

Monikielinen ohjelmointityökalu väliohjelmistojen tarpeisiin

Alexi Kallio

Helsinki 19.1.2006

Pro gradu

HELSINGIN YLIOPISTO

Tietojenkäsittelytieteen laitos

Monikielinen ohjelmointityökalu väliohjelmistojen tarpeisiin

Tietojenkäsittelytiede

Pro gradu

19.1.2006

84 sivua

Tutkielmassa kuvataan Tieteen tietotekniikan keskus CSC:ssä toteutetun Sysbio-projektin tarpeita varten toteutettu Flow-ohjelmointityökalu. Työkalun avulla voidaan upottaa Java-ohjelmointikielillä toteutettuihin hajautettuihin järjestelmiin yksinkertaisilla sovellusaluekohtaisilla ohjelmointikielillä toteutettuja komponentteja. Sovellusaluekohtaisten kielten avulla voidaan hajautetun järjestelmän toteutuksessa käyttää laajemman ryhmän – Sysbio-projektin tapauksessa bioinformaatikkojen – asiantuntemusta. Yksinkertaisten kielten avulla he voivat toteuttaa laajaankin järjestelmään heidän asiantuntemuksensa alaan kuuluvia komponentteja.

Flow-työkalu koostuu alustakomponentista, XML-käsittelyyn tarkoitettusta Xfunktionäisestä sekä metaohjelmointiin tarkoitettusta Cclang-kielestä. Xfunc on funktionaalinen kieli, jonka XML-ominaisuudet perustuvat joukkoon XML-solmuja käsitteleviä funktioita. Kieleen on yhdistetty imperatiivisia piirteitä niin kutsutun heijastusmenetelmän avulla. Metakieltä käytetään Flow-työkalussa alustan hallintaan, sovellusaluekohtaisten kielten avulla toteutettujen komponenttien kytkentään (putkitukseen) sekä aspektiohjelmointiin.

ACM Computing Classification System (CCS):

C.2.4 [Distributed Systems],

D.2.6 [Programming Environments],

D.3 [Programming Languages]

monikielinen ohjelmointi, väliohjelmistot, aspektiohjelmointi

Sisältö

| | | |
|----------|---|-----------|
| 1 | Johdanto | 1 |
| 2 | Ohjelmointityökalu upotettujen kielten käyttöön Java-sovelluksessa | 3 |
| 2.1 | Analyysiympäristö ja annotaatiokomponentti | 3 |
| 2.2 | Pienet ohjelmointikielet ja monikielisyys | 4 |
| 2.3 | Metaohjelmointi | 6 |
| 2.4 | Flow-ohjelmointityökalun arkkitehtuuri | 8 |
| 2.5 | Työkalun käyttötarkoitus | 12 |
| 3 | Flow-työkalun käyttäminen | 14 |
| 3.1 | Xfunc-kielen käyttö XML-dokumenttien käsittelyyn | 14 |
| 3.1.1 | XML-dokumenttien rakenne ja tehokas käsittely | 14 |
| 3.1.2 | Xfunc ja funktionaalisten ohjelmointikielten perusrakenteet . . | 19 |
| 3.1.3 | Kielen syntaksi | 22 |
| 3.1.4 | Tyyppijärjestelmä | 25 |
| 3.1.5 | Xfunc ja ohjelmointikielten laadulliset piirteet | 27 |
| 3.2 | Cclang-metakielen käyttö | 29 |
| 3.2.1 | Alustan tilan esittäminen logiikkaohjelmointikielessä ja tulk- kien monitorointi | 30 |
| 3.2.2 | Työkalun tilan muuttaminen Cclang-kielellä | 34 |
| 3.2.3 | Komponenttien kytkeminen | 36 |
| 3.2.4 | Aspektiohjelmointi Cclang-kielellä | 37 |
| 4 | Flow-työkalun toteutus | 41 |
| 4.1 | Flow-alustan toteutus | 41 |
| 4.1.1 | Alustan ohjelmointirajapinnan kuvaus | 43 |
| 4.1.2 | Alustan ja tulkkien rajapintojen toteutus | 46 |
| 4.1.3 | Tulkkien rinnakkaistus ja synkronointi | 49 |
| 4.2 | Xfunc-kielen toteutus | 50 |
| 4.2.1 | Funktioiden toteutus | 55 |
| 4.2.2 | Muuttujien ja tyyppien toteutus | 57 |
| 5 | Flow-työkalun arviointi | 61 |

| | | |
|----------|---|-----------|
| 5.1 | Työkalun kehittämisprosessi | 61 |
| 5.2 | Työkalun käyttö | 62 |
| 5.2.1 | Xfunc-kieli ja muut XML-työkalut | 62 |
| 5.2.2 | Cclang-metakieli alustan hallinnan ja monitoroinnin välineenä | 65 |
| 5.3 | Työkalun toteutus | 68 |
| 5.3.1 | Xfunc-kielen toteutus | 70 |
| 5.4 | Työkalun jatkokehitys | 75 |
| 6 | Yhteenveto | 77 |
| | Lähteet | 79 |

1 Johdanto

Ohjelmistotekniikan kehityksessä ehkä merkittävin piirre on ollut abstraktiotason jatkuva nousu. Ohjelmointimenetelmät ja ohjelmointikielten rakenteet ovat kehittyneet yhä kauemmas tietokonelaitteiston suorasta käskytyksestä. Korkeamman abstraktiotason menetelmillä saavutetaan parempi tuottavuus, kun ohjelmoija voi keskittyä varsinaisten tietojenkäsittelyongelmien ratkaisuun teknisten kysymysten sijaan.

Tieteen tietotekniikan keskus CSC:ssä toteutetun Sysbio-projektin yhteydessä havaittiin, että järjestelmän kehitystyötä voitaisiin tehostaa sovellusaluekohtaisten kielten avulla. Projektissa rakennettavan tieteellisen analyysiympäristön toteuttamisessa ja ylläpidossa olisi merkittävä etu, jos osa komponenteista voitaisiin toteuttaa tehtävää varten suunnitellulla kielellä yleiskäyttöisen Java-kielen sijaan. Tällöin komponentteja voidaan toteuttaa nopeammin, eikä niiden toteuttamiseen vaadita laajaa ohjelmointikokemusta. Sovellusaluekohtaisten kielten ongelma hajautetussa ympäristössä on, että yhden kielen avulla voidaan toteuttaa vain pieni osa monimutkaisen järjestelmän toiminnoista ja useiden kielten yhdistely on hankalaa. Ongelmaa tutkittiin toteuttamalla prototyyppi Flow-ohjelmointityökalusta, joka on tarkoitettu sovellusaluekohtaisten kielten käyttämiseen Java-pohjaisissa hajautetuissa järjestelmissä.

Flow-työkalun avulla voidaan upottaa hajautettuihin järjestelmiin sovellusaluekohtaisia ohjelmointikieliä sekä tehdä kielillä toteutettujen komponenttien metaohjelmointia. Ohjelmointikielten yhdistämistä yleisessä tapauksessa on tutkittu melko vähän, eivätkä useimmat metaohjelmoinnin menetelmät ole suoraan sovellettavissa kieliriippumattomaan metaohjelmointiin. Tutkielmassa toteutettavan Flow-ohjelmointityökalun prototyypin avulla kokeillaan, missä määrin olemassa olevia menetelmiä voidaan käyttää hyväksi.

Työkalun rakenne perustuu yhteiseen alustaan, jonka päällä käytetään sovellusaluekohtaisten kielten tulkkeja. Sovellusaluekohtaisten kielten etuna Javaan verrattuna on oppimisen ja käytön helppous. Tällöin komponentteja voidaan toteuttaa nopeammin, eikä niiden toteuttamiseen vaadita laajaa ohjelmointikokemusta. Flow-ohjelmointityökalu tukee myös metaohjelmointia. Sen avulla voidaan kytkeä (putkittaa) ja monitoroida alustassa toimivia komponentteja. Ulkoisen kytkemisen avulla komponenteista voidaan tehdä kapseloidumpia ja uudelleenkäytettävämpiä. Monito-

rinti on tärkeä ominaisuus hajautetuissa järjestelmissä, koska järjestelmän valvonta ja virheiden paikallistaminen vaatii komponenttien toiminnan seuraamista.

