan ontologioilla määritellä myös ei-matemaattisia ominaisuuksia, joihin liittyy luonnollista, "arkipäiväistä" tulkintaa. Tulkinnan käyttökelpoisuus perustuu siihen, että tietylle sanastolle ja sen tulkinnalle saadaan yhteisön hyväksyntä. Tällä tavoin esimerkiksi pankkipalvelua etsittäessä voidaan etsiä palveluita, joista saadaan "Rahaa" ja "Tilitietoja", sen sijaan että saataisiin *Int*-lukuja tai tietyn muotoisia tietotyyppejä. Tulkinnan mukana tietenkin häviää myös matematiikkaan perustuva todistusvoima, mutta käytännössä, tietyn sovellusalueen sisällä tämä ei ole ongelma.

Ontologialla tarkoitetaan tietyn toimialueen tai alan jaettua käsitteistöä ja tietämystä. Ontologia on jonkin maailman looginen teoria, joka muodostuu joukosta käsitteitä ja loogisesta kielestä, joka sitoo käsitteitä toisiinsa [Mae02]. Voidaan sanoa, että ontologia eksplisiittisesti käsitteellistää maailman jonkin näkökulman kautta [UG96].

Ontologia voidaan määritellä formaalisti struktuurina, joka sisältää joukon käsitteitä, joukon käsitteitä luokittelevia relaatioita, käsitteiden taksonomisen luokittelun sekä ontologian aksioomat. Ontologian formaali määritelmä on kuvattuna määritelmässä 4.2 [Mae02].

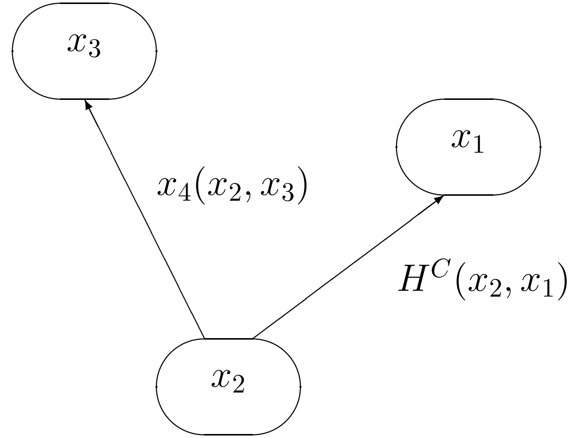
Määritelmä 4.2 (Ontologia)
Ontologia on monikko $\mathcal{O} = \{\mathcal{C}, \mathcal{R}, \mathcal{H}^{\mathcal{C}}, rel, \mathcal{A}^{\mathcal{O}}\}$*, missä*

- $\mathcal{C}$ *on käsitteiden joukko*
- $\mathcal{R}$ *on käsitteiden välisten suhteiden joukko*
- $\mathcal{H}^{\mathcal{C}}$ *on käsitteiden välisten (taksonomisten) hiearkioiden joukko, missä* $\mathcal{H}^{\mathcal{C}}(C_1, C_2)$ *tarkoittaa, että käsite* C_1 *on käsitteen* C_2 *alikäsite*
- $rel : \mathcal{R} \rightarrow \mathcal{C} \times \mathcal{C}$ *on funktio, joka sitoo käsitteitä yhteen ei-taksonomisesti. Jos käsitteet* C_1 *ja* C_2 *ovat* R*-suhteessa toisiinsa, niin* $rel(R) = (C_1, C_2)$*, tätä voidaan merkitä myös* $R(C_1, C_2)$*.*
- $\mathcal{A}^{\mathcal{O}}$ *on joukko ontologian aksioomia, jotka on ilmaistu sopivaa loogista kieltä, esimerkiksi predikaattilogiikkaa, käyttäen.*

Ontologian aksioomat ovat esimerkiksi predikaattilogiilla määriteltyjä lausekkeita, jotka lisäävät ontologisen käsitteistön ja tietämyksen ilmaisuvoimaa. Ontologisiin aksioomiin voisi esimerkiksi kuulua sääntö, joka ilmaisee käsitteen "Sisko" merkityksen henkilösuhteita koskevassa ontologiassa: "*Y on X:n sisko jos ja vain jos Y on nainen ja X:llä ja Y:llä on sama vanhempi*". Tällaisen relaation esittäminen uusia käsitteitä ja suhteita määrittelemällä voi olla hyvinkin hankalaa, jos tarkastelun alaiset käsitteet eivät ole suoranaisesti yhteydessä toisiinsa. Ontologian aksioomat voidaan tulkita käsitteiden välisiksi "dynaamisiksi" suhteiksi tai niiden aihioiksi, jotka luovat uusia käsitteiden välisiä suhteita monimutkaisempien loogisten päättelyiden avulla.

Ontologian käsitteet ja niiden väliset suhteet ovat siis nimeämättömiä maailman objekteja. Määritellään esimerkkiontologia $\mathcal{O}_{esim}$ siten, että $\mathcal{C} = \{x_1, x_2, x_3\}$, $\mathcal{R} = \{x_4\}$, $\mathcal{H}^{\mathcal{C}} = \{(x_2, x_1)\}$

ja $rel(x_4) = (x2, x3)$. Edellä kuvatun ontologian $\mathcal{O}_{esim}$ määrittelemän käsitteiden ja relaatioiden rakenne on mallinnettuna kuvassa 9, jossa ovaalit kuvaavat käsitteitä ja niiden väliset nuolet relaatioita.



Kuva 9: Nimeämättömän ontologian $\mathcal{O}_{esim}$ rakenne

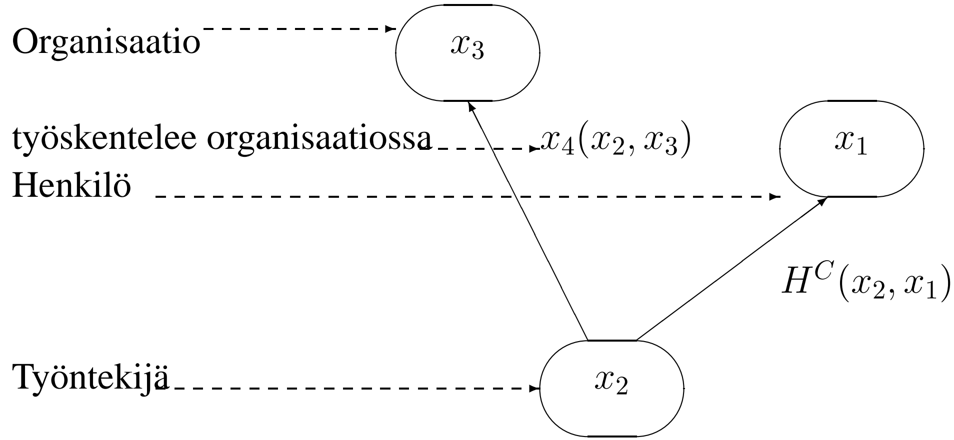
Tällaisena nimeämättömien käsitteiden hierarkiana ontologioita käsitellään vain koneellisesti. Jotta ontologiat olisivat ihmiselle hyödyllisiä sekä käyttö- että määrittelyvaiheessa, käsitteitä ja relaatioita käsitellään yleensä niihin liittyvien viitteiden eli nimien kautta. Tätä varten tarvitaan sanasto, joka luettelee ontologiassa $\mathcal{O}$ käytettävät sanat sekä liittää ne ontologian käsitteisiin ja relaatioihin. Jokainen sanaston alkio voi viitata useampaan käsitteeseen tai relaatioon, ja toisaalta yhteen käsitteeseen tai relaatioon voi olla useampiakin viitteitä [Mae02]. Sanaston kuvaus on annettu määritelmässä 4.3.

Määritelmä 4.3 (Ontologiaan liittyvä sanasto)
Ontologiaan $\mathcal{O}$ liittyvä sanasto $\mathcal{L}$ on monikko $\mathcal{L} = \{\mathcal{L}^{\mathcal{C}}, \mathcal{L}^{\mathcal{R}}, \mathcal{F}, \mathcal{G}\}$, missä

- $\mathcal{L}^{\mathcal{C}}$ *on joukko käsitteitä koskevan sanaston alkioita.*
- $\mathcal{L}^{\mathcal{R}}$ *on joukko suhteita koskevan sanaston alkioita.*
- $\mathcal{F} \subseteq \mathcal{L}^{\mathcal{C}} \times \mathcal{C}$ *on relaatio, jonka alkioita kutsutaan käsitteiden viittauksiksi.* $\mathcal{F}$ *sitoo tietyn käsitteiden sanaston alkion* $\mathcal{L}^{\mathcal{C}}$ *sitä vastaavaan käsitteeseen* $\mathcal{C}$*. Relaation* $\mathcal{F}$ *avulla voidaan määritellä funktiot* $\mathcal{F}(L) = \{C \in \mathcal{C} \mid (L, C) \in \mathcal{F}\}$*, kun* $L \in \mathcal{L}^{\mathcal{C}}$ *ja*
  $\mathcal{F}^{-1}(C) = \{L \in \mathcal{L}^{\mathcal{C}} \mid (L, C) \in \mathcal{F}\}$*, kun* $C \in \mathcal{C}$*.*
- $\mathcal{G} \subseteq \mathcal{L}^{\mathcal{R}} \times \mathcal{R}$ *on relaatio, jonka alkioita kutsutaan suhteiden viittauksiksi.* $\mathcal{G}$ *sitoo tietyn suhteiden sanaston alkion* $\mathcal{L}^{\mathcal{R}}$ *sitä vastaavaan suhteeseen* $\mathcal{R}$*. Relaation* $\mathcal{G}$ *avulla voidaan määritellä funktiot* $\mathcal{G}(L) = \{R \in \mathcal{R} \mid (L, R) \in \mathcal{G}\}$*, kun* $L \in \mathcal{L}^{\mathcal{R}}$ *ja*
  $\mathcal{G}^{-1}(R) = \{L \in \mathcal{L}^{\mathcal{R}} \mid (L, R) \in \mathcal{G}\}$*, kun* $R \in \mathcal{R}$*.*

Nimetty ontologia on pari $(\mathcal{O}, \mathcal{L})$, missä $\mathcal{O}$ on ontologia ja $\mathcal{L}$ siihen liittyvä sanasto. Edelliseen määritelmään perustuen voidaan ontologialle $\mathcal{O}_{esim}$ antaa sanasto, jonka rakenne on seuraavanlainen:
$\mathcal{L}^{\mathcal{C}} = \{$”Henkilö”,”Työntekijä”,”Organisaatio”$\}$,
$\mathcal{L}^{\mathcal{R}} = \{$“työskentelee organisaatiossa”$\}$,

$\mathcal{F}$(“Henkilö”) $= x_1$, $\mathcal{F}$(“Työntekijä”) $= x_2$, $\mathcal{F}$(“Organisaatio$''$) $= x_3$ ja
$\mathcal{G}$(“työskentelee organisaatiossa$''$) $= x_4$.
Nimetty ontologia $(\mathcal{O}_{esim}, \mathcal{L}_{esim})$ muodostaa rakenteen, joka on havainnollistettu kuvassa 10.



Kuva 10: Nimetyn ontologian $(\mathcal{O}_{esim}, \mathcal{L}_{esim})$ rakenne

Nimetyt ontologiat toimivat kehikkona, jonka varaan tiettyyn toimialueeseen, esimerkiksi pankkitoimintaan, liittyvä tietämys voidaan mallintaa. Ontologiat määrittelevät maailman rakenteen ja ovat luonteeltaan staattisia. Tietämystä ylläpidetään tietämyskannoissa, jotka kuvaavat dynaamista maailman tilaa. Ontologian $\mathcal{O}$ mukainen tietämyskanta sisältää joukon käsitteiden instansseja sekä kaksi instantiointifunktiota, jotka ovat relaatioita reaalimaailman instanssien joukoista ontologion $\mathcal{O}$ käsitteisiin $\mathcal{C}$ ja relaatioihin $\mathcal{R}$. Tietämyskannan rakenne on annettu määritelmässä 4.4 [Mae02].

Määritelmä 4.4 (Tietämyskanta $\mathcal{KB}$)
Tietämyskanta on monikko $\mathcal{KB} = \{\mathcal{O}, \mathcal{I}, inst, instr\}$, missä

- *$\mathcal{O}$ on ontologia $\mathcal{O} = \{\mathcal{C}, \mathcal{R}, \mathcal{H}^{\mathcal{C}}, rel, \mathcal{A}^{\mathcal{O}}\}$.*
- *$\mathcal{I}$ on joukko instansseja.*
- *$inst : \mathcal{C} \rightarrow 2^{\mathcal{I}}$ on käsitteiden instantiointifunktio, joka sitoo joukon instansseja tiettyyn ontologian käsitteeseen. Jos I on niiden instanssien joukko, jotka edustavat käsitettä C, niin merkitään $inst(C) = I$. Tämä voidaan ilmaista myös muodossa $C(I)$.*
- *$instr : \mathcal{R} \rightarrow 2^{\mathcal{I} \times \mathcal{I}}$ on relaatioiden instantiointifunktio. Jos instanssit I_i ja I_j on määritelty olevan ontologian suhteessa R, niin merkitään $instr(R) = (I_i, I_j)$, joka voidaan ilmaista myös muodossa $R(I_i, I_j)$.*

Tietämyskanta sisältää kaiken tiedon, mitä sen hetkisestä maailmasta voidaan ilmaista ontologian määrittelemillä käsitteillä. Oletetaan, että ontologiaa $\mathcal{O}_{esim}$ koskevassa maailmassa on olemassa kaksi instanssia $i_1, i_2 \in \mathcal{I}$. Tällöin voidaan määritellä, että i_1 on käsitteen x_2 instanssi ja i_2 on käsitteen x_3 instanssi, joka voidaan merkitä myös $x_2(i_1)$ ja $x_3(i_2)$. Määritellään, että pari (i_1, i_2) on relaation x_4 instanssi. Tämä on eräs tietämyskannan $\mathcal{KB}_{esim}$ instantiointi.

Kuten ontologioille, myös tietämyskannoille määritellään aina sanasto, joka liittää käsitteiden ja niiden välisten suhteiden instansseihin nimet. Tietämyskantaan liitetään sanasto määritelmässä 4.5 kuvatulla tavalla [Mae02].

Määritelmä 4.5 (Tietämyskantaan liittyvä sanasto $\mathcal{L}^{\mathcal{KB}}$)
Tietämyskantaan $\mathcal{KB}$ liittyvä sanasto $\mathcal{L}^{\mathcal{KB}}$ on monikko $\mathcal{L}^{\mathcal{KB}} = \{\mathcal{L}^{\mathcal{I}}, \mathcal{J}\}$, missä

- *$\mathcal{L}^{\mathcal{I}}$ on joukko sanaston alkioita*
- *$\mathcal{J} \in \mathcal{L}^{\mathcal{I}} \times \mathcal{I}$ on viiterelaatio, joka sitoo sanaston alkiot tietämyskannan instansseihin. Relaation $\mathcal{J}$ avulla voidaan määritellä funktiot*
  $\mathcal{J}(L) = \{I \in \mathcal{I} \mid (L, I) \in \mathcal{J}\}$ ja
  $\mathcal{J}^{-1}(I) = \{L \in \mathcal{L}^{\mathcal{I}} \mid (L, I) \in \mathcal{J}\}$

Nimetty tietämyskanta on pari $(\mathcal{KB}, \mathcal{L}^{\mathcal{KB}})$. Edellä kehiteltyyn esimerkkitapaukseen voidaan liittää vielä tietämyskannan nimentä $\mathcal{L}^{\mathcal{KB}}_{esim}$, siten että $\mathcal{L}^{\mathcal{I}} = \{$"Nokia","Tiina"$\}$, $\mathcal{J}($"Nokia"$) = i_2$ ja $\mathcal{J}($"Tiina"$) = i_1$. Tällöin nimetty tietämyskanta $(\mathcal{KB}_{esim}, \mathcal{L}^{\mathcal{KB}}_{esim})$ sisältää tiedon tietyn maailman konfiguraatiosta, jossa "*Tiina on henkilö ja työntekijä, joka työskentelee organisaatiossa nimeltä Nokia*".

Tietämyskannan instantiointifunktiot voidaan myös tulkita predikaateiksi, jolloin instantiointifunktiot ovat yksipaikkaisia ja relaatioiden instantiointifunktiot kaksipaikkaisia predikaatteja. Jos esimerkiksi halutaan tietää, onko instanssi i_1 käsitteen x_2 mukainen, niin voidaan kysyä, onko $x_2(i_1)$, sanastoa käyttäen *Työntekija("Tiina")*, tosi. Predikaatteja voidaan käyttää myös hakufunktioina. Jos halutaan tietää kaikki instanssit, jotka työskentelevät organisaatiossa i_2, niin tämä saadaan selville kyselyllä $x_4(_, i_2)$, sanastoa käyttäen *työskentelee yrityksessä(_,"Nokia")*.

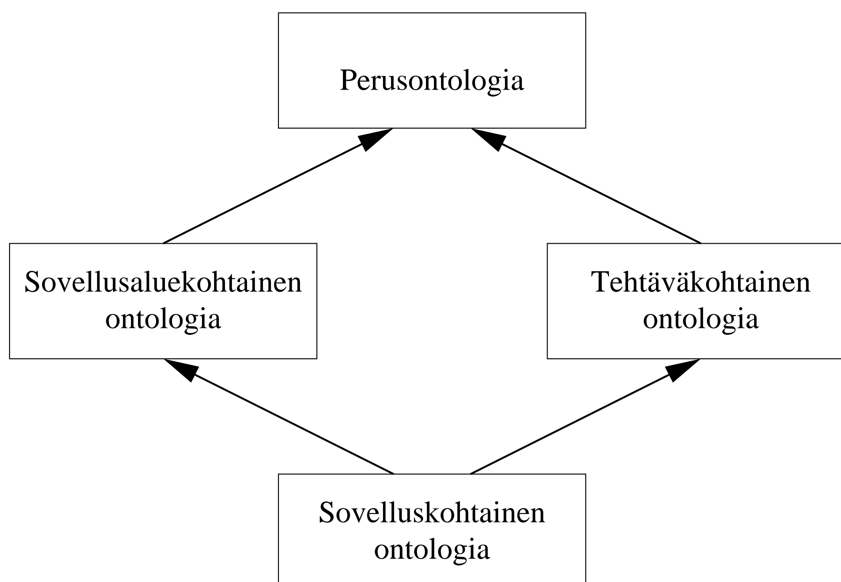
Ontologioita käytetään tyypillisesti useassa kerroksessa. Alimmassa kerroksessa määritellään yleisiä käsitteitä ja suhteita, jotka ovat mahdollisimman paljon sovellusalueriippumattomia [Mae02]. Tällaisia käsitteitä ja suhteita ovat esimerkiksi aikaan, paikkaan ja tapahtumiin liittyvät asiat. Sovelluskehityksessä alin ontologiataso voi määritellä esimerkiksi mitä ovat metodit, luokat ja niiden väliset hierarkiat. Esimerkkiontologiassa voitaisiin perustason käsitteisiin lukea käsitteet x_1*: Henkilö* ja x_3*: Organisaatio*, jotka voisivat määritellä oman aliontologiansa.

Perusontologian käsitteitä ja suhteita erikoistamalla voidaan kehittää sovellusalue- ja tehtäväkohtaisia ontologioita [Mae02]. Esimerkkiontologian voidaan olettaa esittävän sovellusaluekohtaista ontologiaa, joka on kehitetty esimerkiksi yritysten henkilöstöhallinnon tarpeisiin. Tällöin *Henkilö*- ja *Organisaatio*-ontologioita käyttäen on luotu *Henkilöstohallinto*-ontologia. Tämä uusi ontologia määrittelee yhden uuden käsitteen (x_2*: Työntekijä*) ja kaksi käsitteiden välistä suhdetta: x_4*: työskentelee yrityksessä* ja *Työntekijä on henkilö*-suhteen ($H^C(x_2, x_1)$).

Sovellusalue- ja tehtäväkohtaisia ontologioita käyttämällä voidaan muodostaa sovelluskohtaisia ontologioita [Mae02]. Esimerkkiontologiasta *Henkilöstöhallinto* voitaisiin erikoistamalla ja laajentamalla kehittää tietyn yrityksen tai organisaatiorakenteen (osakeyhtiö, kommandiittiyhtiö, jne.) tarpeisiin sopiva käsitekehikko. Edellä kuvailtu tapa jäsentää eri ontologioita kerrosrakenteeksi on kuvattu kuvassa 11 [Mae02].

Ontologioiden ja tietämyskantojen rakenteiden kuvaamiseen tarvitaan kieli. Eräs tällaisiin tarkoituksiin soveltuvista kielistä on RDF- ja siihen perustuva RDF-Schema -kieli, jotka ovat W3C-organisaation standardiluonnoksia [W3C04b, W3C04c]. Molemmat kielet perustuvat XML-standardeihin.

RDF (*Resource Description Language*) on XML-pohjainen kieli, joka on kehitetty erityisesti Web-resurssien kuvaamista varten. Tämä tarkoittaa sitä, että RDF-kielessä resurssit (käsitteet) ja niiden ominaisuudet (käsitteiden väliset suhteet) kuvataan ja identifioidaan URI:en (*Uniform Resource*



Kuva 11: Eräs ontologioiden jäsennys kerrosrakenteeksi

Identifier) avulla [W3C04b]. Resurssit voidaan rinnastaa ontologioiden käsitteisiin sekä niiden instansseihin ja resurssien ominaisuudet ontologisten käsitteiden välisiin suhteisiin.

Käsitteiden ja niiden välisten suhteiden esittämiseen RDF:ssä käytetään kolmikkoa subjekti, predikaatti, objekti, jossa subjekti ja objekti voidaan mieltää ontologian käsitteiksi ja predikaatti niiden väliseksi suhteeksi tai käsitteen ominaisuudeksi [Mae02]. RDF ei itsessään anna mahdollisuutta määritellä esimerkiksi tietotyyppejä muuten kuin eri URI-resursseihin viittaamalla. Taulukossa 21 on esimerkki siitä, miltä instantoitu ontologian tietämys voisi RDF-kielellä ilmaistuna näyttää. Eri ontologioiden käsitteet on ilmaistu XML-standardin nimiavaruuksien avulla.

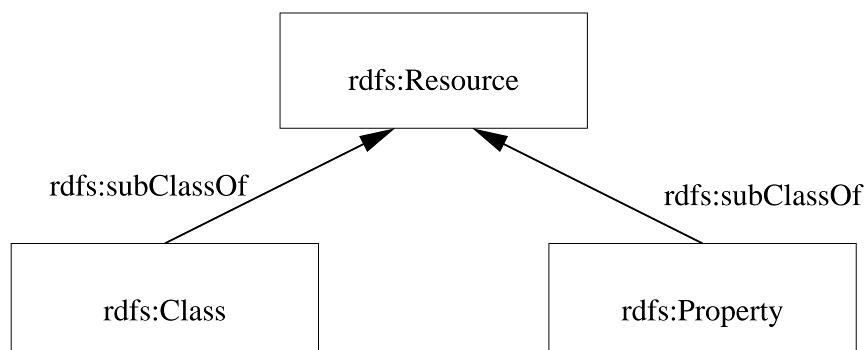
```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
             xmlns:organisaatio="http://www.organisaatiot.com/organisaatio-ns#"
     xmlns:henkilo="http://www.henkilot.com/henkilosto-ns#"
     xmlns:tyontekija="http://www.henkilostohallinta.com/henkilosto-hallinta-ns#"
>

  <tyontekija:Tyolainen rdf:about="http://www.nokia.com/tyolaiset/contact#tiina">
   <henkilo:nimi>Tiina</henkilo:nimi>
   <organisaatio:tyoskentelee_organisaatiossa>Nokia
   </organisaatio:tyoskentelee_organisaatiossa>
  </tyontekija>
</rdf:RDF>
```

Taulukko 21: Esimerkki RDF-pohjaisesta ontologian instanssista

RDF-kielen avulla voidaan kuvata URI-pohjaisia resursseja ja niiden välisiä yksinkertaisia suhteita. RDF:llä ei kuitenkaan voida ilmaista kovin monimutkaisia tai voimakkaita käsitteitä ja niiden välisiä suhteita. Lisäksi epämääräisiin URI-pohjaisiin resursseihin pohjautuva ontologian määrittely ei välttämättä ole tarpeeksi formaalia ja yhtenäistä. Tämän vuoksi on RDF:n tarjoaman resurssikuvauskielen päälle määriteltä RDF-Schema -kieli (RDF-S), jonka avulla voidaan määritellä ontologian käsitteitä ja käsitehierarkioita kuvaavia kieliä [W3C04c].

RDF-S -kielessä on kolme peruskäsitettä: resurssit, luokat ja ominaisuudet. Resurssi

(`rdfs:Resource`) on kaikkien käsitteiden yliluokka. Luokka- (`rdfs:Class`) sekä ominaisuusmääritykset (`rdfs:Property`) ovat resurssien aliluokkia. Tämä luokkahierarkia on mallinnettu kuvassa 12.



Kuva 12: RDF-kielen pääresurssien luokkahierarkia

Luokilla kuvataan ontologian käsitteitä ja niiden välisiä taksonomisia perimyssuhteita ilmaistaan `rdfs:subClassOf` määreellä. Luokkien instanssit perivät kaikkien yliluokkiensa ominaisuudet ja alityypitys on luonteeltaan transitiivinen operaatio [Mae02]. Resurssien ominaisuudet `rdfs:Property` kuvaavat ontologian käsitteiden ei-taksonomisia suhteita, eli ne määrittelevät kielen predikaatit [W3C04c]. Ominaisuuksille määritellään subjekti (`rdfs:domain`) sekä objekti (`rdfs:range`).

Taulukossa 22 on esimerkki osittaisesta RDF-Schema -kuvauksesta, joka määrittelee kuvan 10 mukaisen nimetyn ontologian. Kuvauksessa määritellään ensin kolme luokkaa, “Organisaatio”, “Henkilo” ja “Tyontekija”. Luokan “Tyontekija” määritellään olevan luokan “Henkilo” aliluokka. Luokkamäärittelyiden jälkeen kuvauksessa määritellään “HenkilonNimi”- ja “OrganisaationNimi”-ominaisuudet, jotka liittävät resursseihin “Henkilo” ja “Organisaatio” XML-skeeman mukaiset merkkijonot. Ominaisuus “tyoskentelee_organisaatiossa” on suhden “Tyontekija”-instanssien joukosta “Organisaatio”-instanssien joukkoon.

RDF-S kielestä puuttuvat usein ontologioiden kuvaamisessa käytettäviä piirteitä, kuten käsitteisiin ja suhteisiin perustuvat rajoitteet ja aksioomat. Tämän vuoksi on W3C-organisaatiossa kehitteillä erityisesti ontologioiden kuvaamiseen tarkoitettu OWL-kieli (*Web Ontology Language*), jonka avulla voidaan ilmaista esimerkiksi suhteiden transitiivisuus, symmetrisyys ja erilaisia rajoitteita käsitteiden ja suhteiden välille [W3C04a]. Lisäksi OWL-kielellä voidaan eri ontologioiden käsitteitä suhteuttaa toisiinsa käyttämällä $equivalentClass$- ja $equivalentProperty$ -rakenteita käyttäen. Tämä ominaisuus helpottaa ontologioiden uudelleenkäyttöä uusiin käyttötarkoituksiin.

Taulukossa 23 on esimerkki OWL-kielellä määritellystä luokasta “NokiaEmployee”. Nimiavaruus “organisaatio” viittaa esimerkin 22 määritelmän tapaiseen OWL-kieliseen ontologiaan. Määrittelyn mukaan “NokiaEmployee”-aliluokan instansseja ovat vain ne “Tyontekija”-luokan instanssit, joiden “OrganisaationNimi”-ominaisuudessa on merkkijono “Nokia”.

4.4 Toimintasemantiikan määrittelystä ja käyttötavoista

Semanttisen yhteensopivuuden tarkistaminen perustuu siihen, että on käytettävissä jokin perussanasto, jota käytetään komponentin funktioiden ja kommunikoitavien alkioiden semanttisten ominaisuuksien ilmaisemiseen. Sanastoa ja sopivaa logiikkaa käyttäen muodostetaan loogisia lauseita,

```
<rdfs:Class rdf:ID=''Organisaatio'' />

<rdfs:Class rdf:ID=''Henkilo'' />

<rdfs:Class rdf:ID=''Tyontekija'' >
    <rdfs:subClassOf rdf:resource=''#Henkilo'' />
</rdfs:Class>

<rdfs:Property rdf:ID=''HenkilonNimi''>
    <rdfs:domain rdfs:resource=''#Henkilo'' />
    <rdfs:range rdfs:resource=''&xsd;string'' />
</rdfs:Property>

<rdfs:Property rdf:ID=''OrganisaationNimi''>
    <rdfs:domain rdfs:resource=''#Organisaatio'' />
    <rdfs:range rdfs:resource=''&xsd;string'' />
</rdfs:Property>

<rdfs:Property rdf:ID=''tyoskentelee_yrityksessa''>
    <rdfs:domain rdfs:resource=''#Tyontekija'' />
    <rdfs:range rdfs:resource=''#Organisaatio'' />
</rdfs:Property>
```

Taulukko 22: Esimerkki nimetyn ontologian RDF-Schema -muotoisesta osittaisesta määritelmästä

```
...
<owl:Class rdf:ID="NokiaEmployee">
  <rdfs:subClassOf rdf:resource="&organisaatio;#Tyontekija"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="&organisaatio;#OrganisaationNimi"/>
      <owl:hasValue rdf:resource="#Nokia'' />
    </owl:Restriction>
  </rdfs:subClassOf>
  ...
</owl:Class>
...
```

Taulukko 23: Esimerkki OWL-rajoituksen käytöstä uuden käsitteen mallintamisessa

jotka kuvaavat funktioiden etu- ja jälkiehtoja.

Sanaston käyttöön ja määrittelemiseen liittyy muutamia ongelmia, joista seuraavaksi hieman keskustellaan. Sanaston tulisi olla mahdollisimman yleinen, jotta palveluiden kuvaukset olisivat mahdollisimman suuren yleisön ymmärrettävissä. Liiallinen sanaston yleistäminen toisaalta taas aiheuttaa sen, että yhteensopivuustarkistukset johtavat vääriin positiivisiin tuloksiin.

Otetaan esimerkiksi sovellusalue, jonka ainoana tehtävänä on neliöjuuren laskeminen. Jos funktion etuehtona on, että mikä tahansa luonnollinen luku kelpaa syötteeksi ja funktion pitäisi laskea neliöjuuri, niin negatiivisen luvun neliöjuuren laskijoille voi välillä tulla ikäviä yllätyksiä. Toisaalta, jos neliöjuurifunktiota etsitään pelkästään funktiona, joka ottaa kokonaisluvun, voi olla että suurin osa positiivisen neliöjuuren laskevista funktioista jää löytämättä. Käytännössä ontologioihin perustuvat sanastot ovat yleensä "hyvin" määriteltyjä ja rajattuja, koska sanaston kehittämiselle on jo käytännön todellisuudesta valmiiksi mietitty rakenne ja käsitteistö.

Etu- ja jälkiehtojen määrittelemisellä on kaksi eri funktiota, jotka on syytä mainita. Yleensä etu- ja jälkiehtoja käytetään yhteentoimivien palveluiden etsintään ja yhteentoimivuuden tarkistamiseen. Funktioiden etu- ja jälkiehtoja voidaan toisaalta käyttää myös ajonaikaisen ympäristön valvomiseen. Jos jonkin pankkipalvelun etuehdossa on esimerkiksi määritelty, että "*nostoraja kello*

18.00 jälkeen on 200 euroa", ei tällaista ehtoa voida tarkistaa ennakolta tietämättä kellonaikaa ja haluttua rahamäärää. Etuehdoista voidaan kuitenkin tuottaa funktioiden ajonaikaisia valvontakoneistoja, jotka esimerkiksi tekevät poikkeuksen, jos etuehtoa rikotaan. Jälkiehdoilla ei sen sijaan yksittäisen funktion käyttötilanteessa ole tällaista ajonaikaista käyttöä.

Toimintasemantiikan määrittelyn yhteydessä esitellyt abstraktit tietotyypit ja ontologia liittyvät myös olennaisesti syntaktisen- ja rajapintaprotokollien yhteentoimivuuden tarkistamiseen. Jotta komponentin rajapinnan syntaktinen rakenne ja palvelun käyttöprotokolla voidaan määritellä, täytyy ensin määritellä perussanasto, joihin syntaksin ja käyttäytymisen kuvaus perustuu. Abstraktien tietotyyppien ja ontologioiden avulla voidaan määritellä perussanasto, joka sisältää myös määriteltyihin tietotyyppeihin kohdistuvat operaatiot, predikaatit ja mahdollisesti muita käyttökelpoisia ominaisuuksia.

5 Protokollatason yhteentoimivuus

Edellisissä luvuissa käsiteltiin ohjelmistokomponenttien yhteentoimivuuden staattista puolta, jossa yhteentoimivuuden tarkistaminen liittyi pääasiassa yksittäisen komponentin ominaisuuksiin ja toimintaan. Sanan "yhteentoiminta" englanninkielinen vastine, *interoperation*, kuvaa paremmin niitä tilanteita, joissa yhteentoimivuustarkistuksia todella tarvitaan. Yhteentoimivuudessa on kyse erillisten komponenttien yhteenliittämisestä (*inter*) siten, että ne pystyvät toimimaan keskenään (*operation*). Tässä luvussa tarkastellaan yhteentoimivuuden dynaamista puolta, niin kutsuttua protokollatason yhteentoimivuutta.

Komponenttien yhteentoimivuuden tarkistaminen protokollatasolla perustuu siihen, että jokaiselle komponentille on määritelty niin kutsuttu *rajapintaprotokolla*. Rajapintaprotokolla kuvaa komponentin rajapinnan käyttötavan määrittelemällä järjestyksen ja rajoitteet metodikutsuille ja viestien kommunikoinnille.

Luvussa 5.1 määritellään yleisellä tasolla rajapintaprotokollan käsite sekä rajapintaprotokollien välinen yhteentoimivuus ja korvautuvuus. Tämän jälkeen luvussa 5.2 kuvataan, kuinka rajapintaprotokollia voidaan määritellä niin kutsuttujen prosessialgebrojen avulla ja tutustutaan prosessialgebroiden periaatteisiin. Prosessialgebroista käsitellään erityisesti mobiilien ja rakenteeltaan adaptiivisten järjestelmien kuvaamiseen kehitettyä π-kalkyyliä.

