Arkkitehtuurinen reflektio

Toni Ruokolainen
Toni.Ruokolainen@cs.helsinki.fi

Helsinki 1.12.2003

Seminaarikirjoitelma

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

HELSINGIN YLIOPISTO —HELSINGFORS UNIVERSITET —UNIVERSITY OF HELSINKI

Tiedekunta/Osasto — Fakultet/Sektion — Faculty	Laitos — Institution — Department
Matemaattisluonnontieteellinen tiedekunta	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Toni RuokolainenToni.Ruokolainen@cs.helsinki.fi

Työn nimi — Arbetets titel — Title

Arkkitehtuurinen reflektio

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Seminaarikirjoitelma	1.12.2003	16

Tiivistelmä — Referat — Abstract

Järjestelmän arkkitehtuuria käytetään perinteisesti vain suunnitteluvaiheessa. Toteutettavan järjestelmän arkkitehtuuri esiintyy laatikoiden, viivojen ja niiden attribuuttien kokoelmana suunnitteludokumenteissa. Nämä suunnitteluvaiheen eksplisiittiset arkkitehtuurikuvaukset muutetaan toteutusvaiheessa implisiittisiksi säännöiksi ja oletuksiksi järjestelmän toteutuksen sisään. Jotta komponenttien ja valmiiden järjestelmien ylläpito ja uudelleenkäyttö tulisi helpommaksi, pitäisi eksplisiittiset arkkitehtuurikuvaukset säilyttää osana toteutusta.

Eksplisiittiset, toteutuksen mukana kulkevat arkkitehtuurikuvaukset ovat hyödyllisiä monissa tilanteissa. Usein on esimerkiksi tarpeen tehdä järjestelmässä ylläpitotoimia, kuten vanhojen komponenttien päivittämistä uudempiin, vikaantuneen komponentin vaitamista toimivaan yms. Jotta tällaisia komponenttien korvaamisia tai niiden konfiguraatiotietojen muutoksia voitaisiin tehdä järjestelmän ollessa käytössä, tulee jokaisella komponentilla olla tieto siitä, missä kontekstissa niitä käytetään ja kuinka itseensä tai toisiin komponentteihin tehdyt muutokset vaikuttavat järjestelmän käyttäytymiseen.

Järjestelmän arkkitehtuurin tasolla tapahtuvat ajonaikaiset muutokset ja niihin mukautuminen asettavat huomattavia vaatimuksia itse infrastruktuurijärjestelmälle ja sen informaatiosisällölle. Arkkitehtuurikuvauksen lisäksi pitäisi järjestelmän ajonaikainen tilatieto olla käytössä siinä laajuudessa, että arkkitehtuurin kannalta olennaiset ominaisuudet saadaan selville. Lisäksi arkkitehtuurikuvausten ja järjestelmän tilatietojen pitää olla synkronoitu varsinaisen tilan kanssa siten, että kuvausten avulla voidaan tehdä johtopäätelmiä järjestelmän nykytilasta Arkkitehtuurikuvauksiin perustuvaa laskentaa ja reagoimista kutsutaan *arkkitehtuuriseksi reflektioksi*.

Avainsanat — Nyckelord — Keywords
architecture,reflection,component,connector,adl

Säilytyspaikka — Förvaringsställe — Where deposited
Tietojenkäsittelytieteen laitoksen kirjasto

Muita tietoja — övriga uppgifter — Additional information

Reflektiiviset väliohjelmistot-seminaari, syksy 2003

Sisältö

Kuvat

1 Johdanto

Nykyään suuri osa yritysten palveluista on saavutettavissa Internetin kautta koska tahansa ja mistä tahansa. Kauppaa käydään vuorokauden ympäri sekä yritysten että ihmisten välillä. Yritysten välinen kilpailu ja asiakkaiden vaatimukset luovat yrityksille paineita päivittää tarjoamiensa palveluiden toteutuksia aika-ajoin. Palveluiden tulisi kuitenkin olla aina saatavilla ja adaptoitavissa muuttuvien tarpeiden mukaan, mutta siitä huolimatta järjestelmien tulisi olla käytettävissä lähes koko ajan. Tätä varten tarvitaan palveluiden ajoympäristö, joka tukee palvelukomponenttien ajonaikaista päivittämistä ja adaptoitumista.

Jotta palvelukomponentteja voitaisiin päivittää ja adaptoida, pitäisi järjestelmän tietää missä tilassa ne ovat. Erityisesti tulisi tietää, mitkä muut palvelukomponentit riippuvat päivitettävästä komponentista ja kuinka ne toimivat yhteen. Komponenttien suhteet ja yhteentoimivuuden kuvaukset muodostavat järjestelmän *arkkitehtuurin* [GAO95].

Järjestelmän arkkitehtuuria käytetään perinteisesti vain suunnitteluvaiheessa. Toteutettavan järjestelmän arkkitehtuuri esiintyy laatikoiden, viivojen ja niiden attribuuttien kokoelmana suunnitteludokumenteissa. Nämä suunnitteluvaiheen eksplisiittiset arkkitehtuurikuvaukset muutetaan toteutusvaiheessa implisiittisiksi säännöiksi ja oletuksiksi järjestelmän toteutuksen sisään. Jotta komponenttien ja valmiiden järjestelmien ylläpito ja uudelleenkäyttö tulisi helpommaksi, pitäisi eksplisiittiset arkkitehtuurikuvaukset säilyttää osana toteutusta.

Eksplisiittiset, toteutuksen mukana kulkevat arkkitehtuurikuvaukset ovat hyödyllisiä monissa tilanteissa. Usein on esimerkiksi tarpeen tehdä järjestelmässä ylläpitotoimia, kuten vanhojen komponenttien päivittämistä uudempiin, vikaantuneen komponentin vaitamista toimivaan yms. Jotta tällaisia komponenttien korvaamisia tai niiden konfiguraatiotietojen muutoksia voitaisiin tehdä järjestelmän ollessa käytössä, tulee jokaisella komponentilla olla tieto siitä, missä kontekstissa niitä käytetään ja kuinka itseensä tai toisiin komponentteihin tehdyt muutokset vaikuttavat järjestelmän käyttäytymiseen.

Järjestelmän arkkitehtuurin tasolla tapahtuvat ajonaikaiset muutokset ja niihin mukautuminen asettavat huomattavia vaatimuksia itse infrastruktuurijärjestelmälle ja sen informaatiosisällölle. Arkkitehtuurikuvauksen lisäksi pitäisi järjestelmän ajonaikainen tilatieto olla käytössä siinä laajuudessa, että arkkitehtuurin kannalta olennaiset ominaisuudet saadaan selville. Lisäksi arkkitehtuurikuvausten ja järjestelmän tilatietojen pitää olla synkronoitu varsinaisen tilan kanssa siten, että kuvausten avulla voidaan tehdä johtopäätelmiä järjestelmän nykytilasta [CSST98]. Arkkitehtuurikuvauksiin perustuvaa laskentaa ja reagoimista kutsutaan *arkkitehtuuriseksi reflektioksi* [TSCS01].

