Komponenttien yhteentoimivuus

Toni Ruokolainen
thruokol@cc.helsinki.fi

Helsinki 15.5.2003

Väliohjelmistot kurssin 2003 essee

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Sisältö

Taulukot

1 Johdanto

Komponenttipohjaisen ohjelmistotuotannon ja komponenttipohjaisten järjestelmien avulla pyritään ratkaisemaan ongelmia, jotka ovat olleet osa ohjelmistotuotantoa alusta alkaen. Tällaisia ongelmia ovat esimerkiksi suurten ohjelmistojen kompleksisuuden hallinta ja ohjelmistojen alttius virheille. Näitä ongelmia voidaan helpottaa käyttämällä koostettavassa ohjelmistossa valmiita ja virheettömäksi todettuja komponentteja aina kuin mahdollista. Jotta ohjelmiston kehityskaari olisi mahdollisimman joustava ja mahdollistaisi ohjelmiston vaivattoman päivittämisen, tulisi sekä ohjelmiston kehitysympäristön että ajon aikaisen ympäristön tukea yhteentoimivien komponenttien yhteistyötä.

Tässä esseessä tutustutaan komponenttien yhteentoimivuuteen sekä miten yhteentoimivuus huomioidaan eräissä tämän hetkisissä väliohjelmistomalleissa- tai palveluissa. Luvussa 2 käsitellään yhteentoimivuuden käsitettä ja mitä eri aspekteja ja tasoja se pitää sisällään. Tämän jälkeen luvussa 3 tutustumme käytännön tekniikoihin, jotka lähestyvät eri tavoilla yhteentoimivuuden ongelmaa. Yhteenveto on luvussa 4.

2 Komponenttien yhteentoimivuus

Tulevaisuudessa ohjelmistot tulevat varmasti kehittymään yhä dynaamisempaan ja avoimempaan suuntaan. Ohjelmistot kehitetään valmiita komponentteja käyttäen. Komponentit voivat olla eri komponenttituottajien kehittämiä ja ne voivat sijaita verkkoyhteyden takana. Ohjelmisto voi käyttää ajonaikaista sidontaa käyttökelpoisten komponenttien etsintään ja valmiita ohjelmistoja olisi hyvä voida päivittää keskellä ohjelmiston elinkaarta ilman että siitä koituu loppukäyttäjillä ylimääräistä päänvaivaa. Tämän kehityskulun vuoksi tulee yhä tärkeämmäksi se, että tarvittavat komponentit ovat yhteentoimivia.

Perinteisissä väliohjelmistoissa komponenttien yhteentoimivuus tarkistetaan vain syntaktisella tasolla ja sielläkin yhteentoimivuuden kriteerit ovat toisaalta liian epäformaaleja ja toisaalta liian tiukkoja.

Komponenttien toiminnallisuus kuvaillaan tätä nykyä epäformaalisti ohjelmiston dokumentoinnin avulla, jonka vuoksi avoimesti toteutettujen komponenttien yhteenliittäminen on pahimmillaan olettamuksiin perustuvaa arpapeliä. Toisaalta nykyiset väliohjelmistot tukeutuvat liiaksi komponenttirajapintojen syntaktisten kuvauksien eksaktiin vertailuun. Jos esimerkiksi kaksi komponenttia ovat ekvivalentit käyttäjän tarpeiden kannalta, mutta niiden metodit ovat eri järjestyksessä, ei nykyaikainen väliohjelmisto välttämättä pysty huomioimaan tätä korvautuvuusekvivalenssia. Liian tiukka ekvivalenssivaatimus tekee komponenttien avoimen kehittämisen ja järjestelmän päivittämisen vaikeaksi.

Edellä mainituista syistä johtuen komponenttien rajapintakuvausten ekvivalenssirelaatioita olisi syytä saada joustavimmiksi. Tämän lisäksi komponenttien toimintaa

pitäisi pystyä kuvaamaan formaalilla ja yhteismitallisella tavalla.

Tässä luvussa määritellään mitä ohjelmistokomponenttien tapauksessa tarkoitetaan termeillä *yhteentoimivuus*, *yhteesopivuus* ja *korvattavuus*. Tämän lisäksi esitellään kolme eri tasoa, joilla edellä mainittuja ominaisuuksia voidaan tarkastella. Yhteentoimivuuden tasot ovat syntaktinen, semanttinen ja protokollaso.

2.1 Yhteentoimivuus

Komponenttipohjaisessa ohjelmisto koostuu keskenään *yhteentoimivista* (*interoperable*) komponenteista. Yhteentoimivuudella tarkoitetaan kahden tai useamman komponentin kykyä keskinäiseen yhteistyöhön [VHT02]. Yhteentoimivien komponenttien tulisi yhdistämisen jälkeen toimia oikeellisesti ja ennalta määritellyllä tavalla. Perinteinen ohjelmistotuotantoprosessi, jossa komponenttien toiminnallisuus kuvataan implisiittisesti dokumentoinnissa, ei takaa komponenttien yhteentoimivuutta [BS00]. Erityisesti yhteentoimivuusongelmia tavataan avoimessa ohjelmistotuotannossa ja avoimissa järjestelmissä, joissa komponentit voidaan toteuttaa eri kehittäjien toimesta ja erilaisilla työkaluilla sekä ohjelmointikielillä.