Luvussa 5.3 tutustutaan mallintarkistuksen periaatteeseen. Mallintarkistuksen avulla voidaan tarkistaa komponenttien välisen yhteistyökuvauksen ominaisuuksia. Mallintarkistusta käyttäen voidaan siis toteuttaa osa yhteensopivuustarkistuksista. Tarkistettavia ominaisuuksia voivat olla esimerkiksi lukkiutumattomuus tai viestinvaihdon tietynlainen käyttäytyminen, esimerkiksi viestien järjestyksen säilyminen.

Komponenttien välisen korvautuvuuden tarkistaminen perustuu rajapintaprotokollien välisen korvautuvuusrelaation käyttöön. Korvautuvuusrelaatio perustuu prosessien välisten ekvivalenssien käyttöön. Näihin prosessiekvivalensseihin ja kuinka ne käyttäytyvät erityisesti π-kalkyylin tapauksessa tutustutaan kappaleessa 5.4.

Lopuksi kappaleessa 5.5 tarkastellaan erityisesti komponenttien rajapintaprotokollien yhteistoimintaa mallintavia yhteensopivuus- ja korvautuvuusrelaatioita. Yhteensopivuus- ja korvautuvuusrelaatiot ovat rajapintaprotokollien välisiä relaatioita, jotka käsittelytavoiltaan muistuttavat hyvin paljon luvussa 5.4 esiteltyjä prosessiekvivalensseja. Yhteensopivuus- ja korvautuvuusekvivalenssit kuitenkin poikkeavat perinteisistä prosessiekvivalensseista, koska ohjelmistotuotannon periaatteiden noudattaminen tuo mukanaan vaatimuksia, joita ei perinteisin keinoin voida mallintaa.

Tämän luvun tarkoituksena on esitellä rajapintaprotokollien yhteentoimivuuden tarkistuksessa tarvittavia metodologioita. Valmiita ja täydellisiä ratkaisuja ei esitellä. Kuvattujen periaatteiden mukaisesti, perinteisiä lähestymistapoja käyttäen on komponenttien välinen yhteentoimivuus mahdollista formalisoida, mutta täydellisen yhteensopivuus- ja korvautuvuusrelaatioiden formalisointia ei voida kuitenkaan viedä loppuun asti tässä opinnäytetyössä.

5.1 Rajapintaprotokollien yhteensopivuus ja korvautuvuus

Rajapintaprotokolla on komponentin ulkoisen käyttäytymisen kuvaus. Se määrittelee missä järjestyksessä komponentin metodeja voidaan kutsua ja minkälaisia viestejä kommunikoidaan mihinkin suuntaan. Rajapintaprotokolla kuvaa komponentin ulkoisen käyttäytymisen ottamatta kantaa sen sisäiseen toteutustapaan tai tilaan.

Rajapintaprotokollien yhteentoimivuuden tarkistaminen perustuu siihen, että komponentti käyttäytyminen kuvataan äärellisenä automaattina, joka kommunikoi ympäristönsä kanssa vaihtamalla viestejä nimettyjen kanavien kautta [Nie95]. Jokainen viestintätapahtuma (esimerkiksi metodikutsu) on tilasiirtymä, joka muuttaa komponenttia kuvaavan automaatin tilaa. Komponentin rajapintaprotokolla määrittelee eksplisiittisesti komponentin ulkoisen käyttäytymisen ja toimintatavan: missä järjestyksessä komponentin metodeja täytyy kutsua eli kuinka komponentti käyttäytyy asiakkaalle päin. Soveltamalla metodeja, jotka on kehitetty prosessien ja protokollien ominaisuuksien ja ekvivalenssien tarkistukseen, voidaan rajapintaprotokollien yhteensopivuus ja korvautuvuus verifioida formaalisti.

Rajapintaprotokollien yhteensopivuus määritellään komponenttien rajapintaprotokollien kyvyksi toimia yhdessä siten, että niiden rinnakkainen yhteistyö ja kommunikointi etenee oikeellisesti [AG97]. Tämä edellyttää erityisesti sitä, että rajapintaprotokollien välinen yhteistyö ei johda lukkiutumaan ja jos yhteistyön on tarkoitus terminoitua, niin terminointi tapahtuu yhteisesti ja hyvin määriteltyyn lopputilaan [CPT01]. Terminoituminen tulee tapahtua siten, että molemmat rajapintaprotokollat terminoituvat oikeelliseen lopputilaan joko täysin samanaikaisesti tai siten, että toinen osapuoli ei enää yritä kommunikoida jo terminoituneen osapuolen kanssa.

Yhteensopivuuden takaamiseksi rajapintaprotokollien yhteistoiminta tulee verifioida, toisin sanoen rajapintaprotokollien muodostaman yhteistila-automaatin ominaisuudet pitää pystyä todentamaan. Tärkeimmät vaadittavat ominaisuudet ovat yleensä niin kutsuttuja turvallisuusominaisuuksia. Jos protokolla on turvallinen, niin ei ole mahdollista että mitään pahaa ja epätoivottavaa pääsee tapahtumaan. Tämä tarkoittaa esimerkiksi sitä, että keskinäisen poissulkemisen tarjoava komponentti on toteutettu sellaista rajapintaprotokollaa käyttäen, että se ei päästä kahta asiakasta yhtä aikaa kriittiselle alueelle. Tärkein komponentin rajapintaprotokollilta vaadittava ominaisuus on se, että komponentin käyttäminen ei johda lukkiutumaan.

Rajapintakomponenttien välinen yhteensopivuus tarkistetaan mallintarkistusta käyttäen. Mallintarkistuksessa äärellinen struktuuri, tässä tapauksessa rajapintaprotokollien muodostama yhteistila-automaatti, verifioidaan jonkin loogisen kaavan ϕ suhteen. Kaava ϕ on määritelty sellaisella logiikalla, jolla pystytään ilmaisemaan tarvittavat ominaisuudet, esimerkiksi lukkiutumattomuus. Tällaisia logiikoita ovat esimerkiksi modaali- tai temporaalilogiikoiksi kutsutut predikaattilogiikan laajennokset. Mallintarkistukseen ja siinä käytettäviin logiikkoihin tutustutaan tarkemmin luvussa 5.3.

Komponenttien rajapintaprotokollien välinen korvautuvuus tarkoittaa sitä, että kahden komponentin ulkoinen käyttäytyminen on tarpeeksi samankaltaista, jotta ne voidaan korvata keskenään. Komponenttien rajapintaprotokollien korvautuvuus takaa sen, että asiakkaat voivat esimerkiksi ohjelmistopäivityksen jälkeen käyttää uutta komponenttia samalla tavalla kuin korvattua komponenttia. Rajapintaprotokollien välisen korvautuvuuden tarkistamiseksi tarvitaan jokin menetelmä, jolla pystytään todentamaan rajapintaprotokollien ulkoisten käyttäytymiskuvausten samankaltaisuus.

Rajapintaprotokollien välinen korvautuvuus tarkistetaan käyttämällä relaatiota, jota kutsutaan korvautuvuusrelaatioksi. Korvautuvuusrelaatio perustuu niin kutsuttuun bisimulaatiorelaatioon. Rajapintaprotokollien P ja Q välinen bisimulointi $P\mathcal{R}Q$ voidaan tulkita siten, että P voi simuloida rajapintaprotokollaa Q ja vastaavasti Q voi simuloida rajapintaprotokollaa P.

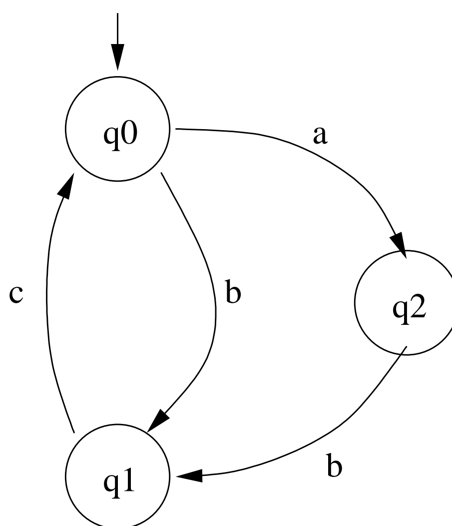
Rajapintaprotokollien korvautuvuuden tarkastamista varten kehitellään prosessien bisimulointirelaation kaltainen ekvivalenssi, joka säilyttää protokollien välisen yhteentoimivuuden. Yhteentoimivuuden tulee säilyä, kun komponenteista kehitetään uusia versioita joko toimintoja laajentamalla tai niitä erikoistamalla. Rajanpintaprotokollien käyttäytymisen laajentaminen ja ominaisuuksien periytyminen ovat ominaisuuksia, jotka korvautuvuusrelaation tulee säilyttää ja ylläpitää. Raja-

pintaprotokollien välistä korvautuvuutta ja siihen liittyviä käsitteitä ja mallinnustapoja käsitellään luvussa 5.5.2.

5.2 Rajapintaprotokollien kuvaaminen prosessialgebralla

Rajapintaprotokollien yhteentoimivuuteen liittyvät tarkistukset suoritetaan protokollia kuvaavien äärellisten automaattien ominaisuuksia verifioimalla ja vertailemalla. Äärellisistä automaateista käytetään tässä tapauksessa erityisesti niin kutsuttuja nimettyjä siirtymäsysteemejä (*labeled transition system*, LTS, tästä lähtien vain siirtymäjärjestelmä). Siirtymäjärjestelmä on äärellinen automaatti, joka koostuu äärellisestä joukosta tiloja Q, äärellisestä aakkostosta A, siirtymärelaatiosta $T \subseteq Q \times A \times Q$ sekä alkutilasta $q_0 \in Q$. Siirtymäjärjestelmän M formaali kuvaus on täten monikko muotoa $M = (Q, A, T, q_0)$.

Kuvassa 13 on esimerkki eräästä äärellisestä siirtymäjärjestelmästä M_{esim}. Siirtymäjärjestelmän M_{esim} alkutila on q_0, jonka lisäksi sillä on tilat q_1 ja q_2. Siirtymäjärjestelmän aakkosto on nimien joukko $A = \{a, b, c\}$ ja siirtymärelaatio $T = \{(q_0, a, q_2), (q_0, b, q_1), (q_1, c, q_0), (q_2, b, q_1)\}$. Äärellisen automaatin ja rajapintaprotokollan välinen yhteys tulee selväksi, jos siirtymät a, b ja c tulkitaan esimerkiksi komponentin metodikutsuiksi. Tällöin automaatti M_{esim} määrittelee rajapintaprotokollan, jonka mukaan esimerkiksi metodikutsujen sekvenssi a, b, c ja b, c ovat kelvollisia. Automaatin M_{esim} mukaan epäkelpoja metodien kutsujärjestyksiä ovat muun muassa a, c tai b, a.



Kuva 13: Esimerkki siirtymäjärjestelmästä

Äärellinen automaatti on kuitenkin hankala tapa kuvata rajapintaprotokollia ja erityisesti niiden yhteistoimintaa. Äärellisestä automaatista tulee hyvin nopeasti niin suuri, ettei sen muodostaminen käsin ole järkevää. Lisäksi puhtaasti tiloihin ja niiden välisiin siirtymiin perustuvan äärellisen automaatin perustalle on hyvin hankalaa määritellä helposti käsiteltävässä muodossa esimerkiksi arvojen kommunikointia kuvaavaa semantiikkaa. Muun muassa näiden käytännöllisten syiden vuoksi tilasiirtymäsysteemejä ei käytetä verifiointiin suoraan, vaan ne muodostetaan käyttämällä niin kutsuttuja prosessialgebroita.

Prosessialgebralla tarkoitetaan rinnakkaisten järjestelmien ominaisuuksien määrittelemistä prosesseja kuvaavien termien ja lausekkeiden avulla [BW90]. Prosessialgebroita on useita erilaisia

eri sovellusalueita varten ja ne eroavat toisistaan syntaksin ja semantiikan suhteen. Esimerkiksi CSP-prosessialgebrassa on kielen määrittelyssä eroteltu prosessin sisäinen (epädeterministinen) ja ulkoinen (deterministinen) valinta, kun taas CCS ja π-kalkyyli nimisissä prosessialgebroissa on käytössä vain deterministinen valinta [BHR84, Mil89b, MPW89]. Toisaalta, CSP ja CCS prosessialgebroissa kommunikointi perustuu arvojen välitykseen, kun taas π-kalkyylissä kommunikoidaan nimiä.

Tässä opinnäytetyössä käytettävä lähestymistapa prosessialgebroihin ja niiden verifiointiin perustuu kolmeen perusideaan: havaitoekvivalenssiin, synkroniseen kommunikointiin ja prosessilausekkeiden operationaaliseen tulkintaan [Mil82].

Havainnolla tarkoitetaan ulospäin näkyvää tai koettavaa käyttäytymistä. Havaintoekvivalenssi on kahden tai useamman prosessin välinen relaatio, joka tulkitaan siten, että prosessit P ja Q ovat relaatiossa keskenään jos ja vain jos prosessit käyttäytyvät havainnoitsijan mielestä samalla tavalla.

Synkronisoidussa kommunikoinnissa prosessien kommunikointi on samalla prosessien synkronointia, toisin sanoen, prosessissa $a.P \mid \bar{a}.Q$, joka kuvaa viestin lähettävän ja vastaanottavan prosessin rinnakkaista toimintaa, tapahtumat a (vastaanotto) ja $\bar{a}$ (lähetys) tapahtuvat yhtä aikaa [Mil82]. Toisistaan riippumattomien tapahtumien yhtäaikaisuus käsitetään yleensä lomitussemantiikan mukaisesti, jossa rinnakkaiset tapahtumat a ja b luovat kaksi mahdollista suoritusjälkeä, ab tai ba.

Prosessialgebran tulkinta määrittelee, mitä jokin prosessialgebran lauseke "todella" tarkoittaa. Prosessialgebran lausekkeille voidaan antaa tulkinta kolmella eri tavalla, jotka ovat keskenään vaihtokelpoisia [Hen88]. Ensimmäinen tapa tulkita prosessilauseketta on sitoa prosessilausekkeelle *operationaalinen* tulkinta äärellisen nimetyn siirtymäjärjestelmän (*labeled transition system*, LTS) kautta. Prosessilausekkeiden ekvivalenssi määritellään operationaalisessa tulkinnassa prosesseja kuvaavien suorituspuiden avulla [Mil82].

Toinen tulkinta prosessilausekkeelle saadaan, kun se sidotaan johonkin matemaattiseen käsitteistöön, kuten esimerkiksi joukko-opillisiin struktuureihin. Prosessialgebran lausekkeelle voidaan tulkita esimerkiksi lausekkeen generoimien suoritusjälkien joukkona (*trace-set*). Tällöin kaksi prosessilauseketta ovat ekvivalentteja, jos ja vain jos niiden suoritusjälkien joukot sisältävät samat alkiot. Prosessialgebran operaattorit, kuten rinnakkainkytkentä, voidaan tämän jälkeen määritellä joukkoihin kohdistuvina operaatioina [BHR84].

Kolmas tulkintatapa on algebrallinen ja sen käyttäminen perustuu prosesseja kuvaavien teoreemien todistamiseen ja ekvivalenssiin. Prosessilausekkeiden todistaminen (automaattisesti) on kuitenkin hyvin hankalaa ellei jopa mahdotonta ja se vaatii yleensä ihmisen väliintuloa. Jo perus-CCS:n, joka ei sisällä arvojen kommunikointia, todistaminen on yleisessä tapauksessa mahdotonta, koska kyseinen prosessialgebra ei ole äärellisesti aksiomatisoitavissa [Hen89]. Toisaalta teoreemantodistamiseen perustuva verifiointi mahdollistaa äärettömien ja parametrisoitavien järjestelmien vertailun ja niiden ominaisuuksien tarkastamisen [MB97]. Tähän eivät äärellistilallisiin automaatteihin perustuvat operationaalisen tulkinnan metodit kykene.

Edellä mainituista prosessialgebrojen tulkintatavoista operationaalinen tulkinta on kaikista yleisin ja suosituin. Tämä johtunee suureksi osaksi siitä, että se on hyvin intuitiivinen tulkinta prosessin käyttäytymiselle. Operationaalisessa tulkinnassa tarvittavien laskennallisten rakenteiden käsittelemiseen on kehitetty tehokkaita ratkaisuja. Näistä voidaan mainita esimerkkeinä niin kutsutut symboliset menetelmät ja on-the-fly–menetelmät [JEK$^+$90, FM90]. Tämän opinnäytetyön tapauksessa prosessien semantiikka määritellään operationaalisen tulkinnan kautta.

Eri prosessialgebroissa on semanttisista ja syntaktisista eroista huolimatta tunnistettavissa saman-

kaltaiset peruskäsitteet (synkroninen kommunikointi, lomitussemantiikka ja ulkoiseen käyttäytymiseen perustuvat ekvivalenssit) ja operaattorit. Prosessi muodostetaan atomisilla toiminnoilla, joita merkitään kirjaimilla $a, b, \ldots$. Atomisia toimintoja käytetään prosessilausekkeiden etuliitteenä .-operaattorin avulla. Esimerkiksi $a.P$ on prosessi, jossa ensin suoritetaan toiminto a ja tämän jälkeen jatketaan prosessin P mukaan.

Valintaoperaattorilla $+$ merkitään prosessin haarautumista ja valintaa. Prosessi $P + Q$ voi jatkaa toimintaansa kuten P tai Q. Näiden lisäksi on vielä käytössä rinnakkaisoperaattori $|$, jossa $P \mid Q$ tarkoittaa sitä, että prosessit P ja Q voivat edetä rinnakkain. Yleensä prosessien rinnakkaissuoritus tulkitaan käytännön syistä toimintojen vuorottaiseksi lomitukseksi [Mil82]. Prosessi $a \mid b$ tulkitaan siten, että sillä on kaksi vaihtoehtoista etenemistapaa: joko toiminto a suoritetaan ensin tai sitten b suoritetaan ensin. Tällä tavalla ei tietenkään pystytä täydellisesti mallintamaan oikeata rinnakkaisuutta, mutta se on kompromissi, joka voidaan hyväksyä [Mil82].

5.2.1 Yksinkertainen prosessialgebra

Prosessialgebroiden periaatteisiin tutustutaan seuraavaksi prosessialgebran CCS kautta [Mil82, Mil89b]. CCS-prosessialgebran ominaisuuksista käydään läpi vain pieni osajoukko, jonka avulla prosessialgebrojen perusperiaatteet tulevat ymmärretyksi. Täysi prosessialgebra sisältää tässä esiteltyjen ominaisuuksien lisäksi mm. mahdollisuuden rekursiivisten prosessien määrittelyyn sekä arvojen kommunikointiin. Tarkastelun kohteena olevassa täyden CCS:n osajoukossa kommunikointi on puhdasta synkronisointia, toisin sanoen, prosessien välinen kommunikointi ei muuta prosessien semantiikkaa synkronointipisteen jälkeen.

CCS koostuu joukosta nimiä $a, b, \ldots \in \mathcal{A}$ ja joukosta vastanimiä (*co-name*) $\bar{a}, \bar{b}, \ldots \in \bar{\mathcal{A}}$ [Mil89b]. Nimi a tarkoittaa lähetystapahtumaa nimettyä kanavaa a pitkin ja $\bar{a}$ vastaanottotapahtumaa nimettyä kanavaa a pitkin. Nimien joukot A ja $\bar{A}$ muodostavat kaikkien nimien joukon $\mathcal{L} = A \cup \bar{A}$. Edellä mainittujen nimien lisäksi on käytössä tunniste τ, joka merkitsee prosessin sisäistä tapahtumaa. Täten CCS:n kaikkien tapahtumien joukko on $Act = \mathcal{L} \cup \tau$. Yleisen käytännön mukaan yleistä nimeämätöntä tapahtumaa merkitään joko kirjaimilla $\alpha, \beta, \ldots \in Act$ tai $l \in \mathcal{A}$ ja $\bar{l} \in \bar{\mathcal{A}}$.

Prosessilausekkeet (agentit) muodostetaan siten, että prosessialgebrassa käytettäviä nimiä (joukon $\mathcal{L}$ alkioita) yhdistetään toisiinsa prosessialgebran operaattoreilla. Prosessilausekkeet muodostetaan määritelmän 5.1 syntaktisten sääntöjen mukaisesti [Mil82, Mil89b].

Määritelmä 5.1 (Prosessilausekkeiden muodostussäännöt) *P ja Q ovat prossessilausekkeita:*

$\mathbf{0}$, *Tyhjä prosessi*

2. $\alpha.P$, *Etutoiminto* $(\alpha \in \mathcal{L})$
3. $P + Q$, *Valinta*
4. $E_1 \mid E_2$, *Rinnakkaisoperaattori*
5. $E \backslash L$, *Näkyvyyden rajaus*
6. $E[f]$, *Uudelleennimentä* (f *uudelleennimentäfunktio*)
7. $A \stackrel{def}{=} P$, *Vakioagentti A*

Tyhjä prosessi **0** kuvaa prosessia, josta ei ole yhtään siirtymää eteenpäin. Se voidaan tulkita päättyneeksi prosessiksi. Etutoiminto-sääntö määrittelee kuinka yksittäisiä nimiä voidaan kytkeä sarjalliseksi tapahtumajonoksi. Valinta määrittelee kaksi vaihtoehtoista etenemistapaa: $P + Q$ voi jatkua kuten prosessi P tai Q. Useamman prosessin P_i summausta merkitään $\Sigma_{i \in I} P_i$, missä I on jokin indeksointijoukko.

Näkyvyyden rajaus on operaatio, joka piilottaa jonkin kommunikointikanavan käytön prosessin sisäiseksi. Esimerkiksi prosessissa $(a.P \mid \bar{a}.Q)\backslash a$ nimi a on prosessin sisäinen eli se ei näy prosessin ulkopuolelle. Jos nimi α on prosessin P sisäinen nimi, niin mikään prosessi ei voi kommunikoida prosessin P kanssa portin a kautta.

Uudelleennimentä on operaatio, jossa prosessin P nimi α on muunnettu kuvauksen $f : \mathcal{L} \rightarrow \mathcal{L}$ antamaksi nimeksi. Jos esimerkiksi prosessi on muotoa $P = a.b.\mathbf{0}$ ja $f : f(a) = c, f(b) = d$, niin prosessi $P[f] = c.d.\mathbf{0}$. Uudelleennimentäfunktion pitää olla bijektio, toisin sanoen $f(\bar{\alpha}) = \bar{f(\alpha)}$ [Mil89b]. Uudelleennimennällä voidaan toteuttaa erilaisia instansseja samasta prosessista ja sitä yleensä käytetään käytännössä muodossa $P[a/c, b/d]$, joka kuvaa edellisen prosessin uudelleennimennän $P[f]$.

Vakioagentti A käyttäytyy kuten prosessi P, se on prosessia P vastaava nimi. Vakioagentin määrittelyä voitaisiin verrata ohjelmointikielissä käytettäviin funktiomäärittelyihin.

Prosessialgebran semantiikka määrittelee kuinka kielen operaattorit käyttäytyvät tapahtumien α yhteydessä. CCS-prosessialgebran semantiikka määritellään operationaalisesti siirtymäsääntöjen avulla, jotka on lueteltuna taulukossa 24.

$$ACT: \quad \frac{}{\alpha.P \xrightarrow{\alpha} P}$$

$$SUM_1: \quad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad SUM_2: \quad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$$

$$COM_1: \quad \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \qquad COM_2: \quad \frac{Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} Q'}$$

$$COM_3: \quad \frac{P \xrightarrow{l} P' \quad Q \xrightarrow{\bar{l}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

$$RES: \quad \frac{P \xrightarrow{\alpha} P'}{P\backslash L \xrightarrow{\alpha} P'\backslash L} \; \alpha, \bar{\alpha} \notin L \qquad REL: \quad \frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]}$$

$$CON: \quad \frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'} \; (A \stackrel{def}{=} P)$$

Taulukko 24: CCS-prosessialgebran siirtymäsäännöt

Siirtymäsäännöt määrittelevät prosessialgebran semantiikan eli kuinka erilaiset siirtymät muuttavat järjestelmän tilaa. Siirtymäsääntöjä käytetään logiikan päättelysääntöjen tapaisesti. Siirtymäsääntöjä prosessilausekkeisiin soveltamalla saadaan aikaan suorituspuu, joka sisältää kaikki prosessille mahdolliset suoritusjäljet (*trace*). Prosessilausekkeen suoritusjäljet ovat kuitenkin usein äärettömiä, joten suorituspuiden sijaan prosessilausekkeesta luodaan siirtymäjärjestelmä, LTS (*labeled transition system*), joka kuvaa prosessin kaikki mahdolliset tilat ja siirtymät [Mil89b]. LTS on äärellinen automaatti $(S, Act, \rightarrow, s_0)$, jossa S on prosessien joukko, Act toimintojen joukko,

$\rightarrow$ on siirtymärelaatio $\rightarrow \subseteq S \times Act \times S$ ja s_0 automaatin alkutila [Mil89b, Hen88]. Jos tilasta p on a-siirtymä tilaan p', $(p, a, p') \in \rightarrow$, niin tätä yleensä merkitään $p \xrightarrow{a} p'$.

Seuraavaksi annetaan esimerkki prosessialgebran käytöstä ohjelmistokomponenttien käyttäytymisen kuvaamisessa. Taulukossa 25 on CCS-prosessialgebran mukaisella syntaksilla kuvattu kolme käyttöliittymäprotokollaa. Ne ovat kuvitteellisen pankkiautomaattijärjestelmän automaatti- ja asiakaskomponenttien käyttäytymiskuvauksia. Prosessilausekkeet tunnistaa isolla kirjaimella alkavasta nimestä (esim. *UseBank*), kun taas kommunikointikanavien nimet alkavat pienellä kirjaimella (*login*).

Rajapintaprotokolla *BankProtocol* kuvaa yksinkertaisen pankkiautomaatin käyttötavan. Pankkiautomaatti tukee vain yhtä varsinaista toiminnallisuutta, *ShowBalance*. Komponentin protokolla määrittelee käyttäytymismallin, jonka mukaan asiakaskomponentti voi yrittää sisäänkirjautumista kaksi kertaa. Jos sisäänkirjautuminen onnistuu, niin asiakas pääsee käyttämään pankkiautomaattia *ShowBalance*-prosessin kuvaamalla tavalla. Kahden epäonnistuneen sisäänkirjautumisyrityksen jälkeen pankkiautomaatti menee asiakkaan näkökulmasta lukkoon. Sen sijaan rajapintaprotokollaa *ClientProtocol2* käyttävä asiakas toimii yhteen rajapintaprotokollaa *BankProtocol* käyttävän pankkiautomaatin kanssa.

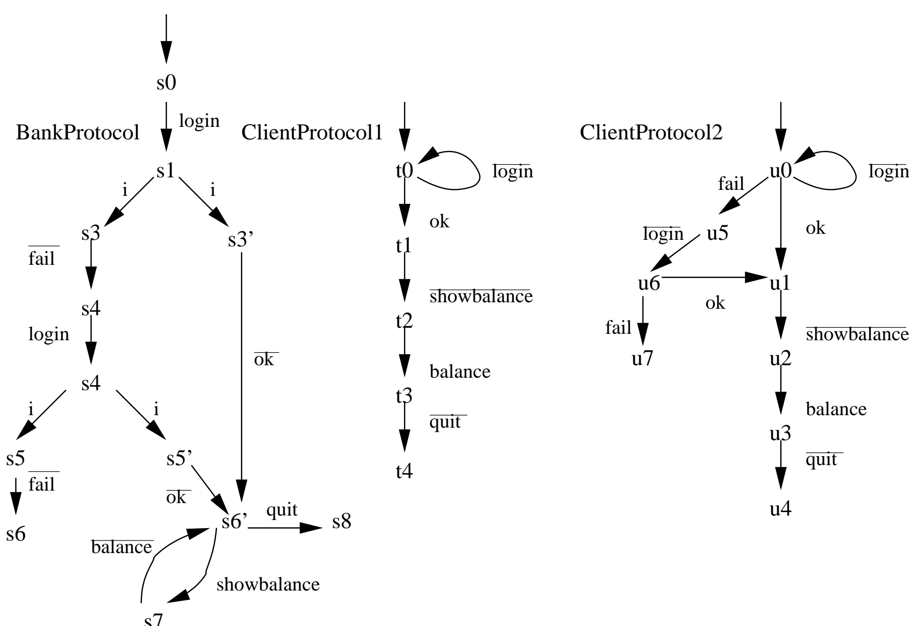
Protokolla *ClientProtocol1* kuvaa erään asiakkaan käyttäytymistyypin. Asiakas käyttäytyy siten, että *login*-kutsua tehdään niin kauan, kunnes takaisin saadaan *ok*-kuittaus. Asiakkaan käyttäytymisestä on myös toinen malli, *ClientProtocol2*, joka eroaa edellisestä siten, että siinä on alkutilasta *fail*-siirtymä.

$$\begin{aligned}
BankProtocol &= login.(\tau.\overline{ok}.UseBank + \tau.Retry)\\
Retry &= \overline{fail}.(login.(\tau.\overline{ok}.UseBank + \tau.\mathbf{0}))\\
UseBank &= ShowBalance + Quit\\
ShowBalance &= showbalance.\overline{balance}.UseBank\\
Quit &= quit.\mathbf{0}\\
\\
ClientProtocol1 &= \overline{login}.(ok.GetBalance + ClientProtocol1)\\
GetBalance &= \overline{showbalance}.balance.\overline{quit}\\
\\
ClientProtocol2 &= \overline{login}.(ok.GetBalance2 + ClientProtocol2 + fail.RetryLogin)\\
RetryLogin &= \overline{login}.(ok.GetBalance2 + fail.0)\\
GetBalance2 &= \overline{showbalance}.balance.\overline{quit}
\end{aligned}$$

Taulukko 25: Esimerkki prosessialgebran käytöstä

Pankki- ja asiakasprotokollien siirtymäjärjestelmät ovat kuvassa 14. Prosessin sisäiset τ-siirtymät on kuvassa esitetty i-nimisinä siirtyminä esitysteknisistä syistä johtuen.

Taulukossa 25 kuvaillut protokollat ovat erittäin yksinkertaisia esimerkkejä komponentin käyttäytymiskuvauksista. Siitä kuitenkin saa käsityksen prosessialgebran käyttämisestä ohjelmistoarkkitehtuureissa. Ilman suurempaa hankaluutta voidaan jo silmämääräisesti *BankProtocol*- ja *ClientProtocol1*-prosesseista nähdä, että ne eivät ole yhteensopivia. Kahden epäonnistuneen *login*-kutsun jälkeen järjestelmä *ClientProtocol1* | *BankProtocol* on jo lukkiutunut: pankkijärjestelmän näkökulmasta prosessi on tilassa $s6$, mutta asiakas ei ole huomioinut $fail$-viestejä. Asiakkaan mielestä ollaan vielä tilassa $t0$ ja voidaan tehdä joko *login*-kutsuja tai saada *ok*-viesti. Jos asiakas käyttää *ClientProtocol2* määrityksen mukaista protokollaa, joka huomioi *fail*-viestin vastaanoton ja määrittelyn mukaisen toiminnan, niin lukkiutumalta vältytään.



Kuva 14: Pankkiautomaattiesimerkin prosessilausekkeiden LTS-mallit

Komponenttien väliset epäyhteensopivuudet voidaan sopivia, verifiointia tukevia, kehitystyökaluja käyttäen huomioida jo ohjelmiston suunnitteluvaiheessa. Monimutkaisempien protokollien spesifikaatioihin voidaan liittää esimerkiksi turvallisuus- tai elävyysominaisuuksia, joita protokollan toteutuksien tulee kunnioittaa. Rajapintaprotokollien ominaisuudet tarkistetaan käyttämällä mallintarkistusta (*model checking*), johon tutustutaan lähemmin luvussa 5.3.

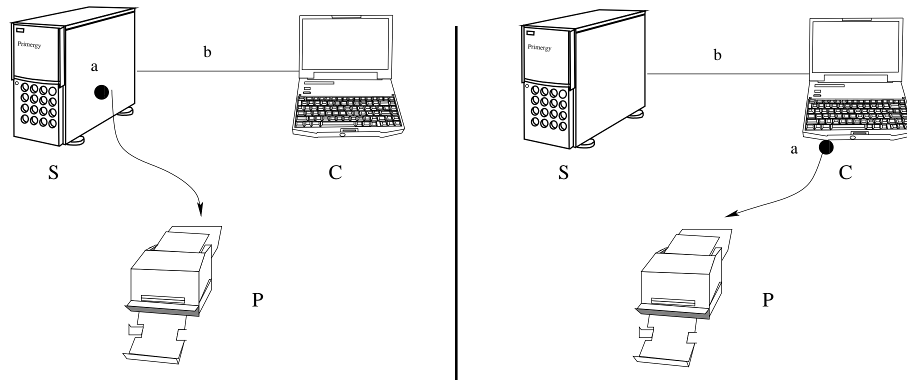
Yhteensopivuuden lisäksi tulee eteen esimerkkiä tarkastellessa kysymys siitä, voidaanko esimerkiksi *BankProtocol* korvata jollain muulla rajapintaprotokollalla, joka pääpiirteittäin käyttäytyy kuten *BankProtocol*, mutta esimerkiksi lisää toiminnallisuutta *UseBank*-prosessiin. Toisin sanoen, jos on olemassa protokolla *EnhancedBankProtocol*, jossa *UseBank* = *ShowBalance* + *DrawMoney* + *Quit*, niin ovatko protokollat *BankProtocol* ja *EnhancedBankProtocol* korvattavissa keskenään? Tällaisiin kysymyksiin saadaan vastaus prosessien ekvivalenssivertailulla, jota käsitellään luvussa 5.4.

Tässä vaiheessa voidaan jo mainita, että ohjelmistotuotannon käytössä sekä mallintarkastus että ekvivalenssivertailut osoittautuvat hankalammiksi kuin niiden perinteiset vastineet esimerkiksi laitteistoverifioinnin puolella. Syynä tähän on se, että (komponenttipohjaisessa) ohjelmistotuotannossa komponentteja pitää pystyä laajentamaan ja korvaamaan vanhoja versioita uusilla, laajennetuilla versioilla. Intuitiivisesti tämä on selvää, koska olisi liian rajoittavaa, jos edellä kuvaillun *BankProtocol*-komponentin korvaaminen *EnhancedBankProtocol*-komponentilla aiheuttaisi sen, että kaikki ennen *BankProtocol*-protokollan kanssa yhteensopivat asiakkaat eivät enää pystyisi käyttämään pankkiautomaattipalvelua. Vaatimukset yhteensopivuudelle ja korvattavuudelle ovat vaikeasti formalisoitavissa, kun protokollia pitää pystyä laajentamaan siten ,että tärkeäksi luokitellut ominaisuudet pysyvät kuitenkin koskemattomina.