Tässä seminaarikirjoitelmassa tutustutaan arkkitehtuurisen reflektion käsitteeseen. Arkkitehtuurinen reflektio on mahdollista vain, jos arkkitehtuurin kuvaus on eksplisiittisesti kuvailtuna. Tämän vuoksi tutustutaan myös siihen, kuinka ohjelmistoarkkitehtuureja voidaan mallintaa arkkitehtuurinkuvauskielillä ja minkälaista tietoa kuvausten tulee sisältää. Viimeiseksi käsitellään sitä, kuinka arkkitehtuurikuvausten perusteella voidaan käytännössä suorittaa arkkitehtuurista reflektiota.

2 Arkkitehtuurinen reflektio

Arkkitehtuurinen reflektio määritellään ohjelmiston laskennaksi oman arkkitehtuurisen kuvansa perusteella [CSST98]. Arkkitehtuurista reflektio jaetaan kahteen osaan: topologiseen ja strategiseen. Topologinen tieto on järjestelmän konfiguraatiota koskevaa tietoa. Se sisältää kuvaukset siitä, mitä komponentteja järjestelmä sisältää ja kuinka eri komponentit liittyvät toisiinsa. Yleisesti järjestelmän kuvataan koostuvan *komponenteista*, jotka edustavat toiminallisia yksiköitä ja *konnektoreista*, jotka kuvaavat komponenttien välisiä yhteyksiä ja niiden ominaisuuksia [AG94]. Strateginen tieto käsittelee kokonaisjärjestelmän käytöstä tai käyttötarkoitusta. Strategia määrittelee esimerkiksi sen, missä vaiheessa tietty toiminnallisuus tulee tehdä [CSST98].

Tässä luvussa käsittelemme arkkitehtuurisen reflektion toteuttamiseen vaadittavia kuvausmekanismeja ja käsitteitä. Tutustumme komponenttien ja konnektoreiden kuvaamiseen aliluvussa 2.1. Luvussa 2.2 kuvataan järjestelmän topologian käsite ja sen kuvaamiseen tarvittavat käsitteet. Strategisen reflektion käsittelemiseen tarvittavat välineet kuvataan luvussa 2.3.

2.1 Komponentit ja konnektorit

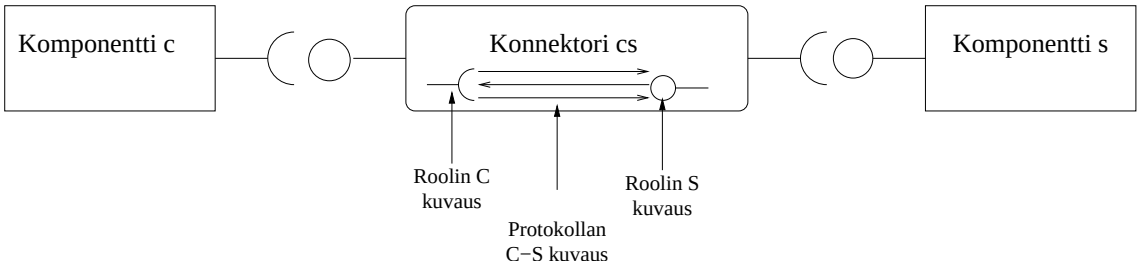
Arkkitehtuuri määrittelee järjestelmän komponentit ja niiden väliset vuorovaikutussuhteet [LVM95]. Arkkitehtuurin kuvaamiseen käytetään arkkitehtuurinkuvauskieliä, joiden välillä on eroavaisuuksia sen suhteen, mihin tarkoitukseen ne on suunniteltu. Eräs eroavaisuus on kielien tapa kuvata komponenttien välistä vuorovaikutusta. Vuorovaikutuksen kuvaaminen voidaan jakaa karkeasti kolmeen ryhmään: implisiittiset konnektorit, esimääritellyt konnektorit ja eksplisiittiset konnektorit [BP]. *Darwin* on esimerkki kielestä, joka käyttää implisiittisiä konnektoreita. Siinä komponenttien vuorovaikutus ilmaistaan komponenttien välisinä $provides$- ja $requires$-suhteina. *Unicon*-kielessä on määriteltynä valmiita konnektorityyppejä, kuten $Pipe$,$FileIO$ ja $ProcedureCall$, jotka kuvaavat tyypillisten vuorovaikutussuhteiden abstraktioita [SDK$^+$95].

Eksplisiittisten konnektoreiden mallia käytetään esimerkiksi *Wright*-kielessä, jossa komponenttien vuorovaikutuksen käyttäytyminen voidaan eksplisiittisesti määritellä [AG94]. Hajautetuissa arkkitehtuureissa on hyödyllistä mallintaa konnektorit eksplisiittisesti, vaikka konnektorit voitaisiinkin mallintaa vain yhtenä komponenttina muiden joukossa. Konnektoreiden määritteleminen tavallisiksi komponenteiksi yksinkertaistaa kuvauskieltä, mutta toisaalta konnektoreiden toimintasemantiikka voi erota komponenttien vastaavasta. Toteutusvaiheessa konnektorit voidaan instantioida automaattisesti esimerkiksi kahdeksi *Proxy*-luokaksi kommunikointikanavan päihin. Jos konnektori olisi määritelty tavallisen komponentin semantiikkaa käyttäen, olisi tällainen automaattinen osiin jako vaikeaa. Lisäksi eksplisiittisten konnektoreiden käyttö erottaa komponentin varsinaisen toiminnallisuuden ja vuorovaikutuksen omiksi, uudelleenkäytettäviksi kokonaisuuksiksi. Tällä tavoin komponenttia voidaan helpommin käyttää ympäristöissä, joihin sitä ei aluperin oltu suunniteltu. Myös komponenttien ja konnektoreiden elinkaaret voivat erota: konnektorien elinkaari on yleensä vaihtelevampi kuin komponenttien [BP].

Komponentit kuvaavat järjestelmän toiminnallisia yksiköitä, tai rooleja, ja konnektorit niiden välisiä yhteyksiä. Esimerkiksi tavallisessa client-server–arkkitehtuurin kuvauksessa on kaksi komponenttia, asiakas ja palvelin. Komponentit kuvataan määrittelemällä niiden käyttötapa ja käyttäytyminen. Käyttötapa kuvataan yleensä rajapintakuvauksen avulla ja käyttäytyminen esimerkiksi prosessialgebralla.