Yhteentoimivuudella on kaksi ulottuvuutta: yhteensopivuus (*compatibility*) ja korvattavuus (*substitutability*). Yhteensopivuudella tarkoitetaan sitä, että komponentit toimivat oikein, jos ne yhdistetään keskenään [VHT02]. Tämä sisältää mm. vaatimukset siitä, että komponentit ymmärtävät vaihtamiensa viestien sisällön oikein ja että komponentit kutsuvat toistensa metodeja oikeassa järjestyksessä sekä oikeellisilla parametreilla.

Komponenttien korvattavuus tarkoittaa sitä, että kaksi eri komponenttia ovat korvattavissa keskenään ilman että järjestelmän funktionaalisuus muuttuu. Keskenään korvattavien komponenttien tulee siis toteuttaa ulkoisesti sama toiminnallisuus. Komponenttien korvattavuus mahdollistaa esimerkiksi ohjelmiston päivittämisen uudella komponentilla.

Tämän hetkisissä väliohjelmistoissa, kuten CORBA, korvattavuus tarkoittaa joko rajapintojen eksaktia "yhtäsuuruutta" tai komponenttiluokkien alityypitysrelaatiota. Komponenttien korvattavuus on kuitenkin toiminnallisuuden ekvivalenssirelaatio: jos komponentit ovat korvattavissa keskenään, niin niiden pitäisi olla ekvivalenteja nimenomaan toiminnallisuudeltaan.

2.2 Yhteentoimivuuden tasot

Yhteentoimivuutta voidaan tarkastella kolmella eri tasolla, jotka määrittävät eri tasoisia ominaisuuksia komponentin rakenteesta ja toiminnallisuudesta. Komponentin rakenne ja toiminnallisuus voidaan määritellä syntaktisten ja semanttisten ominaisuuksien sekä käytettävien protokollien avulla.

2.2.1 Syntaktinen yhteentoimivuus

Syntaktisen tason yhteentoimivuudella tarkoitetaan sitä, että komponenttien rajapinnat ja niiden käyttämät tietorakenteet ovat rakenteellisesti ekvivalentteja. Perinteisesti tämä on tarkoittanut sitä, että keskenään korvattavat komponentit ovat olleet nimeämistä myöten eksakteja yksi-yhteen–kuvauksia toisistaan.

Syntaktisen rakenteen eksakti yhtäsuuruusehto on kuitenkin usein liian vahva ja rajoittava [ZW95]. Esimerkiksi komponenttia päivitettäessä uuden toiminnon (metodin) lisääminen komponenttiin voi saada aikaan sen, että uutta komponenttia ei enää tulkita ekvivalentiksi korvatun komponentin kanssa vaikka usein näin haluttaisiinkin.

Avoimessa ympäristössä voi yhtenäisten nimeämiskäytäntöjen löytäminen ja käyttäminen eri komponenttivalmistajien välillä olla hankalaa. Tehokas, joustava ja avoin (komponenttipohjainen) ohjelmistokehitys helpottuu, jos komponenttien yhteentoimivuus ei riipu esimerkiksi metodien tai muuttujien nimeämistavoista tai siitä, missä järjestyksessä ne kussakin komponentissa esitellään. Yllämainituiden syiden takia komponenttirajapintojen yhteentoimivuus pitäisikin pystyä tarkistamaan niiden syntaktisten rakenteiden perusteella, ei nimien tai tyyppien esiintymisjärjestyksen.

Syntaktisten rakenteiden vertailusta saadaan joustava, jos komponentti tulkitaan sisältämiensä tietotyyppien monijoukoksi. Tällöin ekvivalenssitarkistus redusoituu isomorfismin löytämiseen kahden monijoukon välillä haluttujen sääntöjen vallitessa. Esimerkiksi seuraavat joukot $(int, long)$ ja $(long, int)$ tulkittaisiin ekvivalenteiksi, jos kommutatiivisuus sallitaan. Kommutatiivisuuden lisäksi käyttökelpoisia aksioomia ovat joukkojen assosiatiivisuus ja ns. funktiotyyppien distributiivisuus [ZGC03].