Tutkielmassa kuvataan Flow-työkalun käyttö, rakenne ja suunnitteluun liittyvät valinnat. Erityisesti keskitytään siihen, onko upotettujen tulkkien ajamiseen tarkoitettu alusta hyvä tapa toteuttaa monikielisyttä ja tukea metaohjelmoinnille. Kokeusten pohjalta voidaan arvioida, missä tilanteissa useiden sovellusaluekohtaisten ohjelmointikielten käyttö on suositeltavaa.

Flow-prototyyppeihin toteutettiin toiminnallisuus, jonka avulla voidaan rakentaa Sysbio-projektissa kehitettävään analyysiympäristöön kuuluva annotaatiotyökalu, joka hakee annettuun geeniin liittyvää tietoa Web Services -palveluista. Työkaluun toteutettiin yksi sovellusaluekohtainen kieli, XML-dokumenttien käsittelyyn tarkoitettu Xfunc, sekä yksi metakieli, komponenttien liittämiseen ja monitorointiin tarkoitettu Cclang. Koska tutkielmassa rajoitetaan yhteen komponenttikieleen, ei eri sovellusaluekohtaisten kielten integrointia kokeilla käytännössä.

Luvussa 2 esitellään Flow-ohjelmointityökalun taustalla olleet tarpeet, kehitysajatukset ja niiden pohjalta toteutetun työkalun korkean tason arkkitehtuuri. Luvussa 3 käsitellään työkalua matalammalta tasolta, työkalun käyttäjän näkökulmasta. Luvussa 4 esitellään työkalun toteutuksen merkittävimmät osat. Työkalun soveltuvuutta käyttötarkoitukseensa arvioidaan luvussa 5.

2 Ohjelmointityökalu upotettujen kielten käyttöön Java-sovelluksessa

Tieteen tietotekniikan keskus CSC:ssä toteutetaan nelivuotisessa Sysbio-projektissa analyysiympäristö DNA-mikrosirudatalle. Analyysiympäristö on toteutettu Java-ohjelmointikielellä, mutta se sisältää myös komponentteja, joiden toteuttamiseen yksinkertainen ja rajattuun käyttöön suunniteltu kieli olisi soveltuvampi. Luvussa tarkastellaan upotettujen kielten käyttöön liittyviä ongelmia analyysiympäristön rakenteen kannalta.

2.1 Analyysiympäristö ja annotaatiokomponentti

Analyysiympäristö on laaja ja monimutkainen hajautettu järjestelmä, joka on toteutettu viestinvälitysarkkitehtuuriin pohjautuen, Javan JMS-rajapintaa käyttäen. Järjestelmä muodostuu DNA-tutkijan tietokoneella toimivasta graafisesta asiakasohjelmistosta, joukosta superlaskentaympäristöön asennettuja analyysikomponentteja sekä suurikokoisesta mikrosirudataa sisältävästä tietokannasta. Järjestelmä käsittelee suuria laskentatöitä ja suuria datamääriä, mutta yhtäaikaisten käyttäjien ja pyyntöjen määrä on melko pieni, eikä järjestelmän jatkuva saavutettavuus ole kriittistä.

Analyysiympäristön toteuttamisessa käytettävät viestinvälitysarkkitehtuuri sekä Java-ohjelmointikieli ovat raskaita työvälineitä. Arkkitehtuuri on joustava, mutta sen huono puoli on, että uusien toimintojen kehittäminen on työläämpää kuin tiukemmin kytketyissä arkkitehtuureissa. Java on turvallisten ja vakaiden järjestelmien toteuttamiseen tarkoitettu kieli, jolla yksinkertaisten tehtävien toteuttaminen ei yleensä ole vaivatonta. Projektiin valitut tekniikat mahdollistavat joustavien, vakaiden ja turvallisten järjestelmien toteuttamisen, joka olikin valintojen takana oleva peruste.

Projektissa on tarvetta myös keveämmille menetelmille, joilla pystyttäisiin toteuttamaan kokeiluluontoisia sekä ei-kriittisiä tuotantotasoisia ominaisuuksia. Analyysiympäristön tulee tarjota annotaatio-ominaisuuksia, joilla haetaan Web Services (WS) -protokollien avulla geeneihin liittyvää tietoa verkossa tarjolla olevista palveluista. Bioinformatiikka ja erityisesti annotaatiopalvelut ovat hyvin nopeasti kehittyviä ja muuttuvia aloja, joten pitkässä projektissa on lähes mahdoton ennustaa etukäteen, millaisia palveluja projektin päättyessä on tarjolla. Palvelut ovat

useimmiten Web Services -protokollin perustuvia, muuttuvia, jossain määrin epäluotettavia, päällekkäisiä ja myös ristiriitaisia [NL05]. Annotaatioiden keräysmenetelmää joudutaan jatkuvasti – myös jossain tapauksissa järjestelmän suorituksen aikaisesti – sopeuttamaan sen hetkiseen palvelutarjontaan. Javan avulla voidaan toteuttaa oliopohjainen kirjasto, joka mahdollistaa annotaatiotoimintojen helpon toteutuksen, monitoroinnin ja ajonaikaisen muuttamisen. Jotta annotaatioiden koostamisen tapaa voitaisiin muuttaa ajonaikaisesti, kirjaston pitäisi esimerkiksi lukea koostamiseen tarvittavat operaatiot erillisestä asetustiedostosta tai käyttää Java-koodin ajonaikaista uudelleenlatausta. Monitoroinnin toteuttaminen vaatisi joko Java-tavukoodin instrumentointia tai sovelluskehystä, jossa oliot eivät kutsuisi toisiaan suoraan, vaan kehyksen välityksellä. Oliokirjastosta tulisi siis monimutkainen, mikä tekisi siitä vaikean käyttää ja sopimattoman erityisesti kokeiluluontoisten toiminnallisuuksien toteuttamiseen.

Järjestelmän jatkokehityksen kannalta olisi suotavaa, että bioinformaatikot pystyisivät mahdollisimman vapaasti kytkemään uusia annotaatiotoiminnallisuuksia osaksi analyysiympäristöä. Bioinformaatikot ovat erikoistuneet biologisen datan ohjelmalliseen käsittelyyn, joten yksinkertaisten ohjelmointityökalujen käyttö ei ole heille ongelma. Monimutkaisten Java-pohjaisten oliomallien ja sovelluskehysten käsittely sen sijaan on useimmiten liian vaativaa. Upottamalla sopiva ja yksinkertainen skriptikieli osaksi analyysiympäristöä voitaisiin järjestelmän annotaatiotoimintojen ylläpito tehdä helpommaksi ja antaa bioinformaatikkojen hoidettavaksi, koska he tuntevat annotaatiopalvelut parhaiten. Myös mikäli sovellusta kehittävät henkilöt vastaavat paremmin sovellusta käyttäviä henkilöjä, on todennäköistä, että myös sovellus vastaa paremmin käyttäjien tarpeita [KP05].

2.2 Pienet ohjelmointikielet ja monikielisyys

Viime vuosina on tullut tarjolle monenlaisia skriptikieliä, joita voidaan upottaa osaksi Java-sovelluksia. Ensimmäinen laajaa huomiota saanut upotettu skriptikieli oli BeanShell, jota ovat seuranneet muun muassa Jython, Sleep, Anvil ja Groovy. Skriptikieliä käytetään, koska ne mahdollistavat toimintojen toteuttamisen isäntäkieltä, tässä tapauksessa Javaa, yksinkertaisemmin ja nopeammin. Sysbio-projektin annotaatiotyökalu olisi voitu toteuttaa esimerkiksi Groovy-skriptikieltä käyttäen, toteuttaen Javalla ainoastaan muuhun järjestelmään JMS-rajapinnan kautta kytkeytyvä komponentti.