Konnektorit kuvataan samalla tavoin kuin varsinaiset komponentitkin. Konnektorit muodostavat eräänlaisen "putken" komponenttien välille, jonka kautta komponentit voivat kommunikoida keskenään. Konnektori määrittelee joukon *rooleja*, jotka kuvaavat vaadittavat rajapinnat ja odotetun käytöksen konnektoriin liitettäviltä komponenteilta. Roolikuvausten lisäksi konnektorissa määritellään roolien välinen vuorovaikutus, eräänlainen "liima", roolien välille [AG97]. Tätä roolien välistä synkronointikuvausta kutsutaan tästä lähtien *protokollaksi*.

Kuvassa 1 on esimerkki yksinkertaisesta asiakas-palvelin–arkkitehtuurista, joka selventää edellä esiteltyjä käsitteitä. Komponentti c kuvaa asiakaspuolen komponenttia, joka on yhteydessä palvelinkomponenttiin s konnektorin cs kautta. Konnektori cs määrittää hyväksymänsä komponentit roolikuvauksilla C ja S. Roolikuvaukset kuvaavat konnektoriin liittyvältä komponentilta odotettua (lokaalia) käyttäytymistä [AG97]. Toisin sanoen, rooli on formaali käyttäytymiskuvaus, jonka perusteella komponentin yhteensopivuus kyseiseen rooliin tarkistetaan. Jotta komponentti s olisi yhteensopiva konnektorin määrittelemään rooliin S, tulee näiden käyttäytymisten olla ekvivalentteja tarvittavalla tasolla. Kuvausten ekvivalenssi voi olla esimerkiksi käyttäytymiskuvausten jokin (bi)similaarisuusrelaatio.



Kuva 1: Esimerkki arkkitehtuurikuvauksesta

Roolien C ja S välinen yhteys kuvataan protokollakuvauksen $C - S$ avulla. Se kuvaa konnektorin asiakas- ja palvelinroolien välistä koordinointia. Koordinointi voi esimerkiksi määritellä asiakas- ja palvelinroolien viestienvaihdolle (metodikutsuille) tiukan järjestysehdon. Esimerkissä oleva konnektori on binäärinen, mutta yleisessä tapauksessa konnektori voi määritellä myös yksi-moneen– ja moni-moneen–yhteyksiä [AG97].

Taulukossa 1 on kuvattu Cazzolan tutkimusryhmän kehittämää APIL-notaatiota (*Architectural Programming-In-the-Large*) käyttäen sekä komponentti- että konnektorimäritelmien syntaksit [CSST]. Komponentin spesifikaatio määrittelee komponentin käyttäytymisen tila-automaattien muodossa sekä sidonnan varsinaisen funktionaalisen komponentin toiminteisiin. Käyttäytymisen kuvaaminen tehdään porttikuvauksilla, jotka määrittelevät komponentin vuorovaikutuspisteet [CSST]. Portit määrittelevät komponentin tietyn rajapinnan käyttäytymisen. Ne kuvaavat tilakoneen, jossa tilat kuvaavat portin tilaa ja siirtymät ovat joko ulkoisia tapahtumia tai sisäisiä reaktioita.

```
<componentSpecification> ::=
 component <componentName> {
  {<portSpecifications>}
  <portDependencies>
 }

<portSpecification> ::=
 port <portName> {
  initial_state: <stateName>
  transitions: [<portTransition>
             [{,<portTransition>}]]
}

<portTransition> ::=
 <componentEvent>: <pre> -> <portActions> -> <post>

<portActions> ::= [core.<command>
               [{, core.<command>}]]
<portDependencies> ::=
 [dependencies: <dependence>{[<dependence>]}

<dependence> ::=
 <portName>.<stateName> -> <portName>.<stateName>
```

```
<connectorSpecification> ::=
 connector <connectorName> {
  roles: <roleName> {,<roleName>}
  initial_state: {<stateName>}
  transitions: <connectorTransision>
             {,<connectorTransition>}
 }

<connectorTransition> ::=
 <cooperationEvent>:
  <pre> -> <connectorActions> -> <post>

<connectorActions> ::=
 [<roleName>.<componentEvent>
 [{, <roleName>.<componentEvent>}]]
```

Taulukko 1: Komponentin ja konnektorin määritelmät

Portin määritelmä sisältää alkutilan määrityksen sekä siirtymäsäännöt. Siirtymäsäännöt ovat muotoa $event : pre \rightarrow actions \rightarrow post$, missä $event$ on ulkoinen tapahtuma, joka laukaisee toiminteet $actions$. Toiminteet laukaistaan vain, jos etuehto pre on voimassa. Toiminteiden suorittamisen jälkeen portti menee tilaan $post$. Toiminteet ovat komponentin sisäisen ytimen (*core*) toimintoja, eli varsinaisen toteutuskomponentin metodikutsuja.

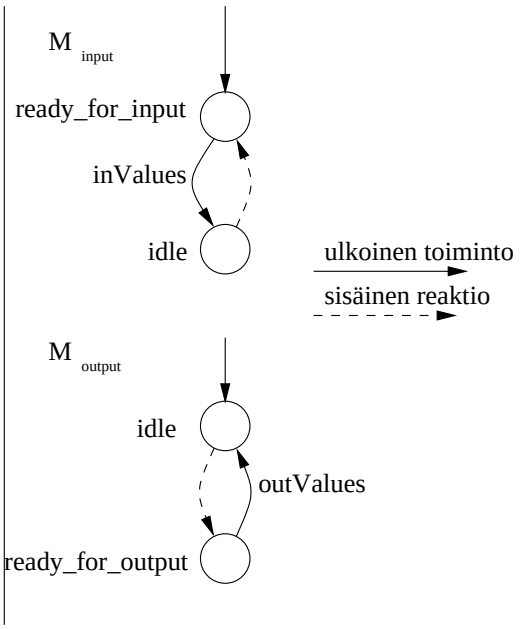
Ulkoiset tapahtumat ovat esimerkiksi metodikutsuja tai muita tapahtumia. Sisäiset reaktiot taas yhdistävät eri portit toisiinsa. Reaktiosäännöt kuvataan komponenttikuvauksen *<portDependencies>* osiossa ja ne ovat muotoa $p.s \rightarrow q.t$, missä $p.s$ tarkoittaa sitä, että komponentti p on tilassa s ja $q.t$ sitä, että komponentti q on tilassa t. Reaktiosääntöjen tulkinta on se, että aina kun jonkin reaktiosäännön vasen puoli on tosi ($p.s$), niin komponentti q:n tila muutetaan tilaan t [CSST] Tällä tavalla komponenteista saadaan aktiivisia siinä mielessä, että ulkoinen ärsyke tietyssä portissa voi saada aikaan joukon / jonon tapahtumia, jotka vaikuttavat eri porttien tiloihin.