Syntaktisiin rakenteisiin perustuvat ekvivalenssitarkistukset ovat huomattavasti joustavampia kuin nykyisissä väliohjelmistoissa käytetyt ekvivalenssitarktistukset. Taulukossa 1 on kaksi komponenttia, jotka kuvaavat saman toiminnollisuuden tarjoavia pankkitilipalveluita. Nykyisissä väliohjelmistoissa *Account* ja *Account*2 miellettäisiin eri komponenteiksi, koska esimerkiksi jotkin metodeista ovat hieman eri nimisiä ja ottavat parametrinsa vastaan eri järjestyksessä. Jos käytössä olisi joustavampi, pelkästään syntaktiseen rakenteeseen perustuva ekvivalenssitarkistus, voisi väliohjelmisto tarjota asiakkaan käyttöön kumpaa tahansa näistä komponenteista.

Eräät nykyisistä väliohjelmistoista (mm. Microsoftin DCOM ja CORBA:n komponenttilaajennos CCM) mahdollistavat usean eri rajapinnan määrittelemisen yhdelle komponentille. Monirajapintaisuudella voidaan joissain määrin kiertää komponenttien päivitysten tuottamia yhteentoimivuusongelmia, mutta pidemmän päälle kaikkien mahdollisten rajapintarakenteiden tukeminen erikseen voi tulla liian raskaaksi ja rajoittavaksi.

2.2.2 Semanttinen yhteentoimivuus

Semanttinen yhteentoimivuus takaa, että komponenttien välinen vuorovaikutus on järkevää. Toisin sanoen, semanttisesti yhteentoimivilla komponenteilla on yhteinen

```
component Account {
  bool openAccount(string userId,unsigned long accountId);
  int getBalance();
  void deposit(unsigned long amount);
  void withDraw(unsigned long amount);
};

component Account2 {
  bool openAccount(unsigned long accountId,string userID);
  int getSavings();
  void withDraw(unsigned long sum);
  void deposit(unsigned long sum);
};
```

Taulukko 1: Kaksi saman toiminnallisuuden toteuttavaa komponenttia, joiden rajapinta on tekstuaalisesti eri näköinen

käsitys siitä, miten komponenttien väliset palvelupyynnöt ja niiden välillä välitettävä data vaikuttavat järjestelmän tilaan [VHT02]. Komponentin toimintojen semantiikka voidaan määritellä esimerkiksi metodikohtaisilla etu-/jälkiehdoilla. Taulukossa 2 on esimerkki metodin *withDraw* etu- ja jälkiehdoista. Metodin etuehtoina ovat vaatimukset siitä, että asiakas on tunnistettu ja että tililtä veloitettava summa ei ylitä tilin katetta. Jälkiehto takaa, että veloituksen jälkeen tililtä tosiaan on veloitettu haluttu summa.

```
//@ pre: _verified = true; (_balance - amount) > 0;
//@ post: _balance = _Oldbalance - amount;
void withDraw(unsigned long amount);
```

Taulukko 2: Esimerkki metodin etu- ja jälkiehdoista.

Komponenttien täydellisen toimintasemantiikan kuvaileminen on kuitenkin käytännössä mahdotonta [VHT02]. Semanttisten tarkistusten tekeminen on yleensä hyvin raskas prosessi, joka sisältää matemaattiseen logiikkaan perustuvien teoreemojen todistamista. Erityisen hankalaksi asian tekee se, että yleisessä tapauksessa algebran todistaminen konfluentiksi (nk. *Church-Rosser*-ominaisuus; jos termillä on normaalimuoto, niin se on yksikäsitteinen) ja normalisoituvaksi (jos termillä on normaalimuoto, niin on olemassa jokin redusointistrategia, joka saattaa termin kyseiseen normaalimuotoon) ei ole laskennallisesti ratkeava ongelma. Jos käytettävässä tietotyyppialgebra sisältää esimerkiksi kommutatiivisuussäännön ($a \circ b = b \circ a$), jota voidaan soveltaa missä tahansa vaiheessa, niin ongelmaksi muodostuu tämän säännön soveltaminen oikeassa kohdassa ja niin, että teoreeman todistaminen päättyy.

Komponenttien oikeelliseen toimintasemantiikkaan kuuluu myös se, että käytettävät tietorakenteet toimivat oikein. Tätä tarkoitusta varten käytettävät tietorakenteet

täytyy kuvata abstrakteina tietotyyppeinä (ADT), jotka kuvaavat tietotyyppiin kohdistuvien toimenpiteiden semantiikan. Taulukossa 3 on esimerkki tietotyypistä, joka määrittelee totuusmuuttujan *bool* toimintojen semantiikan. Syntaksi mukailee ACT-ONE–kieltä, jota käytetään LOTOS-spesifikaatioissa abstraktien tietotyyppien kuvailemiseen. ADT:n *sorts* osio esittelee spesifikaatiossa käytettävät tietotyypit, *opns* esittelee operaatioiden syntaksin ja *eqns* operaatioiden semantiikan.

```
bool=
sorts:bool
opns:   TRUE,FALSE -> bool
        NOT:bool -> bool
        AND:bool bool -> bool
eqns:b in bool
        NOT(TRUE) = FALSE
        NOT(NOT(b)) = b
        b  AND TRUE = b
        b  AND FALSE = FALSE
```

Taulukko 3: Totuusmuuttujan *bool* toimintojen semantiikka kuvattuna ACT-ONE–kielellä.