Annotaatiotyökalu ei ole kuitenkaan ainoa osa analyysiympäristössä, jossa kevyemmän ohjelmointikielen käyttö olisi perusteltua. Analyysikomponentit käärivät sisään-
sä suuren joukon olemassa olevia analyysiohjelmistoja. Ohjelmistot käyttävät erilaisia tiedostoformaatteja ja tarjoavat statustietoa laskennan etenemisestä eri tarkkuudella ja eri muodoissa. Analyysikomponentteja yhdistellessä joudutaan usein suorittamaan tiedon muuntamista muodosta toiseen. Koska analyysiympäristön tulisi sisältää parhaat tunnetut mikrosirudatan analyysimenetelmät ja koska analyysimenetelmät kehittyvät jatkuvasti, analyysikomponenttien mahdollisimman joustava ja vaivaton yhdistely on tärkeää. Skriptikielen avulla voitaisiin analyysikomponenttien kytkemistä nopeuttaa ja erityisesti helpottaa, jolloin analyysimenetelmiin perehtyneet bioinformatikot voisivat tehdä sen itse.

Kun hajautettuun järjestelmään tuodaan useita upotettuja skriptikieliä, kohdataan toisaalta opittavuuteen ja toisaalta järjestelmän rakenteeseen liittyviä ongelmia. Kielten opettelu vie aikaa, vaikkakin yksinkertaisempien kielten opettelu ei ole yhtä vaativaa kuin Javan kaltaisten kielten. Opittavuus on ongelma erityisesti silloin, kun ohjelmointityöhön osallistuu eri alojen asiantuntijoita, joiden pohjatiedot ohjelmointitekniikasta voivat olla heikkoja. Useiden kielten myötä järjestelmän rakenne monimutkaistuu, koska kielten toteutuksilla on erilaiset rajapinnat, joiden suunnittelun taustalla ovat erilaiset oletukset kielen käytöstä ja suhteesta isäntäkieleen. Lisäksi skriptikielisten ohjelmien integrointi toisella skriptikielellä toteutettuihin ohjelmiin on työlästä, koska liitos joudutaan toteuttamaan Java-välikerroksen avulla.

Kielten integroimisen lisäksi useissa olemassa olevissa skriptikielissä on analyysiympäristön kannalta se ongelma, että ne ovat yleiskäyttöisiä kieliä, vaikka niitä käytettäisiin vain rajattuihin tehtäviin. Skriptikielten intuitiivisuutta ja ilmaisuvoimaisuutta voitaisiin parantaa tekemällä niistä sovellusaluekohtaisia. Tällöin kielen syntaksi, rakenteet ja ominaisuudet voitaisiin suunnitella vain tietyn tyyppisiä tehtäviä varten, jolloin onnistunut kieli olisi yleiskäyttöisiä kieliä tehokkaampi työkalu rajatuissa tehtävissä [Ben86, RMHB88, vDK98]. Kielten etu on myöskin, että niiden avulla ohjelmat voivat käyttää hyväkseen sovellusaluekohtaisia idiomia ja merkintätapoja, jolloin niillä kirjoitetut ohjelmat ovat kompakteja ja silti helppoja ymmärtää, mikäli itse sovellusalue on tuttu. Paul Hudak toteaa, että abstraktioiden rakentaminen on tärkein menetelmä hyvien ohjelmien kirjoittamisessa ja että paras mahdollinen abstraktio on sovellusta varten suunniteltu ohjelmointikieli [Hud96]. Esimerkiksi bioinformatikot voivat ottaa sovellusaluekohtaisia kieliä vaivattomammin käyttöönsä. Vaikka ohjelmoijille uusien kielten oppiminen onkin melko vaivatonta,

eivät se ole yhtä vaivatonta bioinformaatikoille, vähäisemmän ohjelmointitekni-
seiden taustansa vuoksi.

Tuotantokäyttöä ajatellen skriptikielten ongelmana pidetään niiden puutteellista
turvallisuutta: tarkistuksia ei tehdä yhtä paljon kuin Javan kaltaisissa kielissä ja
virheiden käsittely on yleensä löyhempää. Sovellusaluekohtaisista kielistä voidaan
tehdä luotettavampia, koska monet operaatiot voidaan varmentaa sovellusalueen ta-
solla – yleiset virheet, kuten nollalla jako ja virheelliset muistiosoittimet, ovat ma-
talan tason virheitä, jotka voidaan välttää jos kieli sisältää pelkästään korkean ta-
son rakenteita. Lisäksi koska sovellusaluekohtaiset kielet ovat yleensä yleiskäyttöisiä
kieliä yksinkertaisempia, on kielen toteutuksen virheettömyys helpompi saavuttaa.

Sovellusaluekohtaisten kielten ongelma on, että niitä tarvitaan yleiskäyttöisiä skrip-
tikieliä useampia. Niinpä laajassa hajautetussa sovelluksessa voitaisiin hyötyä siitä,
että itsenäisten upotettujen tulkkien sijaan käytetään yhteistä sovelluskerrosta, jon-
ka päälle upotetut tulkit on toteutettu. Tällöin pohjalla oleva sovelluskerros eli alus-
ta mahdollistaa upotettujen kielten integroimisen keskenään ja tarjoaa yhtenäisen
rajapinnan isäntäkieleen.

2.3 Metaohjelmointi

Analyysiympäristössä upotettuja kieliä käytettäisiin järjestelmän useimmin muut-
tuvissa ulkoisissa rajapinnoissa. Tällöin niillä toteutettujen komponenttien monito-
rointi on tärkeää, koska esimerkiksi ulkopuolisia WS-palveluja käytettäessä muutok-
set tai ongelmat palvelussa aiheuttavat analyysiympäristön osittaisen toimimatto-
muuden. Tällaisten ominaisuuksien toteuttaminen sovellusaluekohtaisilla kielellä ei
ole järkevää, koska ne ovat teknisiä seikkoja, joita ei tulisi sotkea sovellusaluekoh-
taiseen toiminnallisuuteen. Monitorointi on yhteinen vaatimus kaikille upotetuille
komponenteille, joten hyvä ratkaisu mahdollistaisi saman monitorointilogiikan käy-
tön kaikkien upotettujen kielten kanssa.

Mikäli upotettujen kielten toteuttamiseen käytetään yhteistä sovelluskerrosta, on
kerrokseen helppo yhdistää tuki metaohjelmoinnille. Yleisesti metaohjelmointi tar-
koittaa ohjelmia käsittelevien ohjelmien toteuttamista. Näin ollen myös perinteiset
ohjelmointityökalut kuten kääntäjät ovat metaohjelmia. Tässä yhteydessä rajoitum-
me refleksiiviseen metaohjelmointiin, jolla tarkoitetaan itseään käsitteleviä ohjelmia.
Refleksiivisyys voi tarkoittaa ohjelman lähdekoodimuotoisen rakenteen käsittelyn li-
säksi ohjelman ajonaikaisten rakenteiden, kuten olioiden välisten kutsujen, käsitte-

lyä. Metaohjelmointimenetelmät ovat erittäin ilmaisuvoimaisia, mutta niitä on kritisoitu siitä, että niiden avulla tuotetut ohjelmat ovat vaikeita ymmärtää ja ylläpitää. Kritiikki pätee erityisesti itseään muokkaavaan koodiin. Refleksiivisyyden selkeyttä voidaan parantaa siten, että järjestelmän komponentit jaetaan kahteen ryhmään: ainoastaan metakomponentit käsittelevät reflektion avulla pelkästään tavallisia komponentteja.