Kuvassa 2 on esimerkkinä komponentin $util$ määritelmä sekä sen portteja $input$ ja $output$ vastaavat tilakoneet M_{input} ja M_{output}. Portin $input$ tilakone aloittaa tilasta $ready\_for\_input$. Tapahtuma $inValues$ muuttaa tilakoneen tilaan $idle$, joka saa puolestaan aikaan sen, että $dependencies$ osiosta sovelletaan sääntöä $input.idle \rightarrow output.ready\_for\_output$. Tämä saa aikaan sen, että portti $output$ siirtyy tilasta $idle$ tilaan $ready\_for\_output$.

Konnektorit kuvaavat eksplisiittisesti komponenttien välisen liitännän ja kommunikaation. Ne määrittelevät komponenttien yhteistoimintaan liittyviä tapahtumia tila-automaattien muodossa. Konnektorimääritelmä sisältää joukon roolien nimiä ja konnektorin tilasiirtymäsäännöt. Roolit ovat komponenttien abstrakteja edustajia, joihin komponentteja voidaan liittää varsinaisessa topologian kuvauksessa. Konnektorin tilasiirtymäsäännöt ovat komponentin tilasiirtymäsääntöjen kaltaisia: ne määrittelevät tapahtuman johon reagoidaan, sekä etu- ja jälkiehdot toiminnolle.

Kuvassa 3 on määriteltynä kaksi konnektoria. Konnektori $inputConnector$ määrittelee kaksi roolia, in ja $unit$ sekä alkutilan $empty$. Konnektori $inputConnector$ reagoi tapahtumaan $transfer(real * d)$, jonka seurauksena se laukaisee roolissa in tapahtuman $getValues(real * d)$ ja roolissa $unit$ tapahtuman $inValues(real * d)$. Tämän jälkeen konnektori palaa alkutilaansa. Konnektorin $outputConnector$ määritelmä on symmetri-

```
component unit {
 initial_state: waiting_for_input
 port input {
  initial_state: ready_for_input
 transitions:
  inValues(real *data):
    ready_for_input −> core.evaluate(real *data) −> idle
}
port output {
 initial_state: idle
 transitions:
  outValues(real *data):
   ready_for_output −>core.retrieve(real *data) −> idle
}
dependencies:
 input.idle −> output.ready_for_output
 output.idle −> input.ready_for_input
}
```



Kuva 2: Esimerkki komponentin $util$ määritelmästä sekä sen portteja $input$ ja $output$ vastaavat tilakoneet

nen $inputConnector$:in kanssa.

```
connector inputConnector {
  roles: in, unit
  initial_state: empty
 transitions:
  transfer(real *d):
   empty −> in.getValues(real *d),
     unit.inValues(real *d) −> empty
}
```

```
connector outputConnector {
 roles: out, unit
 initial_state: empty
 transitions:
  transfer(real *d):
    empty −> unit.outValues(real *d),
      out.putValues(real *d) −> empty
}
```

Kuva 3: Konnektorit $inputConnector$ ja $outputConnector$

Komponenttien ja konnektoreiden portteja edustavien tilakoneiden avulla voidaan järjestelmän tilaa mallintaa ja ylläpitää. Mallinnusvaiheessa voidaan komponenttien ja konnektoreiden muodostamista järjestelmistä voidaan myös tarkistaa mielenkiintoisia ominaisuuksia liittyen järjestelmän käyttäytymiseen ja oikeellisuuteen. Oletetaan, että järjestelmä koostuu komponenteista C_1, C_2, C_3 sekä konnektoreista $c_{1\to2}$ ja $c_{2\to3}$. Tällöin järjestelmältä vaaditut ominaisuudet ϕ voidaan verifioida muodostamalla yhteistila-automaatti $M : C_1||c_{1\to2w}||C_2||c_{2\to3}||C_3$, missä $A||B$ tarkoittaa automaattien A ja B rinnakkaisyhdistettä (automaatteja ajetaan rinnakkain ja niiden välillä voi ilmetä synkronointitapahtumia), ja tarkistamalla, että $M \models \phi$. Proseduuria, jossa tarkistetaan että tietty rakenne on annetun mallin mukainen, kutsutaan mallintarkistukseksi. Ominaisuudet ϕ voivat esittää esimerkiksi lukkiutumattomuutta, poissulkemisominaisuuksia tai muita järjestelmältä

vaadittuja ominaisuuksia.

Komponenttien ja konnektoreiden muodostamien automaattien avulla voidaan järjestelmän suoritustua myös tarkkailla. Koska järjestelmän toiminta on kuvattu eksplisiittisesti tilakoneiden avulla, voidaan järjestelmän tilaa kuvata ylläpitämällä tietoja tilakoneiden tiloista ja siirtymistä. Konnektoreiden ja komponenttien tilakoneista voidaan muodostaa niin kutsuttuja monitoreita, jotka valvovat sitä, että arkkitehtuurikuvauksessa määriteltyjä käyttäytymiskuvioita ei rikota ajon aikana.

2.2 Topologinen reflektio

Tämän luvun alussa tutustuimme siihen, kuinka arkkitehtuurin peruspalikat, komponentit ja konnektorit, voidaan mallintaa arkkitehtuurinkuvauskielellä ja kuinka niiden käyttäytyminen voidaan tulkita tilakoneena. Järjestelmän topologialla tarkoitetaan komponenttien ja konnektoreiden muodostaman arkkitehtuurin struktuuria. Arkkitehtuurin kuvaus on joukko komponentteja ja konnektoreja sekä joukko liitoksia, jotka määrittelevät komponenttien ja konnektoreiden yhteydet. Eräs topologian määrittelykieli (arkkitehtuurikuvaus) on kuvattuna kuvassa 2. Topologia on nimetty arkkitehtuurikuvaus, joka määrittää joukon komponentteja ja konnektoreita. Topologiakuvauksen *attachments*-osiossa komponenttien portit liitetään konnektoreiden rooleihin.

```
<topologySpecification> ::=
 topology <topologyName> {
  components: {<componentName>}
  connectors: {<connectorName>}
  attachments:
   {<componentName>.<portName> plays <connectorName>.<roleName>}
```