Komponenttien semanttisen yhteentoimivuuden tasolla yhteensopivuus tarkoittaa sitä, että komponentin tarjoama toiminnallisuus vastaa semantiikaltaan asiakkaan odottamaa käyttäytymistä. Semanttisella korvautuvuudella tarkoitetaan yleensä alityypitysrelaatiota toimintasemantiikan tasolla [VHT02]. Alityypin toimintasemantiikan tulee olla todistettavasti yhteneväinen ylityypin toimintasemantiikan kanssa. Esimerkiksi kooltaan rajoitettu pino on rajoittamattoman pinon semanttinen alityyppi [LW94].

2.2.3 Palveluprotokollien yhteentoimivuus

Komponentin protokollakuvauksilla voidaan määritellä, missä järjestyksessä komponentin toimintoja voidaan kutsua ja millä tavalla komponentti kommunikoi muiden komponenttien kanssa. Lisäksi protokollakuvauksien avulla voidaan tarkistaa yhteenliitettävien komponenttien turvallisuus, toisin sanoen se, että komponenttien kommunikointi ja vuorovaikutus ei johda esimerkiksi järjestelmän lukkiutumiseen [VHT02]. Komponentin käyttämät protokollat voidaan määritellä esimerkiksi prosessialgebroilla. Erityisesti mobiilien tai dynaamisten ympäristöjen (kuten väliohjelmistojen meklauspalveluiden) protokollakuvauksiin voidaan käyttää π-kalkyyliä, joka on mobiileja järjestelmiä tukeva prosessialgebra [MPW89].

Protokollatasolla kahta komponenttia kutsutaan yhteensopiviksi, jos niiden protokollien määrittelemät roolit sopivat yhteen. Jos esimerkiksi komponentti A on palvelinkomponentti ja B asiakaskomponentti, niin niiden sanotaan olevan yhteensopivia, jos niiden protokollakuvaukset ovat ristiriidattomia. Toisin sanoen, A ja B

ovat yhteensopivia, jos B kutsuu A:n palveluita siten kuin A:n palvelinrooli olettaa asiakkaiden käyttäytyvän ja päinvastoin.

Komponenttien korvattavuus protokollien tasolla tarkoittaa sitä, että korvattavan ja korvaavan komponentin roolit ovat yhteneväiset. Toisillaan korvattavien komponenttien tulee toteuttaa samat protokollat eli niiden tulee hyväksyä täsmälleen samanlaiset käyttötapaukset ja viestit. Protokollatason korvattavuus varmistaa sen, että esimerkiksi päivityksen jälkeen asiakkaat voivat kutsua uutta palvelinkomponenttia samalla tavalla kuin vanhaakin.

Taulukossa 4 on yksinkertainen esimerkki pankkitilin protokollakuvauksesta. Esimerkissä on kuvattuna vain tilinosto-operaatio *withDraw* [CFP+01]. Esimerkki on laadittu π-kalkyylillä, jota on laajennettu ohjelmointikielen kaltaisella syntaksilla (metodien kutsuminen ja niiden parametrien välittäminen sulkeissa). Protokollakuvauksella *Account* on yksi parametri, joka vastaa viitettä asiakkaaseen (viite on itseasiassa π-kalkyylin kanava). $ref?withDraw(amount, rep, notEnough)$ määrittelee prosessin, joka vastaanottaa ref-kanavaa pitkin metodikutsun *withDraw*. Metodikutsulla *withDraw* on syötteenään *amount* sekä paluukanavat *rep* ja *notEnough*. Onnistuneen pankkitilin veloituksen jälkeen asiakkaalle palautetaan kuittaus (*receipt*) kanavaa *rep* pitkin. Jos tilillä ei ole tarpeeksi rahaa, niin tilin saldo (*balance*) palautetaan kanavassa *notEnough*. Tämän jälkeen prosessi palaa alkutilaansa.

```
protocol Account {
  Account(ref) =
   ref?withDraw(amount,rep,notEnough) .
     ( tau . (^receipt) rep!(receipt) . Account(ref)
     +
       tau . (^balance) notEnough!(balance) . Account(ref)
     )
};
```

Taulukko 4: Pankkitilin protokollakuvaus, jossa automaatti palaa aina alkutilaansa.