Joissain tilanteissa metaohjelmoinnin avulla voidaan kuitenkin tuottaa yksinkertainen ja selkeä ratkaisu, kun taas perinteisellä ohjelmointityökalulla tuotettu ratkaisu olisi monimutkaisempi, työläämpi toteuttaa ja alttiimpi ohjelmoijan tekemille virheille. Esimerkiksi pyyntöjen käsittelyn seuraaminen on työläs toteuttaa ilman metaohjelmointia. Web Services -rajapintoja käsittelevään koodiin halutaan kytkeä verkko-ongelmia havaitseva toiminnallisuus, joka rekisteröi liian hitaasti vastaavat WS-palvelut ja jonka parametrit, kuten seurattavat järjestelmän osat, ovat säädettävissä ajonakaisesti. Ilman metaohjelmointia joudutaan WS-toteutukseen lisäämään seurantaominaisuus ja WS-koodia käyttävät järjestelmän osat joudutaan muokkaamaan siten, että ne kutsuva olio voidaan tunnistaa, jotta seurantakoodi pystyy tarkistamaan, tuleeko kyseistä järjestelmän osaa seurata. Metaohjelmoinnilla seuranta voidaan lisätä ilman muutoksia WS-toteutukseen tai kutsuja tekeviin järjestelmän osiin. Hajautettuihin järjestelmiin liittyy usein laajoja komponenttijoukkoja koskevia piirteitä, kuten pyyntöjen auktorisointi tai järjestelmän monitorointi [Lin03, s. 405 - 422], jotka voidaan toteuttaa modulaarisesti ja uudelleenkäytettävästi metaohjelmoinnin avulla [Sch03, FGM05].

Refleksiivistä metaohjelmointia käytetään erityisesti koodin punomiseen ja komponenttien kytkemiseen [EGK⁺99, BCRP98]. Koodin punomisella tarkoitetaan sitä, että koodin suoritukseen lisätään kutsuja aspektikoodiin. Kutsut voidaan lisätä joko muokkaamalla alkuperäistä koodia tai seuraamalla ohjelman komponenttien toisilleen välittämiä viestejä [Paw00]. Hajautettujen järjestelmien yhteydessä punontaa käytetään erityisesti järjestelmän toiminnan monitorointiin. Komponenttien kytkeminen, monitorointi ja koodin tuottaminen ovat ominaisuuksia, joita tarvittaisiin myös Sysbio-projektin analyysiympäristössä.

Varsinkin hajautettujen komponenttien kytkemiseen ja punomiseen on kehitetty paljon erilaisia ratkaisuja, sekä tieteellisen tutkimuksen että ohjelmistoteollisen tuotekehityksen piirissä. Hajautettujen järjestelmien yhteydessä instrumentointia on tutkittu avoimen sidonnan (englanniksi open binding) avulla [FBC⁺98]. Avoimet sidonnat ovat yhdistelmäolioita, joiden avulla voidaan käsitellä useita hajautetun

ympäristön rajapintoja. Tällä tavoin viittaukset toisiin olioihin ovat käsiteltävissä monipuolisemmin: viittauksen kohteita voidaan muuttaa ja viittauksia voidaan yhdistellä. Olioviittauksiin perustuva ratkaisu on sidottu olioparadigmaan, mikä tekee siitä vaikeasti yhdistettävän sovellusaluekohtaisiin kieliin, jotka yleensä perustuvat muihin paradigmoihin.

Myös ohjelmistoteollisuudessa kehitetyissä komponenttiarkkitehtuureissa on metaohjelmointiominaisuuksia. Enterprise JavaBeans (EJB) -komponenttitekniikka mahdollistaa instrumentoinnin rajatusti: komponentit voidaan kytkeä osaksi transaktioita ulkoisten määrittelyjen avulla, jolloin EJB-alusta kaappaa komponenttien välisiä viestejä hoitaakseen transaktiot. EJB ei ole löyhästi kytketty komponenttimalli, koska kutsuvalla komponentilla tulee olla viittaus kutsuttavaan komponenttiin. Asynkroninen viestinvälitystekniikka Java Message Service (JMS) mahdollistaa viestivien osapuolten kytkemisen ilman, että kummallakaan on viittausta toiseensa. Tällä tavoin voidaan toteuttaa komponenttien kytkeminen ulkopuolelta. Koska komponenttien välisiä viestejä voidaan käsitellä ennen niiden toimittamista vastaanottajalle, voidaan JMS-tekniikan avulla toteuttaa monia yleisesti metaohjelmointiin liitettyjä toiminnallisuuksia.

EJB- ja JMS-tekniikat on suunniteltu käytettäväksi suurten itsenäisten komponenttien kanssa. Sovellusaluekohtaisilla kielillä toteutetut komponentit vaatisivat kuitenkin hienojakoisempaa monitorointia. Kielet pohjautuvat sovellusaluekohtaisiin idiomiin, jotka eivät välttämättä vastaa hajautuksen tarpeiden mukaan tehtyä komponenttijakoa. Esimerkiksi XML-dokumentteja käsitellessä voidaan haluta seurata yhtä käsittelyvaihetta, mutta sovellusaluekohtaisella kielellä toteutetussa käsittelykomponentissa käsittelyvaiheet ovat sulautuneet yhteen ja yksittäisen vaiheen seuraaminen vaati komponentin toteutuksen matalan tason instrumentointia.

2.4 Flow-ohjelmointityökalun arkkitehtuuri

Ulkoisesti Flow-työkalu muistuttaa tietokantaa: se on palvelinsovellus, johon voidaan kytkeytyä joko komentopohjaisen liittymän (kuten SQL-liittymä tietokannoissa) tai Java-ohjelmointirajapinnan (kuten JDBC tietokannoissa) avulla. Työkalu on suunniteltu upotettavaksi Javalla toteutettuun hajautettuun sovellukseen. Nimi Flow valittiin neljästä syystä: työkalua käytetään hajautetuissa järjestelmissä toteuttamaan komponenttien välistä viestiliikennettä, itse alusta toimii tulkkien välisen vuorovaikutuksen hallinnoijana, työkalun tarkoitus on tukea sujuvaa ohjelmistokehitystä se-

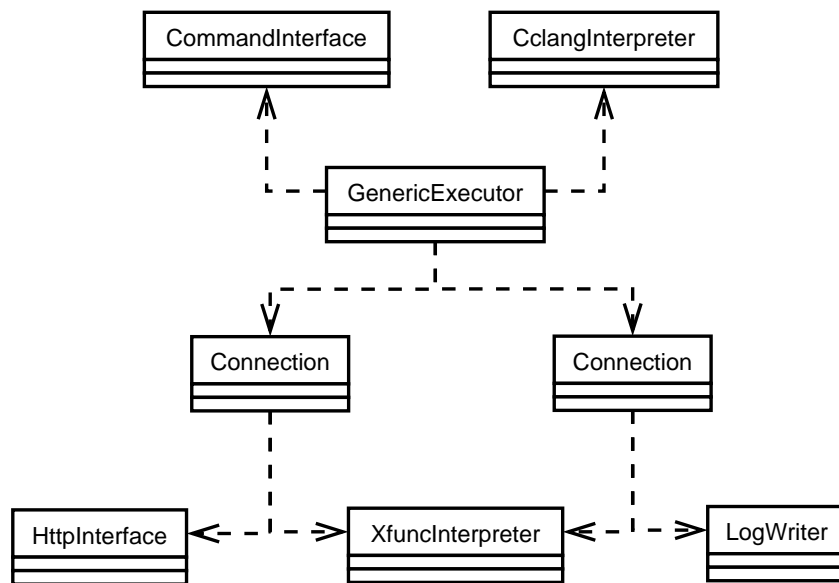
kä työkalun on tarkoitus tarjota yksittäisen ohjelmoijan näkökulmasta luonteva – eli sovellusaluelähtöinen – tapa rakentaa ohjelmia.