Taulukko 2: Järjestelmän topologian kuvaus

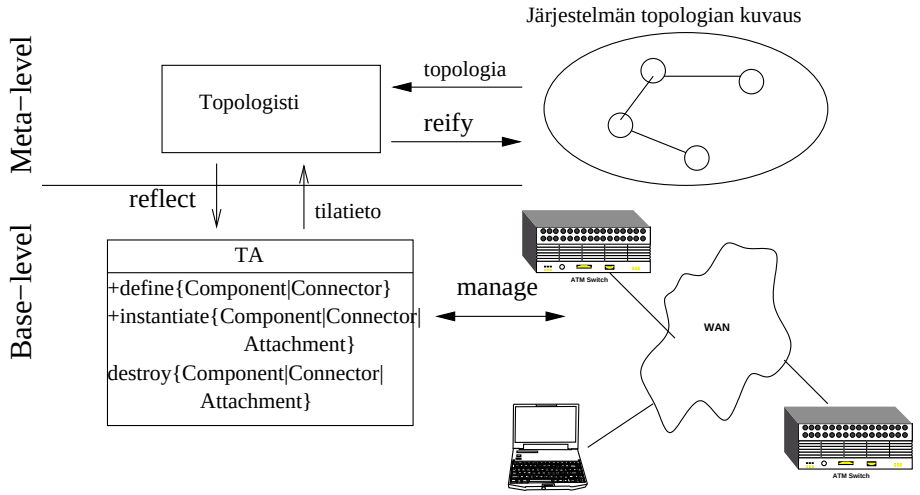
Topologinen reflektio on järjestelmän suorittamaa laskentaa oman rakenteensa perusteella [CSST98]. Topologinen reflektio voi ilmetä komponenttien tai konnektoreiden lisäämisenä / poistamisena järjestelmästä tai niiden uudelleenkonfigurointina. Jotta topologinen reflektio olisi mahdollista, täytyy järjestelmän ylläpitää kuvauksia hajautetun ohjelmiston tilasta ja rakenteesta. Tilatiedot käsittävät komponenttien ja konnektorien kuvaukset sekä niitä vastaavien tilakoneiden tiedot. Topologinen tieto sisältää kuvauksen siitä, miten järjestelmän eri komponentit ovat yhteydessä toisiinsa konnektoreiden kautta.

Topologisesta reflektiosta huolehtivaa järjestelmän osaa kutsutaan *topologiksi*. Sen tehtävänä on ylläpitää kausaalista suhdetta topologisen mallin ja järjestelmän rakenteen välillä [CSST]. Topologi ylläpitää tietoja järjestelmän rakenteesta ja sitä vastaavasta metadatasta sekä reflektiosta näiden välillä. Topologi toimii siltana topologian kuvauksen ja varsinaisen topologian ylläpitäjän (*topology actuator, TA*) välillä [CSST98]. Toisaalta topologian ylläpitäjä (tästälähtien TA) toimii komponenttien ja konnektoreiden abstrahoijana topologille, toisin sanoen, se piiloittaa turhat toteutusyksityiskohdat topologilta.

Kuvassa 4 on kuvattuna järjestelmän topologian, topologin ja TA:n suhteet. Topologi ylläpitää topologiakuvausta TA:lta tulevien tapahtumien mukaan. Toisaalta, topologi toi-

mii reflektion periaatteiden mukaan myös toiseen suuntaan. Kun metadatassa tapahtuu muutoksia, topologi ilmoittaa tästä muutoksesta TA:lle. TA puolestaan tulkitsee topologilta saamansa viestit ja päättää tulkintansa mukaan siitä, mitä muutoksia varsinaisille komponenteille ja konnektoreille tulee tehdä. Tällä tavoin varsinainen topologi voidaan pitää yksinkertaisempana, koska sen ei tarvitse huomioida esimerkiksi varsinaisten komponenttien hajautettua luonnetta. Lisäksi erilaisille järjestelmille suunniteltuja topologian ylläpitäjiä voidaan käyttää saman topologistin alla.

Topologinen reflektio toimii perinteisen laskennallisen reflektiokäsityksen mukaisesti: TA toimii topologisen reflektion perustasona, jota kuvataan topologisella kuvauksella. Reflektio tapahtuu topologistin välityksellä näiden kerrosten välillä. Reflektiivisen järjestelmän periaatteiden mukaisesti topologisia kuvauksia voi olla myös useassa eri kerroksessa, jolloin reflektointi tapahtuu näiden kerrosten välillä.



Kuva 4: Topologisen reflektion toteuttaminen

Topologian ylläpitäjä (TA) antaa topologin käyttöön seuraavat palvelut: uusien topologisten tyyppien ja niiden instanssien luonti, uusien sidosten luonti ja instanssin tuhoaminen (katso kuva 4). Metodikutsulla $define$ topologisti voi luoda esimerkiksi uuden komponenttityypin järjestelmään ja $instantiate$-kutsulla liittää komponentit osaksi topologiaa. Komponenttien portteja voidaan dynaamisesti sitoa konnektoreiden rooleihin samaisella $instantiate$-metodilla. Komponentteja, konnektoreita ja niiden välisiä liitoksia voidaan poistaa $destroy$-metodilla.

Järjestelmän topologinen reflektiomekanismi saadaan aikaan määrittelemällä topologistin käyttäytyminen. Topologisti määritellään meta-komponenttina (avainsanana $meta-component$ avainsanan $component$ sijaan), jolle määritellään tavallisen komponentin tavoin joukko portteja, alkutila sekä tilasiirtymät [CSST]. Toisin kuin tavallisilla, perustason komponenteilla, on metatason topologistikomponenteilla käytössään TA:n palvelut. Taulukossa 3 on erään topologistin kuvaus, jonka tehtävänä on tuplata järjestelmän komponenttien määrä tapahtuman $doubleComponents()$ sattuessa. Tapahtuma $doubleComponents()$ tulee järjestelmän strategistilta (katso seuraava luku), joka liitetään topologistiin tavallisella konnektorilla. Topologistin ja strategistin välinen yhteys tulee paremmin esiin alilu-

```
meta-component topologist {
 port double {
  initial_state: idle
  transitions:
   doubleComponents():
    idle -> core.instantiateComponents(
     components:
      unit{n+1...2n} : unit;),
     connectors:
      ... Määrittele konnektorit
     attachments:
      ... Sido komponentit ja konnektorit
    ) -> idle
 }
}
```

Taulukko 3: Komponenttien määrän tuplaava topologisti

vun 2.3 lopussa, jossa esitellään sekä strategisti-metakomponentti että metakomponentit sitova konnektori.

2.3 Strateginen reflektio

Järjestelmän strategia määritellään joukolla sääntöjä, jotka rajoittavat ja ohjaavat järjestelmän tapahtumia [CSST99]. Strategia määrittelee missä järjestyksessä ja mihin aikaan tietyt tapahtumat voivat laueta. Strategian ylläpidosta järjestelmässä vastaa strategian ylläpitäjä (*strategy actuator, SA*), joka annettujen sääntöjen pohjalta koordinoi järjestelmän kokonaistoimintaa. Strategian ylläpitäjä vuorovaikuttaa järjestelmään konnektoreiden kautta laukaisemalla tapahtumia konnektoreissa [CSST].