Taulukossa 5 on kuvattu muuten samanlainen veloitusprotokolla, mutta liian suuren veloitusyrityksen jälkeen automaatti meneekin virhetilaan. Huomioitavaa on, että nämä kaksi erilaista protokollaa käyttävää komponenttia eivät ole korvattavissa keskenään, sillä niillä on asiakkaan kannalta huomattavan erilaiset roolit. Näistä kahdesta esimerkistä huomataan myös, kuinka tärkeää protokollien eksplisiittinen määrittäminen on. Vaikka komponentit olisivat sekä syntaktisella että toimintosemanttisella tasolla keskenään korvattavissa, voivat ristiriitaiset oletukset käytettävistä protokollista johtaa lukkiutumaan.

```
protocol Account {
  Account(ref) =
   ref?withDraw(amount,rep,notEnough) .
     ( tau . (^receipt) rep!(receipt) . Account(ref)
     +
       tau . (^balance) notEnough!(balance) . AccountError(ref,balance)
     )

  AccountError(ref,balance) =
   ref?withDraw(amount,rep,notEnough) .
   notEnough!(balance) .
   AccountError(ref,balance)
};
```

Taulukko 5: Pankkitilin protokollakuvaus, jossa automaatti menee virhetilaan.

3 Esimerkkejä väliohjelmistotekniikoista

Tässä luvussa käsitellään tällä hetkellä käytössä olevia tai tutkimuksen alla olevia väliohjelmistotekniikoita ja kuinka ne käsittelevät komponenttien yhteentoimivuuden ongelmia. Esimerkkinä perinteisestä väliohjelmistotekniikasta käsitellään CORBA:n komponenttilaajennosta eli CCM:ää. Tämän jälkeen käsitellään COTStrader–arkkitehtuuria, joka on yritys toteuttaa komponenttien meklauspalvelu, jossa komponenttien hakuehdot voivat sisältää myös sekä semanttisia ehtoja että vaatimuksia käytettäville protokollille.

3.1 Corba Component Model

CORBA Component Model (CCM) ei suoranaisesti ota kantaa komponenttien operaatioiden semantiikkoihin tai käytettäviin protokolliin. Kyseinen väliohjelmistoalusta edustaa komponenttipohjaisten järjestelmien tämän hetkistä tilaa: komponenttien yhteentoimivuus voidaan tarkistaa vain rajapintojen rakenteiden, ennalta määriteltyjen nimien ja luokkahierarkioiden välisten alityypitysrelaatioiden avulla.

Toisin kuin CORBA:n edellisissä versioissa, voidaan CCM-komponentille määritellä ulkoisia riippuvaisuussuhteita sekä useita rajapintoja [WSO01]. Ulkoiset riippuvaisuussuhteet voidaan määritellä *uses*-rakennetta käyttäen. Ulkoisten riippuvuuksien määrittelemisellä voidaan välttää tilanne, jossa komponentti on asiakkaan käytettävissä, vaikka kyseinen komponentti itseasiassa on käyttökelvottomassa kunnossa.

CCM-komponentti voi myös antaa asiakkaidensa käyttöön useita eri rooleja itsestään. Jokainen rooli kuvaa erilaista "näkökulmaa" komponentin toiminnallisuudesta. CORBA:n edellisissä versioissa komponentin laajentaminen uudella toiminnallisuudella oli hyvin hankalaa [WSO01].

CCM-mallissa komponentin toiminnallisuuden laajentaminen on huomattavasti helpompaa, koska malli tukee tukee monirajapintaisuutta. Moniroolisuus ja useiden rajapintojen tukeminen on hyödyllistä esimerkiksi siinä tapauksessa, kun komponentista halutaan saada käyttöön uusi ja laajennettu version ilman että vanhat asiakkaat joutuvat päivittämään ohjelmistojaan laajennettua palvelua vastaavaksi.

Hyvin rajoitetulla tasolla voidaan komponenttien yhteensopivuus tarkistaa CCM-arkkitehtuurissa komponenttien *uses/provides*–parien yhteensovittamisella. Jos käyttöön otettavalle komponentille ei löydy tarvittavaa palveluntarjoajaa, saa komponentti poikkeuksen [OMG01]. Poikkeuksen saatuaan jää komponentin tehtäväksi päättää, selviääkö se ilman ilmoittamiaan riippuvuussuhteita. Yhteensopivuustarkistukset voivat olla avointa komponenttikehitystä silmällä pitäen liian tiukkoja, koska ne perustuvat pelkästään metodien ja attribuuttien tekstuaaliseen yhtäsuuruuteen sekä luokkahierarkiseen alityypitysrelaatioon

Komponenttien keskinäistä korvattavuutta voidaan CCM:ssä tarkistaa sitäkin vain hyvin rajoitetuilla tavoilla. Komponentit ovat keskenään korvattavissa CCM-mallissa vain, jos ne ovat täsmälleen samat komponentit tai jos niiden välillä on alityyppirelaatio.