Flow-työkalu sisältää alustakomponentin, johon voidaan kytkeä ohjelmointikielten tulkkeja. Alustan päälle voidaan siis toteuttaa useita upotettuja sovellusaluekohtaisia kieliä. Alustaan kuuluu kaksi erilaista tulkkirajapintaa: rajapinnat sekä tavallisille että metakielen tulkeille. Alustaan voidaan kytkeä useita eri sovellusaluekohtaisten ohjelmointikielten tulkkeja, mutta ainoastaan yksi metakielen tulkki. Tulkkirajapintojen lisäksi alusta sisältää myös ulkoiset rajapinnat: komentokielipohjaisen liittymän sekä Java-ohjelmointirajapinnan.

Alustassa käytettävistä tavallisista ohjelmointikielistä käytetään aspektiohjelmoinnin terminologiasta lainattua nimitystä komponenttikieli. Komponenttikielellä toteutettuja lähdekielisiä ohjelmia ja niiden ajonaikaisia vastineita kutsutaan komponenteiksi. Tutkielman yhteydessä kehitettiin funktionaalinen XML-dokumenttien käsittelyyn tarkoitettu komponenttikieli Xfunc (nimi on lyhennelmä sanoista “XML” ja “functional”) ja toteutettiin sille Flow-alustassa toimiva tulkki. Xfunc-kieli suunniteltiin yleiseen XML-dokumenttien käsittelyyn, mutta projektin puitteissa sitä sovellettiin erityisesti XML-muotoisten geeniannotaatioiden käsittelyyn.

Tutkielmassa kehitettiin metaohjelmointikieli Cclang (nimi on lyhennelmä nimestä “component control language”) ja toteutettiin sille alustassa toimiva tulkki. Cclang-metakieltä käytetään alustan ajonaikaiseen hallintaan, komponenttien yhteenliittämiseen ja aspektiohjelmointiin. Kolmas metakielten yleinen käyttökohde, koodin generointi, jätetään myöhemmin kehitettäväksi. Kokonaisuudesta käytetään nimitystä Flow-työkalu. Ympäristö on rakenteeltaan abstrakti, koska sen tulee mahdollistaa eri kielellä toteutettujen ohjelmien suorittaminen saman alustaohjelman päällä.

Kuvassa 2.1 on esitetty esimerkki alustan kokoonpanosta. Flow-työkalun alustaosa perustuu *GenericExecutor-ytimeen*. *GenericExecutor* kapseloi sisäänsä kaikki työkaluun kuuluvat osat, siis erityisesti komponenttikielten tulkit ja metakielen tulkin. Kuvassa on metakielen tulkki *CclangInterpreter* ja komponenttitulkki *XfuncInterpreter*, joka on kytketty HTTP-rajapintaan ja järjestelmän lokiin. *GenericExecutor* tarjoaa Java-ohjelmointirajapinnan, jonka kautta muut hajautetun järjestelmän osat voivat käyttää työkalua. *GenericExecutor* tarjoaa myös komentokielipohjaisen liittymän, jonka kautta ohjelmoija voi hallita alustaa ajonaikaisesti. Molempien liittymien kautta voidaan käynnistää ja sammuttaa tulkkeja sekä muuttaa komponenttien välisiä kytkentöjä. Komentokieliliittymä on liittymä suoraan metakielen tulkkiin. *GenericExecutor* ei siis itse tue mitään ohjauskieltä. Alustaa voidaan käyttää

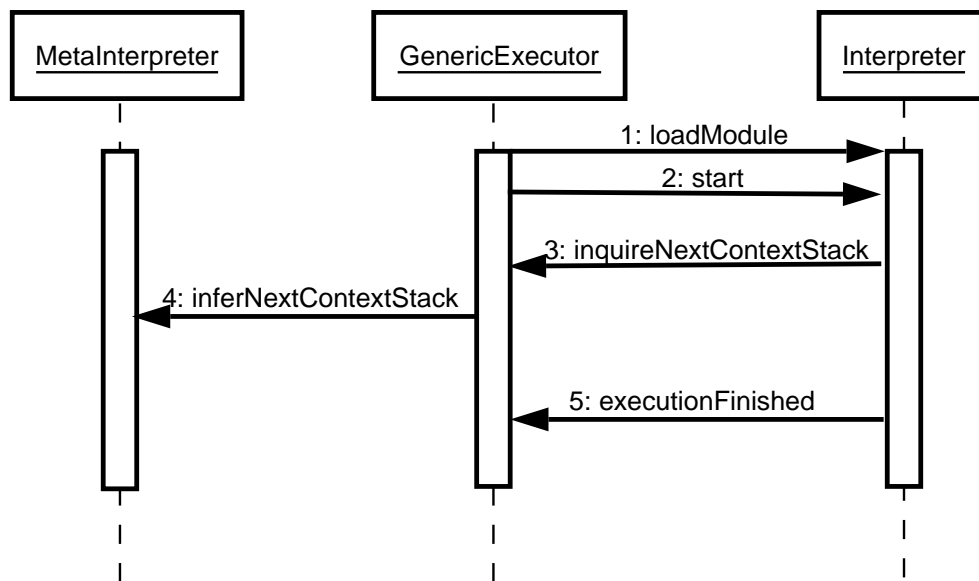


Kuva 2.1: Esimerkki alustan kokoonpanosta.

myös ilman kytkettyä metakielen tulkkia, mutta tällöin sitä voidaan hallita vain Java-ohjelmointirajapinnan kautta.

Yksi *GenericExecutor*-ydin kykenee ajamaan mielivaltaisen monia komponenttikielten tulkkia. Tarvittaessa ytimiä voidaan kuitenkin myös luoda useita. Tällöin eri *GenericExecutor*-ytimissä suoritettavia komponentteja ei voida yhdistää toisiinsa, mutta toisaalta joissain tilanteissa tämä voi olla toivottua tietoturvan vuoksi: *GenericExecutor* toimii siis virtuaalikoneen kaltaisena hiekkalaatikkona. Lisäksi koska metakieli ylläpitää tietorakenteissaan esitystä alustaan kytkettyjen komponenttien tilasta, voi suorituskykyisistä olla tarpeen jakaa eri järjestelmän osiin kuuluvia komponentteja eri metakielen tulkki-instanssien alaisuuteen. Tämä voidaan tehdä erillisten *GenericExecutor*-ytimien avulla.

Kuvassa 2.2 on esitetty Flow-työkalun osien välinen kutsusekvenssi komponentin tulkinnan aikana. *GenericExecutor* hallinnoi komponenttien ja niitä suorittavien tulkkien elinkaarta. Askeleessa 1 se lataa ohjelmakoodin, luo uuden instanssin tulkista ja askeleessa 2 käynnistää sen omassa säikeessään. Käynnistyksen yhteydessä tulkille välitetään *ExecutorCallback*-rajapinta, jonka kautta tulkki kutsuu *GenericExecutor*-ydintä jokaisen suoritusaskelen jälkeen. Suoritusaskelen määritelmä riippuu komponenttikielestä ja sen toteutuksesta. Se tulisi valita siten, että yhden askelen aikana ei tapahdu kahta tai useampaa merkittävää muutosta tulkin tilassa. Tällöin vain viimeisin muutoksista voitaisiin havaita instrumentoinnin kautta. *Xfunc*-komponenttikielessä suoritusaskel on yksi funktiokutsu.



Kuva 2.2: Alustan, komponenttikielen ja metakielen välinen kutsusekvenssi tulkin aikana.