5.2.2 π-kalkyyli

Eräs prosessialgebroista on π-kalkyyli, jonka Robin Milner kehitti CCS-prosessialgebran pohjalta [MPW89]. Toisin kuin monilla muilla prosessialgebroista, kuten CCS [Mil89b], CSP [BHR84] ja Petri-verkot [Pet77], voidaan π-kalkyylin avulla ilmaista prosessien liikkuvuus luonnollisesti nimien välityksen avulla. π-kalkyylin ominaisin piirre on se, että sekä kommunikointikanavat että viestitettävät objektit ovat nimiä [Par01]. Nimet voidaan tulkita esimerkiksi käyttöoikeuksiksi, viittauksiksi tai tietorakenteiksi tarpeen mukaan. Itse asiassa, π-kalkyylissä käsitellään vain kahdenlaisia olioita, nimiä ja agentteja eli prosessilausekkeita [MPW89].

Kuvassa 15 on esimerkki tilanteesta, jossa jaetun resurssin (kirjoitin) käyttöä hallitsee palvelin S[Par01]. Tämä ominaisuus on mallinnettu siten, että palvelimen S hallussa on kommunikointikanava nimeltään a, joka "viittaa" kirjoitinprosessiin P. Kirjoittimen käyttöoikeuden siirto asiakkaalle C mallinnetaan siten, että prosessi S lähettää kanavan (nimen) a prosessille C kanavan b kautta. Tämän jälkeen asiakkaalla C on "käyttöoikeus" kirjoittimeen. Kirjoittimen käyttö mallinnetaan prosessien C ja P välisenä synkronointina kanavan a (tai sitä kautta saatujen uusien kanavien) kautta.



Kuva 15: Esimerkki nimen viestintään perustuvasta mobiliteetista

Seuraavaksi esitellään π-kalkyylin syntaksi. Varsinaisia agentteja eli prosessilausekkeita merkitään kirjaimilla $P, Q, R, \ldots$. Käytössä on myös joukko $\mathcal{K}$ (agenttien) tunnuksia, joita merkitään $A, B, C, \ldots \in \mathcal{K}$ Agentin tunniste voidaan mieltää ohjelmointikielten funktiomääritelmää vastaavaksi käsitteeksi. Jos agentin tunniste on määritelty $A(x_1, \ldots, x_n) \stackrel{def}{=} P$, niin tunniste $A(x_1, \ldots, x_n)$ käyttäytyy kuten agentti $P\{y_1/x_1, \ldots, y_n/x_n\}$, missä merkintä $P\{y_1/x_1, \ldots, y_n/x_n\}$ tarkoittaa agentin P vapaiden nimien x_i yhtäaikaista korvaamista nimillä y_i [MPW89]. Näiden lisäksi π-kalkyylissä oletetaan olevan ääretön määrä nimiä $a, b, \ldots, z \in \mathcal{N}$, jotka toimivat kommunikointiportteina, muuttujina ja data-arvoina [MPW89]. Taulukossa 26 on kuvattuna π-kalkyylin agenttien syntaksi.

Syntaksin lisäksi π-kalkyylin tarkasteluun tarvitaan muutama apukäsite. Prosessin P *vapaat nimet*, $fn(P)$, ovat nimiä, jotka eivät ole sidottu sitovan operaattorin toimesta. Sitovia operaattoreita ovat input-toiminto $a(x).P$ sekä nimen näkyvyysalueen rajoitusoperaattori $(x)P$. Ne molemmat sitovat nimen x prosessiin P. Sidotut nimet $bn(n)$ ovat nimiä, jotka ovat etuliitteiden $a(x).P$ tai $(x)P$ parametreina. Prosessin P kaikkien nimen joukko on $n(P) = fn(P) \cup bn(P)$ [MPW89]. Etuliitteissä $\bar{a}x$ ja $a(x)$ nimeä a kutsutaan subjektiksi ja nimeä x objektiksi [MPW89]. Nimen korvaus $P\{x/y\}$ on operaatio, jossa kaikki nimen y vapaan ilmentymät prosessissa P korvataan

1. Tyhjä agentti 0, joka ei voi suorittaa mitään siirtymiä. Kuvaa päättynyttä prosessia.
2. Output-etuliite $\bar{a}x.P$. Nimi x lähetetään nimen a kautta ja sen jälkeen agentti jatkaa suoritustaan agenttina P.
3. Input-etuliite $a(x).P$. Nimi saadaan nimeä a pitkin ja se sijoitetaan nimen x paikalle. Tämän jälkeen agentti jatkaa suoritustaan agenttina P, jossa x on korvattu arvolla, joka saatiin a:n kautta.
4. Tyhjä toiminto $\tau.P$ kuvaa agenttia, joka voi siirtyä käyttäytymään kuten agentti P ilman näkyvää vuorovaikutusta ympäristön kanssa.
5. Summa $P + Q$ kuvaa agenttia, joka voi toimia joko agentin P tai Q mukaisesti.
6. Rinnakkaiskompositio $P|Q$, edustaa agenttien P ja Q yhteiskäytöstä, kun agentit toimivat rinnakkain. Agentit P ja Q voivat toimia riippumattomasti toisistaan, mutta myös kommunikoida keskenään, jos toinen suorittaa toiminnon $\bar{a}x$ ja toinen $a(y)$.
7. Nimien yhtäsuuruus (match) $if\ x = y\ then\ P$ toimii kuten P jos ja vain jos nimet x ja y ovat ekvivalentit.
8. Näkyvyyden rajoitus $(x)P$ toimii kuten P, mutta nimi x on paikallinen. Toisin sanoen nimeä x ei voida käyttää kommunikointikanavana P:n ja sen ympäristön välillä. Kuitenkin sitä voidaan käyttää P:n komponenttien väliseen kommunikointiin.
9. Agentin tunniste $A(y_1, \ldots, y_n)$, nissä n on sama kuin A:n ariteetti. Jokaisella tunnisteella on määritelmä $A(x_1, \ldots, x_n) \overset{def}{=} P$

Taulukko 26: π-kalkyylin agenttien syntaksi

nimellä x. Jos x on sidottu P:ssä, niin ennen korvausoperaatiota tulee kaikki P:n x-ilmentymät uudelleennimetä: $P\{x/y\}, x \in bn(P) \rightarrow (P\{x'/x\})\{x/y\}$ [MPW89].

Yllä mainittujen syntaktisten perusrakenteiden lisäksi π-kalkyylissä käytetään myös sidottua output-toiminnetta (*bound output action*), jonka määritelmä on seuraava: $\bar{x}(y) = (y)\bar{x}y$ [MPW89]. Tällaista yksityisen nimen kommunikointia ei voida tehdä esimerkiksi CCS-prosesialgebrassa [MPW89].

Muiden prosessialgebrojen tavoin myös π-kalkyylillä on omat siirtymäsääntönsä, jotka on lueteltuna taulukossa 27.

Kuten taulukosta 26 voidaan huomata, on π-kalkyylissä neljä erilaista siirtymätyyppiä: tyhjä siirtymä τ, output-siirtymä $\bar{x}y$, input-siirtymä $x(y)$ sekä sidottu output-siirtymä $\bar{x}(y)$ [MPW89]. Taulukossa 27 on π-kalkyylin siirtymäsäännöt.

Syy π-kalkyylin perinteisiä prosessialgebroja (esimerkiksi CCS) parempaan ilmaisukykyyn on siinä, että π-prosessien välillä voidaan siirtää paikallisen nimen näkyvyys prosessista toiseen (*scope extrusion*) [MPW89]. Tavallisissa prosessialgebroissa on käytössä näkyvyyden rajoitusoperaattori, joka tyypillisesti rajaa kommunikointiportin näkyvyyden tietyn prosessin alueelle. CCS-prosessialgebrassa rajoitus $(P|Q)\backslash a$ tekee luku- ja kirjoitusporteista a ja $\bar{a}$ näkymättömiä prosessin $(P|Q)$ ulkopuolella [Mil89b]. Toisin sanoen, kommunikointikanava a on varattu vain P:n ja Q:n väliseen kommunikointiin, eikä mikään prosessi voi kommunikoida $(P|Q)$:n kanssa portin a kautta.

Yleensä prosessialgebrojen rajoitusoperaattorit koskevat CCS:n tavoin nimen omaan kommunikointiportteja (subjekteja), joita ei voida kommunikoida agenttien kesken. Täten rajoitetun nimen näkyvyys on staattista, eli sen näkyvyysalue ei muutu [Par01]. Nimen näkyvyyden rajoitus merkitään π-kalkyylissä $(a)(P|Q)$, mutta CCS:stä poiketen kommunikointiportti a on vain nimi, joka on kommunikoitavissa muille prosesseille samalla tavoin kuin mikä tahansa muukin nimi [Par01].

Ulkoisesti samalla tavoin käyttäytyvä järjestelmä voidaan mallintaa usealla erilaisella prosessialgebran kaavalla. Yksinkertaisin esimerkki samoin käyttäytyvistä, mutta eri kaavalla mallinnetusta

$$TAU-ACT: \frac{-}{\tau.P \xrightarrow{\tau} P} \qquad OUTPUT-ACT: \frac{-}{\bar{x}y.P \xrightarrow{\bar{x}y} P}$$

$$INPUT-ACT: \frac{-}{x(z).P \xrightarrow{x(w)} P\{w/z\}} \; w \notin fn((z)P)$$

$$SUM: \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \qquad MATCH: \frac{P \xrightarrow{\alpha} P'}{[x=x]P \xrightarrow{\alpha} P'}$$

$$IDE: \frac{P\{\tilde{y}/\tilde{x}\} \xrightarrow{\alpha} P'}{A(\tilde{y}) \xrightarrow{\alpha} P'} \; A(\tilde{x}) \stackrel{def}{=} P$$

$$PAR: \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \; bn(\alpha) \cap fn(Q) = 0$$

$$COM: \frac{P \xrightarrow{\bar{x}y} P' \quad Q \xrightarrow{x(z)} Q'}{P|Q \xrightarrow{\tau} P'|Q'\{y/z\}} \qquad CLOSE: \frac{P \xrightarrow{\bar{x}(w)} P' \quad Q \xrightarrow{x(w)} Q'}{P|Q \xrightarrow{\tau} (w)(P'|Q')}$$

$$RES: \frac{P \xrightarrow{\alpha} P'}{(y)P \xrightarrow{\alpha} (y)P'} \; y \notin n(\alpha) \qquad OPEN: \frac{P \xrightarrow{\bar{x}y} P}{(y)P \xrightarrow{\bar{x}(w)} P'\{w/y\}} \; y \neq x, \; w \notin fn((y)P')$$

Taulukko 27: π-kalkyylin siirtymäsäännöt

prosesseista ovat prosessit $(P \mid Q)$ ja $(Q \mid P)$. Näiden prosessien tulisi selvästikin olla ekvivalentteja rinnakkaiskompositio-operaattorin intuitiivisen tulkinnan mukaan. Tällaisten prosessien välisten yksinkertaisten ekvivalenssien tunnistamiseksi käytetään rakenteellisen kongruenssin sääntöjä (*structural congruence*). Rakenteellista kongruenssia käytetään lähinnä erilaisten ekvivalenssien ja siirtymäsääntöjen määrittelyssä ja niiden yksinkertaistamiseen. Esimerkiksi ilman sääntöä $P \mid Q \equiv Q \mid P$ täytyisi siirtymäsäännöissä ottaa erikseen tällainen yksinkertainen rakenteellinen samankaltaisuus huomioon.

Rakenteellisia kongruensseja on useita, jotka poikkeavat hieman toisistaan käyttötarpeen mukaan [Par01]. Taulukossa 28 on eräs versio rakenteellisen kongruenssin säännöistä [Par01].

Rakenteellisen kongruenssin määrittelyn jälkeen voidaan π-kalkyylin siirtymäsääntöihin (taulukko 27) lisätä sääntö $STRUCT$, joka ilmaisee että rakenteellisesti kongruentit agentit tulkitaan ekvivalenteiksi:

$$STRUCT: \frac{P' \equiv P \quad P \xrightarrow{\alpha} Q \quad Q \equiv Q'}{P' \xrightarrow{\alpha} Q'}$$

Edellä määriteltyjä siirtymäsääntöjä (taulukko 27 + sääntö $STRUCT$) hyödyntämällä voidaan π-kalkyylin kaavoja todistaa sekä tarkistaa niiden ominaisuuksia. Prosessikuvauksesta, joka on määritelty π-kalkyylillä, kehitetään semanttinen puu (*derivation tree*) siirtymäsääntöjä soveltamalla. Semanttista puuta voidaan prosessialgebrojen kontekstissa kutsua myös siirtymäpuuksi, koska se kuvaa prosessin kaikki mahdolliset siirtymät.

Tämän luvun alussa esitelty esimerkki (kuva 15) on tyypillinen esimerkki järjestelmästä, jonka mallintamiseksi tarvitaan paikallisen nimen kommunikointia. Kuvan 15 järjestelmässä voidaan palvelimen S ja asiakkaan C välinen toiminta kuvata seuraavanlaisella π-kalkyylin lausekkeella: $(a)(\bar{b}a.S) \mid b(c).\bar{c}d.C$. Jos nyt oletetaan, että $a \notin fn(S)$ ja $a \notin fn(C)$, niin haluaisimme että palvelin- ja asiakasprosessien toiminta johtaisi prosessiin $S \mid \bar{a}d.C$, joka vastaa kuvan 15 intuitiivista tulkintaa siitä, että nimen kommunikoinnin jälkeen vain asiakkaalla C on “käyttöoikeus”

Rakenteellinen kongruenssi $\equiv$ on pienin kongruenssi, joka täyttää seuraavat ehdot:

1. Jos P ja Q ovat sidottujen nimien uudelleennimeämistä vaille samat prosessit, niin $P \equiv Q$. Esimerkiksi $a(x).\bar{b}x \equiv a(y).\bar{b}y$. ($\alpha$-konversio)

2. Abelin monoidiehdot operaattoreiden $|$ ja $+$ suhteen:

   – $P \mid Q \equiv Q \mid P$ (vaihdantalaki)
   – $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$ (liitäntälaki)
   – $P \mid 0 \equiv P$ (neutraalialkio)
   – sekä samat säännöt symmetrisesti operaattorille $+$

3. Prosessin lavennuslaki: $A(\tilde{y}) \equiv P\{\tilde{y}/\tilde{x}\} \stackrel{def}{=} P$.

4. Näkyvyysalueen laajennuslait:

   – $(x)0 \equiv 0$
   – $(x)(P \mid Q) \equiv P \mid (x)Q$, jos $x \notin fn(P)$
   – $(x)(P\ +\ Q) \equiv P\ +\ (x)Q$, jos $x \notin fn(P)$
   – $(x)[u = v]P \equiv [u = v](x)P$, jos $x \neq u$ ja $x \neq v$
   – $(x)(y)P \equiv (y)(x)P$

Taulukko 28: π-kalkyylin eräs rakenteellinen kongruenssi

tulostimeen P. Seuraavaksi todistamme, että π-kalkyylin semantiikkaa käyttäen näin tosiaan tapahtuu eli $(a)(\bar{b}a.S) \mid b(c).\bar{c}d.C \stackrel{\tau}{\rightarrow} S|\bar{a}d.C$ on validi päätelmä π-kalkyylissä:

$$\dfrac{\dfrac{\dfrac{}{\bar{b}a.S \stackrel{\bar{b}(a)}{\rightarrow} S}\ OUTPUT-ACT}{(a)(\bar{b}a.S) \stackrel{\bar{b}a}{\rightarrow} (a)S}\ RES \qquad \dfrac{}{b(c).\bar{c}d.C \stackrel{b(c)}{\rightarrow} \bar{c}d.C}\ INPUT-ACT}{(a)(\bar{b}a.S) \mid b(c).\bar{c}d.C \stackrel{\tau}{\rightarrow} S|\bar{a}d.C}\ CLOSE$$

5.3 Mallintarkistus yhteensopivuuden tarkastamisessa

Rajapintaprotokollien yhteensopivuus tarkistetaan siten, että ensin tarkistetaan rajapintaprotokollien varsinainen yhteensopivuus käyttäen yhteensopivuusrelaatiota. Yhteensopivuusrelaatio määritellään luvussa 5.5. Kun varsinainen yhteensopivuus on tarkistettu, voidaan komponenttien välisen yhteistoiminnan ominaisuuksia verifioida ja varmistaa. Yhteistoiminnan ominaisuuksien verifiointi suoritetaan mallintarkistusta käyttäen.

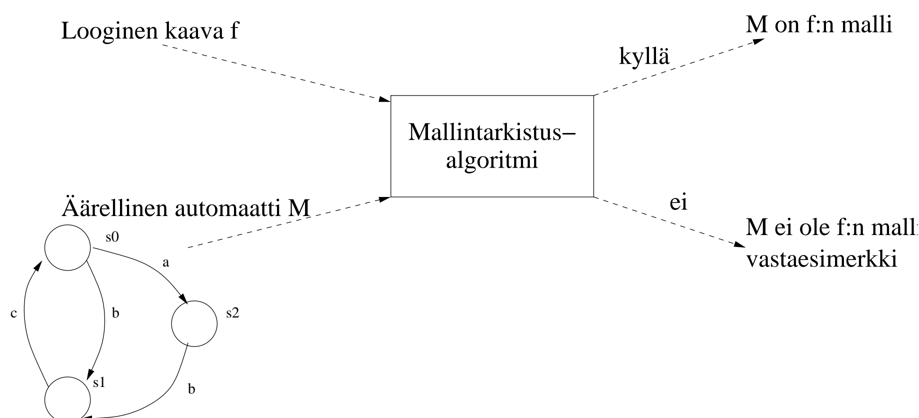
Rajapintaprotokollien P ja Q yhteistoiminta voidaan kuvata niiden rinnakkaiskytkentänä $P \mid Q$. Rajapintaprotokollien yhteentoimivuuden takaamiseksi rinnakkaiskytkennän määrittelemältä yhteistilaverkolta vaaditaan tiettyjä ominaisuuksia. Komponenttien välinen yhteistoiminta ei saa esimerkiksi johtaa järjestelmän lukkiutumiseen. Lukkiutumattomuuden lisäksi yhteistoiminnalta voidaan vaatia muitakin ominaisuuksia, kuten terminointi oikeelliseen lopputilaan tai keskinäinen poissulkeminen.

Tärkeimmät verifioitavista ominaisuuksista ovat joko elävyys- (*liveness*) ja turvallisuusominai-

suuksia (*safety*). Elävyysominaisuudet kertovat, että jotain hyvää tapahtuu, toisin sanoen, järjestelmä ei jää tekemään tyhjää työtä tai lukkiudu. Turvallisuusominaisuudet ovat väittämiä, jotka ilmaisevat että mitään pahaa ei voi tapahtua. Esimerkki yksinkertaisesta eläväisyysominaisuudesta on: "*aina on niin, että uusi asiakas voi kirjautua sisään*". Yksinkertainen esimerkki turvallisuusominaisuudesta on keskinäisen poissulkemisen ominaisuus, joka voitaisiin ilmaista esimerkiksi näin: "*aina on voimassa, että jos asiakas c_i on kriittisellä alueella K, niin kaikilla $j \neq i$ asiakas c_j ei ole kriittisellä alueella*". Rajapintaprotokollien tapauksessa ehdottomasti tärkein turvallisuusominaisuus on se, että rajapintaprotokollien yhteistoiminta ei johda lukkiutumaan.

Rajapintaprotokolla määrittelee äärellisen siirtymäjärjestelmän, jonka avulla komponentin käyttäytymisen ominaisuudet voidaan tarkistaa, kun ominaisuudet tai väittämät muutetaan siirtymäjärjestelmän tiloja tai siirtymiä koskeviksi väitteiksi. Täten esimerkiksi poissulkemisominaisuus voi muuttua muotoon "*missään ohjelman tilassa kriittisellä alueella olevien asiakkaiden määrä ei ole suurempi kuin 1*". Kun komponentilta vaaditut ominaisuudet on muutettu tilaa tai siirtymiä koskeviksi loogisiksi väittämiksi, voidaan ominaisuuksien voimassaolo tarkistaa käymällä komponenttia vastaavan äärellisen automaatin generoimat tilat läpi. Jokaisessa tilassa tarkistetaan, pätevätkö annetut loogiset lausekkeet. Tätä toimintoa kutsutaan *mallintarkistukseksi*.

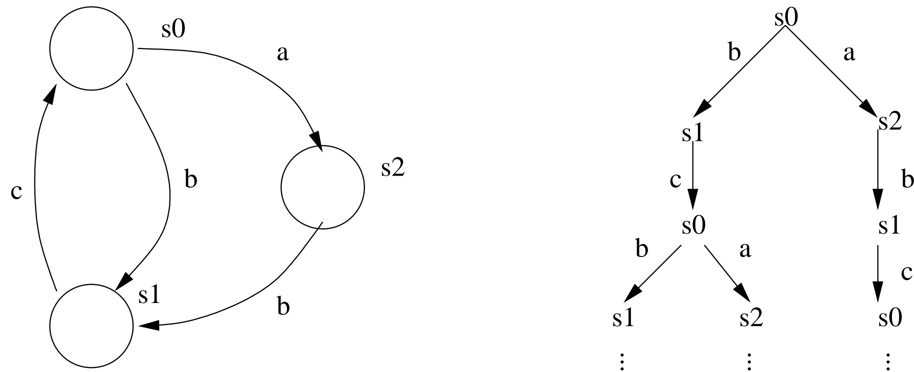
Kuvassa 16 on kuvattu mallintarkistuksen periaate. Mallintarkistusalgoritmi ottaa vastaan tarkastettavan automaatin kuvauksen M sekä loogisen lausekkeen f, joka ilmaisee äärelliseltä automaatilta M vaaditut ominaisuudet. Mallintarkistuksessa tarkistetaan, onko annettu struktuuri, automaatti M, annetun loogisen lausekkeen f malli [CES86]. Jos struktuuri M on kaavan f malli, niin merkitään $M \models f$. Toisaalta, jos $M \not\models f$, niin mallintarkistusalgoritmit yleensä lopettavat toimintansa siihen tilaan, missä kaava f ei enää pätenyt. Purkamalla tilapino saadaan vastaesimerkki, joka edustaa virheelliseen tilaan johtanutta suoritusjälkeä.



Kuva 16: Mallintarkistuksen periaate

Jokaisessa automaatin tilassa tarkistetaan, ovatko kaikki vaaditut väittämät voimassa. Väittämät koskevat joko automaatin tiloja tai suoritusjälkiä. Tiloja koskevat väittämät voivat olla esimerkiksi muotoa *"kaikissa tiloissa on voimassa ominaisuus ϕ"*. Suoritusjälkiä koskeva väittämä on esimerkiksi "*on olemassa sellainen suoritusjälki, jonka kaikille tiloille pätee ϕ"*. Kuvassa 17 on äärellinen automaatti, jolla on kolme tilaa (s_0, s_1 ja s_2) sekä neljä siirtymää. Automaatista saadaan suorituspuu, joka sisältää useita suoritusjälkiä. Äärellisen automaatin suoritusjälkien alut on esitetty kuvan 17 vasemmalla laidalla.

Äärelliseltä automaatilta halutut ominaisuudet määritellään *temporaalilogiikan* avulla. Temporaa-



Kuva 17: Äärellinen automaatti ja siitä generoituvat suoritusjäljet [CES86]

lilogiikka on predikaattilogiikan laajennus, joka lisää logiikkaan ajallisia ja modaalisia väittämiä [1]. Temporaalilogiikan kaava ϕ on tosi jos ja vain jos väittämä on tosi oikeana ajan hetkenä.

Temporaalilogiikat voidaan jakaa kahteen ryhmään sen mukaan kuinka ne tulkitsevat aikaa ja siihen liittyviä ilmaisuja: lineaarisiin ja haarautuvan ajan temporaalilogiikkoihin [EH86]. Lineaarisen ajan tulkinnan mukaan mahdollisia tulevaisuuksia on kussakin tilassa olemassa vain yksi. Tällaista ajan tulkintaa tukevaa logiikkaa kutsutaan lineaariseksi temporaalilogiikaksi. Lineaarisessa temporaalilogiikassa aikaan ja modaliteetteihin liittyvät ilmaisut käsittelevät tapahtumia yksittäisen aikajanan sisällä ja lineaarisen temporaalilogiikan kaavan totuus riippuu tietyn suoritusjäljen ominaisuuksista [EH86].

Haarautuvan ajan tulkinnan mukaan kussakin suorituksen tilassa voi olla valittavissa useita mahdollisia tulevaisuuksia. Ajalla on haarautuva, puumainen rakenne ja temporaalilogiikan kaavan totuus tulkitaan tilakohtaisesti [EH86]. Tälläisia logiikoita kutsutaan haarautuvan ajan temporaalilogiikoiksi ja niistä ehkä tunnetuin on temporaalilogiikka nimeltään CTL [CES86].

Tässä opinnäytetyössä tutustutaan temporaalilogiikkaan $ACTL$-kielen (*Action-based Computation Tree Logic*) avulla [DFGR93]. ACTL-logiikka on tilasiirtymäjärjestelmien verifioimiseen kehitetty haarautuvan ajan temporaalilogiikka, joka pohjautuu CTL-temporaalilogiikkaan. ACTL-logiikan väittämät koskevat tilasiirtymäjärjestelmän $\mathcal{A} = (Q, A_\tau, \rightarrow, 0_Q)$ siirtymiä, kun taas perinteisissä, CTL:n kaltaisissa temporaalilogiikoissa väittämät koskevat tiloja ja kaavat tulkitaan niin kutsuttujen Kripke-struktuurien avulla [DFGR93, CES86]. Kripke-struktuurit ovat tilasiirtymäjärjestelmiä, joissa jokaiseen tilaan on liitetty tieto siitä, mitkä temporaalilogiikan kaavojen atomisista propositioista ovat voimassa kussakin tilassa [CES86].

ACTL-logiikan kaavoissa käytetään *siirtymäkaavoja*, jotka ovat muotoa $\mathcal{X} = a \mid \neg\mathcal{X} \mid \mathcal{X} \vee \mathcal{X}'$, $a \in A$. Siirtymäkaavat määrittelevät siirtymiä koskevia rajoitteita. Esimerkiksi kaavalla $a \vee b$ tarkoitetaan sitä, että ainoat mahdolliset havaittavat siirtymät tietystä tilasta ovat joko a tai b. Siirtymäkaavoissa voidaan käyttää myös lyhennemerkintöjä $\mathbf{true} = \forall a_0 : a_0 \vee \neg a_0$, missä a_0 on mikä tahansa siirtymä ja $\mathbf{false} = \neg\mathbf{true}$. Siirtymäkaava $\mathbf{true}$ ilmaisee, että tilasta voidaan suorittaa mikä tahansa siirtymä, kun taas $\mathbf{false}$ merkitsee sitä, että tilasta ei voida suorittaa mitään havaittavaa siirtymää [DFGR93]. Siirtymä $\alpha \in A$ toteuttaa siirtymäkaavan $\mathcal{X}$ määritelmän 5.2 sääntöjä noudattaen[DFGR93].

[1]Tässä opinnäytetyössä käytetään yleisen käytännön mukaisesti ilmausta *temporaalilogiikka* viittaamaan verifioinnissa käytettävään logiikkaan, jolla voidaan määritellä sekä ajallisia että modaalisia väittämiä. Oikea termi tällaiselle logiikalle olisi temporaalinen modaalilogiikka

Määritelmä 5.2 (Siirtymäkaavojen toteutuminen)
Siirtymä α toteuttaa siirtymäkaavan $\mathcal{X}$, merkitään $\alpha \models \mathcal{X}$, seuraavien sääntöjen mukaisesti:

$\alpha \models \mathbf{true}$	*pätee aina*
$\alpha \models \neg\mathcal{X}$	*jos ja vain jos* $\alpha \not\models \mathcal{X}$
$\alpha \models \mathcal{X} \vee \mathcal{X}'$	*jos ja vain jos* $\alpha \models \mathcal{X}$ *tai* $\alpha \models \mathcal{X}'$

Tilakaavat, joita merkitään symboleilla $\phi, \phi', ...$, ovat väittämiä tietyn tilan suhteen. Kaava $q_0 \models \phi$ on voimassa, jos ja vain jos tilakaava ϕ on tosi tilassa q_0.

Polkukaavat, joita merkitään symboleilla $\gamma, \gamma', ...$ ovat kaavoja, joiden totuus määräytyy jonkin tilojen polun (suoritusjäljen) $\pi = q_1,\ q_2,\ ..., q_n, n \geq 0$ suhteen. Kaava $\pi \models \gamma$ on voimassa, jos ja vain jos tilojen polku π on polkukaavan γ määrittelemien ominaisuuksien mukainen polku [DFGR93].

ACTL-temporaalilogiikan kaavat muodostetaan yhdistämällä siirtymä-, tila- ja polkukaavoja loogisiksi väittämiksi seuraavaksi esiteltävien operaattoreiden avulla. Temporaalilogiikan kaavat muodostetaan käyttämällä kahdenlaisia operaattoreita: temporaali- ja polkuoperaattoreita. Temporaalilogiikan kaavojen yhteydessä käytetään merkintätapoja, jotka on annettu taulukossa 29.

q, q_0, q_i	äärellisen automaatin tiloja
$\mathcal{X}, \mathcal{X}', ...$	siirtymäkaavoja
$\phi, \phi', ...$	tilakaavoja
$\gamma, \gamma', ...$	polkukaavoja
π	tilojen polku $q_0, q_1, ..., q_i, i \geq 0$, joka voidaan myös merkitä $\pi(0), \pi(1), ..., \pi(i)$.
$\Pi(q)$	kaikkien tilasta q_0 lähtevien polkujen $\pi_1, \pi_2, ..., \pi_n$ joukko

Taulukko 29: ACTL-temporaalilogiikan kaavoissa käytettävät merkintätavat

Temporaalioperaattoreiden avulla voidaan muodostaa aikaa koskevia väittämiä ja niiden avulla määriteltyjen kaavojen totuus määräytyy jonkin tilojen polun π suhteen. CTL-logiikkaan pohjautuville temporaalilogiikoille on tyypillisesti määritelty kaksi temporaaliperaattoria, $Until$ ja $Next$ [CES86]. Näiden kahden temporaalioperaattorin avulla voidaan ilmaista lähes kaikki verifioinnissa tarvittavat ajalliset ominaisuudet, kuten ajalliset invariantit ("aina on voimassa ominaisuus ϕ") ja välttämättömyydet ("lopulta on voimassa ominaisuus ϕ") [CES86, DFGR93].

ACTL-logiikan temporaalioperaattori $Until$ tulkitaan siten, että kaava $f_1 Until f_2$ on tosi polulle π, merkitään $\pi \models (f_1 U f_2)$, jos ja vain jos $\exists i[i \geq 0 \wedge \pi(i) \models f_2 \wedge \forall j[0 \leq j < i \rightarrow \pi(j) \models f_1]]$ on voimassa. Temporaalilauseke $f_1\ U\ f_2$ ilmaisee siis sen, että f_1 on voimassa aina siihen asti kunnes f_2 tulee voimaan [CES86]. Toinen ACTL-logiikan temporaalioperaattoreista on $Next$ (merkitään temporaalikaavoissa muodossa $X\phi$), jonka tulkinta on se, että kaava $\pi \models X\phi$ on voimassa, jos ja vain jos $\pi(1) \models \phi$ [CES86].

Haarautuvan ajan temporaalilogiikan polkuoperaattorit ovat suurin ero lineaarisen ajan temporaalilogiikkoihin verrattuna. Polkuoperaattoreiden avulla voidaan ilmaista väittämiä, jotka ottavat huomioon ajan tulkinnan puumaisen rakenteen. Polkuoperaattorit ovat eksistentiaalikvanttori $\exists$ ja universaalikvanttori $\forall$. Polkuoperaattoreita käytetään yhdessä polkukaavojen γ kanssa, jotka esittävät väittämiä yhtä puun polkua koskien. Kaava $q_0 \models \exists\gamma$ on tosi, jos ja vain jos on olemassa jokin tilasta q_0 lähtevä polku π siten, että $\pi \models \gamma$ [DFGR93]. Universaalikvanttoria käytetään samalla tavoin kuin eksistentiaalikvanttoria ja $q_0 \models \forall\gamma$ on tosi, jos ja vain jos kaikille poluille $\pi_i \in \Pi(q_0), i \geq 0$ on voimassa $pi_i \models \gamma$.

ACTL-logiikan kaavojen lopullinen syntaksi on annettu taulukossa 30. Tilakaavojen termit ovat

taulukossa 30 määritellyn ϕ-termin mukaisia ja polkukaavojen termit noudattavat γ-termin mukaista syntaksia.

$$\begin{aligned} \phi &::= \mathbf{true} \mid \neg\phi \mid \phi \wedge \phi' \mid \exists\gamma \mid \forall\gamma \\ \gamma &::= X_{\mathcal{X}}\ \phi \mid X_\tau\ \phi \mid \phi\ {}_{\mathcal{X}}U_{\mathcal{X}'}\ \phi' \mid \phi\ {}_{\mathcal{X}}U\ \phi' \end{aligned}$$

Taulukko 30: ACTL-logiikan kaavojen syntaksi

Taulukossa tiloja koskevia kaavoja on merkitty symboleilla $\phi, \phi', ...$, polkukaavaa symbolilla γ ja siirtymäkaavoja symboleilla $\mathcal{X}, \mathcal{X}', ...$.

ACTL-logiikan kaavalle ϕ annetaan tulkinta siirtymäsystemin $\mathcal{A} = (Q, A_\tau, \rightarrow, 0_Q)$ avulla, missä Q on joukko tiloja, A_τ on automaatin aakkosto, joka sisältää sisäisen siirtymän τ, siirtymäfunktio on $\rightarrow$ ja alkutila on 0_Q [DFGR93]. ACTL-logiikan kaavat tulkitaan määritelmän 5.3 mukaisesti.