3.2 COTStrader

Eräs väliohjelmistojen tarjoamista palveluista on niin kutsuttu meklauspalvelu, joka julkaisee palvelutarjouksia ja ohjaa asiakkaat tarvitsemiensa palveluiden äärelle. Meklarin avulla asiakas voi hakea palvelua tietämättä kyseisen palvelun nimeä tai sijaintia. Palvelun hakeminen tapahtuu nimettyjen attribuuttien avulla, toisin sanoen haut tapahtuvat palvelun ominaisuuksiin perustuen. Yksinkertainen esimerkki asiakkaan esittämästä palvelunetsintäpyynnöstä voisi olla esimerkiksi *"etsi tulostuspalvelu, jolla voin tulostaa värillistä grafiikkaa 300 dpi:n tarkkuudella"*. Tämän jälkeen meklauspalvelu etsii annettujen attribuuttien mukaiset palvelut käyttäen hyväkseen väliohjelmiston tyyppienhallintapalvelua ja rajapintavarastoa (*interface repository* [OMG00].

Edellä mainittu meklauspalvelu on erittäin käyttökelpoinen. Sen avulla voidaan ohjelmistoissa hyödyntää komponenttien välistä ajonaikaista sidontaa, jossa käytettävän komponentin sijainti selvitetään vasta ajon aikana. Siitä on erityisesti hyötyä esimerkiksi adaptiivisissa ja vikasietoisissa järjestelmissä. Meklauspalvelu on kuitenkin hyödyllinen vain siinä tapauksessa, jos asiakas voi aina olla varma, että tarjotun palvelun toiminnot käyttäytyvät oletetulla tavalla. Tulevaisuudessa, kun ohjelmistot käyttävät yhä enemmän valmiita, uudelleen käytettäviä black-box–komponentteja, täytyy meklauspalveluidenkin kuvailla kattavammin tarjomiensa palveluiden toimintoja [1]. Palvelukuvauksiin täytyy liittää tietoja sekä toimintasemantiikasta että käytettävistä protokollista.

[1] Englanninkielisessä kirjallisuudessa puhutaan yleensä COTS (Commercial Of The Shelf)-komponenteista, kun tarkoitetaan uudelleenkäyttöön suunniteltuja kaupallisia komponentteja

COTStrader–projektin [2] tarkoituksena on on luoda väliohjelmistojen meklauspalvelu, jossa palveluiden hauissa huomioidaan mm. toimintojen semanttiset kuvaukset sekä palveluiden protokollakuvaukset. Nykyiset meklauspalvelut toimivat yleensä vain syntaktisella tasolla ja sielläkin yhteentoimivuusehdot rajoittuvat eksaktiin tekstuaaliseen ekvivalenssiin [ITV01]. Lisäksi nykyiset meklaukspalvelut eivät vielä tue komponenttiajattelua ja sen mukanaan tuomia ominaisuuksi kuten monirajapintaisuutta.

ODP:n mallin mukainen meklari määrittelee palvelun tyypin rajapinnan syntaksin ja palvelun ominaisuuksia kuvaavien nimettyjen attribuuttien mukaan [OMG00]. COTStrader mallissa sen sijaan palvelun tyyppi muodostuu neljästä erillisestä ominaisuuksien joukosta: funktionaalisista, epäfunktionaalisista, pakkaus- (*packaging*) ja markkinointiominaisuuksista (*marketing*).

Funktionaaliset ominaisuuksiin kuuluvat esimerkiksi tuettujen rajapintojen syntaksien kuvaukset, toimintojen semantiikka ja käytettävien protokollien kuvaukset. ODP:n mallista poiketen komponentilla voi olla useampia rajapintoja.

Epäfunktionaalisilla ominaisuuksilla kuvataan esimerkiksi palvelunlaatutakeita tai -vaatimuksia ja riippuvuussuhteita muista komponenteista.

Pakkausominaisuudet antavat informaatiota siitä, kuinka kyseinen komponentti tulisi asentaa ja minkälaisia rajoituksia komponentin käyttö asettaa ympäristölle.

Markkinointiominaisuudet kuvaavat esimerkiksi lisensöintipolitiikan, hintatietoja ja tietoja komponentin toteuttajasta.

Palvelutyypin ominaisuudet kirjataan XML-dokumenttiin. Taulukossa 6 on kuvattuna pankkitilikomponentin tyyppi COTStrader-arkkitehtuurin mukaisena XML-dokumenttina.

Rajapintojen syntaksitason, toimintojen semantiikan ja palveluprotokollakuvausten tarkistamiseen voidaan halutessa määritellä itse omat rutiinit. Yhteentoimivuusrutiineista voidaan tarjota kaksi eri vaihtoehtoa: $exactMatching$-rutiini suorittaa tiukan ekvivalenssitarkistuksen, joka esimerkiksi perustuu palveluiden luokkahierarkioiden alityyppirelaation, kun taas $softMatching$-rutiinin tulee suorittaa joustavampi ekvivalenssitarkistus. Joustavampaa tarkistusrutiinia käytetään esimerkiksi sellaisessa tilanteessa, jossa asiakas haluaa löytää palvelun, joka täyttää *vähintään* asiakkaan määräämät ehdot.