Komponenttikielen tulkki kutsuu *GenericExecutor*-ydintä askeleessa 3, jotta ydin voi seurata komponenttien tilaa ja tarvittaessa tehdä siihen muutoksia. Tulkki esittää tilansa *FrameStack*-rajapinnan kautta. Rajapinta on abstraktio, jonka avulla voidaan esittää tulkin tila pinorakenteena itse komponenttikielestä riippumatta. Tila mallinnetaan pinossa olevina *Frame*-olioina, jotka vastaavat suorituksen kontekstia. Suorituksen kontekstin käytännön tulkinta riippuu kielestä ja sen toteutuksesta, mutta useimmissa tapauksissa se on funktiokutsu. Muussa tapauksessa alusta välittää askeleessa 4 tiedon komponentin tilasta metakielen tulkille, joka muuttaa tilaa, mikäli tulkkiin ladattu metaohjelma niin käsklee. Tämän jälkeen kutsu palautuu komponenttikielen tulkkiin, joka siirtyy suorittamaan seuraavaa suoritusaskelta, eli askelia 3 ja 4 toistetaan. Lopulta komponenttitulkki ilmoittaa alustalle askeleessa 5 suorituksen päättyneen. Tällöin alusta poistaa komponenttiin liittyvät merkinnät kirjanpidostaan ja päättää säikeen.

Metakielen tulkin ja alustan välinen rajapinta poikkeaa merkittävästi komponenttikielten rajapinnasta. Metakielen tulkkia ei suoriteta omassa säikeessään, vaan alusta kutsuu tulkkia komponenttitulkkien suoritusaskelten välillä ja ohjelmoijan syöttäessä komentoja komentokielisen liittymän kautta. Kutsussa alusta välittää *ExtendedExecutorCallback*-rajapinnan, joka laajentaa komponenttikielten käytössä olevaa *ExecutorCallback*-rajapintaa. Rajapinnan kautta metatulkki voi kutsuja

käsitellessään lukea ja muuttaa alustan, käynnissä olevien komponenttien ja komponenttien välisten kytkentöjen tilaa.

GenericExecutor ylläpitää myös tietoa komponenttien välisistä kytkennöistä. Komponenttikielten tulkeilla on käytössä tekstidataa kuljettavat syöte- ja tulostevirrat. Komponenttien avulla ei voida ohjata virtoja, vaan ne kytketään ulkopuolelta joko komentokielistä liittymää tai ohjelmointirajapintaa käyttäen. Komponenttien lisäksi alustaan on kytketty ulkoisina rajapintoina tiedostojärjestelmä sekä HTTP-yhteydet, jotka voidaan kytkeä syöte- ja tulostevirtoihin.

Liittämisen lisäksi metakieltä voidaan käyttää komponenttien punontaan. Esimerkiksi monitorointi voidaan toteuttaa niin, että metakieli tutkii komponentin tilaa esittävää tietorakennetta ja kutsuu toista komponenttia, mikäli tila vastaa annettuja ehtoja.

2.5 Työkalun käyttötarkoitus

Flow-työkalu on tarkoitettu käytettäväksi upotettuna Java-sovelluksiin. Sysbio-projektin yhteydessä työkalua tullaan käyttämään väliohjelmistoille ja sovellusintegraatiolle tyypillisiin tehtäviin: XML-muotoisen tiedon muuntamiseen ja koostamiseen sekä näiden operaatioiden monitorointiin. Työkalu on periaatteessa käytettävissä kaikenlaisien Java-sovellusten kanssa, vaikka sitä kehitetäänkin väliohjelmistojen tarpeita ajatellen. Tässä tutkielmassa väliohjelmistoilla tarkoitetaan vain Web Services -tekniikoihin perustuvia ohjelmistoja, jotka koostavat ja jalostavat muista WS-palveluista saatavaa tietoa. Laajasti ymmärrettynä väliohjelmistoilla voidaan tarkoittaa mitä tahansa sovelluksia, jotka tukevat kahden järjestelmän välistä viestintää [Lin03, s. 115 - 136]. Ehdottomat työkalun käyttöä rajoittavat piirteet ovat Java-sidonaisuus, korkeasta abstraktiotasosta johtuva ylimääräinen laskentakuorma sekä tarjolla olevien tulkkitoiteutusten valikoima, joka tällä hetkellä on suunnattu analyysiympäristön annotaatio toimintoihin eli XML-käsittelyyn ja komponenttien monitorointiin.

Työkalulla on kolme käyttäjäryhmää: uusia komponenttikieliä toteuttavat kehittäjät, metakieltä käyttävät ohjelmoijat ja komponenttikieliä käyttävät ohjelmoijat. Uusien komponenttikielten kehittäjät ovat pääasiassa tietojenkäsittelytieteen ja erityisesti ohjelmointitekniikan asiantuntijoita ja metakielen käyttäjät edistyneempiä ohjelmoijia tai ylläpitäjiä, jotka käyttävät metakieltä järjestelmän monitorointiin.

Komponenttikieliä käyttävät voivat olla mahdollisesti vähäisenkin ohjelmointiosaamisen omaavia oman sovellusalueensa tuntijoita.

Kehittämällä uusia komponenttikieliä voidaan luoda hyvin ilmaisuvoimaisia ja helposti opittavia abstraktioita laajemmalle käyttäjäkunnalle. Tällä tavoin kokeneimpien ohjelmoijien ongelmanratkaisu- ja mallinnusosaamista voidaan siirtää muille järjestelmän kehittäjille. Sovellusaluekohtaiset komponenttikielet voivat olla hyvin yksinkertaisia: esimerkiksi analyysikomponenttien hallintaan suunnitellun kielen ei tarvitse olla Turing-täydellinen ohjelmointikieli, vaan se voi olla omalla sovellusalueellaan hyödyllinen ilmankin täyttä laskennallista voimaa. Alusta mahdollistaa komponenttikielten integroinnin keskenään ja isäntäkieleen, joten yksinkertaisen kielen ilmaisuvoimaa voidaan tarvittaessa täydentää toisella kielellä.

Flow-työkalun avulla voidaan helpottaa järjestelmän ylläpitoa ja jatkokehitystä monella tavalla. Metaohjelmoinnin avulla voidaan monitoroida järjestelmää ja tehdä siihen muutoksia tarvittaessa ajonaikaisesti. Sovellusaluekohtaisten kielten avulla järjestelmän eri osia voivat ylläpitää kyseisen sovellusalueen asiantuntijat, ilman laajaa ohjelmointiosaamista. Helpottamalla kielten integrointia voidaan järjestelmään tuoda uusia komponenttikieliä vaivattomammin ja tarvittaessa käyttää rinnakkain saman kielen eri versioita.

3 Flow-työkalun käyttäminen

Flow-työkalun avulla ohjelmoija voi luoda sovellusaluekohtaisia kieliä hyväksikäyttäen tiettyyn tehtävään tarkoitettuja komponentteja. Komponentit toimivat Flow-alustaan kuuluvissa tulkeissa, joiden välistä kommunikaatiota alusta hoitaa. Kommunikaatio perustuu yhteyksiin, jotka kytkevät kaksi komponenttia toisiinsa. Yhteyksiä ja muita alustan hallinnoimia objekteja voidaan käsitellä alustaan kuuluvan Cclang-metaohjelmointikielen avulla. Metaohjelmointikielen avulla voidaan myös punoa komponenttien suoritukseen ulkopuolisia toiminnallisuuksia eli tehdä aspektiohjelmointia.

3.1 Xfunc-kielen käyttö XML-dokumenttien käsittelyyn

XML-dokumentit ovat tavallisia tietorakenteita monipuolisempia, joten niiden tehokas käsittely vaatii korkeamman tason rakenteita kuin mitä perinteiset ohjelmointikieliset sisältävät tietorakenteiden käsittelyä varten. Xfunc-kieli rakentuu XML-dokumenttien puurakenteen varaan: kieli on tarkoitettu soveltuvaksi erityisesti puun solmujen valintaan ja muokkaamiseen. Koska Xfunc-kielen käyttöalue on suppea, on se pyritty pitämään hyvin yksinkertaisena, sekä sitä käyttävän sovellusohjelmoijan että kieltä kehittävän tai laajentavan työkaluohjelmoijan näkökulmasta.