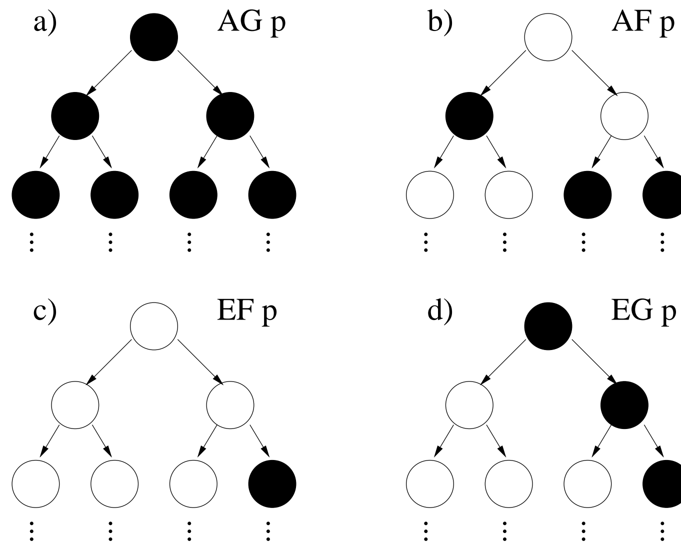
Määritelmä 5.3 (ACTL-temporaalilogiikan semantiikka)
Olkoon $\mathcal{A} = (Q, A_\tau, \rightarrow, 0_Q)$ siirtymäjärjestelmä. Tällöin tilakaavan ϕ (polkukaavan γ) totuus tulkitaan seuraavien sääntöjen mukaan, kun $q \in Q, \pi \in \Pi$:

$q \models \mathbf{true}$	*aina*
$q \models \phi \wedge \phi'$	*jos ja vain jos $q \models \phi$ ja $q \models \phi'$*
$q \models \phi \vee \phi'$	*jos ja vain jos $q \models \phi$ tai $q \models \phi'$*
$q \models \neg\phi$	*jos ja vain jos $q \not\models \phi$*
$q \models \exists\gamma$	*jos ja vain jos on olemassa polku $\pi \in \Pi(q)$ siten, että $\pi \models \gamma$*
$q \models \forall\gamma$	*jos ja vain jos kaikille poluille $\pi \in \Pi(q)$ on voimassa $\pi \models \gamma$*
$\pi \models \phi\ {}_{\mathcal{X}}U_{\mathcal{X}'}\ \phi'$	*jos ja vain jos $\exists i > 0$ siten että*
	$\pi(i) \models \phi', (\pi(i-1) \xrightarrow{a} \pi(i)) \in\rightarrow, a \models \mathcal{X}'$ *ja*
	$\forall 0 \leq j < i : \pi(j) \models \phi$ *ja* $\forall(\pi(j-1), \xrightarrow{\alpha} \pi(j)) \in\rightarrow: \alpha \models \mathcal{X}$ *tai* $\alpha = \tau$
$\pi \models \phi\ {}_{\mathcal{X}}U\ \phi'$	*jos ja vain jos $\exists i > 0$ siten että*
	$\pi(i) \models \phi'$ *ja* $\forall 0 \leq j < i : \pi(j) \models \phi$ *ja*
	$\forall(\pi(j-1), \xrightarrow{\alpha} \pi(j)) \in\rightarrow: \alpha \models \mathcal{X}$ *tai* $\alpha = \tau$
$\pi \models X_{\mathcal{X}}\ \phi$	*jos ja vain jos* $(\pi(0) \xrightarrow{a} \pi(1)) \in\rightarrow$ *ja* $a \models \mathcal{X}, \pi(1) \models \phi$
$\pi \models X_\tau\ \phi$	*jos ja vain jos* $(\pi(0) \xrightarrow{\tau} \in\rightarrow$ *ja* $\pi(1) \models \phi$

Määritelmässä 5.3 esiteltyjen sääntöjen avulla voidaan johtaa uusia apuoperaattoreita, jotka helpottavat temporaalilogiikan kaavojen kirjoittamista käytännössä. Tällaisia johdettuja operaattoreita ovat esimerkiksi operaattorit $AF\phi = \forall(\mathbf{true}\ {}_{\mathbf{true}}U\ \phi)$ ja $EF\phi = \exists(\mathbf{true}\ {}_{\mathbf{true}}U\ \phi)$, joista ensimmäinen ilmaisee, että ϕ on välttämättä tosi ja jälkimmäinen määrittelee, että ϕ on potentiaalisesti tosi [CES86]. Näiden lisäksi erittäin usein verifioinnissa tarvittavia ilmauksia ovat $AG\phi = \neg EF\neg\phi$ ja $EG\phi = \neg AF\neg\phi$. Näistä ensimmäinen määrittelee ominaisuuden ϕ olevan invariantti, eli ei ole olemassa yhtään suorituspolkua, missä ϕ ei olisi voimassa. Kaava $EG\phi$ ilmaisee, että on olemassa ainakin yksi polku π siten, että ϕ on voimassa kaikissa polun π tiloissa [CES86].

Kuvassa 18 on neljä esimerkkiä suorituspuista, sekä edellä määritellyistä johdetuista ACTL-kaavoista. Jokaisen puun viereen on merkitty temporaalilogiikan kaava, joka pätee kyseisen puun juuressa, kun täytetty ympyrä kuvaa tilaa, jossa tilakaava p pätee.

Mallintarkistuksella voidaan äärellisestä automaatista tarkistaa monia mielenkiintoisia ominaisuuksia. Esimerkiksi lukkiutumattomuus voidaan mallintaa ACTL-kaavana $AG(\exists X_\alpha \mathbf{true})$ (“jo-



Kuva 18: Esimerkkejä temporaalilogiikan kaavoista

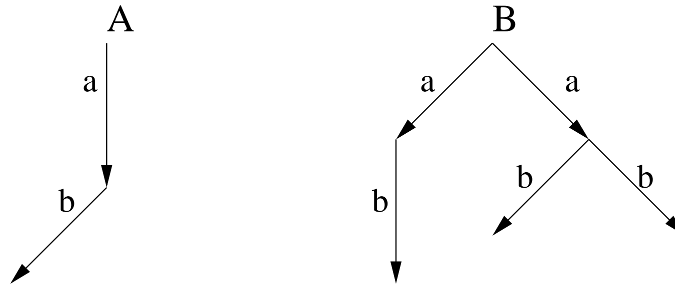
kaisessa tilassa on voimassa, että on olemassa jokin tila, johon on siirtymä α"). Turvallisuusominaisuudesta esimerkkinä on kaava $AG(\forall(\neg k))$, missä $k \in A$ on jokin siirtymä (kielletty toiminto), jolloin kaava ilmaisee, että milloinkaan ei saada joutua tilaan, missä olisi edes mahdollista suorittaa siirtymää k. Elävyysominaisuudet jonkin tapahtuman ϕ suhteen voidaan ilmaista kaavalla, joka on muotoa $AG(EF(\phi))$. Tämä kaava on tosi, jos ja vain jos kaikista tiloista on mahdollista päästä sellaiseen tilaan, missä ϕ on tosi.

5.4 Rajapintaprotokollien korvautuvuus ja prosessien ekvivalenssit

Rajapintaprotokollien välinen korvautuvuus tulkitaan ulkoisen tarkkailijan näkökulmasta: kaksi rajapintaprotokollaa ovat keskenään korvattavissa, jos ja vain jos ne ulkoiselta toiminnaltaan ovat erottamattomat. Prosessialgebroille on kehitetty useita ekvivalensseja, joista osa keskittyy nimenomaan ulkoisen käyttäytymisen samankaltaisuuteen. Tässä aliluvussa tutustutaan prosessien välisiin ekvivalensseihin ja nimenomaan niin kutsuttuihin bisimulointiekvivalensseihin.

Tämän aliluvun tarkoituksena on tutustua perinteisiin prosessiekvivalensseihin ja niiden periaatteisiin. Komponenttien rajapintaprotokollien korvautuvuusrelaatioksi tässä luvussa esitellyt prosessiekvivalenssit eivät suoraan kelpaa, kuten tullaan huomaamaan luvussa 5.5. Prosessiekvivalenssien periaatteet ovat kuitenkin melko suoraan siirrettävissä rajapintaprotokollien korvautuvuusrelaation käyttöön.

Ulkoiseen käyttäytymiseen perustuvan prosessien tarkastelutavan mukaan kaksi prosessia ovat ekvivalentteja, jos mikään ulkopuolinen tarkkailija ei pysty erottamaan niitä toisistaan ulkoisen käyttäytymisen perusteella [HM85]. Jos kahta eri prosessia vastaavat siirtymäpuut ovat identtisiä, niin prosessit ovat varmasti ekvivalentteja, koska siirtymäpuut sisälsivät kaikki prosessin mahdolliset suoritusjäljet [Mil89b]. Siirtymäpuiden identtisyys on kuitenkin liian vahva ehto, jos prosesseja halutaan verrata niiden ulkoisen käyttäytymisen perusteella [Mil89b]. Kuvassa 19 on kaksi agenttia A ja B. Sekä agentti A että B käyttäytyvät ulospäin samalla tavalla, koska niillä on vain yksi mahdollinen suoritusjälki, ab. Niiden siirtymäpuut ovat kuitenkin erilaiset, johtuen agentin B epädeterminismistä.



Kuva 19: Agenttien A ja B siirtymäpuut

Mikä sitten on sopiva ekvivalenssi kahden prosessikuvauksen ulkoisen käyttäytymisen vertaamiseen? Ensimmäiseksi tulee mieleen, että prosessien tulisi olla ekvivalentteja, jos ja vain jos niillä on samat ulospäin havaittavat siirtymät ja ne voidaan suorittaa samassa järjestyksessä. Toisin sanoen, prosessit ovat ekvivalentteja, jos ja vain jos niillä on (täsmälleen) samat suorituspolut. Tämä suoritusjälkiekvivalenssiksi (*trace equivalence*) kutsuttu relaatio ei kuitenkaan ole tarpeeksi voimakas ehto, koska se ei pysty erottamaan päättyikö prosessi onnistuneesti vai lukkiutuiko se [MPW89]. Esimerkiksi prosessit $P = a.b$ ja $Q = a + (a.b)$ ovat suoritusjälkiekvivalentteja, niillä molemmilla on suoritusjäljet $\{\epsilon, a, ab\}$, mutta Q voi lukkiutua siirtymän a jälkeen toisin kuin P [Shi93].

Prosessialgebrojen yhteydessä yleisimmin käytetään niin kutsuttua *bisimulointia* prosessien (ulkoisen) käyttäytymisen vertaamiseen. Intuitiivisesti bisimulaatio merkitsee sitä, että prosessi P voi simuloida prosessia Q ja prosessi Q voi simuloida prosessia P. Toisin kuin prosessien suoritusjälkiekvivalenssi, joka on lineaarisen ajan ekvivalenssi, on bisimulointirelaatio haarautuvan ajan relaatio. Lineaarisen ajantulkinnan prosessiekvivalensseissa prosessit identifioidaan niiden mahdollisten suoritusjälkien perusteella, kun taas haarautuvan ajan prosessiekvivalensseissa aika tulkitaan puumaisena, haarautuvana rakenteena [vW96].

Haarautuvan ajan huomioiva relaatio erottaa siirtymien lisäksi ajanhetken, jolloin jokin valinta voidaan tehdä. Bisimulointi esimerkiksi erottaa, kuten yleensä prosessien yhteydessä on hyödyllistä, prosessit $a(b + c)$ ja $(ab + bc)$ toisistaan [Mil89b]. Edelliset prosessit ovat polkuekvivalentteja, mutta niissä tehdään haarautumisen valinta eri ajanhetkenä.

Määritelmä 5.4 kuvaa niin sanotun *vahvan* bisimulaation yleisen muodon, jonka mukaan prosessit ovat ekvivalentit, jos ja vain jos niillä on täsmälleen samat siirtymät, mukaan lukien sisäiset siirtymät τ.

Määritelmä 5.4 (Vahva bisimulaatio) *(Vahva) bisimulaatio on relaatio $\mathcal{R} \subseteq S \times S$, missä S on prosessitilojen joukko, siten että*

- *jos $(s, t) \in \mathcal{R}$ ja $s \xrightarrow{\alpha} s'$, niin $\exists t'$ siten että $t \xrightarrow{\alpha} t'$ ja $(s', t') \in \mathcal{R}$*
- *jos $(s, t) \in \mathcal{R}$ ja $t \xrightarrow{\alpha} t'$, niin $\exists s'$ siten että $s \xrightarrow{\alpha} s'$ ja $(s', t') \in \mathcal{R}$*

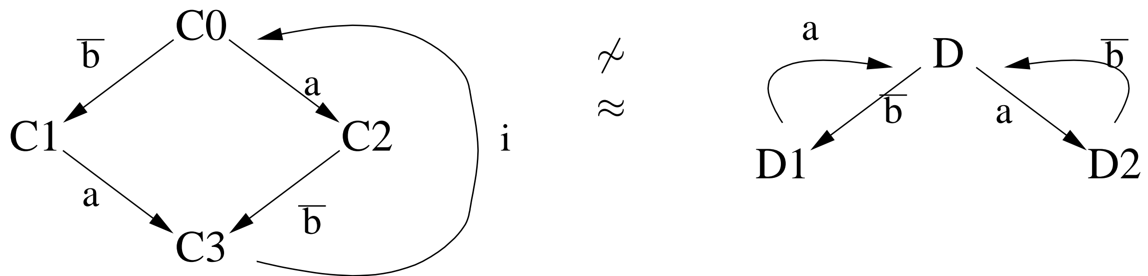
Vahvan bisimulaation määritelmä on siis tavallaan “rekursiivinen”. Jotta tilat s ja t olisivat bisimulaatiorelaatiossa, niin vaatimuksena on että myös niiden jälkeläiset ovat bisimulaatiorelaatiossa ja niin edelleen.

Vahvan bisimulaation avulla voidaan prosessien välillä määritellä prosessien vahva ekvivalenssi eli vahva bisimilaarisuus $\sim$. Määritelmä on annettu määritelmässä 5.5.

Määritelmä 5.5 (Vahva bisimilaarisuus) *Prosessit P ja Q ovat vahvasti ekvivalentit eli vahvasti bisimilaariset, merkitään $P \sim Q$, jos ja vain jos $(P, Q) \in \mathcal{R}$ jollakin vahvalla bisimulaatiolla $\mathcal{R}$.*

Määritelmän 5.5 mukainen yhteys prosessien bisimuloinnin ja prosessien välisen bisimilaarisuuden (ekvivalenssin) välillä on voimassa kaikilla tässä opinnäytetyössä esiteltävillä bisimuloinnin versioilla.

Prosessit P ja Q ovat siis bisimilaariset jos ja vain jos on olemassa jokin bisimulointi ja (P, Q) on tämän bisimuloinnin alkio. Kuvassa 20 on kaksi prosessia C_0 ja D, jotka bisimulaatio intuitiivisen tulkinnan mukaan tulisivat olla ekvivalentit, koska ne käyttäytyvät ulkoisesti samalla tavalla. Prosessien C_0 ja D bisimulaatio $\mathcal{R}$ on joukko prosessien tilojen välisiä relaatioita: $\mathcal{R} = \{(C_0, D), (C_1, D1), (C_2, D_2), (C_3, D)\}$. Bisimulointi $\mathcal{R}$ ei ole kuitenkaan vahva bisimulointi, koska $C_3 \not\sim D$. Tämä johtuu siitä, että $C_3 \xrightarrow{\tau} C_0$ mutta $D \stackrel{\tau}{\not\to} D'$.



Kuva 20: Vahva ja heikko bisilaarisuus

Vahva bisimulointi on liian rajoittava, koska se ei "kätke" sisäisiä τ-siirtymiä. Tämän vuoksi tarvitaan relaatio, joka ei ota huomioon prosessien sisäisiä siirtymiä. Tätä relaatiota kutsutaan *heikoksi bisimulaatioksi* ja sen avulla saatavaa prosessien ekvivalenssirelaatiota *havaintoekvivalenssiksi* [Mil89b]. Kuvan 20 automaatit ovat heikosti bisimilaarisia. Heikon bisimulaation yleinen määritelmä on annettu määritelmässä 5.6. Heikon bisimulaation määritelmässä käytetään merkintää $p \stackrel{\alpha}{\Rightarrow} p'$ kuvaamaan suoritusjälkeä, joka on muotoa $p \xrightarrow{\tau^*} q \xrightarrow{\alpha} q' \xrightarrow{\tau^*} p'$, missä $\xrightarrow{\tau^*}$ tarkoittaa nollan tai useamman τ-siirtymän jonoa [Mil82]. Siirtymä $p \stackrel{\alpha}{\Rightarrow} p'$ identifioi siis kaikki suoritusjäljet, joissa esiintyy siirtymä α ja kaikki muut mahdollisesti esiintyvät siirtymät tilojen p ja p' välillä ovat τ-siirtymiä.

Määritelmä 5.6 (Heikko bisimulaatio) *Heikko bisimulaatio on relaatio $\mathcal{R} \subseteq S \times S$, missä S on prosessitilojen joukko, siten että*

– *jos $(s, t) \in \mathcal{R}$ ja $s \xrightarrow{\alpha} s'$, niin $\exists t'$ siten että $t \stackrel{\alpha}{\Rightarrow} t'$ ja $(s', t') \in \mathcal{R}$*

– *jos $(s, t) \in \mathcal{R}$ ja $t \xrightarrow{\alpha} t'$, niin $\exists s'$ siten että $s \stackrel{\alpha}{\Rightarrow} s'$ ja $(s', t') \in \mathcal{R}$*

Ohjelmistoteknisestä näkökulmasta tärkein ominaisuus käytettävälle prosessien ekvivalenssirelaatiolle on se, että se identifioi kaksi prosessia P ja Q siten, että prosessit ovat vaihdettavissa keskenään *missä ympäristössä tahansa*. Tätä relaation ominaisuutta tietyssä algebrassa kutsutaan algebran kongruenssiksi, joka intuitiivisesti voidaan tulkita siten, että kahden algebran termin välinen suhde säilyy muuttumattomana riippumatta siitä ympäristöstä, missä niitä käytetään.

Prosessialgebran tapauksessa kongruessin määrittely saa muodon, joka on esitetty määritelmässä 5.7. Prosessialgebran kongruenssin määritelmässä käytetään hyväksi kontekstin käsitettä. Kontekstiksi kutsutaan jotakin prosessia, jossa on "tyhjä paikka" [ ]. Tähän tyhjään paikkaan voidaan

sijoittaa mikä tahansa prosessi. Jos esimerkiksi C on konteksti $C[\,] = a.[\,] + \overline{a}.0$, niin prosessin P sijoitusta kontekstiin C merkitään $C[P] = a.P + \overline{a}.0$.

Määritelmä 5.7 (Ekvivalenssirelaation kongruenssi) *Jos relaatiolle $\mathcal{R} \in \mathcal{S} \times \mathcal{S}$ ja prosesseille $P, Q \in \mathcal{S}$, missä $\mathcal{S}$ on prosessilausekkeiden joukko, on voimassa $P\mathcal{R}Q$, niin $\mathcal{R}$ on kongruenssi jos ja vain jos $C[P]\ \mathcal{R}\ C[Q]$ kaikille konteksteille $C[\,]$.*

Käytännössä relaation $\mathcal{R}$ todistaminen kongruenssiksi tapahtuu siten, että sen todistetaan olevan suljettu kaikkien algebran operaatioiden suhteen. Esimerkiksi vahva bisimulaatio CCS-prosessialgebrassa on kongruenssi koska, jos oletetaan että $P \sim Q$, niin $\sim$ on suljettu kaikkien CCS-prosessialgebran operaattoreiden suhteen [Mil89b]. Tämä ominaisuus on kuvattu lausessa 5.1.

Lause 5.1 (CCS-prosessialgebran vahva kongruenssi) *Jos $P \sim Q$, niin*

1. $\alpha.P \sim \alpha.Q$
2. $P + R \sim Q + R$
3. $P \mid R \sim Q \mid R$
4. $P \backslash L \sim Q \backslash L$
5. $P[f] \sim Q[f]$

(Todistus sivuutetaan)

Erityisesti täytyy huomata, että toisin kuin vahva bisimulointi, heikko bisimulointi ei ole kongruenssirelaatio CCS-prosessialgebrassa. Syynä tähän on se, epädeterministisen valinta ei säilytä heikkoa bisimilaarisuutta prosessien välillä [Mil89b]. Esimerkiksi $b.\mathbf{0} \approx \tau.b.\mathbf{0}$, mutta prosessit $a.\mathbf{0} + b.\mathbf{0}$ ja $a.\mathbf{0} + \tau.b.\mathbf{0}$ eivät ole heikosti bisimilaariset. CCS-prosessialgebraan on tämän vuoksi määritelty niin kutsuttu havaintokongruenssi, joka ilmaisee sen, että prosessit P ja Q ovat havaintoekvivalentit jos ja vain jos $P + R \approx Q + R$ kaikilla prosesseilla R [Mil89b]. Tällainen ekvivalenssirelaation ulkopuolinen ehto kongruenssin saavuttamiseksi tietenkin monimutkaistaa prosessien ekvivalenssin tarkistamista. Ideaalinen havaintoekvivalenssi olisi sellainen, joka olisi jo määritelmänsä puolesta kongruenssi käytetyn algebran suhteen. Tällöin komponenttien vertailuun voidaan kehittää tehokkaampia tarkistusalgoritmeja.

On myös toinen perustavaa laatua oleva syy, miksi Milnerin määrittelemä havaintoekvivalenssi $\approx$ ei sovellu komponenttien yhteentoimivuuden tarkistamiseen. Vaikka bisimulaation tulisi olla haarautuvan ajan ekvivalenssi, niin heikko bisimulaatioekvivalenssi ei tietyssä mielessä säilytä haarautumisen ominaisuuksia prosessien välillä [vW96]. Esimerkiksi prosessit $a(\tau.b + c)$ ja $a(\tau.b + c) + a.b$ ovat heikosti bisimilaariset, mutta jälkimmäisessä valinta suorituspolkujen ab ja ac voidaan tehdä eri ajankohtana kuin ensimmäisessä prosessissa. Kuitenkin komponenttien toiminnallisuuden laajennuksen yhteydessä tarvitaan relaatiota, joka säilyttää prosessin ominaisuudet haarautuvan ajan suhteen myös τ-polkujen varrella. Tälle vaatimukselle annetaan perustelu luvussa 5.5.

5.4.1 Prosessien ekvivalenssi π-kalkyylissä

Bisimulaatiorelaatiota voidaan käyttää myös π-kalkyylissä prosessien käyttäytymisen yhteneväisyyden tarkistamiseen. Vahvan bisimulaation määrittely π-kalkyylissä on annettu määritelmässä 5.8 [MPW89].

Määritelmä 5.8 (Vahva bisimulointi π-kalkyylissä)
Vahva bisimulointi $\mathcal{R}$ on relaatio $\mathcal{R} \in S \times S$, siten että

1. *jos $(P, Q) \in \mathcal{R}$ ja $P \xrightarrow{\alpha} P'$, α on vapaa (τ- tai vapaa output-siirtymä) siirtymä, niin $\exists Q'$ siten että*
   $Q \xrightarrow{\alpha} Q'$ *ja* $(P', Q') \in \mathcal{R}$

2. *jos $(P, Q) \in \mathcal{R}$ ja $P \xrightarrow{x(y)} P'$, $y \notin n(P, Q)$ niin $\exists Q'$ siten että*
   $Q \xrightarrow{x(y)} Q'$ *ja* $\forall w, (P'\{w/y\}, Q'\{w/y\}) \in \mathcal{R}$

3. *jos $(P, Q) \in \mathcal{R}$ ja $P \xrightarrow{\bar{x}y} P'$, $y \notin n(P, Q)$ niin $\exists Q'$ siten että*
   $Q \xrightarrow{\bar{x}y} Q'$ *ja* $(P', Q') \in \mathcal{R}$

Prosessit P ja Q ovat (vahvasti) bisimilaarisia, merkitään $P \dot{\sim} Q$, jos on olemassa jokin bisimulointi $\mathcal{R}$ ja $(P, Q) \in \mathcal{R}$ [MPW89].

Bisimuloinnin periaatteet eivät kuitenkaan ole suoraan siirrettävissä CCS:stä π-kalkyyliin. Syy siihen, miksi vahva bisimulointi ei onnistu π-kalkyylin tapauksessa, löytyy määritelmän kohdasta kaksi, joka määrittelee, että input-etuliitteen jälkeen prosessien P ja Q tulisi pystyä simuloimaan toisiaan kaikilla mahdollisilla w:n arvoilla. Tämä määritelmä johtaa kuitenkin hankaluuksiin, sillä input-etuliite, ja yleensäkin rajoittamaton vapaiden nimien korvaaminen, ei säilytä vahvaa bisimilaarisuutta [MPW89]. Esimerkiksi prosessit

$$P \equiv a(x).[x = b]\bar{b}b \quad Q \equiv a(x).0$$

näyttäisivät selvästikin käyttäytyvän samoin. Molemmat prosessit käyttäytyvät kuten prosessi **0** siirtymän $a(x)$ jälkeen, koska $[x = b]$ ei ole tosi. Jos nämä prosessit yhdistetään rinnakkain prosessin $\bar{a}b$ kanssa, ne käyttäytyvätkin toisin. Tällöin $a(x).[x = b]\bar{b}b \mid \bar{a}b \xrightarrow{\tau} [b = b]\bar{b}b$, kun taas $a(x) \mid \bar{a}b \xrightarrow{\tau} 0$ [San93]. Tästä esimerkistä käy ilmi, että perinteinen bisimulointi ei yleisessä tapauksessa päde π-kalkyylissä, vaan tarvitaan relaatio, joka huomioi π-kalkyylin muista prosessialgebroista poikkeavan nimien käsittelyn.

Edellä huomattiin, että perinteinen bisimulaation määritelmä ei toimi, koska vapaiden nimien korvaus voi jossain tapauksissa muuttaa prosessien semantiikan täysin. Perinteinen vahva bisimulaatio toimii vain tapauksissa, joissa nimet tulkitaan vakioiksi [MPW89]. Nimien tulkitseminen vakioiksi ei kuitenkaan sovi π-kalkyylin tapauksessa, jossa nimenomaan halutaan että nimet tulkitaan muuttujiksi ja paikanpitimiksi instantioituville nimille. Tämän vuoksi π-kalkyylin bisimulointirelaatiolta vaaditaan, että prosessien yhteneväisyys säilyy kaikilla mahdollisilla nimien korvauksilla. Tämä on ilmaistu määritelmässä 5.9 [MPW89].

Määritelmä 5.9 (π-prosessien ekvivalenssi)
Prosessit P ja Q ovat (vahvasti) ekvivalentteja, $P \sim Q$, jos $P\sigma \dot{\sim} Q\sigma$ on voimassa kaikilla vapaiden nimien korvauksilla σ.

Täysin vapaa nimien korvaus ei tietenkään toimi, mutta toisaalta ilman nimien korvausta ei tulla toimeen. Tämän vuoksi tarvitaan jokin kompromissi näiden kahden nimien käytön ääripään väliltä, jonka avulla voidaan määritellä toimiva bisimulointi π-kalkyylin käyttöön. Tätä tarkoitusta varten täytyy ottaa käyttöön nimien korvauksia rajoittava sääntö (*distinction*) [MPW89]:

Määritelmä 5.10 (Nimien korvauksien rajoitus)
*Nimien korvauksien rajoitus (*distinction*) on π-kalkyylin nimien joukon $\mathcal{N}$ irrefleksiivinen ja symmetrinen binäärinen relaatio D, joka ilmaisee pysyvän nimien erisuuruuden. Nimien korvaus σ on rajoituksen D mukainen, jos $(x, y) \in D$, ja $\forall\sigma : x\sigma \neq y\sigma$.*

Nimien korvauksien rajoitus $(a, b) \in D$ siis ilmaisee, että mikään nimien korvaus ei saa johtaa tilanteeseen, jossa $a = b$. Nimien korvauksien rajoitusta käyttäen saadaan määritelmän 5.11 π-prosesseille käyttökelpoinen ekvivalenssirelaatio $\sim_D$ [MPW89].

Määritelmä 5.11 (Prosessien vahva D-ekvivalenssi)
Prosessit P ja Q ovat vahvasti $D - ekvivalentit$, merkitään $P \sim_D Q$, jos $P\sigma \dot{\sim} Q\sigma$ kaikille σ, jotka ovat rajoituksen D mukaisia.

Tällöin saadaan D-ekvivalenssia käyttäen, että $a(x).[x = b]\ \bar{b}b \sim_D a(x).\mathbf{0}$, missä $D = \{(x, b)\}$, mikä ei vahvan bisimulaation tapauksessa ollut voimassa.

5.4.2 Eräs π-kalkyylin kongruenssi relaatio

Kuten luvussa 5.4.1 huomattiin, nimien instantiointi tekee π-prosessien vertailun hankalaksi. Ensinnäkin π-kalkyylin prosessien välisen vahvan bisimuloinnin $\dot{\sim}$ laskennassa tulisi ottaa huomioon *kaikki* mahdolliset arvot prosessin vapaille nimille α. Toiseksi, juuri vaatimus nimien vapaalle korvattavuudelle aiheuttaa sen, että relaatio $\dot{\sim}$ ei ole kongruenssi, koska se ei säily input-toiminnon, $a(x)$, suhteen. Kuitenkin π-kalkyylissä tarvitaan jokin, mieluummin algoritmisesti tehokas, tapa, jolla prosesseja voidaan vertailla.

Avoin bisimulointi (*open bisimulation*) on π-kalkyyliä varten kehitetty bisimulointirelaation versio. Toisin kuin tähän asti esitellyt, "perinteiset" bisimulointirelaatiot, on vahva avoin bisimulointi myös kongruenssi kaikkien π-kalkyylin operaattoreiden suhteen [San93].

Avoimessa bisimulointirelaatiossa nimien instantioinnin ongelmasta päästään eroon sillä, että instantiointi viedään relaation sisälle. Tämän seurauksena avoimen bisimuloinnin määritelmässä ei tarvitse erikseen käsitellä eri siirtymiä, $a, x(y), \bar{x}y$, vaan määritelmä voidaan antaa yleisessä muodossa koskien kaikkia siirtymiä α.

Määritelmä 5.12 ((Vahva) avoin bisimulointi)
(Vahva) avoin bisimulointi on relaatio $\mathcal{R} \in S \times S$, siten että kaikille σ on voimassa

1. *jos $(P, Q) \in \mathcal{R}$ ja $P\sigma \xrightarrow{\alpha} P'$, niin $\exists Q'$ siten että $Q\sigma \xrightarrow{\alpha} Q'$ ja $(P', Q') \in \mathcal{R}$*
2. *jos $(P, Q) \in \mathcal{R}$ ja $Q\sigma \xrightarrow{\alpha} Q'$, niin $\exists P'$ siten että $P\sigma \xrightarrow{\alpha} Q'$ ja $(P', Q') \in \mathcal{R}$*

Prosessit P ja Q ovat avoimesti bisimilaariset merkitään $P \sim_o Q$, jos $(P, Q) \in \mathcal{R}$ jollakin avoimella bisimulaatiolla $\mathcal{R}$.

Avoin bisimulointi on prosessien ekvivalenssirelaatio, joka on suljettu nimien korvausten suhteen, toisin sanoen jos $P \sim_o Q$, niin $P\sigma \sim_o Q\sigma$ kaikilla σ [San93].

Propositio 5.1 *$\sim_o$ on suurin avoin bisimulaatio ja se on suljettu kaikkien nimien korvausten σ suhteen.*

Propositiosta 5.1 seuraa, että relaatio $\sim_o$ säilyy kaikkien π-kalkyylin operaattoreiden, erityisesti input-toiminnon $a(x)$, suhteen. Tästä seuraa, että avoin bisimulointi $\sim_o$ on kongruenssi π-kalkyylin relaatio [San93]. Määritelmän 5.12 mukaisen vahvan avoimen bisimuloinnin lisäksi voidaan muiden bisimulointirelaatioden tapaan määritellä myös heikko avoin bisimulointi. Avoimen bisimuloinnin heikko versio ei ole kongruenssi, koska valinta-operaattori tuottaa ongelmia. Jos valinta määritellään ehdolliseksi (*guarded*), niin myös heikko avoin bisimulointi on kongruenssi [San93].

Avointa bisimulointia käyttäen perinteiselle bisimulointirelaatiolle (määritelmän 5.4 mukainen relaatio $\sim$) ongelmalliset prosessiparit $P \equiv a(x).[x = b]\bar{b}b$ ja $Q \equiv a(x).0$ voidaan erottaa toisistaan. Näille prosesseille pätee $P \sim Q$, mutta $P \not\sim_o Q$, koska $Q \stackrel{\bar{b}b}{\not\rightarrow}$, kun $\sigma = \{(x/b)\}$ eli avoimen bisimuloinnin ehto 1 ei täyty. Avoimen bisimulaatioekvivalenssin laskemiseen on kehitetty Paigen ja Tarjanin ositusalgoritmia käyttävä algoritmi, jonka aikavaativuus on luokkaa $\mathcal{O}(mn \log n)$, missä m on osioitavan verkon siirtymien ja n tilojen määrä [PS96].

Prosessien bisimilaarisuuden tarkistamisessa on nimien instantioinnin lisäksi vielä yksi ongelma. Koska π-kalkyylin semantiikka vaatii universaalia nimien korvausta σ ja nimiä oletetaan olevan ääretön määrä, voisi luulla että prosessien ekvivalenssin tarkistaminen on käytännössä hyvin raskasta ellei jopa mahdotonta. Tästä ongelmasta päästään eroon käyttämällä niin kutsuttuja symbolisia menetelmiä, joissa ääretöntä nimiavaruutta käsitellään symbolisten siirtymäjärjestelmien avulla [HL95].

Symboliset siirtymäjärjestelmät ovat samantapaisia kuin tavalliset tilasiirtymäjärjestelmät, mutta symbolisissa siirtymäjärjestelmissä jokainen siirtymä on muotoa (b, α), missä b on jokin totuusarvoinen lauseke. Esimerkiksi perinteisen järjestelmän tilasiirtymä $p \stackrel{\alpha}{\rightarrow} p'$ voidaan ilmaista symbolisessa siirtymäjärjestelmässä siirtymänä $p \stackrel{true,\alpha}{\rightarrow} p'$ [HL95]. Avoimen bisimuloinnin laskennassa siirtymät ovat muotoa $P \stackrel{M,\alpha}{\rightarrow} P'$, missä joukko M määrittelee siirtymän α välttämättömät ehdot [San93]. Avoimen bisimuloinnin laskemisessa ilmenevä symbolinen siirtymä voisi olla esimerkisi $[a = b]\alpha.P \stackrel{[a=b],\alpha}{\rightarrow} P'$, jossa $M = \{[a = b]\}$.