Palveluiden etsiminen tapahtuu käyttäen kahta erillistä XML-dokumenttia. Ensimmäinen niistä ilmaisee hakukriteerit meklarille ja toinen kuvailee halutun tyyppisen palvelun. Hakukriteereissä ilmoitetaan meklarille mm. epäfunktionaaliset ominaisuudet (esim. $securityLevel = SAFE$) ja halutessaan asiakas voi määritellä omat versionsa $exactMatching$- ja $softMatching$-rutiineista jokaiselle funktionaalisen ominaisuuden osiolle erikseen [ITV01].

Taulukossa 7 on asiakkaan palvelunetsintädokumentit. Niiden perusteella meklarin pitää palauttaa kaikki *Account*-palvelut, jotka toteuttavat lukkiutumattoman ver-

[2] www.cotstrader.com

sion *withDraw*-operaatiosta ja joiden *securityLevel*-ominaisuus on $SAFE$. Kuten huomataan, asiakas ei tässä tapauksessa ole kiinnostunut kuin siitä, että palautettavat palvelut sisältävät *Account*-rajapinnan, jonka tulee sisältää ainakin metodit $getBalance, deposit$ ja $withDraw$. Lisäksi palveluiden pitää toteuttaa asiakkaan määrittelemä rooli protokollatasolla. Protokolla määritellään $MyQuery.xml$-dokumentissa *serviceAccessProtocol*-tietueessa.

4 Yhteenveto

Tulevaisuudessa ohjelmistojen kehittäminen ja käyttäminen menevät yhä avoimempaan suuntaan. Ohjelmistokehityksen kannalta tämä tarkoittaa sitä, että ohjelmistoja kehitetään hyväksikäyttämällä uudelleenkäyttöä varten suunniteltuja komponentteja. Ohjelmistojen käytön kannalta avoimuus tarkoittaa sitä, että tarvittavia ohjelmistokomponentteja voidaan hakea jopa ajonaikaisesti verkkoyhteyden takaa. Molemmissa tapauksissa ongelmaksi muodostuu oikealla tavalla käyttäytyvien komponenttien löytäminen. Käytettävät komponentit voivat olla eri ohjelmointikielillä toteutettuja ja eri kehitysryhmien tuottamia. Tämän takia komponenttien käyttökelpoisuutta ei voida enää päätellä pelkästään rajapinnan syntaksin tekstuaalisella (nimiin perustuvalla) vertailulla, sillä tämä rajoittaisi turhaan yhteentoimivien komponenttien määrää. Toisaalta, pelkästään syntaksiin perustuva yhteentoimivuustarkistus ei takaa yhteenliitettyjen komponenttien virheetöntä toimintaa.

Yllä mainittujen syiden takia tarvitaan tulevaisuuden väliohjelmistoihin toiminnallisuutta, jota toisaalta mahdollistaa joustavammat yhteentoimivuustarkistukset syntaktisella tasolla ja toisaalta mahdollistaa komponentin toiminnallisuuden formalisoinnin. Näitä tarpeita varten voidaan kehittää järjestelmiä, jotka käyttävät yhteentoimivuutta tarkistaessaan hyväkseen komponenttityyppien strukturaalista yhteneväisyyttä (joko eksaktia tai joustavampia ehtoja), toimintojen semantiikan kuvauksia sekä palveluprotokollien kuvauksia. Näillä keinoin voitaisiin toteuttaa esimerkiksi meklausjärjestelmä, jonka avulla ohjelmistojen kehittäminen *todella hyötyisi uudelleenkäytettävistä komponenteista.*

Lähteet

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

CFP+01 C. Canal, L. Fuentes, E. Pimentel, J. M. Troya ja A. Vallecillo. Extending CORBA Interfaces with Protocols. *The Computer Journal*, 44(5):448–462, lokakuu 2001. URL citeseer.nj.nec.com/canal01extending.html.

ITV01 Luis Iribarne, Jose' M. Troya ja Antonio Vallecillo. Trading for COTS Components in Open Environments. *27th Euromicro Conference 2001: A Net Odyssey (EUROMICRO'01)*, sivut 30–40. IEEE, syyskuu 2001.

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

MPW89 Robin Milner, Joachim Parrow ja David Walker. A Calculus of Mobile Processes, Parts I and II. Tekninen raportti -86, University of Edinburgh, 1989. URL citeseer.nj.nec.com/milner89calculus.html.

OMG00 Trading Object Service Documentation. Tekninen raportti 00-06-27, Object management Group, toukokuu 2000. URL http://www.omg.org/technology/documents/formal/trading_object%_service.htm.

OMG01 CORBA 3.0 New Components Chapters. Tekninen raportti ptc/2001-11-03, Object Management Group, marraskuu 2001.