3.1.1 XML-dokumenttien rakenne ja tehokas käsittely

XML on SGML-kuvauskielen pohjalta kehitetty yksinkertainen metakieli, joka on tarkoitettu yleiskäyttöiseksi tiedon esittämisen muodoksi [W3Ca]. XML-tekniikoiden avulla voidaan luoda uusia kuvauskieliä. Kuvauskielten määrittelyyn käytetään XML-skeemaa, jonka avulla voidaan tarkemmin määritellä XML-dokumenttien rakennetta [W3Ca, W3Cb, W3Cc]. Useimmat uudet kuvauskielet perustuvatkin XML-tekniikkaan, kuten biologisen datan kuvaamiseen tarkoitettut BSML¹ ja MAGE-ML². Myös Web Services -järjestelmien rakentamiseen käytetyt SOAP³ ja WSDL⁴ ovat XML-pohjaisia.

XML-muotoinen data koostuu dokumenteista, jotka soveltuvat myös ihmisen luettaviksi, vaikka ovatkin tarkoitettu pääasiassa ohjelmallista käsittelyä varten. Doku-

¹<http://www.labbook.com>

²<http://www.mged.org>

³<http://www.w3.org/TR/soap>

⁴<http://www.w3.org/TR/wsdl>

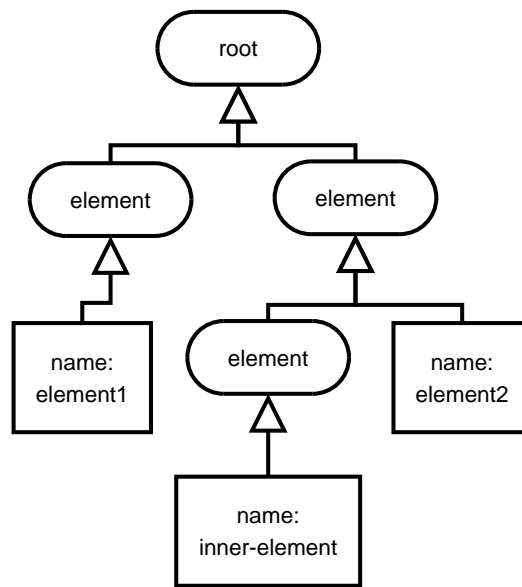
mentit muodostuvat elementeistä, jotka voivat olla sisäkkäisiä. Dokumentilla on yksi juurielementti, jonka sisällä ovat kaikki muut elementit. XML-dokumentti on siis rakenteeltaan elementtien muodostama puu. Elementeillä voi lisäksi olla attribuutteja, jotka ovat elementtiin kytkettyjä nimi-arvo -pareja. Attribuutteja ja elementtejä kutsutaan yhdessä solmuiksi. Esimerkki XML-dokumentista on esitetty alla ja sitä vastaava puurakenne on kuvassa 3.1.

```
<?xml version="1.0" ?>
<root>
  <element name="element1">
  </element>
  <element name="element2">
    <element name="inner-element">
    </element>
  </element>
</root>
```

Esimerkkidokumentti sisältää juurielementin nimeltään *root*, jolla on kaksi lapsisolmua. Molemmat lapsisolmut ovat elementtejä, joilla on attribuutti *name*. Lisäksi jälkimmäisellä lapsella on vielä oma lapsielementti. XML-dokumenteissa usein samantyyppiset elementit toistuvat eri syvyyksillä, kuten esimerkissä *element*-tyyppiset elementit toistuvat. XML-dokumentit ovatkin siis usein perinteisiä tietorakenteita löyhempirakenteisia.

XML-dokumenttien käsittely on tehtävä, johon sovellusaluekohtaisen kielen kehittäminen on perusteltua. Vaikkakin useat XML-työkalut, kuten DOM-rajapinta, perustuvat yleiskäyttöiseen ohjelmointikieleen, on myös XML-spesifejä käsittelykieliä kehitetty useita. Ne ovat yleensä joko imperatiivisia tai funktionaalisia. Molemmilla lähestymistavoilla on omat heikkoutensa. Imperatiiviset työkalut soveltuvat hyvin tilanteisiin, joissa XML-dokumenttiin tehdään pieniä paikallisia muutoksia. Tällöin dokumentista voidaan poimia halutut elementit ja tehdä niihin muutokset suoraan. Imperatiivinen lähestymistapa soveltuu kuitenkin huonosti koko dokumentin uudelleen muotoiluun, kuten esimerkiksi dokumentin muuntamiseen toisen skeeman mukaiseen muotoon.

Funktionaalisten työkalujen ongelma sen sijaan on, että pienten muutosten tekeminen dokumenttiin on työlästä [Sco00, s. 622 - 623]. Esimerkiksi yhden elementin nimen vaihtaminen on hankalaa, koska silloin joudutaan dokumentista kopioimaan



Kuva 3.1: Esimerkkidokumenttia vastaava puurakenne.

myös rekursiivisesti juuresta alkaen kaikki muut dokumentin elementit attribuutteineen. Imperatiivisella työkalulla yhden elementin muuttaminen on hyvin suoraviivaista, koska mitään ei tarvitse kopioida.

Pienten muutosten tekemisen hankaluus on erikoistapaus yleisemmästä ongelmasta funktionaalisisissa kielissä. Funktionaalinen tietorakenteiden käsittely perustuu funktioihin, jotka ottavat syötteenä alkuperäisen rakenteen ja palauttavat uuden rakenteen. Funktionaaliset kielet eivät kannusta ohjelmoimaan sivuvaikutusten avulla, jotta tietorakenteen muuttaminen paikallaan eli sitä kopioimatta on. Yksinkertaisten funktionaalista tietorakenteiden – kuten listojen – tapauksessa kyse on vain suorituskykyongelmasta: kopiointi ohjelmointityylinä on yksinkertainen ja hallittava. XML-dokumenttien rakenne on kuitenkin monimutkaisempi, jolloin myös kopiointi on monimutkaisempaa ja virhealttiimpaa. Monimutkaisuus voidaan piilottaa funktiokirjaston avulla, mutta tällöin kirjaston laajentaminen on hankalaa.

Xfunc-kielen suunnitteluratkaisut

Xfunc-kielessä yhdistetään imperatiivisten ja funktionaalisten XML-työkalujen hyvät piirteet. Tarkoitus ei ollut kehittää uusia menetelmiä XML-dokumenttien käsittelyyn, vaan suunnitella kieli kokonaan XML-dokumenttien käsittelyä varten, jo tunnettuja käsittelymenetelmiä käyttäen.

Xfunc-kieli suunniteltiin täydeksi – vaikkakin yksinkertaiseksi – ohjelmointikieliksi. Valinnan taustalla oli merkittävin XML:n muokkausta varten suunniteltu kieli, W3C:n piirissä kehitetty XSLT [W3Ce]. Se on suunniteltu alunperin XSL-kokonaisuuden osaksi eikä niinkään itsenäiseksi ohjelmointityökaluksi. Niinpä XSLT onkin muotoilukielen ja ohjelmointikielen välimuoto, mutta erityisesti ohjelmointiominaisuuksiltaan hyvin yksinkertainen ja moniin tarkoituksiin riittämätön. Kielen ilmaisuvoima on rajoittunut, sitä on hyvin vaikea laajentaa ja tuotettuja tyyllisivuja on vaikea modularisoida.

Pyrkimyksenä oli tuottaa XSLT:n kaltainen kieli, jonka ohjelmointiominaisuudet ovat monipuolisemmat ja jonka syntaksi ei pohjautu XML-syntaksiin. XML-syntaksin päälle rakennetusta ohjelmointikielen syntaksista tulee kohtuullisen monimutkaista ja hidasta kirjoittaa, mikä on vastoin sovellusaluekohtaisten kielten tavoitteita. Sama suunnittelupäätös on tehty W3C:n XQuery-kielessä [W3Cd].