Edellisessä esimerkissä esiintynyttä joukkoa M kutsutaan sovitussekvenssiksi (*matching sequence*) ja se sisältää nolla tai useampia nimien yhtäsuuruuksia (*guard*) muotoa $[\alpha = \beta]$ [San93]. Yleensä sovitusjoukkoja merkitään kirjaimilla $M, N, ...$ Merkinnällä σ_M tarkoitetaan erityistä nimien korvaustoimintoja, jossa kaikki nimet korvataan sovitussekvenssin ekvivalenssijoukkojen edustajilla. Jos esimerkiksi $M = \{[a = b][b = c][d = e]\}$, niin eräs vaihtoehtoinen σ_M on nimien korvaus $\sigma_M = \{(b \rightarrow a), (c \rightarrow a), (d \rightarrow a), (e \rightarrow d)\}$, joka korvaa nimet b, c, d nimellä a ja nimen e nimellä d. Jos sekvenssin M testien $[\alpha = \beta]$ totuudesta seuraa, että myös kaikkien joukon N testit onnistuvat, niin merkitään $M \rhd N$.

Avoimelle bisimuloinnille $\sim_o$ voidaan nyt määritellä symbolinen vastine, relaatio $\asymp$, joka määrittelee symbolisen bisimulointirelaation prosessien välille [San93]. Määritelmässä 5.13 annettu $\asymp$-bisimulointi voidaan todistaa olevan ekvivalentti avoimen bisimuloinnin kanssa. Toisin kuin avoimessa bisimuloinnissa, ei $\asymp$-bisimuloinnin toteutuksessa tarvitse nimien instantioinnissa käsitellä universaalisti kvantifioituja nimiä vaan nimiä tarvitsee instantioida vain sen verran kuin on välttämätöntä ehtojen täyttämiseksi [San93].

Määritelmä 5.13 ($\asymp$-simulointi) *$\asymp$-simulointi on relaatio $\mathcal{R} \in S \times S$ siten että*

- *jos $(P, Q) \in \mathcal{R}$ ja $P \stackrel{M,\alpha}{\rightarrow} P'$, niin $\exists N, \beta, Q'$ siten että $Q \stackrel{N,\beta}{\rightarrow} Q'$ ja*
  - *$M \rhd N$,*

– $\alpha\sigma_M \equiv \beta\sigma_M$,
– $P'\sigma_M \mathcal{R} Q'\sigma_M$

$\mathcal{R}$ on $\asymp$-bisimulointi jos sekä $\mathcal{R}$ ja $\mathcal{R}^{-1}$ ovat $\asymp$-simulointeja. Kaksi prosessia P ja Q ovat $\asymp$-bisimilaarisia, merkitään $P \asymp Q$, jos $(P, Q) \in \mathcal{R}$ jollekin $\asymp$-bisimuloinnille $\mathcal{R}$.

Määritelmän 5.13 tulkinta on se, että kaksi prosessia ovat samankaltaiset, jos niissä voidaan suorittaa samanlainen siirtymä samoja nimiä koskevia rajoitteita käyttäen. Vahvan avoimen bisimuloinnin laskennassa $\asymp$-bisimulointia käyttäen yhdistetään sekä D-ekvivalenssin käyttö (katso määritelmä 5.11) että symbolisen bisimuloinnin menetelmät sopivaan osointialgoritmiin soveltaen [PS96, FJ01]. Tämän tarkempaa kuvausta kyseisen algoritmin toimintaperiaatteista ei kuitenkaan tämän opinnäytetyön laajuudessa pystytä antamaan. Tärkeintä on huomata, että myös π-kalkyylille on olemassa tehokkaita algoritmeja, joiden avulla prosessien välinen ekvivalenssi voidaan ratkaista.

5.5 Rajapintaprotokollien yhteentoimivuuden tarkistaminen

Tässä aliluvussa tarkastellaan kuinka edellisissä aliluvuissa esiteltyjä formaaleja menetelmiä soveltaen voidaan todistaa kahden komponentin välinen yhteentoimivuus, kun rajapintaprotokollat kuvataan π-kalkyylillä. Ohjelmistokomponenttien yhteentoimivuuden tarkistuksessa tarvitaan käytettäviltä menetelmiltä vaatimuksia, joita ei välttämättä tarvitse huomioida perinteisissä verifiointiongelmissa. Nämä vaatimukset liittyvät yhteensopivuuden formalisointiin sekä yhteensopivuuden säilymiseen korvautuvuusrelaation yhteydessä.

Rajapintaprotokollien yhteensopivuus tarkistetaan käyttämällä siihen soveltuvaa relaatiota, joka formalisoi komponenttien yhteentoimivuuden. Yhteensopivuus takaa, että komponenttien yhteistoiminta ei päädy lukkiumaan, vaan molempien komponenttien suoritus päättyy hyväksyttävään lopputilaan. Yhteensopivuusrelaatio määrittelee yhteensopivuuden sellaiseksi komponenttien yhteiskäytökseksi, jossa duaalisten operaatioiden (esimerkiksi a:n lähetys ja vastaanotto muodostavat duaaliset operaatiot) suorittaminen johtaa hyväksyttyyn lopputilaan.

Rajapintaprotokollien korvautuvuuden tarkastamista varten kehitellään prosessien bisimulointirelaation kaltainen ekvivalenssi, joka säilyttää protokollien välisen yhteentoimivuuden. Yhteentoimivuuden tulee säilyä, kun komponenteista kehitetään uusia versioita joko toimintoja laajentamalla tai niitä erikoistamalla.

Komponenttien yhteentoimivuustarkistuksien oletetaan olevan binäärisiä kahden komponentin välisiä rajapintaprotokollien tarkistuksia eli komponenttien oletetaan olevan niin sanotusti yksiroolisia. Komponentin yhteentoimivuus useamman komponentin kanssa yhtäaikaa, moniroolisuus, tuo mukanaan ongelmia, joita ei tämän opinnäytetyön rajoissa voida käsitellä. Näistä ongelmista voidaan mainita esimerkiksi toimintojen välinen synkronointi eri komponenttien kesken, kuvausten uudelleenkäyttö (korvattavuus) eri ympäristöissä ja yhteentoimivuuden laskennallisen vaativuuden hallitseminen. Nämä asiat tulevat vastaan ohjelmistoarkkitehtuureita verifioitaessa, jonka osana käytetään ohjelmistokomponenttien yhteentoimivuustarkistuksia.

5.5.1 Yhteensopivuuden formalisointi

Rajapintaprotokollien yhteensopivuus tarkoittaa sitä, että kaksi komponenttia voivat turvallisesti kommunikoida keskenään käyttäen määriteltyjä protokollia [AG97]. Turvallinen kommunikointi

tarkoittaa, että järjestelmään ei muodostu lukkiutumia (*deadlock*) tai elävälukkiutumia (*livelock*) [2]. Turvallisen kommunikoinnin lisäksi vaaditaan, että komponenttien vuorovaikutus päättyy hyvin määriteltyyn lopputilaan [CPT01]. Nämä vaatimukset rajapintaprotokollien välillä tarkistetaan käyttämällä erityista yhteensopivuusrelaatiota.

Tärkein yhteensopivuuden tae on lukkiutumattomuus, joka takaa, että komponentin on mahdollista lopettaa suorituksensa hyvin määriteltyyn tilaan. Lukkiutunut komponentti käyttäytyy ulospäin kuin toimintansa lopettanut komponentti $\mathbf{0}$, mutta toisin kuin $\mathbf{0}$-prosessi, voi lukkiutunut komponentti toimia oikein toisenlaisessa ympäristössä.

Edellä kuvaillusta syystä on tärkeää erottaa toisistaan onnistuneesti päättyneet prosessit niistä, jotka ovat lukkiutuneet. Yhteensopivuuden määrittelyä varten otetaan käyttöön tulkinta onnistuneelle prosessin suoritukselle, jonka mukaan prosessin suoritus on onnistunut, jos ja vain jos se on päättynyt tilaan, joka on strukturaalisesti ekvivalentti $\mathbf{0}$-prosessin kanssa. Prosessi on epäonnistunut, jos sen suoritus päättyy johonkin muuhun tilaan kuin $\mathbf{0}$, josta se ei voi edetä τ-siirtymin [CPT01].

Määritelmä 5.14 (Agentin onnistunut ja epäonnistunut suoritus) *Agentin P suoritus on epäonnistunut (päättynyt virhetilaan), jos on olemassa P' siten että $P \Rightarrow P'$, $P' \stackrel{\tau}{\not\rightarrow}$ ja $P \not\equiv \mathbf{0}$.*
Jos agentti ei ole epäonnistunut, niin se on onnistunut.

Agenttien välisen yhteensopivuuden formalisointia varten tarvitaan muutamia apukäsitteitä, joista ensimmäisenä esitellään syötteen välittävä agentti. Syötteen välittävä agentti on yksinkertaisesti prosessi, joka lähettää syötteen toiselle agentille. Syötteen välittävä agentti toimii agenttien välisen synkronisoinnin aktiivisena osapuolena.

Määritelmä 5.15 (Syötteen välittävä agentti) *Agentti P välittää syötteen agentille Q jos $\exists P', Q'$ siten että*

1. $P \stackrel{\bar{x}y}{\Rightarrow} P'$ *ja* $Q \stackrel{x(y)}{\Rightarrow} Q'$ *tai*
2. $P \stackrel{\bar{x}(z)}{\Rightarrow} P'$ *ja* $Q \stackrel{x(z)}{\Rightarrow} Q'$

Kaksi agenttia P ja Q ovat synkronisoituvia, jos niiden välillä on jokin kommunikointitapahtuma.

Määritelmä 5.16 (Synkronisoituvat agentit) *Kaksi agenttia P ja Q ovat synkronisoituvia jos P välittää syötteen Q:lle tai Q välittää syötteen P:lle*

Edellä esiteltyjen käsitteiden avulla voidaan määritellä agenttien välinen relaatio, joka takaa agenttien yhteensopivuuden [CPT01]. Yhteensopivuuden takaamiseksi agenttien yhteistoiminnalta vaaditaan kolme ominaisuutta, jotta niiden välinen vuorovaikutus ei johtaisi virhetilanteisiin [CPT01]. Ensimmäinen vaatimus on, että molemmat voivat suorittaa ainakin yhden siirtymän. Tämä on varsin selkeä olettamus, sillä jos kumpikaan ei voi suorittaa yhtään siirtymää, niin agentit ovat lukkiutuneita (tai täysin hyödyttömiä $\mathbf{0}$-prosesseja). Toiseksi, mikään agentin tekemä paikallinen valinta ($\tau.a + \tau.b$ on paikallinen valinta, kun taas $a + b$ on globaali valinta siirtymien a ja b välillä) ei saa johtaa epäyhteensopiviin prosesseihin. Tämä tarkoittaa sitä, että jos komponentti A tekee paikallisen valinnan, johon B ei voi vaikuttaa, niin B:llä täytyy olla samat etenemismahdollisuudet sekä ennen, että jälkeen A:n valintaa. Kolmas vaatimus on se, että agenttien välinen kommunikointi ei saa johtaa epäyhteensopiviin prosesseihin.

[2]Suomennos perustuu Roope Kaivolan opinnäytetyöhön “Aikalogiikkajärjestelmien muodostaminen ja hyödynnettävyys”, 1990

Määritelmä 5.17 (Agenttien yhteensopivuusrelaatio) *Binäärinen relaatio $\mathcal{E}$ on agenttien P ja Q välinen semi-yhteensopivuus, merkitään $P\mathcal{E}Q$, jos*

1. *jos P ei ole onnistunut, niin P ja Q ovat synkronisoituvia*
2. *jos $P \xrightarrow{\tau} P'$, niin $P'\mathcal{E}Q$*
3. *jos $P \xrightarrow{x(w)} P'$ ja $Q \xrightarrow{\bar{x}y} Q'$, niin $P'\{y/w\}\mathcal{E}Q'$*
4. *jos $P \xrightarrow{x(w)} P'$ ja $Q \xrightarrow{\bar{x}(w)} Q'$, niin $P'\mathcal{E}Q'$*

Relaatio $\mathcal{E}$ on yhteensopivuus jos sekä $\mathcal{E}$ ja $\mathcal{E}^{-1}$ ovat semi-yhteensopivuuksia. Agenttien yhteensopivuus $\diamond$ on suurin yhteensopivuus.

Yhteensopivuusrelaation ensimmäinen kohta takaa, että yhteensopivien agenttien tulee synkronisoitua ainakin kerran. Toinen kohta määrittelee sen, että jos P tekee sisäisen siirtymän (esimerkiksi paikallisen valinnan), niin sen jälkeen P:n ja Q:n tulee olla yhteentoimivat. Kolmas ja neljäs kohta määrittelevät, että yhteensopivien prosessien välinen kommunikointi pitää johtaa yhteensopiviin prosesseihin.

Määritellään lopuksi agenttien välinen yhteensopivuus, joka π-kalkyylin bisimulaatiorelaation tapaan tulee olla suljettu nimien korvauksien suhteen.

Määritelmä 5.18 (Yhteensopivat agentit) *Agentit P ja Q ovat yhteensopivia, merkitään $P\Diamond Q$, jos $P\sigma\diamond Q\sigma$ kaikilla nimien korvauksilla σ, missä σ on nimien korvauksien rajoituksen $D : fn(P) \times fn(Q)$ mukainen.*

Yhteensopivuuden määrittelyssä kaikki muut nimien korvaukset ovat sallittuja paitsi sellaiset, jotka identifioivat prosessien P ja Q vapaita nimiä toisiinsa. Tämä oletus on tarpeellinen, koska vapaat nimet edustavat agenteista ulospäinnäkyviä nimiä, kuten ohjelmistotuotannossa mallinnettavia kommuninointikanavia. Jos esimerkiksi kaksi eri kommunikointikanavaa identifioitaisiin samoiksi, voi se johtaa komponenttien yhteentoimimattomaan käyttäytymiseen.

Seuraava esimerkki havainnollistaa, miksi nimien korvauksien rajoittaminen on välttämätöntä ohjelmistokomponenttien yhteentoimivuuden tarkistuksessa. Prosesseille $P \stackrel{def}{\equiv} \bar{a}(x).\bar{x}.\mathbf{0}$ ja $Q = a(y).y.\mathbf{0} + b(z).\mathbf{0}$ on voimassa $P\dot{\diamond}Q$ melkein kaikilla nimien korvauksilla σ. Jos $\sigma = \{a/b\}$, niin $P\{a/b\} \xrightarrow{\bar{a}(x)} \bar{x}.\mathbf{0}$, $Q\{a/b\} \xrightarrow{a(x)} \mathbf{0}$ ja $\bar{x}.\mathbf{0} \not\dot{\diamond} \mathbf{0}$ [CPT99]. Prosessin P vapaan nimen a, joka edustaa prosessien välistä kommunikointikanavaa, identifioiminen prosessin Q vapaaseen nimeen b muuttaa kokonaisjärjestelmän toimintaa siten, että prosessit eivät ole enää yhteensopivia.

Tärkeä huomio määritelmän 5.18 mukaisesta yhteensopivuusrelaatiosta $\dot{\diamond}$ on se, että se määrittelee yhteensopivuuden nimenomaan duaalisia, synkronoituvia siirtymäpareja käyttäen. Esimerkiksi kaksi agenttia $P = a(x).\mathbf{0} + b(y).\mathbf{0}$ ja $Q = \bar{a}u.\mathbf{0} + \bar{b}v.0 + \bar{c}Error$ ovat yhteensopivia, sillä $P \diamond Q$ [CPT01]. Yhteensopivuus ei siis tarkoita sitä, että komponenttien tulisi tukea toistensa kaikkia käyttäytymismalleja, vaan yhteensopivuus tarkistetaan vain niiden siirtymien mukaan, jotka ovat duaalisia, synkronisoituvia siirtymäpareja.

Toinen tärkeä yhteensopivuusrelaation ominaisuus koskee rajapintaprotokollien sisäisiä τ-siirtymiä. Yhteensopivuusrelaation mukaan agentit ovat yhteensopivia ainoastaan, jos kaikki sisäiset siirtymät johtavat yhteensopiviin agentteihin. Edellistä esimerkkiä jatkaen, jos prosessi Q korvataan

prosessilla $Q' = \bar{a}u.\mathbf{0} + \bar{b}v.0 + \tau.\bar{c}Error$, niin se ei ole enää yhteensopiva prosessin P kanssa, $P \diamond Q'$ [CPT01]. Tämä johtuu siitä, että prosessin Q' siirtymä $\tau.\bar{c}Error \xrightarrow{\tau} \bar{c}Error$ johtaa prosessiin $\bar{c}Error$, joka ei selvästikään ole yhteensopiva prosessin P kanssa.

Yhteensopivuuden tarkistukseen voi liittyä myös muiden kuin tässä luvussa esiteltyjen ominaisuuksien verifiointia. Muut yhteistoiminnan ominaisuudet verifioidaan käyttäen mallintarkistusta rajapintaprotokollien yhteistoimintaa kuvaavaan prosessiin $P \mid Q$, missä P ja Q ovat rajapintaprotokollia. Mallintarkistuksen perusteisiin ja ominaisuuksien mallintamiseen temporaalilogiikalla tutustuttiin kappaleessa 5.3.

5.5.2 Korvautuvuuden vaatimukset

Rajapintaprotokollien korvautuvuus tarkoittaa, että kahden komponentin ulkoinen käyttäytyminen ovat tarvittavalla tarkkuustasolla yhteneväistä. Edellisissä luvuissa ollaan kuvailtu erilaisia bisimulaatiorelaatioita, jotka identifioivat ulkoiselta käyttäytymiseltään samankaltaisia prosesseja. Bisimulaatiorelaation kaltainen prosessiekvivalenssi on hyvä lähtökohta rajapintaprotokollien välisen korvautuvuuden tarkistamiseen.

Perinteinen prosessien (heikko tai vahva) bisimulointi ei kuitenkaan suoraan sovellu suoraan ohjelmistokomponenttien korvautuvuusrelaatioksi. Jos esimerkiksi on määritelty prosessit $P \stackrel{def}{\equiv} \mathbf{0}$, $Q \stackrel{def}{\equiv} \tau.\mathbf{0}$ ja $P' \stackrel{def}{\equiv} (a)a(x).\mathbf{0}$, niin $P \sim P'$, $P \dot{\diamond} Q$, mutta $P' \not\dot{\diamond} Q$, koska P' ei ole onnistunut mutta ei koskaan pysty synkronoitumaan minkään prosessin kanssa [CPT01]. Esimerkistä huomataan, että perinteinen bisimulointi ei ole sopiva relaatio rajapintaprotokollien keskinäiseen vertailemiseen, koska korvautuvuusrelaation tulisi säilyttää rajapintaprotokollien yhteensopivuus. Tämän vuoksi bisimulaation tilalle tarvitaan paremmin komponenttipohjaisen ohjelmistotuotannon tarpeisiin soveltuva korvautuvuusrelaatio.

Protokollien korvautuvuus tarkistetaan korvautuvuusrelaatiota $\mathcal{R}$ käyttäen. Korvautuvuusrelaation tulee olla sellainen, että jos $P \; \mathcal{R} \; Q$, niin rajapinnat P ja Q käyttäytyvät (minkä tahansa) ympäristön näkökulmasta samalla tavalla.

Tämä vaatimus tunnetaan yleisesti Liskovin korvautuvuusperiaatteena, joka koskee niin kutsuttua käyttäytymisalityypitystä [LW94]. Käyttäytymisalityypitys tulkitaan "*is-a*" tyyliseksi alityypitykseksi, joka ilmaisee että käyttäytymisalityyppiä voidaan aina käyttää ylityypin sijaan. Käyttäytymisen alityypitys ei siis saa poistaa ylityypin käyttäytymismalleja eli alityypitys on ensisijaisesti käyttäytymismalleja lisäävä operaatio [HK02].

Korvautuvuuden tulee olla käsite, joka sallii komponentin toiminnallisuuden laajentamisen siten, että yhteentoimivuus säilyy laajentamisen yhteydessä. Tämä tarkoittaa sitä, että jos $P \diamond Q$ ja P' on prosessin P toiminnallinen laajennus, niin $P' \diamond Q$ on voimassa. Koska vahva bisimilaarisuus tarkoittaa käytännössä prosessien ekvivalenssia, eikä siten salli prosessin laajentamista, on vahva bisimilaarisuus liian vahva ehto komponenttien korvautuvuusrelaatioksi. Tämän vuoksi mielenkiinto kääntyy relaatioihin, jotka heikon bisimilaarisuuden tapaan voivat abstrahoida tiettyjä, yhteensopivuustarkistuksen kannalta tarpeettomia ominaisuuksia.

Rajapintakomponenttien korvautuvuusrelaation tulisi tukea komponenttipohjaisen ohjelmistotuotannon erikoistamisen ja laajentamisen käsitteitä, jotta järjestelmiä voitaisiin päivittää turvallisesti. Korvautuvuusrelaation tulee olla sellainen, että käyttäytyminen on tiettyyn rajaan asti periytyvää. Tämä tarkoittaa sitä, että jos komponenttia P laajennetaan enemmän toiminnallisuutta sisältäksi versioksi P', niin komponenttia P' voidaan käyttää täysin samalla tavalla kuin komponenttia P.

Koska komponenttien yhteentoimivuustarkistuksia tullaan käyttämään ohjelmistotuotannossa ja ehkä jopa komponenttipohjaisten väliohjelmistojen osana, tulee komponenttien korvautuvuuden tarkistamisen olla algoritmisesti tehokasta. Käytännössä tämä tarkoittaa sitä, että käytettävä relaatio komponenttien välillä voidaan laskea käyttäen soveltuvaa verkon ositusalgoritmia tai vastaavaa metodia käyttäen [PT87, Fer90].

Yllä käydyn alustuksen pohjalta voidaan yhteensopivuustarkistuksessa käytettävälle korvautuvuusrelaatiolle $\mathcal{R}$ antaa seuraavat vaatimukset:

1. $\mathcal{R}$ on rajapintaprotokollien kuvaamisen käytettävän prosessialgebran kongruenssi relaatio.

2. $\mathcal{R}$ abstrahoi korvautuvuuden kannalta turhat toteutusyksityiskohdat ja irrelevantit vaihtoehdot.

3. $\mathcal{R}$ säilyttää prosessien ominaisuudet haarautuvan ajan suhteen.

4. relaation $P\ \mathcal{R}\ R$ laskeminen voidaan toteuttaa tehokkaalla tavalla, esimerkiksi verkon ositusalgoritmiä soveltamalla.

5. $\mathcal{R}$ säilyttää prosessien välisen yhteensopivuuden.

Ensimmäinen vaatimus on hyvin luonnollinen, sillä algebrallisen relaation kongruenssi on juuri se ominaisuus, mitä tarvitaan komponenttien turvallisessa keskinäisessä korvauksessa. Kuten määritelmässä 5.7 ilmaistaan, tarkoittaa relaation kongruenssi käytännössä sitä, että jos prosessien välisellä kongruenssilla relaatiolla $\mathcal{R}$ pätee $P\ \mathcal{R}\ Q$, niin P ja Q ovat käyttäytymiseltään erottamattomat missä ympäristössä tahansa. Rajapintaprotokollien kuvaamiseen käytettävä prosessialgebran ominaisuudet vaikuttavat siihen, minkälainen kongruenssista prosessirelaatiosta muodostuu. Kun kuvaamiseen käytetään π-kalkyyliä, niin korvautuvuusrelaation lähtökohtana tulee käyttää jotakin π-kalkyylin kongruenssia prosessiekvivalenssia. Eräs π-kalkyylien kongruenssi ekvivalenssirelaatio on luvussa 5.4.2 kuvattu avoin bisimulointi.

Toinen vaatimus koskee prosessien välistä abstrahointi. Abstrahointi tarkoittaa sitä, että korvautuvuusrelaatio ei huomioi prosessien sisäisiä siirtymiä tai muuten tarkistuksen kannalta tarpeettomia siirtymiä. Korvautuvuusrelaation pitää siis olla heikon bisimilarisuuden (määritelmä 5.6) tapainen relaatio, joka identifioi kaksi prosessia samankaltaiseksi, vaikka niissä suoritettavat suoritusjäljet eivät olisikaan täysin identtisiä.

Korvautuvuusrelaation tulee sallia toimintojen turvallinen laajentaminen. Tämä tarkoittaa sitä, että esimerkiksi prosessit $P = \bar{a}.\mathbf{0} + b.\mathbf{0}$ ja $P' = \bar{a}.\mathbf{0} + b.\mathbf{0} + \bar{c}.\mathbf{0}$ tulisi identifioida keskenään, kun haetaan rajapintaprotokollalle P korvaavia rajapintaprotokollia. Korvautuvuusrelaatio voi siis olla luonteeltaan epäsymmetrinen relaatio varsinaisten rajapintaprotokollien suhteen. Tutkittavaksi jää, miten bisimulaation kaltaisesta prosessiekvivalenssista saadaan kehitettyä ohjelmistotuotannon tarpeisiin paremmin soveltuva korvautuvuusrelaatio.

Kolmas kohta vaatimuksista esittää, että korvautuvuusrelaation tulisi säilyttää kaikki ominaisuudet haarautuvan ajan suhteen. Tällä tarkoitetaan erityisesti sitä, että sisäisten siirtymien luoma epädeterminismi tulisi huomioida ja prosessin haarautumiskohtien ajanhetket pitää säilyttää yhteensopivuuden säilyttämiseksi. Ajan haarautumisominaisuuksien säilyminen takaa sen, että kun esimerkiksi palvelinkomponentti korvataan uudella komponentilla, on asiakkailla korvauksen jälkeen käytettävissä täsmälleen samat käyttäytymismallit. Keskenään korvautuvissa komponenteissa tulisi erityisesti suorituksen haarautuminen valintatilanteissa suorittaa täsmälleen samoissa kohdissa suorituspuuta.

Perinteinen heikko bisimulointi, niin kuin sen kuvattu määritelmässä 5.6, ei sovellu suoraan ohjelmistotuotannon tarpeisiin. Syynä tähän on se, että heikko bisimulointi ei säilytä kaikkia prosessien haarautumisominaisuuksia ja siten voi virheellisesti identifioida kaksi prosessia, jotka eivät välttämättä ole korvattavissa keskenään.

Esimerkiksi prosessit $a(\tau b + c)$ ja $a(\tau b + c) + ab$ ovat havaintoekvivalentit eli heikosti bisimilaariset. Ensimmäisessä prosessissa valinta toimintojen b ja c välillä tehdään aina toiminnon a jälkeen, kun taas jälkimmäisessä suoritus ab voidaan valita jo prosessin alussa [vW96]. Jotta tällaiset prosessit, jotka tekevät ulkoiseen käyttäytymiseen vaikuttavat päätökset eri kohdassa suorituspuuta voitaisiin erottaa toisistaan, täytyy ottaa käyttöön niin kutsuttu haarautuva bisimulointirelaatio (*branching bisimulation*) [vW96].

Määritelmä 5.19 (Haarautuva bisimulaatio) *Haarautuva bisimulaatio on prosessien tilojen joukon S relaatio $\mathcal{R} \subseteq S \times S$ siten että*

- *jos $(s,t) \in \mathcal{R}$ ja $s \xrightarrow{\alpha} s'$, niin joko $\alpha = \tau$ ja $(s',t) \in \mathcal{R}$ tai on olemassa suorituspolku $t \Rightarrow t_1 \xrightarrow{\alpha} t_2 \Rightarrow t'$ siten, että $(s,t_1) \in \mathcal{R}$, $(s',t_2) \in \mathcal{R}$ ja $(s',t') \in \mathcal{R}$.*

Prosessien haarautuva bisimilaarisuus takaa, että jos $(s,t) \in \mathcal{R}$ ja $(s',t') \in \mathcal{R}$, niin suorituspolulle $s \xrightarrow{\tau} s_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} s_m \xrightarrow{\tau} s'$ on voimassa $\forall i \in \{1,\ldots,m\}, (m \geq 0) : (r_i, s) \in \mathcal{R}$ [vW96]. Haarautuva bisimulointi on prosessien välinen relaatio, joka on heikompi kuin vahva bisimulointi, koska se abstrahoi τ-siirtymät, mutta on vahvempi kuin heikko bisimulointi, joka ei kaikissa tapauksissa erota prosessien (epädeterminististä) haarautumista toisistaan.

Neljäs relaatiolta vaadittava ominaisuus on luonnollinen ja triviaali olettamus, joka ei kuitenkaan ole itsestäänselvyys. Jotta korvautuvuus $\mathcal{R}$ voidaan laskea ositusalgoritmia käyttäen, tulee käytettävän relaation $\mathcal{R}$ täyttää tietyt vaatimukset. Paigen ja Tarjanin osiointialgoritmiä voidaan soveltaa korvautuvuusrelaation laskemiseen ainoastaan, jos laskettava relaatio on ekvivalenssirelaatio. Osiointialgoritmia käyttäen kahden siirtymäjärjestelmän välinen ekvivalenssi voidaan tarkistaa ajassa $\mathcal{O}(mlogn)$, missä m on siirtymien ja n tilojen lukumäärä [Fer90].

Viides ja kenties tärkein vaatimus korvautuvuusrelaatiolle on yhteensopivuuden säilyttäminen. Sovellettavan korvautuvuusrelaation $\mathcal{R}$ pitää myös säilyttää yhteensopivuus, niin kuin se on määritelty luvussa 5.5.1. Jos komponentit P ja Q ovat yhteensopivia, $P \Diamond Q$ ja $P \mathcal{R} P'$, niin pitäisi olla siten että myös $P' \Diamond Q$. Tämä ei kuitenkaan toteudu bisimulaatioekvivalenssia käytettäessä.

Mikään tähän asti määritellyistä bisimulaatioista, vahva tai heikko, ei pysty erottamaan prosessin onnistunutta ja epäonnistunutta suoritusta [CPT01]. Esimerkiksi prosesseille $P = \mathbf{0}, P'(a) = (a)a(x).\mathbf{0}$ ja $Q = \tau.\mathbf{0}$ on voimassa $P \sim P'$ ja $P \Diamond Q$, mutta $P' \Diamond Q$ ei päde. Tämän takia täytyy määritellä niin kutsuttu $\mathbf{0}$-bisimulointirelaatio (määritelmä 5.20), joka erottaa onnistuneen suorituksen epäonnistuneesta [CPT01].

Määritelmä 5.20 (Vahva $\mathbf{0} - bisimulointi$) *Vahva $\mathbf{0}$-bisimulointi on relaatio $\mathcal{R}_0 \in S \times S$, siten että*

1. *jos $(P,Q) \in \mathcal{R}_0$ ja $P \xrightarrow{\tau} P'$, niin $\exists Q'$ siten että $Q \xrightarrow{\tau} Q'$ ja $(P',Q') \in \mathcal{R}_0$*
2. *jos $(P,Q) \in \mathcal{R}_0$ ja $P \xrightarrow{\bar{x}y} P'$, niin $\exists Q'$ siten että $Q \xrightarrow{\bar{x}y} Q'$ ja $(P',Q') \in \mathcal{R}_0$*
3. *jos $(P,Q) \in \mathcal{R}_0$, $P \xrightarrow{x(y)} P'$ ja $y \notin n(P,Q)$, niin $\exists Q'$ siten että $Q \xrightarrow{x(y)} Q'$ ja $\forall w(P'\{w/y\}, Q'\{w/y\}) \in \mathcal{R}_0$*

4. *jos* $(P, Q) \in \mathcal{R}_0$, $P \stackrel{\bar{x}(y)}{\rightarrow} P'$ *ja* $y \notin n(P, Q)$, *niin* $\exists Q'$ *siten että* $Q \stackrel{\bar{x}(y)}{\rightarrow} Q'$ *ja* $(P', Q') \in \mathcal{R}_0$
5. *jos* $P \equiv \mathbf{0}$, *niin* $Q \equiv \mathbf{0}$

Relaatio $\mathcal{R}_\mathbf{0}$ *on vahva* $\mathbf{0}$*-bisimulointi, jos myös* $\mathcal{R}_\mathbf{0}^{-1}$ *on vahva* $\mathbf{0}$*-bisimulointi. Prosessien* $\mathbf{0}$*-bisimilaarisuus* $\dot{\sim}_\mathbf{0}$ *on suurin prosessien* $\mathbf{0}$*-bisimulointi* $\{\bigcup R : R \; on \; \mathbf{0} - bisimulointi\}$*. Prosessit* P *ja* Q *ovat vahvasti* $\mathbf{0}$*-bisimilaariset, merkitään* $P \sim_\mathbf{0} Q$*, jos* $P\sigma \sim_\mathbf{0} Q\sigma$ *kaikilla nimien korvauksilla* σ*.*

Vahva $\mathbf{0}$-bisimulointi määrittelee prosessien bisimuloinnille lisäehdon, jonka mukaan prosessit joista ei ole yhtään siirtymää eteenpäin, ovat $\mathbf{0}$-bisimilaarisia jos ja vain jos molemmat ovat $\mathbf{0}$-prosesseja. Lisäehto takaa sen, että prosessien onnistuneet ja epäonnistuneet suoritukset voidaan erottaa toisistaan.

5.5.3 Mietteitä korvautuvuusrelaation muodostamisesta

Komponenttien rajapintaprotokollien korvautuvuusrelaation tulisi sisältää kaikki viisi luvussa 5.5.2 esiteltyä ominaisuutta. Mikään tässä opinnäytetyössä esitellyistä prosessien välisistä relaatioista ei suoraan sovellu näiden vaatimuksien mukaiseksi korvautuvuusrelaatioksi. Esitellyistä relaatioista saadaan kuitenkin jonkinlainen tuntuma sille, minkälainen korvautuvuusrelaatiosta tulisi kehittää.

Korvautuvuusrelaatiossa tarvittavien ominaisuuksien määrittely ja ominaisuuksien formalisointi sekä todistaminen vaatii tarkempaa analyysiä ohjelmistotuotannon käsitteiden luonteesta ja mahdollisesti matemaattisesti raskasta todistuskoneistoa. Tämän vuoksi korvautuvuusrelaation varsinainen muodostaminen joudutaan toistaiseksi jättämään avoimien tutkimusongelmien joukkoon ja tässä luvussa tyydytään vain valoittamaan mahdollisia ratkaisumalleja korvautuvuusrelaatiolle.

Korvautuvuusrelaatio yhdistää ominaisuuksia useista edellisissä luvuissa esitellyistä prosessiekvivalensseista. Korvautuvuusrelaatio tulisi pohjautua prosessien välisen heikon bisimuloinnin (määritelmä 5.6) kaltaiseen relaatioon, jotta rajapintaprotokollien sisäiset siirtymät saadaan abstrahoitua. Lisäksi korvautuvuusrelaatioon tulee sisällyttää haarautuvan bisimimuloinnin (määritelmä 5.19) mukainen τ-polkujen tulkinta, jotta komponenttien korvaus säilyttää ulkoisen käyttäytymisen mahdollisimman eheänä. Korvautuvuuden tulisi myös $\mathbf{0}$-bisimuloinnin (määritelmä 5.20) tavoin erottaa lukkiutuma onnistuneesta terminoinnista. Lisäksi korvautuvuusrelaation tulisi olla avoimen bisimulaation (määritelmä 5.12) periaatteiden mukaisesti määritelty π-kalkyylin kongruenssi relaatio.