Taulukossa 4 on määritelty strategian muodostavien sääntöjen rakenne [CSST99, CSST]. Jokainen sääntö koostuu joukosta etu- ja jälkiehtoja sekä aktiviteetteja. Etuehtojen joukko on jaettu kahteen osajoukkoon: niihin jotka käsittelevät konnektoreiden tilaa sekä niihin jotka käsittelevät aikaa. Nämä ominaisuudet ovat kaikki, mitä SA:n täytyy tietää järjestelmästä.

Konnektoreiden tilaa käsittelevät etuehdot koostuvat joukosta *connectorEnabledEvents* alkioita. Nämä alkiot luettelevat tietyn nimetyn konnektorin laukaisukelpoiset aktiviteetit. Konnektorin tilalla ei siis tarkoiteta sen sisäistä tilaa, vaan ulospäin näkyvää tilaa. Aikaa käsittelevät ehdot koostuvat avainsanasta *time*, sitä seuraavasta operaattorista (esim. $<, =, >$) ja ajan määreestä *timeExpression*. Aikaa käsittelevät etuehdot määrittävät aikaan perustuvia rajoituksia ($time < 2:00pm$) tai tapahtumia ($time = 3:00pm$) [CSST].

Strategiasääntöjen aktiviteetit ovat konnektoreiden *connectorActions*-tyyppisiä tapahtumia. Koska yksi sääntö voi määritellä useita aktiviteetteja laukaistavaksi, voi niiden aktiviteettijoukon tulkinta olla vaikeaa: missä järjestyksessä ne pitää laukaista, ovatko ne rinnakkaisia ja erityisesti, onko laukaisujen joukko atominen. Nämä laukaisujen semanttisen ominaisuudet tulee määritellä järjestelmässä tarkasti jotta virheellisiltä tulkinnoilta vältytään. Aktiviteetit voidaan esimerkiksi laukaista peräkkäin yksi kerrallaan siten, että jokaisen aktiviteetin välissä tarkistetaan uudestaan esiehdot. Toinen vaihtoehto on suorittaa

```
rule <ruleName> {
 <rulePreconditions> -> <ruleActions> [-> <rulePostConditions>]
}

<rulePreconditions> ::= <stateOfConnectors>
<rulePreconditions> ::= <stateOfTime>
<rulePreconditions> ::= <stateOfConnectors>, <stateOfTime>

<stateOfConnectors> ::= <connectorEnabledEvents> {, connectorEnabledEvents>}
<connectorEnabledEvents> ::= <connectorName> <enabledCooperationEvent>
  {,<enabledCooperationEvent>}

<stateOfTime> ::= time <relop> <timeExpression>

<ruleActions> ::= <connectorAction> {,<connectorAction>}
<connectorAction> ::= <connectorName> <cooperationEvent>  {,<cooperationEvent>}

<rulePostConditions> ::= <stateOfConnectors>, <stateOfTime>
<rulePostConditions> ::= <stateOfConnectors>, <elapsedTime>
<elapsedTime> ::= timeElapsed <relop> <timeExpression>
```

Taulukko 4: Strategiasääntöjen muoto

```
<strategySpecification> ::=
 strategy <strategyName> {
  [<ruleDefinition> [{, <ruleDefinition>}]]
 }
```

Taulukko 5: Järjestelmän strategia

säännössä luetellut aktiviteetit peräkkäin ilman etuehtojen tarkistamista välissä [CSST].

Strategiasääntöjen jälkiehdot määrittelevät joukon totuusarvoisia lausekkeita, jotka liittyvät konnektoreiden tilaan ja aikaan. Etuehtoihin verrattuna jälkiehdot sisältävät yhden uuden lisäpredikaatin $elapsedTime$, joka mallintaa aktiviteettijoukon suorittamiseen kulunutta aikaa.

Järjestelmän strategia muodostetaan koostamalla joukko sääntöjä yhteen. Taulukossa 5 on määriteltynä järjestelmän strategian rakenne. Järjestelmän strategian ylläpitäjä (*strategy actuator, SA*) ylläpitää sääntöjä suorittamalla kutakin sääntöä vastaavat operaatiot varsinaisessa järjestemässä. Varsinainen strategian suunnittelu tapahtuu strategistin avustuksella, joka huolehtii siitä, että strategiametadatan ja varsinaisen järjestelmän välinen kausaalinen suhde säilyy konsistenttina.

Kuva 5 selvittää strategisen metadatan, strategistin, strategian ylläpitäjän (tästä lähtien SA) ja varsinaisen järjestelmän suhteita. Metadata, strategisti ja SA muodostavat ensimmäisen reflektiivisen tason mahdollisesti useampikerroksisessa reflektiivisessä tornissa. Järjestelmän arkkitehtuurista reflektiota ja käyttäytymistä säädellään SA:n tarjoamilla palveluilla. SA:n velvollisuuten on ylläpitää järjestelmän tilaa (konnektoreiden tilaa) sekä saattaa strategiasääntöjen mukaiset aktiviteetit tai tapahtumat suoritettavaksi [CSST].

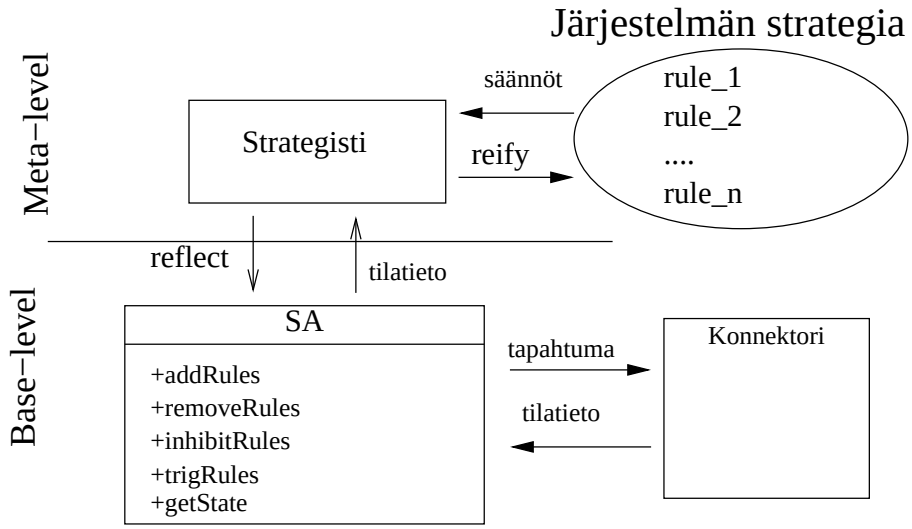
Strategisti voi säädellä järjestelmän toimintaa sääntöjä luomalla, tuhoamalla ja niiden suoritettavuutta säätelemällä. Uusien sääntöjen tuominen järjestelmään ja poistaminen tapahtuu $addRules$- ja $removeRules$-metodeja käyttäen. Sääntöjä voidaan myös väliaikaisesti ottaa pois käytöstä $inhibitRules$-metodilla. $trigRules$ määrittää joukon sääntöjä etuoi-

```
meta-component strategist {
 port double {
  initial_state: idle
  transitions:
   addRules():
    idle -> core.addRules(
     rule advance{n+1...2n} (
      -> INUnit{n+1...2n}.transfer(),
         OUT{n+1...2n}unit.transfer()
      -> )
    -> idle
 }
}
```

```
connector doublingCoordinator {
 roles: topologyDoubler, strategyDoubler
 initial_state: idle
 transitions:
  coordinateDoubling():
   idle -> topologyDoubler.doubleComponents(),
   strategyDoubler.addRule() -> idle -> idle
}
```