VHT02 A. Vallecillo, J. Hernandez ja J.M. Troya. Component Interoperability. Tekninen raportti ITI-2000-37, University of Malaga, heinäkuu 2002. URL citeseer.nj.nec.com/535975.html.

WSO01 Nanbor Wang, Douglas C. Schmidt ja Carlos O'Ryan. *Component-Based Software Engineering: Putting the Pieces Together*, luku 31, sivut 557–572. Addison-Wesley, 2001.

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

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

```
<COTScomponent name=''Account''
 xmlns=''http://www.cotstrader.com/COTS-XMLSchema.xsd''>
 <functional> <!-- Toiminnallisuuden kuvaus -->
  <providedInterfaces>
   <interface name=''Account''>
    <description notation=''CORBA-IDL''>
     interface Account {
      bool _verified;
      unsigned long _OldBalance;

      bool openAccount(string userId,unsigned long accountId);
      int getBalance();
      void deposit(unsigned long amount);
      void withDraw(unsigned long amount);
     };
    </description>
    <exactmatching href=''http://www.foobar.com/corba.exactmatch'' />
    <softmatching href=''http://www.foobar.com/corba.softmatch'' />
    <behavior notation=''MyOwnNotation''>
     //@ pre: _verified = true; (getBalance() - amount) > 0;
     //@ post: _balance = _OldBalance - amount;
     void withDraw(unsigned long amount);
    </behavior>
   </interface>
  </providedInterfaces>
  <serviceAccessProtocol>
   <description notation="pi-protocols">
     Account(ref) =
      ref?withDraw(amount,rep,notEnough) .
      ( tau . (^receipt) rep!(receipt) . Account(ref)
       +
        tau . (^balance) notEnough!(balance) . Account(ref)
      )
   </description>
   <exactMatching href="http://soleres.ual.es/servlet/PI.exactMatching"/>
   <softMatching href="http://soleres.ual.es/servlet/PI.softMatching"/>
  </serviceAccessProtocol>
 </functional>

 <properties notation="W3C"> <!-- Epäfunktionaaliset ominaisuudet -->
   <property name="securityLevel">
     <type>xsd:string</type> <value>SAFE</value>
   </property>
   <property name="capacity">
     <type>xsd:int</type> <value>1</value>
   </property>
   <property name="keywords">
     <type>xsd:string</type> <value>account,safe</value>
   </property>
 </properties>

 <packaging>    <!-- Pakkaustiedot -->
  <description notation=''MyOwnPkgDesc''>
   <author name=''BigBank Corporation'' />
  </description>
 </packaging>

 <marketing>     <!-- Markkinointitiedot -->
  <licence href=''http://www.gnu.org/licenses/gpl.txt'' />
 </marketing>
</COTScomponent>
```

Taulukko 6: Account palvelutyypin XML-kielinen kuvaus.

```
// Tiedosto ClientQuery.xml
<?xml-stylesheet type="text/xsl"
                  href="http://www.cotstrader.com/COTS-XMLStyle.xsl"?>

  <COTSquery name="ClientQuery"
             xmlns="http://www.cotstrader.com/COTS-XMLSchema.xsd">

  // Viittaus alhaalla esitettyyn XML-dokumenttiin
  <COTSdescription href="MyQuery.xml" />

   <functionalMatching>
    <interfaceMatching>
      <softMatching />
    </interfaceMatching>
    <serviceAccessProtocolMatching>
      <exactMatching />
    </serviceAccessProtocolMatching>
   </functionalMatching>

   <propertyMatching>
    <constraints notation="XQuery">
     (securityLevel = SAFE)
    </constraints>
   </propertyMatching>

   <packagingMatching />

   <marketingMatching />
   </COTSquery>
..............
// Tiedosto MyQuery.xml
<COTScomponent name="MyQuery"
                xmlns="http://www.cotstrader.com/COTS-XMLSchema.xsd">

   <functional>
    <providedInterfaces>
     <interface name="Account">
      <description notation="CORBA-IDL">
        interface Account {
         int getBalance();
         void deposit(unsigned long amount);
         void withDraw(unsigned long amount);
        }
      </description>
     </interface>
    </providedInterfaces>

    <serviceAccessProtocol>
     <description notation="pi-protocols">
       Account(ref) =
       ref?withDraw(amount,rep,notEnough) .
       ( tau . (^receipt) rep!(receipt) . Account(ref)
        +
         tau . (^balance) notEnough!(balance) . Account(ref)
       )
     </description>
    </serviceAccessProtocol>
   </functional>

   <properties>
    <property name="securityLevel">
      <type>xsd:string</type>
    </property>
    <property name="isRunningNow">
      <type>xsd:bool</type>
    </property>
   </properties>

  </COTScomponent>
```

Taulukko 7: Asiakkaan kysely, joka etsii *Account*-tyyppisiä palveluita.