Korvautuusrelaation määrittelyssä voitaisiin käyttää apuna esimerkiksi ylimääräisten siirtymien piiloitusta, kätkentää tai rajoitusta [Weh03]. Tällä tavalla tiettyjen korvautuvuusvaatimusten kannalta turhat toteutusyksityiskohdat (esimerkiksi ylimääräiset laajennusmetodit ja uusi toiminnallisuus) voitaisiin jättää korvautuvuustarkistuksesta pois. Korvautuvuusrelaationa voitaisiin ehkä siis käyttää "rajoitettua bisimulointia", jossa bisimulointitarkistus koskisi vain yhteisiä siirtymiä.

Korvautuvuusrelaation ratkaisu ei kuitenkaan ole näin yksiselitteinen ja suoraviivainen, sillä korvautuvuusrelaatiossa täytyy huomioida esimerkiksi input- ja output-toimintojen mahdollinen epäsymmetrisyys: perinteisesti ensin mainittu on passiivinen, kun taas jälkimmäinen on aktiivinen toiminto. Tällaiset asiat täytyy huomioida, jolloin ei enää välttämättä voida olettaa, että tiettyjä käyttäytymiseen liittyviä ominaisuuksia voitaisiin vain yksinkertaisesti jättää huomioimatta. Lisäksi ei ole lainkaan selvää, kuinka tällainen rajoitetulla bisimuloinnilla toteutettu korvautuvuusrelaatio käyttäytyy π-kalkyylin prosessien tapauksessa (onko se kongruenssi?) ja minkälaisia temporaalilogiikan ominaisuuksia kyseinen relaatio säilyttää.

Toinen mielenkiintoinen vaihtoehto on simulaatioesijärjestyksien käyttö korvautuvuusrelaationa. Simulaatioesijärjestys säilyttää tietyn CTL-logiikan kaltaisen temporaalilogiikan osajoukon, nimittäin $\forall CTL^*$, kaavat [MRR03]. Jotta simulaatioesijärjestystä voisi käyttää korvautuvuusrelaationa, tulisi sen äilyttää komponenttien välisen yhteentoimivuuden, niin kuin se määriteltiin luvussa 5.5.1 täytyy selvittää.

6 Komponenttien yhteentoimivuuden tarkistaminen ohjelmallisesti

Tässä luvussa kuvataan komponenttien yhteentoimivuuden tarkistusohjelmiston toiminta- ja toteutusperiaatteet. Tarkoituksena on kuvailla ohjelmisto, jonka avulla ohjelmistokomponenttien välinen yhteentoimivuus voidaan tarkistaa. Ohjelmistolla voidaan tarkistaa, onko kaksi tarpeellisilla laajennoksilla varustettua WSDL-kuvausta (*Web Service Description Language*) joko yhteentoimivia tai keskenään korvattavissa.

Ohjelmiston toimitaperiaatteet pohjautuvat edellisissä luvuissa esiteltyihin asioihin. Ohjelmistokomponentit tulkitaan monikoksi $C = (\Sigma, \mathcal{P}, \mathcal{T})$, kuten edellisissä luvuissa on esitelty. Tämän tulkinnan kautta voidaan ohjelmisto jakaa kolmeen osaan: yksi osa huolehtii syntaktisten rakennekuvausten välisistä yhteentoimivuustarkistuksista, toisen osan velvollisuuksiin kuuluu antaa tarvittava välineistö rajapintaprotokollien välisten yhteentoimivuustestien toteuttamiseen ja kolmas osa ylläpitää tyypityskontekstin $\mathcal{T}$ vaatimia tyypityssääntöjä ja niihin liittyvää loogista koneistoa.

Aliluvussa 6.1 annetaan kuvaus ohjelmiston yleisestä rakenteesta sekä toimintatavasta. Tämän jälkeen aliluvussa 6.2 esitellään tapa kuvata komponenttien ominaisuuksia. Näiden alustusten jälkeen siirrytään tarkastelemaan ohjelmiston eri komponenttien toteutusperiaatteita aliluvuissa 6.3, 6.4 ja 6.5.

6.1 Ohjelmiston yleiskuvaus

Ohjelmiston näkyvä toiminnallisuus jakautuu kahteen osaan: komponenttien välisen yhteensopivuuden verifioimiseen ja komponenttien keskinäisen korvautuvuuden tarkistamiseen. Ohjelmisto saa kummassakin tapauksessa syötteenään kaksi laajennettua WSDL-kuvausta. Tarvittavat laajennokset kuvataan luvussa 6.2.

Yhteensopivuuden verifioinnissa ohjelmisto palauttaa arvon tosi, jos WSDL-kuvausten mukaiset komponenttien yhdistäminen ohjelmistoarkkitehtuuriksi on turvallista, eli ei johda lukkiutumaan. Yhteentoimivuustarkistus palauttaa epätoden, jos komponentit eivät ole yhteensopivia. Yhteentoimivuustestissä tarkistetaan, että kommunikoitava tieto ja kommunikointitapahtumien järjestys on määritelmien mukaan yhteentoimivaa. Yhteensopivuutta tarkistettaessa varmistetaan myös, että osapuolet vastaanottavat ja lähettävät sekä syntaktisesti että tulkinnallisesti oikeaa tietoa. Yhteensopivuustarkistuksessa voidaan siten tarvita myös kommunikointialkioiden välisiä korvautuvuustestejä.

Korvautuvuustestissä tarkistetaan, voidaanko komponentit korvata keskenään siten, että korvaus ei johda yhteentoimivuusongelmiin. Korvautuvuustesti jakautuu kolmeen osaan. Ensin tarkistetaan, että käytettävät tietotyypit ja komponentit ovat syntaktisella tasolla korvattavissa keskenään. Jos komponentit ovat syntaktisesti korvattavissa keskenään, niin siirrytään testin toiseen osioon, jossa tarkistetaan komponenttien yksittäisten metodien toimintasemantiikkojen korvautuvuus. Jos kaikki metodit ovat toimintasemantiikaltaan keskenään korvattivissa, niin jatketaan korvautuvuustestin kolmanteen osioon, jossa tarkistetaan komponenttien rajapintaprotokollien keskinäinen korvautuvuus. Jos tämäkin testi onnistuu, niin korvautuvuustesti on onnistunut, muuten se on epäonnistunut.

Ohjelmisto rakentuu viidestä peruskomponentista, jotka ovat tyyppisovitin *TypeMatcher*, rajapintaprotokollien sovitin ja verifioija *ProcessMatcher*, sekä tietämysjärjestelmä, joka koostyy päättelijästä *Reasoner* sekä tietämyskannasta.

Komponentti *TypeMatcher* saa kaksi WSDL-kuvauksesta eristettyä tyyppikuvausta, joiden väli-

nen korvautuvuus tulee tarkistaa. Korvautuvuustarkistuksessa tarvitaan tietoa perustyypeistä sekä funktioiden toimintasemantiikasta. Nämä tiedot saadaan *Reasoner*-komponentilta. Komponentin *TypeMatcher* toiminnallisuutta ja toteutusta kuvaillaan tarkemmmin luvussa 6.3.

Prosessisovitin ja -verifioija *ProcessMatcher* saa syötteekseen kaksi rajapintaprotokollan kuvausta. Näiden välillä suoritetaan yhteentoimivuus- tai korvautuvuustestaus. Molemmissa tapauksissa tarvitaan tyyppitietoja, jotka saadaan *TypeMatcher*-komponentilta. Komponentin *ProcessMatcher* toiminnallisuutta ja toteutusta tarkastellaan luvussa 6.5.

Komponentti *Reasoner* ja tietämyskanta muodostavat tietämysjärjestelmän, jota tarvitaan *TypeMatcher*-komponentin toiminnallisuuden tueksi. Tietämyskanta sisältää OWL-ontologioiden kuvauksia ja ontologioiden käsitteillä määriteltyjä faktoja. *Reasoner*-komponentin antaa vastauksia ontologioita koskeviin kysymyksiin ja päättelee uusia faktoja. Tietämysjärjestelmän toimintaan, tietämyksen ylläpitoon ja muodostamiseen, tutustutaan luvussa 6.4.

6.2 Komponenttien kuvaaminen ja kuvaamiskielet

Ohjelmistokomponenttien rajapinnan kuvauksen perustana käytetään WSDL-kieltä [W3C01a]. Niin kutsuttujen Web-palveluiden (*Web Services*) kuvaamiseen kehitetty WSDL-kieli on XML-pohjainen kuvauskieli, jolla voidaan kuvata Web-pohjaisia palveluita. WSDL sisältää perusmuodossaan kuvaukset komponentin metodeille ja kommunikoinnissa käytettäville viesteille. Näiden lisäksi WSDL-kuvaus määrittelee palvelun teknisen kommunikointiprotokollan (yleensä HTTP) ja palvelun osoitteen [W3C01a]. Tulevissa tarkasteluissa keskitytään vain komponentin viesteihin ja metodeihin liittyviin WSDL-kielen abstrakteihin osioihin, jotka eivät ota kantaa teknisiin yksityiskohtiin.

Komponentti kuvataan kokonaisuudessaan WSDL-kielen *definition*-elementin sisällä, joka sisältää yhteentoimivuustarkasteluiden kannalta kolme mielenkiintoista osiota. Nämä osiot ovat *types*-, *message*- ja *portType*-elementit, jotka kuvaavat palvelussa käytettävät tyypit, sanomat ja metodit [W3C01a]. Taulukossa 31 on kuvattuna erään WSDL-kuvauksen abstraktin osuuden yleinen rakenne [Sid01]. Palvelun kuvaus alkaa *definitions*-elementillä, jonka attribuuteissa esitellään kuvauksessa tarvittavat nimiavaruudet.

Palvelukuvauksen *types*-osiossa esitellään kommunikoinnissa käytettävät tietotyypit. Tietotyyppien määrittely tehdään XML-Schema -tekniikkaa hyödyntämällä [W3C01b]. Taulukon 31 esimerkissä määritellään *types*-osiossa tietotyyppi *Vector*, joka kuuluu *MobilePhoneService* nimiavaruuteen. Tietotyyppi *Vector* on luvussa 3.1 määriteltyjä merkintöjä ja käsitteitä käyttäen tulotyyppi $(xsd : String \times xsd : int)$.

Kuvauksen *message*-elementit määrittelevät kommunikoinnin varsinaisen sisällön. Yksi viesti (*message*) voi koostua yhdestä tai useammasta osasta (*part*). Viesti vastaa yleensä metodikutsun parametria tai paluuarvoa. Viestin osa voi olla mikä tahansa XML-skeemalla tai muussa nimiavaruudessa määritelty tietotyyppi [W3C01a]. Taulukon 31 kuvauksessa viestit ovat vain yksiosaisia.

Palvelun varsinaiset metodit kuvataan *portType*-elementissä, joka voi sisältää yhden tai useamman metodikuvauksen. Metodeille määritellään syöteparametrit ja paluuarvo *input*- ja *output*-määreillä, jotka viittaavat kuvausen *message*-määrittelyihin.

Yhteentoimivuustarkistusta varten WSDL-kuvausta täytyy laajentaa. Syntaktisen tarkistuksen osalta WSDL-kuvaus tarjoaa tarpeeksi tietoa, mutta metodien toimintasemantiikkoja ja rajapintaprotokollia ei ole peruskuvauksessa mukana. Tämän vuoksi WSDL-kuvausta laajennetaan tarpeellisin käsittein, jotka määritellään *ServiceInterface* nimisen ontologian avulla.

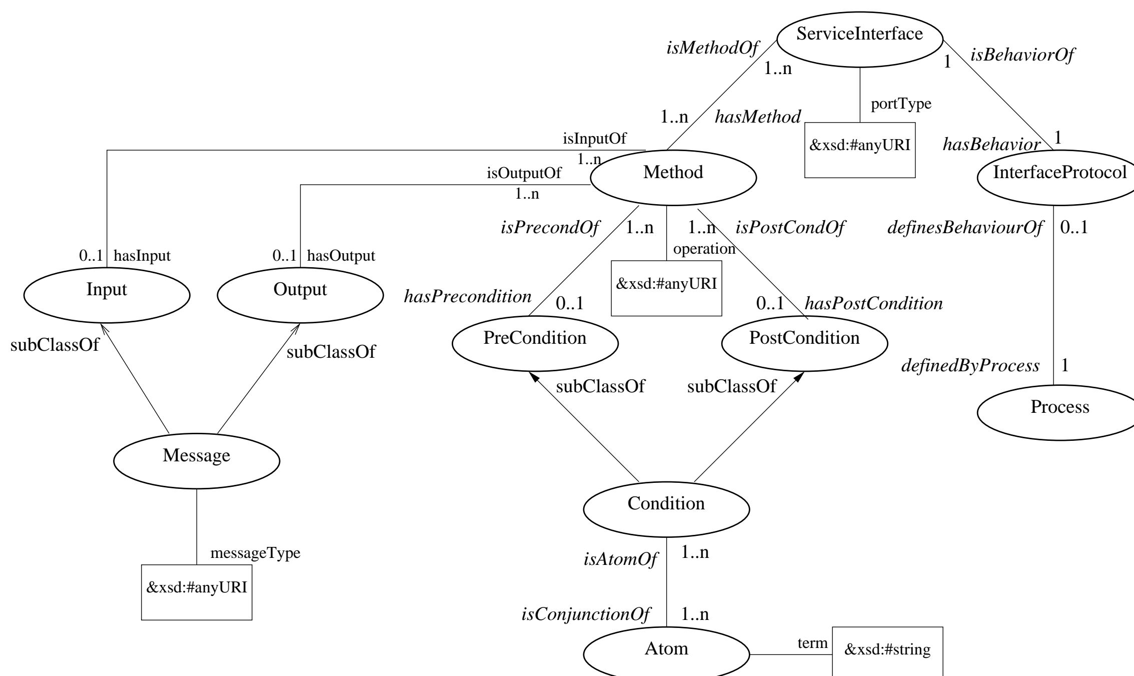
```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions  name="MobilePhoneService"
     targetNamespace="http://www.mobilephoneservice.com/MobilePhoneService-interface"
     xmlns="http://schemas.xmlsoap.org/wsdl/"
     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
     xmlns:tns="http://www.mobilephoneservice.com/MobilePhoneService"
     xmlns:xsd="http://www.w3.org/1999/XMLSchema">

     <types>
         <xsd:schema targetNamespace="http://www.mobilephoneservice.com/MobilePhoneService"
         xmlns="http://www.w3.org/1999/XMLSchema/">
             <xsd:complexType name="Vector">
                 <xsd:element name="elementData" type="xsd:string" />
                 <xsd:element name="elementCount" type="xsd:int" />
             </xsd:complexType>
         </xsd:schema>
     </types>

     <message name="ListOfPhoneModels">
         <part name="models" type="tns:Vector">
     </message>

     <message name="PhoneModel">
         <part name="model" type="xsd:String">
     </message>

     <message name="PhoneModelPrice">
         <part name="price" type="xsd:String">
     </message>

     <portType name="MobilePhoneService_port">
         <operation name="getListOfModels ">
             <output message="ListOfPhoneModels"/>
         </operation>
         <operation name="getPrice">
             <Input message="PhoneModel"/>
             <output message="PhoneModelPrice"/>
         </operation>
     </portType>

</definitions>
```

Taulukko 31: WSDL-kuvauksen yleinen rakenne

WSDL-kuvausta laajennetaan hyödyntämällä nimiavaruuksia [W3C01a]. Nimiavaruuksia käytetään yleisesti Web-palveluiden määrittelyissä ja esimerkiksi WSDL-kuvauksen *binding*-elementissä hyödynnetään SOAP-määrittelyjä esittelevää nimiavaruutta sitomaan palvelu SOAP-siirtoprotokollaan [W3C01a]. Seuraavissa luvuissa käsitellyissä esimerkeissä käytetään nimiavaruutta *http:// www.cs.helsinki.fi/u/thruokol/ServiceInterface.owl* viittaamaan kehiteltävään *ServiceInterface*-ontologiaan.

Ontologiassa *ServiceInterface* määritellään Web-palveluita koskeva käsitteistö, joiden avulla metodien semantiikat ja rajapintaprotokollien toiminta voidaan kuvata. Ontologian kuvaamiseen on käytetty OWL-kieltä (liite B) ja sen rakenne on annettu kuvassa 21.

Käsite *ServiceInterface* määrittelee jonkin WSDL-kuvauksen *portType*-elementin toiminnallisuuden. WSDL-kuvauksen *portType*-elementin URI annetaan *ServiceInterface*-luokan *portType*-elementissä, joka sisältää XML-skeeman *anyURI*-muotoisen tietotyypin [W3C01b]. Palvelurajapinta koostuu tämän lisäksi yhdestä rajapintaprotokollan kuvauksesta ja yhdestä tai useammasta metodin kuvauksesta.



Kuva 21: ServiceInterface-ontologian rakenne

Rajapintaprotokollan kuvaus *InterfaceProtocol* sisältää prosessikuvauksen *Process*-käsitteen muodossa. Prosessikuvausluokasta *Process* voidaan sitten määritellä haluttua formalismia käyttävä aliluokka, joka määrittelee varsinaisen rajapintaprotokollan. Tämän lopputyön liitteessä D on esimerkki OWL-kielellä kuvatusta π-kalkyylin ontologiasta, jonka avulla π-prosesseja voidaan kuvata XML-pohjaisesti. Koska prosessikuvaukset ovat OWL-kielellä kuvattuja, voidaan prosessikuvauskieltä kehittää ja laajentaa helposti tarvittavaan suuntaan. Jos lisäksi on käytössä validoiva OWL-parseri, niin OWL-kieltä voidaan hyödyntää prosessikuvauskielen semantiikan määrittelyyn. Validoivaa OWL-parseria käyttäen voidaan vaikka luoda rajoitteita π-prosessien käyttämille nimille, esimerkiksi erottaa kanavien nimet muista nimistä.

Metodin ontologinen määritelmä sidotaan WSDL-kuvaukseen *operation*-tietotyypillä, joka on URI-muotoinen viittaus WSDL-kuvauksen sisältämään *operation*-elementtiin. WSDL-kuvauksen operaation *input*- ja *output*-alkiolle annetaan semanttinen tulkinta ontogian käsitteiden *Input*- ja *Output*-avulla. Käsitteet ovat aliluokkia *Message*-käsitteelle, joka sisältää *messageType* nimisen URI:n viestin varsinaiseen ontologiseen kuvaukseen. Metodin syöte- ja tulosteparametreille ei ole pakko liittää semanttista tulkintaa. Tällöin yhteensopivuus tarkistetaan vain viestien syntaktisella tasolla.

Metodin kuvaus sisältää määrittelyt etu- ja jälkiehdoille, joita kumpiakin voi olla korkeintaan yksi. Etu- ja jälkiehdot ovat *Condition*-luokan aliluokkia. Jokainen *Condition*-luokka sisältää yhden tai useamman *Atom*-luokan käsitteen. Käsite *Atom* edustaa loogisia, atomisia termejä, joista konjunktion avulla muodostetaan looginen ehto. Tällä hetkellä *Atom* sisältää vain XML-skeeman *string*-muotoisen tietotyypin nimeltään *term*, jonka tulkitaan olevan jonkin ontologian väite. Metodille ei ole pakko antaa etu- ja jälkiehtoa. Jos jompaa kumpaa tai molempia ei olla määritelty, niin ehdot tulkitaan loogisesti todeksi.

Taulukossa 32 on kuvattu, kuinka ontologian *ServiceInterface* käsitteisiin voidaan viitata WSDL-kuvauksesta käsin.

```
<definition ...
 xmlns:interop="http://www.cs.helsinki.fi/u/thruokol/ServiceInterface.owl#"
 xmlns:mybank="http://www.somebank.com/Ontologies/MyBankingService.owl#"
 xmlns:tns="http://..."
>
...
<message name="transferMessage" interop:Message="mybank:moneyTransfer" />

<message name="transferReceipt" interop:Message="mybank:receipt" />

<portType name="BankingService" interop:InterfaceProtocol="mybank:myServiceProtocol">
  <operation name="transferMoney" interop:Method="myBank:transferMoney">
   <input message="#transferMessage" />
   <output message="#transferReceipt" />
  </operation>
</portType>
...
</definition>
```

Taulukko 32: Semanttisten laajennusten liittäminen WSDL-kuvaukseen

Ontologia *ServiceInterface* ei sisällä käsitteitä esimerkiksi ehdollisille lopputuloksille, joissa metodin tulos riippuu loogisesta ehdosta. Toteutettava ontologia on eräänlainen abstraktimpi ja rajoitetumpi versio niin kutsutusta OWL-S -kielestä, joka on DAML-S -yhteenliittymän kehittämä palveluiden kuvausontologia [DAM04].

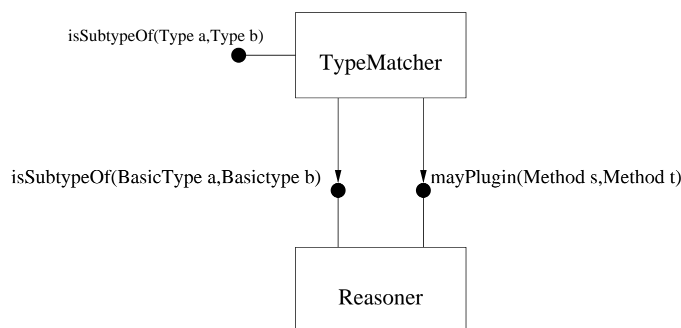
Määritelty palveluontologia ja abstraktit WSDL-kuvaukset muodostavat kehikon, joka soveltuu OWL-S -kieltä paremmin uudelleenkäytettävien palvelukomponenttien kuvaamiseen ja määrittelyyn. Tähän on syynä se, että kuvaukset eivät sisällä turhia toteutusteknisiä yksityiskohtia, kuten eräät OWL-S -kielen rakenteet tekevät. Lisäksi *ServiceInterface*-ontologia on joustavampi komponentin käyttäytymisen määrittelyssä kuin OWL-S, sillä rajapintaprotokollat voidaan kuvata π-kalkyylin syntaksin ja semantiikan rajoissa. OWL-S -kielessä prosessien kuvauskeinot on rajoitettu ennalta määriteltyihin kaavaimiin, kuten *Split-Join* tai *Sequence* [DAM04].

Edellä määritellyn ontologian *ServiceInterface* käyttöä WSDL-kuvauksen laajentamiseen käsitellään luvuissa 6.4 ja 6.5. Ontologiaan *ServiceInterface* viitataan käyttäen URI-muotoista osoitetta *http://www.cs.helsinki.fi/u/thruokol/ServiceInterface.owl*, josta yleensä käytetään XML-nimiavaruuden mukaista lyhennysmerkintää *xmlns:interop* [W3C99a]. Tällöin esimerkiksi *ServiceInterface*-ontologian käsitteeseen *Method* viittausta merkitään *interop:Method*.

6.3 TypeMatcher-komponentin toiminta

Tässä aliluvussa kuvataan järjestelmän *TypeMatcher*-komponentin toimintaa. Tämän komponentin tarkoitus on selvittää, ovatko kaksi annettua tyyppikuvausta keskenään korvattavissa staattisten ominaisuuksiensa puolesta. Komponentti selvittää, onko annettujen tyyppikuvausten välillä korvautuvuussuhde, joka syntaktisen rakenteen ja metodien semantiikan puitteissa ei riko Liskovin korvautuvuusperiaatetta eli on turvallinen (määritelmä 2.3).

Komponentti *TypeMatcher* toteuttaa metodin *isSubtypeOf(Type a,Type b)*. Kyseinen metodi saa syötteekseen kaksi tyyppikuvausta a ja b. Metodi palauttaa totuusarvon *true*, jos $a \leq_{AC}$, muuten palautetaan *false*. Komponentin yleinen rakenne on mallinnettu kuvaussa 22. Komponentti *TypeMatcher* käyttää *Reasoner*-komponentin metodeja *isSubtypeOf* ja *mayPlugin*. Ensin mainittua metodia käytetään kahden perustyypin välisen alityypityssuhteen tarkistamiseen. Metodia *mayPlugin* tarvitaan kahden funktiotyypin välisen semanttisen alityypityksen tarkistamiseen.



Kuva 22: TypeMatcher-komponentin yleiskuva

Korvautuvuustarkistuksen runkona toimii luvussa 3.4.2 kuvatut algoritmit, joiden toimintaa laajennetaan siten, että ne käyttävät tietämysjärjestelmää hyväkseen perustyyppien ja funktiotyyppien alityypitystä tarkistettaessa. Perustyyppien väliset alityypityssuhteet ja metodien semanttinen vastaavuus voidaan selvittää tietämysjärjestelmän *Reasoner*-komponentin avulla.

Ontologisten käsitteiden lisäksi voidaan perustyypeiksi tulkita myös XML-Schema -standardiluonnoksessa määritellyt XML-pohjaiset tietotyypit [W3C01b]. Näitä varten tietämysjärjestelmään on määriteltävä XML-Schema -tietotyyppien väliset alityypityssuhteet. Kun alityyppisuhteet on määritelty, voidaan myös XML-Schema -pohjaisten tietotyyppien välinen korvautuvuus huomioida. Tyyppijärjestelmää voitaisiin laajentaa tarvittaessa samalla periaatteella myös esimerkiksi CORBA-kuvauksia käsitteleväksi järjestelmäksi.

Abstraktin tyyppikuvauksen tarkoituksena on tehdä tyyppitarkistin mahdollisimman riippumattomaksi WSDL-kielestä, jotta sitä voitaisiin käyttää myös esimerkiksi CORBA-väliohjelmiston rakenteiden tarkistamiseen. Lisäksi abstraktissa tyyppikuvauksessa nimiavaruuksien lyhenteet on lavennettu täydelliseen muotoonsa. Jos tarkistettavat tyypit ovat WSDL-kuvauksen abstrakteja tyyppikuvauksia, niin silloin kaikki perustyyppien identifiointiin käytettävät <*string*>-elementit ovat URI-muotoisia viittauksia tyypin määrittelyyn, esimerkiksi "*http://www.w3.org/2001/XMLSchema-#string*" tai "*http://www.cs.helsinki.fi/u/thruokol/MyBankinService.owl#receipt*".

Komponentin *TypeMatcher* metodi *isSubtypeOf(Type a,Type b)* saa parametreinaan kaksi tyyppikuvausta, jotka sisältävät tyyppien rakenteen sekä metodien etu- ja jälkiehdot. Tyyppikuvauksista luodaan termiautomaatit, joiden välinen alityypityssuhde tarkistetaan taulukoissa 16 ja 17 kuvattuja algoritmeja käyttäen. Syntaktisen alityypitysrelaation tarkistusalgoritmit (taulukoissa 16 ja 17) voidaan toteuttaa suoraviivaisesti kuvauksien pohjalta. Vaikein toteutusosio on kuvauksen b laskemisessa käytettävän Hopcroftin ja Karpin algoritmin toteuttaminen tehokkaasti [HK73, Gal86].

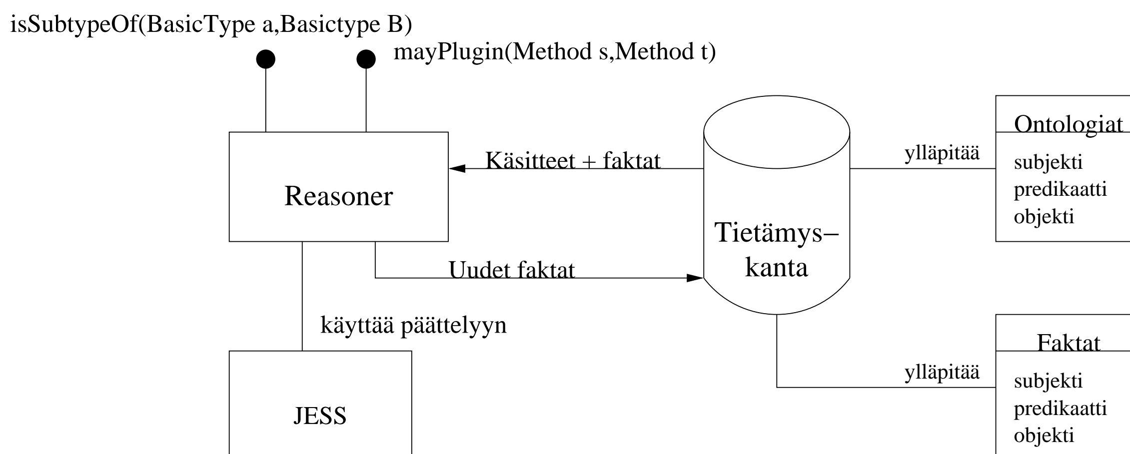
Taulukon 17 algoritmin toiminnallisuutta muokataan siten, että rivillä 4 esiintyvässä perustyyppien alityypitystestissä ($\tau \leq_{TS} \tau'$) tarkistetaan tietämyskannasta, ovatko tyyppejä vastaavat käsitteet toistensa alikäsitteitä. Tämä tarkistus tehdään käyttäen tietämyskannan metodia *isSubtypeOf(BasicType a, BasicType b)*.

Alityypitysrelaation tarkistusalgoritmin riville 8 lisätään toiminnallisuutta, jossa funktioiden välinen semanttinen korvautuvuus tarkistetaan ennen kuin tyypit lisätään joukkoon S. Tämä tarkistus suoritetaan *Reasoner*-komponentin *mayPlugin(Method s,Method t)* metodia käyttäen. Metodi *mayPlugIn* antaa vastauksen sille, onko kahden metodin välillä voimassa määritelmässä 4.1 kuvailtu semanttiseksi alityypitykseksikin kutsuttu $plugin$-suhde. Jos *mayPlugin* palauttaa arvon *true*, niin funktiotyyppien pari lisätään joukkoon S, muuten funktiotyyppien pari lisätään jouk-

koon F ja sen vanhemmat joukkoon W.

6.4 Tietämysjärjestelmän toiminta

Tietämysjärjestelmän tehtävänä on muodostaa ja ylläpitää tyyppejä koskevaa tietämystä. Tietämysjärjestelmä jakaantuu kahteen peruskomponenttiin, jotka ovat tietämyskanta ja päättelijäkomponentti *Reasoner*. Tietämyskannassa ylläpidetään ontologioiden käsitteitä sekä näiden käsitteiden avulla ilmaistuja faktoja. Tietämysjärjestelmän perusrakenne on mallinnetuna kuvassa 23.



Kuva 23: Reasoner-komponentin yleiskuva

Tietämyskanta on perinteinen relaatiotietokanta, joka ylläpitää OWL-kielellä määriteltyihin ontologioihin perustuvaa tietämystä. Tietämyskanta sisältää tiedot sekä ontologioiden määrittelyistä, että maailman faktoista. Kaikki tietämys on esitetty kolmikkona *(subjekti, predikaatti, objekti)*, joka on luonnollinen tapa ylläpitää ontologista tietoutta. Jos esimerkiksi OWL-kielellä on määritelty ontologiassa *http://www.foo.com/Human.owl* luokka *Woman* luokan *Human* aliluokaksi, niin tietämyskannassa on kolmikko *(http://www.foo.com/Human.owl#Woman,http://www.w3.org/2000/01/-rdf-schema#subClassOf,http://www.foo.com/Human.owl#Human)*.

Käsitteisiin ja niiden välisiin suhteisiin perustuva päättely suoritetaan käyttäen niin kutsuttua kuvauslogiikkaa (*description logic, DL*) [NB02]. Perusmuodossaan kuvauslogiikka on ensimmäisen asteen predikaattilogiikan osajoukko, jossa kielen kuvausvoimaa on rajoitettu siten, että päättelyistä saadaan ratkeavia ongelmia. Loogisen implikaation sijasta kuvauslogiikassa käytetään käsitteiden välistä sisältyvyyttä $C \sqsubseteq D$ (*subsumption*), joka tarkoittaa sitä, että käsite D on yleisempi kuin C. Loogisten lausekkeiden sisältyvyys on loogista implikaatiota heikompi ehto eli jos lausekkeille A ja B on voimassa $A \Rightarrow B$, niin on myös voimassa $A \sqsubseteq B$ [NB02].

Tietämysjärjestelmän *Reasoner*-komponentti toteuttaa kaksi metodia. Metodi *mayPlugin(Method s, Method t)* palauttaa totuusarvon *true*, jos metodi s on metodin t semanttinen alityyppi. Parametrit s ja t ovat rakenteita, jotka sisältävät metodien etu- ja jälkiehtojen kuvaukset. Semanttinen alityypitys on taulukossa 18 määritelty $plugin$-suhde, kun implikaation sijasta käytetään käsitteiden sisältyvyyssuhdetta. Tämän metodin toiminnallisuuden toteuttaminen perustuu JESS-ohjelmistoon [FH03].

Tietämysjärjestelmän metodi *isSubtypeOf(BasicType a,BasicType b)* saa syötteekseen kaksi perustyyppiä, joiden käsitteet löytyvät tietämyskannasta. Metodi palauttaa totuusarvon *true*, jos ja

vain jos tietämyskannan ylläpitämistä faktoista löytyy tieto, että peruskäsite a on peruskäsitteen b alikäsite.

Tietämysjärjestelmän *Reasoner*-komponentti vastaa toiminnaltaan asiantuntijajärjestelmää. Asiantuntijajärjestelmä on ohjelmisto, joka annettujen faktojen ja sääntöjen perusteella antaa vastauksen annettuihin kysymyksiin ja tuottaa uusia faktoja. Eräs asiantuntijajärjestelmä on JESS (*Java Expert System Shell*), joka on sääntöpohjainen asiantuntijajärjestelmä [FH03]. Kyseistä asiantuntijajärjestelmää voidaan erityisesti käyttää palveluiden etu- ja jälkiehtoihin perustuvissa sovituksissa ja hauissa [SS03].

Sääntöpohjaisten asiantuntijajärjestelmien voidaan yksinkertaistaen sanoa toimivan siten, että annetut säännöt tulkitaan *if-then* -muotoisiksi rakenteiksi, joita sovelletaan jatkuvasti ylläpidettävään dataan. Uusien sääntöjen ja faktojen lisääminen tietämyskantaan voi siis tuottaa uutta tietämystä, jos lisääminen saa aikaan ehtolausekkeiden laukeamisen [FH03]. Sääntöpohjaiset asiantuntijajärjestelmät ovat tietyssä mielessä reaktiivisia järjestelmiä, jotka reagoivat uusien faktojen ja sääntöjen tuontiin.