Taulukko 6: Strategisti sekä sitä ja topologistia ohjaava konnektori

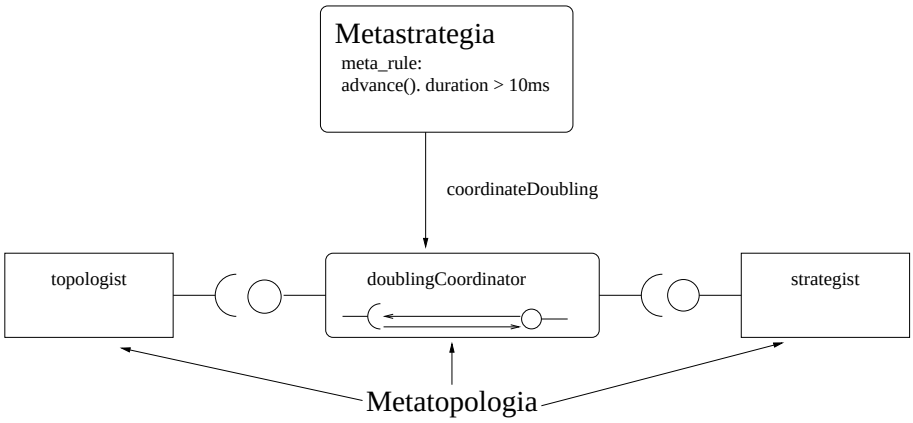
keutetuiksi, toisin sanoen, niiden määrittelemät aktiviteetit suoritetaan ennen muita sääntöjä, jos useampi sääntö on laukaisukelpoinen. Näiden palveluiden lisäksi strategisti voi kysyä järjestelmän tilaa SA:lta $getState$-metodilla.



Kuva 5: Strategisen reflektion toteuttaminen

Järjestelmän strategisti määritellään metakomponenttina, joka muuttaa järjestelmän tilaa SA:n kautta. Taulukossa 6 on esimerkki strategistista, joka lisää järjestelmään säännön $advance n + 1...2n$, jos järjestelmässä tulee vastaan tapahtuma $addRules()$. Strategistille täytyy määritellä myös sitä ohjaava "metastrategia", joka ohjaa strategin käyttäytymistä.

Kun sekä strategisti että topologisti on määritelty, niiden välinen yhteys ilmaistaan konnektorin avulla. Tästä muodostuu järjestelmän metatopologia, joka on mallinnettuna kuvassa 6. Strategistia ja topologistia ohjaava metasääntö tulkitaan siten, että jos perustason aktiviteetti $advance()$ kestää yli kymmenen millisekuntia, niin silloin $doublingCoordinator$-konnektorille aiheutetaan tapahtuma $CoordinateDoubling$. Tämä taas saa aikaan topologissa ja strategissa aktiviteettien laukeamiset, jotka lisäävät varsinaisen järjestelmän strategiaan ja topologiaan säännöt ja rakenteet, jotka tuplaavat käsiteltävien komponenttien määrän [CSST].



Kuva 6: Topologisen reflektion toteuttaminen

3 Kuvauskielien vaatimuksista

Arkkitehtuurinen reflektion toteutus- ja käyttökelpoisuus riippuvat käytettävästä mallinnuskielestä. Mallinnuskielen tulee olla sellainen, että kaikki järjestelmältä vaaditut ominaisuudet voidaan sillä ilmaista eli kielen tulee olla tarpeeksi ilmaisuvoimainen.

Konnektoripohjaisen arkkitehtuurinkuvauskielen tulisi ainakin pystyä esittämään yleisiä arkkitehtuureja, kuten proseduurikutsuja, jaettuja muuttujia ja tapahtumajulkaisuja *event broadcast*. Lisäksi kielellä pitäisi pystyä ilmaisemaan komponenttien välisiä monimutkaisia ja dynaamisia vuorovaikutussuhteita. Lopuksi, kielellä pitäisi pystyä ilmaisemaan konnektoreiden käytöksen pienijakoisia eroja, esimerkiksi kuinka jaetun muuttujaan käyttöön ja alustukseen suhtaudutaan eri konnektoriversioissa [AG94].

Toisaalta, tällaiset erot eivät ole kaikille komponenteille tärkeitä, joten “tarpeeksi yleisellä” tasolla samoin käyttäytyviä konnektoreita / komponentteja tulisi voida korvata keskenään. Erityisesti viimeinen vaatimus voi olla hyvinkin vaikea täyttää formaalilla tasolla, koska sellaisten käsitteiden kuten “*käyttäytyy lähes kuin*” tai “*on melkein sama kuin*” on erittäin vaikea formalisoida.

Arkkitehtuurinkuvauskielen ilmaisuvoiman lisäksi kielellä tulisi olla formaalin analyysin mahdollistavia ominaisuuksi. Kielen perusteella pitäisi pystyä ymmärtämään konnektoreiden ja komponenttien toiminnallisuus vaikka ne ovat irti ympäröivästä kontekstista [AG94]. Esimerkiksi putken toimintaperiaate pitäisi tulla sen spesifikaatiosta selväksi vaikka sen päitä ei olekaan kiinnitetty. Komponenttien ja konnektoreiden yhdisteiden ominaisuudet tulee olla verifioitavissa [AG94]. Tämä tarkoittaa sitä, että kuvauskielellä tulee olla formaali semantiikka ja lisäksi tulee olla jokin tapa kuvata malleja, joita yhdisteiden halutut ominaisuudet (esim. kahdenkeskinen poissulkeminen) voidaan ilmaista. Tyypillisesti kuvauskielen semantiikka perustuu jonkin formaalin kielen, kuten prosessialgebran tai logiikan, semantiikkaan ja mallit kuvataan esimerkiksi temporaalilogiikalla. Edellisessä luvussa tutustuimme järjestelmän strategian käsitteeseen. Strategia määritteli joukon ominaisuuksia, jotka itseasiassa redusoituvat tarkistettavien automaattien malleiksi.

Esimerkiksi Wright-kielen semantiikka perustuu rajattuun CSP-prosessialgebraan ja Darwin-kielen semantiikka voidaan johtaa π-kalkyylin avulla [AG97, MDEK95]. Wright-kielen eduksi voidaan laskea sen yksinkertaisempi semantiikka ja kevyempi verifiointivaativuus. Mobiilien järjestelmien kuvaamiseen tarkoitettuun π-kalkyyliin pohjautuvalla Darwin-kielellä voidaan ilmaista paremmin hajautettuja järjestelmiä, mutta π-kalkyyliin perustuvat verifioinnit ovat huomattavasti raskaampia.

Kielen ilmaisuvoiman ja analysointiominaisuuksien lisäksi arkkitehtuurisen reflektion perustana käytettävällä kuvauskielellä on vielä yksi vaatimus: siitä pitäisi olla mahdollista suoraan generoida tarvittavat toiminnalliset järjestelmän osat. Jos kielen ja järjestelmän ajoympäristön välillä ei ole suoraa yhteyttä, niin arkkitehtuurisen reflektion tukeminen tulisi hyvin hankalaksi. Erityisesti konnektoreiden tulisi olla ajonaikaisia yksilöitä, jotka eivät pelkästään mallinna vaan myös kontrolloivat ja valvovat komponenttien välistä vuorovaikutusta [DR97].

Arkkitehtuurikuvauksen pitää sisältää tarpeeksi informaatiota, jotta järjestelmän ajonaikaista käyttäytymistä voidaan hallita. Komponenttien porttien ja konnektoreiden roolien kuvauksista saadaan rajoitteet ja velvoitteet metodikutsujen keskinäiselle järjestykselle. Aktiviteettien etu- ja jälkiehtoja voidaan käyttää valvomaan esimerkiksi syötteiden oikeellisuutta tai operaatiokutsujen onnistumista. Jos jokin ehdoista ei ole voimassa, niin järjestelmän muita komponentteja tiedoitetaan siitä. Järjestelmän käyttäytymistä hallitaan ja valvotaan esimerkiksi monitoroivilla proxyilla, jotka ylläpitävät edustamansa komponentin keskustelun tilaa yllä. Jokainen tila sisältää kuvauksen siitä, mitkä ovat sen vaatimat ehdot (saatu etu- ja jälkiehdoista). Siirtymä tilasta toiseen tapahtuu ulkoisen tapahtuman, kuten metodikutsun, ilmetessä.

4 Yhteenveto

Tässä seminaaripaperissa tutustuttiin arkkitehtuurisen reflektioon ja sitä tukeviin käsitteisiin. Arkkitehtuurinen reflektio perustuu formaaliin arkkitehtuurin kuvaukseen, joka kuvaa järjestelmän rakenteen joukkona komponentteja ja niitä yhdistävinä komponentteina. Järjestelmän rakenne, eli topologia, kuvataan arkkitehtuurinkuvauskieliä käyttäen. Arkkitehtuurisen reflektion aikaansaavana voimana on sille määritelty strategia. Strategia määrittelee joukon sääntöjä, joita strategisti soveltaa. Strategistin päätösten mukaan järjestelmän rakennetta voidaan muuttaa tarpeen mukaan. Muutokset tapahtuvat konnektoreiden kautta.

Topologi on strategistin apuna reflektiivisen järjestelmän ylläpidossa. Tämän metakomponentin velvollisuuksiin kuuluu ylläpitää järjestelmän topologian kuvausta ja muuttaa järjestelmän topologiaa tarvittaessa.

Arkkitehtuurisen reflektion mahdollistaa järjestelmän kuvaaminen formaalilla kielellä. Kieleltä vaadittavia vaatimuksia ja reflektiomekanismien toteuttamista kuvauskielen pohjalta pohdittiin hieman luvussa 3.

Arkkitehtuurinen reflektio sopii hyvin dynaamisten järjestelmien toteutusperiaatteeksi tilanteisiin, joissa ohjelmiston lopullista rakennetta tai käyttäytymistä ei voida täysin en-

nalta määritellä. Reflektiomekanismien avulla voidaan toteuttaa esimerkiksi adaptiivisia, komponenttipohjaisia ohjelmistoja, jotka sopeutuvat ympäristön ja käyttäjien vaatimuksiin. Adaptiivisuus ilmenee nimenomaan koko järjestelmän ominaisuuksien muutoksena *programming-in-the-large*-periaatteen mukaisesti, toisin kuin perinteisissä adaptiivisissa järjestelmissä, joissa ohjelmiston sopeutuminen vaikuttaa tyypillisesti yhden komponentin konfiguraatioon.

Lähteet

AG94 Robert Allen ja David Garlan. Formalizing architectural connection. *Proceedings of the 16th international conference on Software engineering*, sivut 71–80. IEEE Computer Society Press, 1994.

AG97 Robert Allen ja David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997. URL `citeseer.nj.nec.com/allen97formal.html`.

BP Dusan Balek ja Frantisek Plasil. Software connectors: A hierarchical model. URL `citeseer.nj.nec.com/balek00software.html`.

CSST Walter Cazzola, Andrea Savigni, Andrea Sosio ja Francesco Tisato. Architectural reflection: Concepts, design, and evaluation. URL `citeseer.nj.nec.com/422304.html`.

CSST98 W. Cazzola, A. Savigni, A. Sosio ja F. Tisato. Architectural reflection: Bridging the gap between a running system and its architectural specification, 1998. URL `citeseer.nj.nec.com/cazzola98architectural.html`.

CSST99 Walter Cazzola, Andrea Savigni, Andrea Sosio ja Francesco Tisato. Rule-based strategic reflection: observing and modifying behaviour at the architectural level. *Automated Software Engineering, 1999. 14th IEEE International Conference on*, sivut 263–266. IEEE, lokakuu 1999.

DR97 Stephane Ducasse ja Tamar Richner. Executable connectors: towards reusable design elements. *Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, sivut 483–499. Springer-Verlag New York, Inc., 1997.

GAO95 David Garlan, Robert Allen ja John Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. *Proceedings of the 17th international conference on Software engineering*, sivut 179–185. ACM Press, 1995.

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

MDEK95 J. Magee, N. Dulay, S. Eisenbach ja J. Kramer. Specifying Distributed Software Architectures. W. Schafer ja P. Botella, toimittajat, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, osa 989, sivut 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin. URL `citeseer.nj.nec.com/magee94specifying.html`.

SDK+95 Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young ja Gregory Zelesnik. Abstractions for software architecture and tools to support them. *Software Engineering*, 21(4):314–335, 1995. URL `citeseer.nj.nec.com/shaw95abstractions.html`.

TSCS01 Francesco Tisato, Andrea Savigni, Walter Cazzola ja Andrea Sosio. Architectural reflection: Realising software architectures via reflective activities. *Lecture Notes in Computer Science*, 1999:102–??, 2001. URL `citeseer.nj.nec.com/tisato00architectural.html`.