Tietämysjärjestelmään joudutaan mallintamaan haluttu semantiikka ja syntaksi. Tämä tarkoittaa sitä että esimerkiksi alityypityssuhteet joudutaan mallintamaan ennen kuin luokkien väliseen alityypitykseen perustuvaa päättelyä voidaan tehdä. Taulukossa 33 on esimerkki JESS-järjestelmän hyväksymällä syntaksilla, niin kutsutulla CLIPS-kielellä, määritellystä RDF-Schema -kielen alityypityssuhteesta [KR03a]. JESS-kielessä säännöt määritellään *defrule*-lauseella ja faktat *assert*-lauseella [FH03]. Säännöt ja faktat ovat muodossa *(predikaatti,subjekti,objekti)*.

Taulukon 33 esimerkkisääntö määrittelee, että kaikki alityypin instanssit ovat myös ylityyppiensä instansseja. Samalla tavalla täytyy määritellä käytettävän kieliopin semantiikka ja säännöt. Komponentin *Reasoner* tapauksessa täytyy mallintaa JESS-järjestelmän kielellä OWL-kielen semantiikka sekä syntaksi, jonka jälkeen voidaan päätellä OWL-ontologioiden käsitteiden ja niiden instanssien välisiä suhteita.

```
(defrule subtype-instances
 (PropertyValue http://www.w3.org/2000/01/rdf-schema#subClassOf ?child ?parent)
 (PropertyValue http://www.w3.org/2000/01/rdf-schema#type ?instance ?child)
 =>
(assert
 (PropertyValue http://www.w3.org/2000/01/rdf-schema#type ?instance ?parent)))
```

Taulukko 33: Esimerkki RDF-Schema -kielen alityypityksen määrittelystä JESS-kielellä

Kun OWL-ontologiat on mallinnettu, voidaan niiden käsitteitä käyttää etu- ja jälkiehtojen termeinä. Etu- ja jälkiehdot tulkitaan termiensä loogiseksi konjunktioksi (katso kuva 21), jossa kaikki *Atom*-luokan instanssit ovat joko muotoa *Käsite_1 conn Käsite_2 conn ... conn Käsite_n*, missä käsitteet ovat OWL-ontologian käsitteitä ja jokainen *conn* voi olla joko *or* tai *and* toisistaan riippumatta. Tämän muotoiset ehtolausekkeet voidaan muuttaa helposti JESS-järjestelmän hyväksymälle kielelle muotoon *(conn (Käsite_1) (Käsite_2) ... (Käsite_n))*.

Etu- ja jälkiehdoista luodaan JESS-sääntöjä, jotka lautessaan muuntuvat JESS-faktoiksi *precondition-satisfy* ja *postcondition-satisfy* [SS03]. Taulukossa 34 on esimerkki JESS-säännön muotoon muutetusta esiehdosta, jossa esiehto täyttyy jos ja vain jos tietyn ontologian käsitteiden välinen suhde *CreditcardHolder decisionOn Amex* tai *CreditcarHolder decisionOn Visa* on voimassa.

Metodien semanttisen alityypityksen tarkistus perustuu siihen, että metodien $\{pre_S\}$ S $\{post_S\}$ ja $\{pre_T\}$ T $\{post_T\}$ etu- ja jälkiehdoista muodostetaan JESS-järjestelmän sääntöjä ja faktoja.

```
(defrule precondition-CardAccepted (or (CreditcardHolder decisionOn Amex)
(CreditcarHolder decisionOn Visa)) => (precondition-satisfy))
```

Taulukko 34: Esimerkki metodin etuehdosta JESS-sääntönä

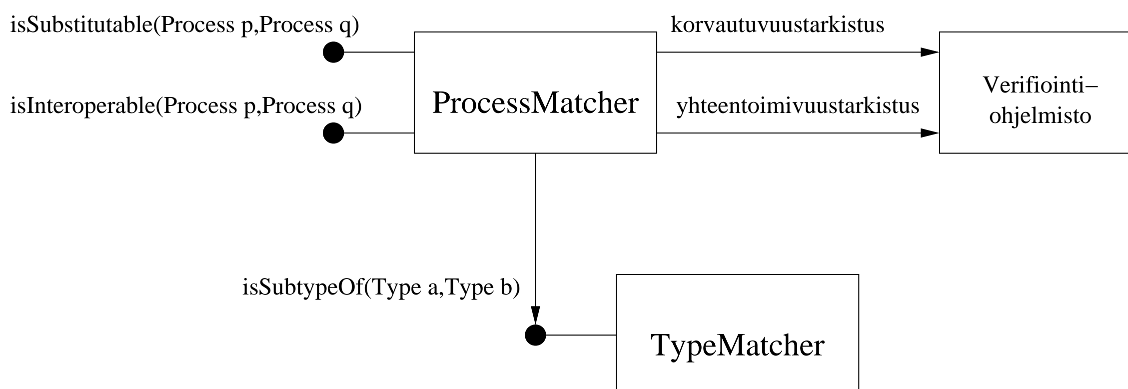
Tietämysjärjestelmään syötetään tietämys OWL-kielestä sekä tarvittavista ontologioista ja niihin liittyvistä faktoista. Käytettävien ontologiat ja faktat saadaan tietämyskannasta. Tietämys OWL-kielestä voi olla sisäänrakennettu ohjelmistoon. OWL-kuvausten muuntaminen JESS-järjestelmän säännöiksi ja faktoiksi voidaan tehdä käyttämällä esimerkiksi XSLT-muunnosta [W3C99b].

Kun tarvittava perustietämys on syötetty, muunnetaan ehdot pre_S ja $post_T$ säännöiksi $precondition - pre_S$ ja $postcondition - post_T$. Edellisten sääntöjen lisäämisen jälkeen järjestelmään lisätään faktat pre_T ja $post_S$. Tällöin säännöt $precondition - pre_S$ ja $postcondition - post_T$ laukeavat jos ja vain jos niiden premissien mukaiset ehdot täyttyvät annettujen faktojen ja ontologioiden mukaisia päättelysääntöjä noudattamalla.

Jos tulokseksi saadaan faktat *precondition-satisfy* ja *postcondition-satisfy*, niin tiedetään että loogisten ehtojen välillä ovat voimassa suhteet $pre_T \sqsubseteq pre_S$ ja $post_S \sqsubseteq post_T$. Tällöin voidaan päätellä, että metodien välillä on voimasssa semanttinen alityypitysrelaatio $S \leq T$ (katso taulukko 18, sääntö $plugin$).

6.5 ProcessMatcher-komponentin toiminta

Komponentti *ProcessMatcher* sisältää kaksi eri toimintoa. Ensimmäinen on komponenttien välisen korvautuvuuden tarkistava funktio *isSubstitutable(InterfaceProtocol P,InterfaceProtocol Q)*, joka palauttaa arvon *true*, jos ja vain jos rajapintaprotokollan Q mukainen komponentti on korvattavissa rajapintaprotokollan P mukaisella komponentilla. Toinen metodi, jonka *ProcessMatcher* toteuttaa, on *isInteroperable(InterfaceProtocol P,InterfaceProtocol Q)*, joka palauttaa totuusarvon *true*, jos ja vain jos rajapintaprotokollat P ja Q ovat keskenään yhteentoimivia.



Kuva 24: ProcessMatcher-komponentin yleiskuva

Komponentin *ProcessMatcher* toiminta voidaan osaksi perustaa valmiiden π-kalkyyliprosessien verifiointiohjelmiston käyttöön. Verifiointiohjelmistolla suoritetaan π-prosessien mallintarkistus sekä prosessien välinen bisimulointitarkistus. Valmiita verifiointiohjelmistoja, joilla nämä toimenpiteet voidaan tehdä ovat esimerkiksi *Mobility Workbench* (MWB) ja *HD Automata Laboratory*

(HAL) [Vic94, FGM+98]. Molempia ohjelmistoja voidaan käyttää π-kalkyylillä mallinnettujen prosessien mallintarkistukseen sekä prosessien välisen bisimuloinnin tarkistamiseen.

MWB-ohjelmistossa mallintarkistus suoritetaan "on-the-fly" -periaatteella kun taas HAL-ohjelmistossa π-prosessista muodostetaan äärellinen automaatti ennen tarkistusta [Vic94, FGM+98]. Rajapintaprotokollat kuvaavat aina palvelun äärellisen käyttötapauksen tai -tavan, joten äärettömille kuvauksille ei ole tarvetta. Tästä syystä sekä MWB- että HAL-ohjelmistoja voidaan kumpaakin käyttää yhtä hyvin. Koska HAL-ohjelmisto eksplisiittisesti rakentaa koko prosessien suorituspuun, voidaan verifioinnin epäonnistuessa antaa käyttäjälle vastaesimerkki. Tämä ei ole mahdollista "on-the-fly" -tyylisessä verifioinnissa.

Mallintarkistukseen MWB-ohjelmistossa käytetään niin kutsuttua modaalista μ-logiikkaa, kun taas HAL-ohjelmistossa käytetään π-logiikkaa. Näistä kahdesta modaalilogiikasta on MWB:ssä käytettävä μ-logiikka ilmaisuvoimaisempi, mutta CTL-aikalogiikkaan perustuva π-logiikkaa on tarpeeksi ilmaisuvoimainen rajapintaprotokollien ominaisuuksien ilmaisemiseen.

Edellä mainittujen ohjelmistojen suurin puute *ProcessMatcher*-komponentin toiminnan kannalta on se, että niissä ei ymmärrettävistä syistä ole valmiiksi toteutettuna luvussa 5.5 esiteltyjä korvautuvuusekvivalensseja. Niitä voitaisiin käyttää kuitenkin "korvautuvuustarkistajan" lähtökohtina, mutta esimerkiksi MWB-toteutus on jo itsessään virheellinen ja HAL-ohjelmistosta ei ole toistaiseksi saatavilla lähdekoodia [3].

Rajapintaprotokollien korvautuvuuden tarkistava metodi *isSubstitutable* saa syötteekseen kaksi *InterfaceProtocol*-käsitteen mukaista XML-dokumenttia. Käsite *InterfaceProtocol* sisältää ominaisuuden *definedByProcess*, joka viittaa varsinaiseen prosessikuvaukseen *Process* (katso kuva 21). Tässä opinnäytetyössä *Process*-luokan instanssit ovat *PiProcess*-aliluokan instansseja. Luokka *PiProcess* määrittelee OWL-kuvauksen muodossa käytettävän π-kalkyyliin pohjautuvat prosessinkuvauksen käsitteet. OWL-kuvaus on annettu liitteenä D.

Käsitteen *PiProcess* mukaiset rajapintaprotokollakuvaukset muunnetaan XSLT-muunnoksella verifiointiohjelmiston hyväksymän π-kalkyylin syntaksille. Kun rajapintaprotokollat on muunnettu π-prosesseiksi P' ja Q', voidaan niiden välinen korvautuvuus tarkistaa verifiointiohjelmistoa käyttäen. MWB-ohjelmistolla voidaan tarkistaa prosessien välinen heikko ja vahva (avoin) bisimilaarisuus [Vic94]. HAL-ohjelmisto perustuu JACK-verifiointiohjelmistoon, jolla voidaan tarkistaa heikon ja vahvan bisimilaarisuuden lisäksi myös haarautuva bisimilaarisuus [BGL94].

Rajapintaprotokollien yhteentoimivuuden tarkistava metodi *isInteroperable* saa syötteekseen kaksi *InterfaceProtocol*-käsitteen mukaista kuvausta P ja Q. Rajapintaprotokollien muunnos π-kalkyylin muotoon tapahtuu samalla tavalla kuin korvautuvuustarkistuksessakin käyttäen XSLT-muunnosta. Verifiointiohjelmistolle annetaan syötteeksi π-prosessi $P' \mid Q'$, missä P' ja Q' ovat sekä temporaalilogiikan kaava ϕ. Tämän jälkeen suoritetaan mallintarkistus, jossa selvitetään, onko prosessi $P' \mid Q'$ mallin ϕ mukainen. Jos vastaus on myöntävä, niin käyttäjälle palautetaan arvo *true*. Jos mallintarkistus päättyi virheeseen, niin käyttäjälle palautetaan joko pelkästään totuusarvo *false* tai mahdollisesti myös lisäksi syy sille, miksi annetut rajapintaprotokollat eivät ole yhteentoimivia. HAL-verifiointiohjelmiston käyttämä AMC-mallintarkistusohjelma tuottaa vastaesimerkin, jos prosessi ei ole annetun temporaalilogiikan kaavan mukainen [BGL94].

Rajapintaprotokollien yhteentoimivuuden tarkistuksessa on tärkeää, että rajapintaprotokollien oikeellinen ja virheellinen terminointi voidaan erottaa toisistaan. Perusmuodossaan π-kalkyylissä prosessin terminointi mallinnetaan **0**-prosessilla, joka mallintaa lukkiutunutta prosessia. Tätä **0**-

[3] Nämä tiedot perustuvat kyseisten ohjelmistojen kehittäjien kanssa käytyihin sähköpostikeskusteluihin kevään 2004 aikana.

prosessia tullaan käyttäämään mallintamaan rajapintaprotokollan virheellistä terminointia. Tätä tulkintaa käyttäen voidaan rajapintaprotokollat P ja Q todeta yhteentoimimattomiksi jos prosessi $P \mid Q$ lukkiutuu.

Prosessien lukkiutuminen voidaan todeta mallintarkistusta käyttäen. CTL-logiikan kaltaista modaalilogiikkaa käyttäen prosessin lukkiutumattomuus voidaan ilmaista kaavalla $NoDeadLock = AG(< - > true)$, missä $< - >$ tarkoittaa sitä, että tilasta on mahdollista suorittaa mikä tahansa siirtymä. Kaava $NoDeadLock$ on siis tosi, jos kaikista tiloista on mahdollista päästä tilaan, missä jonkin siirtymän suorittaminen on mahdollista. Jos mallintarkistuksessa havaintaan, että kaava $NoDeadLock$ on epätosi, niin prosessi on lukkiutunut.

Onnistunut prosessin terminointi määritellään prosessiksi $\overline{exit}.\mathbf{0}$ eli prosessi on päättynyt oikeellisesti jos ja vain jos sen viimeinen siirtymä on $\overline{exit}$-siirtymä. Onnistuneesti päättynyt prosessi voidaan kuvata CTL-logiikan kaavalla $ExitOK = AG(EF(< \overline{exit} > (< - > FF)))$. Kaava $ExitOK$ on tosi, jos aina on mahdollista löytää suorituspolku, joka päättyy $\overline{exit}$-siirtymään ja kyseisen siirtymän jälkeen ei ole mahdollista suorittaa mitään siirtymiä. Kaava $ExitOK$ on siis tosi prosesseille, jotka aina päättyvät prosessiin $\overline{exit}.\mathbf{0}$.

Komponentin *ProcessMatcher* toteuttamiseksi tulee siis joko laajentaa perinteisiä verifiointiohjelmistoja tarvittavin muunnoksin tai toteuttaa korvautuvuus- ja yhteentoimivuustarkistusalgoritmit täysin tyhjästä. Ensimmäinen vaihtoehto on parempi, mutta se vaatisi, että käytettävissä olisi oikeellisesti toimivan verifiointiohjelmisto lähdekoodeineen. Tarvittavien algoritmien tyhjästä toteuttaminen taas voi osoittautua hyvinkin työlääksi.

7 Yhteenveto

Tässä opinnäytetyössä tarkasteltiin komponenttien välistä yhteentoimivuutta ja erityisesti sitä, miten kahden komponentin välinen yhteentoimivuus voidaan tarkistaa. Yhteentoimivuuden käsitteellä tarkoitetaan sitä, että kaksi komponenttia voivat toimia yhdessä oikeellisesti. Yhteentoimivuutta käsiteltiin komponentin syntaksin, semantiikan ja käyttäytymisen alueilla.

Luvussa 2 tutustuttiin yhteensopivuuden käsitteeseen sekä ohjelmistokomponenttien tyypitykseen. Yhteentoimivuus jaettiin kahteen, toisitaan riippuvaan osioon: yhteensopivuuteen ja korvautuvuuteen. Korvautuvuus on käsitteenä staattisempi ja se tulee esiin kaikissa yhteentoimivuuden tasoissa. Yhteensopivuus on taas komponentin dynaamiseen käytökseen liittyvä ominaisuus. Ohjelmistokomponentin tyypityksellä voidaan komponentin jakaa keskenään eri ekvivalenssiluokkiin niiden ominaisuuksien suhteen. Oikeanlainen tyypitys on välttämätön ehto onnistuneelle yhteentoimivuustarkistukselle. Erityisesti luvussa 2 annettiin ohjelmistokomponentin määritelmä monikkona $\mathcal{C} = (\Sigma, \mathcal{P}, \mathcal{T})$ eli komponentti koostuu rajapinnan rakennekuvauksesta Σ, rajapintaprotokollasta $\mathcal{P}$ sekä tyypityskontekstista $\mathcal{T}$. Luvun lopuksi annettiin alityypityksen kuvaus ja erityisesti määriteltiin niin kutsuttu turvallinen alityypitys, joka säilyttää komponenttien ominaisuudet korvausoperaation yhteydessä.

Syntaktisella tasolla korvautuvuus määriteltiin käyttäen perinteisestä tyyppiteoriasta tulevia käsitteitä ja menetelmiä käyttäen. Korvautuvuustarkistus perustuu jonkin tyyppiteorian soveltamiseen. Luvussa 3 esiteltiin tyyppiteoria Th, joka koostuu seitsemästä aksioomasta sekä ekvivalenssin käsitteestä. Aksioomien avulla vertailtavat tyypit saadaan muutettua niin kutsuttuun normaalimuotoon. Tämän jälkeen normaalimuotoisia termejä vertaillaan soveltaen jotakin ekvivalenssirelaatiota. Ekvivalenssin käsite määriteltiin niin kutsuttujen termiautomaattien avulla. Termiautomaattien ekvivalenssi redusoitui niiden väliseksi bisimulointi- tai simulointiekvivalensseiksi. Erityisesti näiden termien välisten relaatioden assosiatiiviset ja kommutatiiviset versiot havaittiin kaikista käyttökelpoisimmiksi ja kiinnostavimmiksi. Luvun 3 lopuksi tutustuttiin termiautomaattien ekvivalenssi- ja alityypitysrelaatioiden tarkistusalgoritmien toimintaperiaatteisiin.

Luvussa 4 määriteltiin kuinka komponentin toimintaan liittyvää semantiikkaa voidaan tuoda ilmi. Semanttiset määrittelyt voidaan jakaa kahteen osioon: niihin, jotka määrittelevät komponentin funktioiden toimintasemantiikkaa ja niihin, jotka määrittelevät käytettävien tietoalkioiden tulkinnat. Funktioiden toimintasemantiikan puolelta mielenkiintoisin korvautuvuusrelaation on *plugin*-tyyppinen looginen suhden funktioiden etu- ja jälkiehtojen välillä. Tämänlainen looginen suhde voidaan myös tulkita eräänlaiseksi toimintasemanttiseksi alityypitykseksi. Komponentin toiminnassa käytettäville tietoalkiolle voidaan antaa semanttinen tulkinta joko käyttämällä niin kutsuttuja abstrakteja tietotyyppejä tai ontologisia käsitteitä. Abstraktit tietotyypit ovat puhtaasti matemaattinen tapa kuvata tietotyyppejä ja siten sillä on omat hyvät ja huonot puolensa. Ontologioihin perustuva käsitteellistäminen taas ei näy käyttäjälle niin matemaattisena ja sillä pystytään kuvaamaan paremmin "arkipäiväisiä" käsitteitä kuin abstrakteilla tietotyypeillä. Ontologioiden käyttö kuitenkin perustuu puhtaasti sopivan predikaattilogiikan sääntöihin.

Komponenttien välinen yhteistoiminta määritellän käyttäen soveltuvaa prosessialgebraa. Luvussa 5 tutustuttiin ensin siihen, mitä tarkoitetaan rajapintaprotokollilla ja niiden välisellä yhteentoimivuudella. Prosessialgebran alkeita käytiin läpi käyttäen yksinkertaista CCS-pohjaista prosessialgebraa, jonka jälkeen siirryttiin erityisesti mobiilien järjestelmien kuvaukseen kehitettyyn π-kalkyyliin. Rajapintaprotokollien yhteensopivuustarkistuksissa voidaan joutua tarkistamaan tiettyjä ominaisuuksia, kuten esimerkiksi viestinvaihdon järjestykseen liittyviä rajoitteita tai vastaavaa. Tämän vuoksi käytiin hieman läpi niin kutsutun mallintarkistuksen periaatetta sekä erityisesti ää-

rellisen automaatin ominaisuuksien kuvaamiseen käytettävään ACTL-temporaalilogiikkaan. Mallintarkistuksen lisäksi toinen perinteisiin verifiointikeinoihin lukeutuu prosessien ekvivalenssitarkistukset, joihin tutustuttiin luvussa 5.4. Kuten syntaktisessa korvautuvuudessa, myös prosessien välinen korvautuvuus perustuu tiettyjen aksioomien ja tarvittavan ekvivalenssirelaation käyttöön. Aksioomat annettiin niin kutsutun strukturaalisen kongruenssin (tai redusointisääntöjen) määritelmässä. Prosessien välinen ekvivalenssirelaation perustuu bisimulointiin, jolle annettiin useita erilaisia vaihtoehtoja. Luvun 5 lopuksi päästiin tarkastelemaan varsinaista komponenttien rajapintaprotokollien välisen yhteentoimivuuden problematiikkaa. Erityisesti huomattiin, että perinteiset verifiointimenetelmät, jotka perustuvat bisimulointiin ja mallintarkistukseen eivät välttämättä ole tässä tapauksessa parhaita vaihtoehtoja. Kuvailtujen yhteensopivuus- ja korvautuvuusrelaatioiden ja perinteisten bisimulointirelaatioiden sekä mallintarkistuksen välillä on kuitenkin selkeä yhteys.

Luvussa 6 annettiin hahmotelma komponenttien yhteensopivuus- ja yhteentoimivuustarkistuksen toteuttavan ohjelmiston rakenteesta ja toteutusperiaatteista. Ohjelmisto koostuu kolmesta komponentista, jotka ovat *TypeMatcher*, *ProcessMatcher* sekä tietämysjärjestelmä. Rajapintakuvausten syntaktisen rakenteen korvautuvuustarkistukset ovat *TypeMatcher*-komponentin vastuualuetta. Tietämysjärjestelmä taas ylläpitää ontologioihin perustuvaa käsitteistöä sekä loogista koneistoa, jolla komponenttien metodien väliset semanttiset alityypitykset voidaan tarkistaa. Käytännössä tietämysjärjestelmä on siis komponentin toiminnan semantiikan tarkistaja. Rajapintaprotokollien välistä yhteensopivuus- ja korvautuvuustarkistuksia toteuttava *ProcessMatcher* on ehdottomasti koko ohjelmiston hankalin toteutettava. Tämä johtuu siitä, että mitään suoraan käytettävissä olevia valmiita ohjelmistoja ei ole ja tyhjästä toteuttaminen voi olla todella työlästä. Järkevin vaihtoehto olisikin ottaa jokin verifiointiohjelmiston toteutus ja laajentaa tai muuttaa sitä tarvittavin osin. Luvun 6 lopussa annettiin hieman osviittaa siitä, minkälaisia ominaisuuksia korvautuvuusrelaatiolla tulisi olla. Näiden vaatimusten lähempi tarkastelu ei ikävä kyllä tämän opinnäytetyön sivumäärän puitteissa ollut mahdollista.

Tässä opinnäytetyössä selvitettiin, kuinka ja millaisia menetelmiä käyttäen ohjelmistokomponenttien välinen yhteentoimivuus voidaan tarkistaa. Yhteentoimivuuden tarkistukselle on käyttöä ohjelmiston suunnittelu- ja toteutusvaiheessa sekä myös ajonaikaisessa ympäristössä. Suunnitteluvaiheessa voidaan tässä opinnäytetyössä kuvattuja menetelmiä käyttäen tarkistaa ohjelmistoarkkitehtuurien toimivuus jo suunnittelupöydällä. Toteutusvaiheessa näitä periaatteita voidaan käyttää esimerkiksi ohjelmistokomponenttien hakemiseen komponenttikirjastoista (tai ehkä jopa markkinoilta) tai ohjaamaan ja avustamaan ohjelmoijan toimintoja siinä vaiheessa, kun eri komponentteja integroidaan ohjelmistoksi.

Lähteet

AC93 Amadio, R. M. ja Cardelli, L., Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15,4(1993), sivut 575–631. URL `citeseer.nj.nec.com/amadio93subtyping.html`.

AG97 Allen, R. ja Garlan, D., A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6,3(1997), sivut 213–249.

AL91 Asperti, A. ja Longo, G., *Categories, Types and Structures: An Introduction to Category Theory for the working computer scientist*. Foundation of Computing Series. M.I.T Press, 1991. URL `http://www.di.ens.fr/users/longo/download.html`. Painos loppu.

BCI+97 Brookes, W., Crawley, S., Indulska, J., Kosovic, D. ja Vogel, A., Types and their management in open distributed systems. *Distri. Syst. Engng*, 4, sivut 177–190.

BCL92 Bruce, K. B., Cosmo, R. D. ja Long, G., Provable Isomorphisms of Types. *Mathematical Structures in Computer Science*, 2,2(1992), sivut 231–247. URL `citeseer.nj.nec.com/article/bruce90provable.html`.

BGL94 Bouali, A., Gnesi, S. ja Larosa, S., The integration project for the JACK environment. Teoksessa *312*, Centrum voor Wiskunde en Informatica (CWI), ISSN 0169-118X, 31 1994, sivu 28, URL `citeseer.ist.psu.edu/bouali94integration.html`.

BHR84 Brookes, S. D., Hoare, C. A. R. ja Roscoe, A. W., A theory of communicating sequential processes. *J. ACM*, 31,3(1984), sivut 560–599.

BS00 Bastide, R. ja Sy, O., Towards components that Plug AND Play, 2000. URL `citeseer.nj.nec.com/bastide00towards.html`.

BW90 Baeten, J. C. M. ja Weijland, W. P., *Process algebra*. Cambridge University Press, 1990.

CES86 Clarke, E. M., Emerson, E. A. ja Sistla, A. P., Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8,2(1986), sivut 244–263.

CFP99 Camarao, C., Figueiredo, L. ja Pimentel, E., Types in programming languages, 1999. URL `citeseer.nj.nec.com/489211.html`.

Cou90 Cousot, P., Methods and logics for proving programs. Teoksessa *Handbook of Theoretical Computer Science, Volume B*, van Leeuwen, J., toimittaja, Elsevier, 1990, sivut 841–003.

CPR04 Cosmo, R. D., Pottier, F. ja Rémy, D., Subtyping Recursive Types modulo Associative Commutative Products. URL `http://pauillac.inria.fr/~remy/publications.html`. Submitted to POPL'04, heinäkuu 2004.

CPT99 Canal, C., Pimentel, E. ja Troya, J. M., Conformance and refinement of behavior in π-calculus. *Electronic Proceedings of the 2nd International Workshop on Component-based Software Development in Computational Logic*, syyskuu 1999, URL http://www.di.unipi.it/~brogi/ResearchActivity/COCL99/proceed%ings/canal.ps.

CPT01 Canal, C., Pimentel, E. ja Troya, J. M., Compatibility and inheritance in software architectures. *Science of Computer Programming*, 41,2(2001), sivut 105–138.

CW85 Cardelli, L. ja Wegner, P., On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17,4(1985), sivut 471–522. URL citeseer.nj.nec.com/cardelli85understanding.html.

DAM04 DAML-S Coalition, *OWL-S 1.0 Release*, helmikuu 2004. URL http://www.daml.org/services/owl-s/1.0/.

DFGR93 De Nicola, R., Fantechi, A., Gnesi, S. ja Ristori, G., An action-based framework for verifying logical and behavioural properties of concurrent systems. *Computer Networks and ISDN Systems*, 25,7(1993), sivut 761–778. URL citeseer.ist.psu.edu/nicola93actionbased.html.

DJ90 Dershowitz, N. ja Jouannaud, J., Rewrite systems. Teoksessa *Handbook of Theoretical Computer Science, Volume B*, van Leeuwen, J., toimittaja, Elsevier, 1990, sivut 243–320.

EH86 Emerson, E. A. ja Halpern, J. Y., šometimesänd ňot neverřevisited: on branching versus linear time temporal logic. *Journal of the ACM*, 33,1(1986), sivut 151–178.

EM85 Ehrig, H. ja Mahr, B., *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, osa 6 sarjasta *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, New York, NY, 1985.

FCB02 Fiore, M., Cosmo, R. D. ja Balat, V., Remarks on Isomorphisms in Typed Lambda Calculi with Empty and Sum Types. *17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, Logic in Computer Science. IEEE, 2002, sivut 147–156, URL http://www.pps.jussieu.fr/~balat/publications/lics02-balat_di%cosmo_fiore-remarks_on_isomorphisms_with_empty_and_sum_type.pdf.

Fer90 Fernandez, J.-C., An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13,2–3(1990), sivut 219–236. URL http://www.inrialpes.fr/vasy/Publications/Fernandez-90.html.

FGM+98 Ferrari, G., Gnesi, S., Montanari, U., Pistore, M. ja Ristori, G., Verifying mobile processes in the HAL environment. *Computer-Aided Verification, CAV '98, 10th International Conference, Vancouver, BC, Canada, Proceedings*, Hu, A. J. ja Vardi, M. Y., toimittajat, osa 1427 sarjasta *LNCS*. Springer, 1998, URL ftp://ftp.di.unipi.it/pub/Papers/pistore/cav98.ps.gz. Tool Poster.

FH03 Friedman-Hill, E. J., *Jess, The Rule Engine for Java Platform*. Sandia National Laboratories, Livermore, CA, USA, marraskuu 2003. URL http://herzberg.ca.sandia.gov/jess/. Version 6.1.

FJ01 Frendrup, U. ja Jensen, J. N., Checking for open bisimilarity in the π-calculus. Tekninen raportti RS-01-08, BRICS, Department of Computer Science, University of Aarhus, helmikuu 2001. URL `http://www.brics.dk/RS/01/8/`.

FM90 Fernandez, J.-C. ja Mounier, L., Verifying bisimulations "on the fly". *FORTE*, 1990, sivut 95–110, URL `citeseer.nj.nec.com/fernandez90verifying.html`.

Gal86 Galil, Z., Efficient algorithms for finding maximum matching in graphs. *ACM Comput. Surv.*, 18,1(1986), sivut 23–38.

GAO95 Garlan, D., Allen, R. ja Ockerbloom, J., Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software*, osa 12 sarjasta *IEEE Software*. IEEE, IEEE Press, marraskuu 1995, sivut 17–26.

Gil01 Gil, J. Y., Subtyping arithmetical types. *ACM SIGPLAN Notices*, 36,3(2001), sivut 276–289.

Gog91 Goguen, J. A., A categorical manifesto. *Mathematical Structures in Computer Science*, 1,1(1991), sivut 49–67. URL `citeseer.nj.nec.com/goguen91categorical.html`.

Hen88 Hennessy, M., *Algebraic theory of processes*. MIT Press, 1988.

Hen89 Hennessy, M., A proof system for communicating processes with value-passing. *Foundations of Software Technology and Theoretical Computer Science*, Berlin - Heidelberg - New York, joulukuu 1989, Springer, sivut 325–339.

HK73 Hopcroft, J. E. ja Karp, R. M., An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2,4(1973), sivut 225–231.

HK02 Harel, D. ja Kupferman, O., On object systems and behavioral inheritance. *IEEE Transactions on Software Engineering*, 28,9(2002), sivut 889–903.

HL95 Hennessy, M. ja Lin, H., Symbolic bisimulations. *Theoretical Computer Science*, 138,2(1995), sivut 353–389.

HM85 Hennessey, M. ja Milner, R., Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32,1(1985), sivut 137–161.

HMU01 Hopcroft, J. E., Motwani, R. ja Ullman, J. D., *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2001.

JEK$^+$90 J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill ja L.J. Hwang, Symbolic Model Checking: 10^{20} States and Beyond. *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, Washington, D.C., 1990, IEEE Computer Society Press, sivut 1–33, URL `citeseer.nj.nec.com/burch90symbolic.html`.

JPZ02 Jha, S., Palsberg, J. ja Zhao, T., Efficient Type Matching. *Lecture Notes in Computer Science*, 2303, sivut 187–206.

Kon93 Konstantas, D., Object-oriented interoperability. *European Conference on Object-Oriented Programming*, Nierstrasz, O., toimittaja, osa 707 sarjasta *Lecture Notes in Computer Science*, Kaiserslautern, Germany, heinäkuu 1993, Springer-Verlag Heidelberg, sivut 80–102.

KPS93 Kozen, D., Palsberg, J. ja Schwartzbach, M. I., Efficient recursive subtyping. *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 1993, sivut 419–428.

KR03a Kopena, J. ja Regli, W. C., Damljesskb: A tool for reasoning with the semantic web. *IEEE Intelligent Systems*, 18,3(2003), sivut 74–77.

KR03b Kuncak, V. ja Rinard, M., Structural Subtyping of Non-Recursive Types is Decidable. *Proceedings in 18th Annual IEEE Symposium on Logic in Computer Science*, Logic in Computer Science. IEEE, kesäkuu 2003, sivut 96–108, URL `citeseer.nj.nec.com/kuncak03structural.html`.

Leh00 Lehtinen, M., Matematiikkalehti Solmu - Matematiikan historia, syyskuu 2000. URL `http://solmu.math.helsinki.fi/2000/mathist/`. Luku 14, Matemaattinen logiikka ja joukko-oppi.

LVM95 Luckham, D. C., Vera, J. ja Meldal, S., Three Concepts of System Architecture. Tekninen raportti CSL-TR-95-674, Stanford University, 1995. URL `citeseer.nj.nec.com/luckham95three.html`.

LW94 Liskov, B. H. ja Wing, J. M., A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16,6(1994), sivut 1811–1841.

Mae02 Maedche, A. D., *Ontology Learning for the Semantic Web*. Kluwer Academic Publishers, 2002.

MB97 Monroy-Borja, R., *Planning Proofs of Correctness of CCS Systems*. Väitöskirja, University of Edinburgh, 1997. URL `http://citeseer.nj.nec.com/monroy-borja97planning.html`.

Mil82 Milner, R., *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., 1982.

Mil89a Milner, R., *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, New York, 1989.

Mil89b Milner, R., *Communication and Concurrency*. Prentice Hall, New York, 1989.

MN97 Meijler, T. D. ja Nierstrasz, O., Beyond Objects: Components. Teoksessa *Cooperative Information Systems: Current Trends and Directions*, Papazoglou, M. P. ja Schlageter, G., toimittajat, Academic Press, 1997, sivut 49–78, URL `citeseer.nj.nec.com/meijler98beyond.html`.

MPW89 Milner, R., Parrow, J. ja Walker, D., A Calculus of Mobile Processes, Parts I and II. Tekninen raportti, University of Edinburgh, 1989. URL `citeseer.nj.nec.com/milner89calculus.html`.

MRR03 Moisan, S., Ressouche, A. ja Rigault, J.-P., Behavioral substitutability in component frameworks: a formal approach. Tekninen raportti 03-11, Department of Computer Science, Iowa State University, 2003. URL `http://www.cs.iastate.edu/SAVCBS/`. Proceedings of SAVCVS 2003 Workshop at ESEC/FSE 2003.

NB02 Nardi, D. ja Brachman, R. *Description Logic Handbook*, luku An Introduction to Description Logics, sivut 5–44. Cambridge University Press, 2002.

Nie95 Nierstrasz, O., Regular types for active objects. Teoksessa *Object-Oriented Software Composition*, Nierstrasz, O. ja Tsichritzis, D., toimittajat, Prentice-Hall, 1995, sivut 99–121, URL `citeseer.nj.nec.com/nierstrasz93regular.html`.

OHE97 Orfali, R., Harkey, D. ja Edwards, J., *Instant CORBA*. John Wiley & Sons, Inc., 1997.

Par81 Park, D., Concurrency and automata on infinite sequences. Teoksessa *Theoretical Computer Science, 5th GI-Conf.*, LNCS 104, Springer-Verlag, Karlsruhe, maaliskuu 1981, sivut 167–183.

Par01 Parrow, J., An introduction to the pi-calculus. *Handbook of Process Algebra*, Bergstra, J., Ponse, A. ja Smolka, S., toimittajat. Elsevier Science, 2001, sivut 479–543, URL `http://www.it.kth.se/~joachim/intro.ps`.

Pet77 Peterson, J. L., Petri nets. *ACM Comput. Surv.*, 9,3(1977), sivut 223–252.

Pie02 Pierce, B. C., *Types and Programming Languages*. The MIT Press, Cambridge, MA, 2002.

PS96 Pistore, M. ja Sangiorgi, D., A Partition Refinement Algorithm for the pi-Calculus (Extended Abstract). *Computer Aided Verification, 8th International Conference, CAV'96, New Brunswick, NJ, USA, July 31 - August 3, 1996*, osa 1102 sarjasta *Lecture Notes in Computer Science*. Springer-Verlag, 1996, sivut 38–49.

PT87 Paige, R. ja Tarjan, R. E., Three partition refinement algorithms. *SIAM Journal on Computing*, 16,6(1987), sivut 973–989.

PZ00 Palsberg, J. ja Zhao, T., Efficient and flexible matching of recursive types. *Logic in Computer Science*, 2000, sivut 388–398, URL `http://citeseer.nj.nec.com/article/palsberg00efficient.html`.

QMT+02 Quirchmayr, G., Milosevic, Z., Tagg, R., Cole, J. ja Kulharni, S., Establishment of virtual enterprise contracts. *Database and Expert Systems Applications : 13th International Conference, DEXA 2002*, Cicchetti, R., Hameurlain, A. ja Traunmuller, R., toimittajat, osa 2453 sarjasta *Lecture Notes in Computer Science*. Springer-Verlag Heiderberg, syyskuu 2002, sivut 236–248.

San93 Sangiorgi, D., A theory of bisimulation for the pi-calculus. *International Conference on Concurrency Theory*, 1993, sivut 127–142, URL `citeseer.nj.nec.com/sangiorgi93theory.html`.

Shi93 Shiple, T. R., Survey of equivalences for transition systems, 1993. URL `citeseer.nj.nec.com/shiple93survey.html`.

Sid01 Siddiqui, B., Deploying web services with wsdl: Part 1, IBM Developerworks tutoriaali, marraskuu 2001. URL `http://www-106.ibm.com/developerworks/webservices/library/ws-%intwsdl/`.

SS03 Shiraree, N. ja Senivongse, T., Discovering web services using behavioural constraints and ontology. *Proceedings of Distributed Applications and Interoperable Systems, DAIS 2003*, osa 2893 sarjasta *Lecture Notes in Computer Science*, marraskuu 2003, sivut 248–259.

SW01 Sangiorgi, D. ja Walker, D. *The Pi-Calculus — A Theory of Mobile Processes*, luku 2. Cambridge University Press, 2001.

UG96 Uschold, M. ja Grüninger, M., Ontologies: principles, methods, and applications. *Knowledge Engineering Review*, 11,2(1996), sivut 93–155. URL `citeseer.nj.nec.com/uschold96ontologie.html`.

Vic94 Victor, B., *A Verification Tool for the Polyadic π-Calculus*. Licentiate thesis, Department of Computer Systems, Uppsala University, Sweden, May 1994. URL `http://user.it.uu.se/~victor/tr/docs-tr-94-50.ps.gz`. Available as report DoCS 94/50.

vW96 van Gabbeek, R. J. ja Weijland, W. P., Branching time and abstraction in bisimulation semantics. *J. ACM*, 43,3(1996), sivut 555–600.

W3C99a W3C, *Namespaces in XML*, tammikuu 1999. URL `http://www.w3.org/TR/REC-xml-names/`. World Wide Web Consortium 14-January-1999.

W3C99b W3C, *XSL Transformations*, marraskuu 1999. URL `http://www.w3.org/TR/xslt`. W3C Recommendation 16 November 1999.

W3C01a W3C, *Web Services Description Language (WSDL) 1.1*, maaliskuu 2001. URL `http://www.w3.org/TR/wsdl`. W3C Note 15 March 2001.

W3C01b W3C, *XML Schema Part 2: Datatypes*, toukokuu 2001. URL `http://www.w3.org/TR/xmlschema-2`. W3C Recommendation 02 May 2001.

W3C04a W3C, *OWL Web Ontology Language Guide*, helmikuu 2004. URL `http://www.w3.org/TR/owl-guide/`. W3C Recommendation 10 February 2004.

W3C04b W3C, *RDF Primer*, helmikuu 2004. URL `http://www.w3.org/TR/rdf-primer/`. W3C Recommendation 10 February 2004.

W3C04c W3C, *RDF Vocabulary Description Language 1.0: RDF Schema*, helmikuu 2004. URL `http://www.w3.org/TR/rdf-schema/`. W3C Recommendation 10 February 2004.

Weg96 Wegner, P., Interoperability. *ACM Computing Surveys (CSUR)*, 28,1(1996), sivut 285–287.

Weh03 Wehrheim, H., Behavioral subtyping relations for active objects. *Formal Methods in System Design*, 23,2(2003), sivut 143–170.

ZGC03 Zibin, Y., Gil, J. Y. ja Considine, J., Efficient algorithms for isomorphisms of simple types. *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 2003, sivut 160–171.

ZW95 Zaremski, A. M. ja Wing, J. M., Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 4,2(1995), sivut 146–170.

ZW97 Zaremski, A. M. ja Wing, J. M., Specification matching of software components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6,4(1997), sivut 333–369.

Liite A. OrderedContainer–tietotyyppi

OrderedContainer–tietotyypin ACT-ONE-tyylinen määritelmä

OrderedContainer =

sorts:	OrderedContainer	
	Bool	
	Int	
	TotalOrder	
opns:	$C \in OrderedContainer$	$, E \in TotalOrder$
	empty:	$\rightarrow$ C
	insert:	E,C $\rightarrow$ C
	delete:	E,C $\rightarrow$ C
	first:	C $\rightarrow$ E
	last:	C $\rightarrow$ E
	max:	C $\rightarrow$ E
	butFirst:	C $\rightarrow$ C
	butLast:	C $\rightarrow$ C
	butMax:	C $\rightarrow$ C
	isEmpty:	C $\rightarrow$ Bool
	isIn	E,C $\rightarrow$ Bool
	size	C $\rightarrow$ Int
	count	E,C $\rightarrow$ Int
eqns:	$\forall$ e,e1:E,c:C	
	last(insert(e,c))	== e
	butLast(insert(e,c))	== c
	first(empty)	== empty
	first(insert(e,empty))	== e
	first(insert(e,c))	== first(c)

Liite B. ServiceInterface-ontologian OWL-kuvaus

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE uridef[
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns">
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema">
  <!ENTITY owl "http://www.w3.org/2002/07/owl">
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
  <!ENTITY interface "http://www.cs.helsinki.fi/u/thruokol/ServiceInterface.owl">
  <!ENTITY DEFAULT "http://www.cs.helsinki.fi/u/thruokol/ServiceInterface.owl">
]>

<rdf:RDF
  xmlns:rdf= "&rdf;#"
  xmlns:rdfs= "&rdfs;#"
  xmlns:owl = "&owl;#"
  xmlns:xsd=  "&xsd;#"
  xmlns= "&DEFAULT;#">

  <owl:Ontology rdf:about="">
    <owl:versionInfo>
      $Id: ServiceInterface.owl,v 1.9 2004/03/11 14:19:32 thruokol Exp $
    </owl:versionInfo>
    <rdfs:comment>
      ServiceInterface-ontologia Web-palveluiden kuvaamiseksi
      Toni Ruokolainen (thruokol@cs.helsinki.fi)
    </rdfs:comment>
    <owl:imports rdf:resource="&owl;" />
  </owl:Ontology>

<!-- Luokkarakenteen maarittely -->
  <owl:Class rdf:ID="ServiceInterface">
    <rdfs:label>ServiceInterface</rdfs:label>
    <rdfs:comment>Ontologian ylakasite</rdfs:comment>
  </owl:Class>

  <owl:Class rdf:ID="Method">
    <rdfs:label>Method</rdfs:label>
    <rdfs:comment>Method-luokka maarittelee WSDL-operaation semantiikan</rdfs:comment>
  </owl:Class>

  <owl:Class rdf:ID="Message">
    <rdfs:label>Message</rdfs:label>
    <rdfs:comment>Message-luokka maarittelee WSDL-viestin semantiikan</rdfs:comment>
    <rdfs:subClassOf>
      <owl:Restriction owl:minCardinality="1">
        <owl:onProperty rdf:resource="#messageType" />
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

  <owl:Class rdf:ID="Input">
    <rdfs:subClassOf rdf:resource="&interface;#Message" />
    <rdfs:label>Input</rdfs:label>
    <rdfs:comment>Input-luokka maarittelee WSDL-viestin semantiikan</rdfs:comment>
  </owl:Class>

  <owl:Class rdf:ID="Output">
    <rdfs:subClassOf rdf:resource="&interface;#Message" />
    <rdfs:label>Input</rdfs:label>
    <rdfs:comment>Input-luokka maarittelee WSDL-viestin semantiikan</rdfs:comment>
  </owl:Class>

  <owl:Class rdf:ID="InterfaceProtocol">
    <rdfs:label>InterfaceProtocol</rdfs:label>
    <rdfs:comment>InterfaceProtocol-luokka maarittelee rajapintaprotokollan
    WSDL-kuvauksen portType-elentille</rdfs:comment>
  </owl:Class>
```

```
  <owl:Class rdf:ID="Condition">
    <rdfs:label>Condition</rdfs:label>
    <rdfs:comment>Condition-luokka maarittelee semanttisen lausekkeen
    joukkona atomeja</rdfs:comment>
  </owl:Class>

  <owl:Class rdf:ID="PreCondition">
    <rdfs:subClassOf rdf:resource="&interface;#Condition" />
    <rdfs:label>PreCondition</rdfs:label>
    <rdfs:comment>Esiehto on ehtoluokan aliluokka</rdfs:comment>
  </owl:Class>

  <owl:Class rdf:ID="PostCondition">
    <rdfs:subClassOf rdf:resource="&interface;#Condition" />
    <rdfs:label>PostCondition</rdfs:label>
    <rdfs:comment>Jalkiehto on ehtoluokan aliluokka</rdfs:comment>
  </owl:Class>

  <owl:Class rdf:ID="Atom">
    <rdfs:label>Atom</rdfs:label>
    <rdfs:comment>Ehtolausekkeen termi</rdfs:comment>
  </owl:Class>

  <owl:Class rdf:ID="Process">
    <rdfs:label>Process</rdfs:label>
    <rdfs:comment>Rajapintalausekkeen kuvaus</rdfs:comment>
  </owl:Class>

<!-- Luokkien valiset suhteet -->
<!-- ServiceInterface- ja Method-luokkien suhteet -->
  <owl:ObjectProperty rdf:ID="hasMethod">
    <rdfs:domain rdf:resource="&interface;#ServiceInterface" />
    <rdfs:range rdf:resource="&interface;#Method" />
    <owl:inverseOf rdf:resource="&interface;#isMethodOf" />
  </owl:ObjectProperty>

  <owl:Class rdf:about="&interface;#ServiceInterface">
    <rdfs:comment>
    </rdfs:comment>
    <rdfs:subClassOf>
      <owl:Restriction owl:minCardinality="1">
      <owl:onProperty rdf:resource="&interface;#hasMethod"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

  <owl:ObjectProperty rdf:ID="isMethodOf">
    <rdfs:domain rdf:resource="&interface;#Method" />
    <rdfs:range rdf:resource="&interface;#ServiceInterface" />
    <owl:inverseOf rdf:resource="&interface;#hasMethod" />
  </owl:ObjectProperty>
<!--
  <owl:Class rdf:about="&interface;#Method">
    <rdfs:comment>
    </rdfs:comment>
    <rdfs:subClassOf>
      <owl:Restriction owl:minCardinality="1">
      <owl:onProperty rdf:resource="&interface;#isMethodOf"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
-->
  <owl:ObjectProperty rdf:ID="hasInput">
    <rdfs:domain rdf:resource="&interface;#Method" />
    <rdfs:range rdf:resource="&interface;#Input" />
    <owl:inverseOf rdf:resource="&interface;#isInputOf" />
  </owl:ObjectProperty>
```

```
  <owl:ObjectProperty rdf:ID="isInputOf">
    <rdfs:domain rdf:resource="&interface;#Input" />
    <rdfs:range rdf:resource="&interface;#Method" />
    <owl:inverseOf rdf:resource="&interface;#hasInput" />
  </owl:ObjectProperty>

  <owl:Class rdf:about="&interface;#Method">
    <rdfs:comment>Jokaisella metodilla on korkeintaan 1 input-viesti
    </rdfs:comment>
    <rdfs:subClassOf>
      <owl:Restriction owl:maxCardinality="1">
      <owl:onProperty rdf:resource="&interface;#hasInput"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

<!--
  <owl:Class rdf:about="&interface;#Input">
    <rdfs:comment>Jokainen input-viesti liittyy ainakin yhteen metodiin
    </rdfs:comment>
    <rdfs:subClassOf>
      <owl:Restriction owl:minCardinality="1">
      <owl:onProperty rdf:resource="&interface;#isInputOf"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
-->

  <owl:ObjectProperty rdf:ID="hasOutput">
    <rdfs:domain rdf:resource="&interface;#Method" />
    <rdfs:range rdf:resource="&interface;#Output" />
    <owl:inverseOf rdf:resource="&interface;#isOutputOf" />
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="isOutputOf">
    <rdfs:domain rdf:resource="&interface;#Output" />
    <rdfs:range rdf:resource="&interface;#Method" />
    <owl:inverseOf rdf:resource="&interface;#hasOutput" />
  </owl:ObjectProperty>

  <owl:Class rdf:about="&interface;#Method">
    <rdfs:comment>Jokaisella metodilla on korkeintaan 1 input-viesti
    </rdfs:comment>
    <rdfs:subClassOf>
      <owl:Restriction owl:maxCardinality="1">
      <owl:onProperty rdf:resource="&interface;#hasOutput"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

<!--
  <owl:Class rdf:about="&interface;#Input">
    <rdfs:comment>Jokainen input-viesti liittyy ainakin yhteen metodiin
    </rdfs:comment>
    <rdfs:subClassOf>
      <owl:Restriction owl:minCardinality="1">
      <owl:onProperty rdf:resource="&interface;#isOutputOf"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
-->

  <!-- ServiceInterface- ja InterfaceProtocol-luokkien suhteet -->
  <owl:ObjectProperty rdf:ID="hasBehaviour">
    <rdfs:domain rdf:resource="&interface;#ServiceInterface" />
    <rdfs:range rdf:resource="&interface;#InterfaceProtocol" />
    <owl:inverseOf rdf:resource="&interface;#isBehaviourOf" />
```

```
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="isBehaviourOf">
    <rdfs:domain rdf:resource="&interface;#InterfaceProtocol" />
    <rdfs:range rdf:resource="&interface;#ServiceInterface" />
    <owl:inverseOf rdf:resource="&interface;#hasBehaviour" />
  </owl:ObjectProperty>

  <owl:Class rdf:about="&interface;#ServiceInterface">
    <rdfs:comment>
    </rdfs:comment>
    <rdfs:subClassOf>
      <owl:Restriction owl:cardinality="1">
      <owl:onProperty rdf:resource="&interface;#hasBehaviour"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

<!--
  <owl:Class rdf:about="&interface;#InterfaceProtocol">
    <rdfs:comment>
    </rdfs:comment>
    <rdfs:subClassOf>
      <owl:Restriction owl:cardinality="1">
      <owl:onProperty rdf:resource="&interface;#isBehaviourOf"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
-->

  <!-- InterfaceProtocol- ja Process-luokkien suhteet -->
  <owl:ObjectProperty rdf:ID="definedByProcess">
    <rdfs:domain rdf:resource="&interface;#InterfaceProtocol" />
    <rdfs:range rdf:resource="&interface;#Process" />
    <owl:inverseOf rdf:resource="&interface;#definesBehaviourOf" />
  </owl:ObjectProperty>

  <owl:Class rdf:about="&interface;#InterfaceProtocol">
    <rdfs:comment>
    </rdfs:comment>
    <rdfs:subClassOf>
      <owl:Restriction owl:cardinality="1">
      <owl:onProperty rdf:resource="&interface;#definedByProcess"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

  <owl:ObjectProperty rdf:ID="definesBehaviourOf">
    <rdfs:domain rdf:resource="&interface;#Process" />
    <rdfs:range rdf:resource="&interface;#InterfaceProtocol" />
    <owl:inverseOf rdf:resource="&interface;#definedByProcess" />
  </owl:ObjectProperty>

<!--
  <owl:Class rdf:about="&interface;#Process">
    <rdfs:comment>
    </rdfs:comment>
    <rdfs:subClassOf>
      <owl:Restriction owl:maxCardinality="1">
      <owl:onProperty rdf:resource="&interface;#definesBehaviourOf"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
-->

  <!-- Metodien etu- ja jälkiehdot -->
  <owl:ObjectProperty rdf:ID="hasPrecondition">
    <rdfs:domain rdf:resource="&interface;#Method" />
```

```
    <rdfs:range rdf:resource="&interface;#PreCondition" />
    <owl:inverseOf rdf:resource="&interface;#isPrecondOf" />
  </owl:ObjectProperty>

  <owl:Class rdf:about="&interface;#Method">
    <rdfs:comment>
    </rdfs:comment>
    <rdfs:subClassOf>
      <owl:Restriction owl:maxCardinality="1">
      <owl:onProperty rdf:resource="&interface;#hasPrecondition"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

  <owl:ObjectProperty rdf:ID="isPrecondOf">
    <rdfs:domain rdf:resource="&interface;#PreCondition" />
    <rdfs:range rdf:resource="&interface;#Method" />
    <owl:inverseOf rdf:resource="&interface;#hasPrecondition" />
  </owl:ObjectProperty>
<!--
  <owl:Class rdf:about="&interface;#PreCondition">
    <rdfs:comment>
    </rdfs:comment>
    <rdfs:subClassOf>
      <owl:Restriction owl:minCardinality="1">
      <owl:onProperty rdf:resource="&interface;#isPrecondOf"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
-->
  <owl:ObjectProperty rdf:ID="hasPostcondition">
    <rdfs:domain rdf:resource="&interface;#Method" />
    <rdfs:range rdf:resource="&interface;#PostCondition" />
    <owl:inverseOf rdf:resource="&interface;#isPostcondOf" />
  </owl:ObjectProperty>

  <owl:Class rdf:about="&interface;#Method">
    <rdfs:comment>
    </rdfs:comment>
    <rdfs:subClassOf>
      <owl:Restriction owl:maxCardinality="1">
      <owl:onProperty rdf:resource="&interface;#hasPostcondition" />
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>


  <owl:ObjectProperty rdf:ID="isPostcondOf">
    <rdfs:domain rdf:resource="&interface;#PostCondition" />
    <rdfs:range rdf:resource="&interface;#Method" />
    <owl:inverseOf rdf:resource="&interface;#hasPostcondition" />
  </owl:ObjectProperty>
<!--
  <owl:Class rdf:about="&interface;#PostCondition">
    <rdfs:comment>
    </rdfs:comment>
    <rdfs:subClassOf>
      <owl:Restriction owl:minCardinality="1">
      <owl:onProperty rdf:resource="&interface;#isPostcondOf"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
-->

  <!-- Condition ja Atom-luokat -->
  <owl:ObjectProperty rdf:ID="isConjunctionOf">
    <rdfs:domain rdf:resource="&interface;#Condition" />
    <rdfs:range rdf:resource="&interface;#Atom" />
```

```
    <owl:inverseOf rdf:resource="&interface;#isAtomOf" />
  </owl:ObjectProperty>

  <owl:Class rdf:about="&interface;#Condition">
    <rdfs:comment>
    </rdfs:comment>
    <rdfs:subClassOf>
      <owl:Restriction owl:minCardinality="1">
      <owl:onProperty rdf:resource="&interface;#isConjunctionOf"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>


  <owl:ObjectProperty rdf:ID="isAtomOf">
    <rdfs:domain rdf:resource="&interface;#Atom" />
    <rdfs:range rdf:resource="&interface;#Condition" />
    <owl:inverseOf rdf:resource="&interface;#isConjunctionOf" />
  </owl:ObjectProperty>
<!--
  <owl:Class rdf:about="&interface;#Atom">
    <rdfs:comment>
    </rdfs:comment>
    <rdfs:subClassOf>
      <owl:Restriction owl:minCardinality="1">
      <owl:onProperty rdf:resource="&interface;#isAtomOf"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
-->

  <!-- Datamäärittelyt -->
  <owl:DatatypeProperty rdf:ID="portType">
    <rdfs:domain rdf:resource="&interface;#ServiceInterface" />
    <rdfs:range rdf:resource="&xsd;#anyURI" />
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="operation">
    <rdfs:domain rdf:resource="&interface;#Method" />
    <rdfs:range rdf:resource="&xsd;#anyURI" />
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="agent">
    <rdfs:domain rdf:resource="&interface;#Process" />
    <rdfs:range rdf:resource="&xsd;#string" />
  </owl:DatatypeProperty>

  <owl:Class rdf:about="&interface;#ServiceInterface">
    <rdfs:subClassOf>
      <owl:Restriction owl:cardinality="1" >
<owl:onProperty rdf:resource="&interface;#portType" />
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

  <owl:Class rdf:about="&interface;#Method">
    <rdfs:subClassOf>
      <owl:Restriction owl:cardinality="1" >
<owl:onProperty rdf:resource="&interface;#operation" />
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

  <rdf:Property rdf:ID="messageType">
    <rdfs:domain rdf:resource="#Message" />
    <rdfs:comment>Voi viitata joko toiseen luokkaan tai XML-skeemaan.
    Siksi range:a ei maaritella.
    </rdfs:comment>
```

```
  </rdf:Property>
</rdf:RDF>
```

Liite C. MyBankingService-ontologian OWL-kuvaus

```
<?xml version='1.0' encoding='iso-8859-1' ?>
<!DOCTYPE uridef[
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns">
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema">
  <!ENTITY owl "http://www.w3.org/2002/07/owl">
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
  <!ENTITY interop "http://www.cs.helsinki.fi/u/thruokol/ServiceInterface.owl">
  <!ENTITY mybank "http://www.cs.helsinki.fi/u/thruokol/MyBankingService.owl">
  <!ENTITY DEFAULT "http://www.cs.helsinki.fi/u/thruokol/MyBankingService.owl">
]>

<rdf:RDF
  xmlns:rdf= "&rdf;#"
  xmlns:rdfs= "&rdfs;#"
  xmlns:owl = "&owl;#"
  xmlns:xsd=  "&xsd;#"
  xmlns:interop = "&interop;#"
  xmlns:mybank = "&mybank;#"
  xmlns= "&DEFAULT;#">

  <owl:Ontology rdf:about="">
    <owl:versionInfo>
      $Id: MyBankingService.owl,v 1.2 2004/03/11 14:19:32 thruokol Exp $
    </owl:versionInfo>
    <rdfs:comment>
      MyBankingService-ontologia. Pro gradu -työn esimerkkiontologia.
      Toni Ruokolainen (thruokol@cs.helsinki.fi)
    </rdfs:comment>
    <owl:imports rdf:resource="&owl;" />
    <owl:imports rdf:resource="&interop;" />
  </owl:Ontology>

<!-- Käytettävien tietotyyppien määrittelyt -->

<owl:Class rdf:ID="CreditCardType">
  <owl:oneOf rdf:parseType="Collection">
    <CreditCardType rdf:ID="MasterCard"/>
    <CreditCardType rdf:ID="VISA"/>
    <CreditCardType rdf:ID="AmericanExpress"/>
    <CreditCardType rdf:ID="DiscoverCard"/>
   </owl:oneOf>
</owl:Class>

<owl:Class rdf:ID="AckMessage">
 <owl:oneOf rdf:parseType="Collection">
  <AckMessage rdf:ID="Ok" />
  <AckMessage rdf:ID="Failed" />
 </owl:oneOf>
</owl:Class>

<owl:Class rdf:ID="SignInData" />

<owl:DatatypeProperty rdf:ID="acctName">
  <rdfs:domain rdf:resource="#SignInData"/>
  <rdfs:range rdf:resource="&xsd;#string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="password">
  <rdfs:domain rdf:resource="#SignInData"/>
  <rdfs:range rdf:resource="&xsd;#string"/>
</owl:DatatypeProperty>

<!-- moneyTransfer-tietotyypin määritelmä -->
<!-- Sisältää merkkijonon ja päivämäärän -->
<owl:Class rdf:ID="moneyTransfer" />
```

```
<owl:DatatypeProperty rdf:ID="destinationAccount">
  <rdfs:domain rdf:resource="#moneyTransfer" />
  <rdfs:range rdf:resource="&xsd;#string" />
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="amount">
  <rdfs:domain rdf:resource="#moneyTransfer" />
  <rdfs:range rdf:resource="&xsd;#decimal" />
</owl:DatatypeProperty>

<!-- receipt-tietotyypin määritelmä -->
<owl:Class rdf:ID="receipt" />

<owl:DatatypeProperty rdf:ID="date">
  <rdfs:domain rdf:resource="#receipt" />
  <rdfs:range rdf:resource="&xsd;#dateTime" />
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="from">
  <rdfs:domain rdf:resource="#receipt" />
  <rdfs:range rdf:resource="&xsd;#string" />
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="to">
  <rdfs:domain rdf:resource="#receipt" />
  <rdfs:range rdf:resource="&xsd;#string" />
</owl:DatatypeProperty>

<!-- ***************************** -->
<!-- Palvelukomponentin määrittely -->
<!-- ***************************** -->
<interop:ServiceInterface rdf:ID="MyServiceInterface" >
 <interop:hasBehaviour rdf:resource="#myServiceBehaviour" />
 <interop:hasMethod rdf:resource="#transferMoney" />

 <!-- URI WSDL-kuvauksen portType-elementtiin -->
 <interop:portType>http://www.foobar.com/myBankingInterface.wsdl#somePortTypeName</interop:portType>
</interop:ServiceInterface>

<!-- ******************************* -->
<!-- Rajapintaprotokollan määrittely -->
<!-- ******************************* -->
<interop:InterfaceProtocol rdf:ID="myServiceBehaviour">
 <interop:definedByProcess rdf:resource="#myServiceProcess" />
</interop:InterfaceProtocol>

<interop:Process rdf:ID="myServiceProcess" >
 <interop:agent>foobar</interop:agent>
</interop:Process>

<!-- ******************* -->
<!-- Metodien määrittelyt -->
<!-- ******************* -->
<interop:Method rdf:ID="transferMoney">

 <interop:hasInput>
  <interop:Input>
   <interop:messageType rdf:resource="#moneyTransfer" />
  </interop:Input>
 </interop:hasInput>

 <interop:hasOutput>
  <interop:Output>
   <interop:messageType rdf:resource="#receipt" />
  </interop:Output>
 </interop:hasOutput>

 <!-- URI WSDL-kuvauksen operation-elementtiin -->
```

```
 <interop:operation>http://www.foobar.com/myBankingInterface.wsdl#someOperationName</interop:operation>
</interop:Method>
</rdf:RDF>
```

Liite D. PiProcess-ontologian OWL-kuvaus

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE uridef[
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns">
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema">
  <!ENTITY owl "http://www.w3.org/2002/07/owl">
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
  <!ENTITY interop "http://www.cs.helsinki.fi/u/thruokol/ServiceInterface.owl">
  <!ENTITY pi "http://www.cs.helsinki.fi/u/thruokol/PiProcess.owl">
  <!ENTITY DEFAULT "http://www.cs.helsinki.fi/u/thruokol/PiProcess.owl">
]>

<rdf:RDF
  xmlns:rdf  = "&rdf;#"
  xmlns:rdfs = "&rdfs;#"
  xmlns:owl  = "&owl;#"
  xmlns:xsd  = "&xsd;#"
  xmlns:interop = "&interop;#"
  xmlns:pi   = "&pi;#"
  xmlns      = "&DEFAULT;#">

  <owl:Ontology rdf:about="">
    <owl:versionInfo>
      $Id: PiProcess.owl,v 1.4 2004/03/12 12:14:13 thruokol Exp $
    </owl:versionInfo>
    <rdfs:comment>
      PiProcess-ontologia pi-kalkyyliprosessien kuvaamiseksi
      Web-palveluiden rajapintaprotokollina.
      Toni Ruokolainen (thruokol@cs.helsinki.fi)
    </rdfs:comment>
    <owl:imports rdf:resource="&owl;" />
    <owl:imports rdf:resource="&interop;" />
  </owl:Ontology>

  <!-- Pi-kalkyylin prefiksit -->
  <owl:Class rdf:ID="Prefix" />

  <!-- tau-siirtyma -->
  <owl:Class rdf:ID="tau" >
    <rdfs:subClassOf rdf:resource="#Prefix" />
  </owl:Class>

  <!-- Input- ja output-prefiksien kantaluokka -->
  <owl:Class rdf:ID="IOPrefix" >
    <rdfs:subClassOf rdf:resource="#Prefix" />
  </owl:Class>

  <owl:ObjectProperty rdf:ID="channel">
    <rdfs:domain rdf:resource="#IOPrefix" />
    <rdfs:range rdf:resource="#channelNames" />
  </owl:ObjectProperty>

  <owl:Class rdf:ID="outputPrefix" >
    <rdfs:subClassOf rdf:resource="#IOPrefix" />
  </owl:Class>

  <owl:Class rdf:ID="outputMessage" >
    <rdfs:subClassOf rdf:resource="#outputPrefix" />
  </owl:Class>

  <owl:Class rdf:ID="outputFunction" >
    <rdfs:subClassOf rdf:resource="#outputPrefix" />
  </owl:Class>

  <owl:Class rdf:ID="inputPrefix" >
    <rdfs:subClassOf rdf:resource="#IOPrefix" />
  </owl:Class>
```

```
  <owl:Class rdf:ID="inputMessage" >
    <rdfs:subClassOf rdf:resource="#inputPrefix" />
  </owl:Class>

  <owl:Class rdf:ID="inputFunction" >
    <rdfs:subClassOf rdf:resource="#inputPrefix" />
  </owl:Class>

  <!-- Pi-kalkyylin prosessit -->
  <owl:Class rdf:ID="PiProcess">
    <rdfs:subClassOf rdf:resource="&interop;#Process" />
    <rdfs:label>Pi-prosessin ontologia</rdfs:label>
    <rdfs:comment></rdfs:comment>
  </owl:Class>

  <!-- Tyhja prosessi -->
  <owl:Class rdf:ID="Nil" >
    <rdfs:subClassOf rdf:resource="#PiProcess" />
  </owl:Class>

  <!-- P + Q -->
  <owl:Class rdf:ID="Sum" >
    <rdfs:subClassOf rdf:resource="#PiProcess" />
  </owl:Class>

  <!-- P | Q -->
  <owl:Class rdf:ID="Par" >
    <rdfs:subClassOf rdf:resource="#PiProcess" />
  </owl:Class>

  <!-- [a = b] P -->
  <owl:Class rdf:ID="Match" >
    <rdfs:subClassOf rdf:resource="#PiProcess" />
    <rdfs:subClassOf>
      <owl:Restriction owl:cardinality="1" >
<owl:onProperty rdf:resource="#lhs" />
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction owl:cardinality="1" >
<owl:onProperty rdf:resource="#rhs" />
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

  <owl:ObjectProperty rdf:ID="lhs">
    <rdfs:domain rdf:resource="#Match" />
    <rdfs:range rdf:resource="#Names" />
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="rhs">
    <rdfs:domain rdf:resource="#Match" />
    <rdfs:range rdf:resource="#Names" />
  </owl:ObjectProperty>

  <!-- (x)P -->
  <owl:Class rdf:ID="Res" >
    <rdfs:subClassOf rdf:resource="#PiProcess" />
  </owl:Class>

  <!-- Nimet -->
  <owl:Class rdf:ID="Names" />

  <owl:ObjectProperty rdf:ID="name">
    <rdf:domain rdf:resource="#Names" />
  </owl:ObjectProperty>
```

```
  <!-- Kanavien nimet -->
  <owl:Class rdf:ID="channelNames" >
    <rdfs:subClassOf rdf:resource="#Names" />
  </owl:Class>

  <owl:ObjectProperty rdf:ID="name">
    <rdf:domain rdf:resource="#Names" />
    <rdf:range rdf:resource="#channelInstances" />
  </owl:ObjectProperty>

  <!-- Viestien ja metodien nimet -->
  <owl:Class rdf:ID="messageNames" >
    <rdfs:subClassOf rdf:resource="#Names" />
  </owl:Class>

  <owl:ObjectProperty rdf:ID="name">
    <rdf:domain rdf:resource="#Names" />
    <rdf:range rdf:resource="communicationInstances" />
    <rdfs:domain>
      <owl:Class>
        <owl:unionOf rdf:parseType="Collection">
          <owl:Class rdf:about="&interop;#Message"/>
          <owl:Class rdf:about="&interop;#Method"/>
        </owl:unionOf>
      </owl:Class>
    </rdfs:domain>

  </owl:ObjectProperty>

</rdf:RDF>
```