Xfunc on täysi funktionaalinen ohjelmointikieli. Myös XSLT on jossain määrin funktionaalinen, esimerkiksi sen *template*-mekanismi sisältää monia funktionaalisesta ohjelmoinnista lainattuja piirteitä. Kuitenkaan XSLT ei kielen rakenteiden tasolla tue korkeamman kertaluvun funktioita, joten se ei tue aitoa funktionaalista ohjelmointia⁵. Xfunc-kielessä funktionaalisia ominaisuuksia ei ole pyritty piilottamaan. XML-dokumentteja voidaan käsitellä puina ja näin ollen rekursio on erittäin käyttökelpoinen menetelmä. Myös monia muita funktionaalisia kieliä on sovellettu XML:n käsittelyyn. Esimerkiksi Malcolm Wallace ja Colin Runciman ovat esitelleet generisiin kombinaattoreihin perustuvan tavan käsitellä XML-dokumentteja Haskell-ohjelmointikielillä [WR99] ja Sven Van Caekenberghe on esitellyt Common Lisp-kielillä toteutetun XML:n käsittelykielen S-XML⁶. Tutkimuksissa on todettu funktionaalisen paradigman soveltuvan hyvin XML-dokumentteja käsitteleviin ohjelmiin. Ratkaisuissa ei kuitenkaan keskitytty kopiointiin liittyviin ongelmiin.

Funktionaalisen paradigman ongelmien kiertämiseksi Xfunc-kieltä laajennettiin imperatiiviseen suuntaan. Laajennuksen avulla pyrittiin helpottamaan pienten muutosten tekoa. Laajennusta kutsutaan heijastettujen muutosten menetelmäksi, lyhyemmin heijastukseksi. Sen avulla voidaan muokatut elementit heijastaa takaisin siihen dokumenttiin, josta ne on alunperin poimittu. Esimerkiksi mikäli halutaan muokata dokumentin tiettyä elementtien osajoukkoa, voidaan elementtien poiminta tehdä

⁵XSLT:n päälle voidaan rakentaa myös korkeamman kertaluvun funktioita, kuten FXSL-kirjasto todistaa: <http://fxsl.sourceforge.net/>. Mielestäni kuitenkin ratkaisu on aivan liian monimutkainen mihinkään käytännölliseen tehtävään.

⁶<http://common-lisp.net/project/s-xml/>

funktionaalisesti valitsijafunktion avulla. Elementtejä käsitellään funktioilla, jotka palauttavat niistä muutetun kopion. Muutetut elementit heijastetaan takaisin alkuperäiseen dokumenttiin, jolloin dokumentin muita elementtejä ei tarvitse erikseen kopioida. Esimerkiksi luvussa 3.1.1 esitetystä dokumentista voidaan poimia kaikki juurielementin lapsielementit, jolloin saadaan alla esitetyt elementit.

```
<element name="element1">
</element>
```

```
<element name="element2">
  <element name="inner-element">
  </element>
</element>
```

Nämä elementit voidaan esimerkiksi käsitellä funktiolla, joka palauttaa uudelleennimetyn kopion alkuperäisestä elementistä.

```
<renamed-element name="element1">
</renamed-element>
```

```
<renamed-element name="element2">
  <element name="inner-element">
  </element>
</renamed-element>
```

Kun muutetut elementit heijastetaan takaisin alkuperäiseen dokumenttiin, saadaan muokattu dokumentti.

```
<?xml version="1.0" ?>
<root>
  <renamed-element name="element1">
  </renamed-element>
  <renamed-element name="element2">
    <element name="inner-element">
    </element>
```

```
</renamed-element>
</root>
```

Heijastus muistuttaa menetelmänä jossain määrin versionhallintajärjestelmää: alkuperäisestä datasta haetaan oma kopio, jota muokataan ja joka sen jälkeen viedään takaisin järjestelmään. Menetelmän etu on, että ohjelmoijan ei tarvitse kopioida tai ylipäänsä huomioida alkuperäisen dokumentin muuta sisältöä millään tavalla. Näin muutokset suorittava komponentti on helpommin sovellettavissa erityyppisten dokumenttien käsittelyyn.

3.1.2 Xfunc ja funktionaalisten ohjelmointikielten perusrakenteet

Xfunc-kieli suunniteltiin XML-käsittelyyn soveltuvaksi kieleksi, joka voidaan Flowtyökalun avulla upottaa Java-sovelluksiin. Kielen avulla voidaan yksinkertaistaa monia XML-käsittelytehtäviä – esimerkkinä dokumenttien lataaminen ja yhdistäminen.

```
(define $document
  (create-document "merged-document"))
(define $part1
  (load-xml "examples/example1.xml"))
(define $part2
  (load-xml "examples/example2.xml"))
(append-children
  $document $part1 $part2)
```

Esimerkissä luodaan uusi dokumentti, jonka juurielementin nimeksi tulee *merged-document*. Dokumentti sidotaan nimeen *\$document*. Sen jälkeen ladataan kaksi XML-dokumenttia tiedostoista ja palautetaan dokumentti, jossa ladatut dokumentit on lisätty luodun dokumentin juurielementin sisään. Funktio *append-children* ottaa ensimmäiseksi parametriksi dokumentin ja palauttaa muutetun dokumentin, johon on lisätty muiden parametrien – joita voi olla mielivaltaisen monta – sisältämät dokumentit. Xfuncin syntaksi muistuttaa Scheme-kieltä, erityisesti funktiokutsut merkitään samalla tavoin: koko kutsu kirjoitetaan sulkeiden sisään, funktion nimi on ensimmäisenä ja parametrit sen perässä. Muuttujien nimet aloitetaan \$-merkillä

sekä niitä määriteltäessä että niihin viitattaessa, kuten Perl- ja PHP-skriptikielissä. Xfuncissa käytetään aina \$-merkkiä, muuttujan tyyppistä huolimatta.

Dokumenttien lataaminen ja yhdistäminen voitaisiin tehdä yksinkertaisemmin ilman välivaiheita seuraavasti.

```
(append-children
  (create-document "merged-document")
  (load-xml "examples/example1.xml")
  (load-xml "examples/example2.xml"))
```

Esimerkkiohjelma on yksinkertaistettuna *append-children*-funktion kutsu. Välituloksia ei tallenneta muuttujiin, vaan ne annetaan suoraan parametrina *append-children*-funktiolle.

Esimerkin osadokumenttien tiedostonimet noudattavat yksinkertaista sääntöä: vain nimiosan lopussa oleva numero kasvaa, muuten nimi säilyy samana. Nimien toistaminen voidaan välttää kapseloimalla tiedostonimen luonti uudeksi funktioksi seuraavasti.

```
(define $load-part
  (function ($number)
    (load-xml (concat "examples/example" $number ".xml"))))
(append-children
  (create-document "merged-document")
  ($load-part "1")
  ($load-part "2"))
```

Esimerkkiohjelmassa määritellään funktio, joka ottaa parametrikseen numeron ja yhdistää siihen *concat*-funktioilla tiedostonimen alku- ja loppuosan, lataa luodun tiedostonimen avulla XML-dokumentin sekä palauttaa sen. Funktio sidotaan nimeen *\$load-part*. Edellisessä esimerkkiohjelmassa olleet kovakoodatut tiedostojen nimet voidaan nyt korvata kutsuilla luotuun funktioon.

Tarkastellaan vielä esimerkin funktioiden käyttöä. Funktion *\$load-part* määrittelyssä oleva parametri *\$number* on muodollinen parametri. Kun funktiota kutsutaan, sijoitetaan kutsussa oleva konkreettinen parametri – “1” tai “2” – muodolliseen parametriin *\$number*. Ohjelmointikielissä parametrinvälitys tehdään yleensä viitteen

tai arvon perusteella [Seb01, s. 358 - 377]. Xfunc-kielessä parametri välitetään arvon perusteella, eli parametrina olevasta arvosta tehdään kopio, joka on funktion käytössä. Tällöin funktion tekemät muutokset parametrina saatuun arvoon eivät näy funktion ulkopuolelle. Funktionaaliossa kielessä tämä on perusteltua, koska funktiot ovat kielen tärkein rakenne ja tällä tavoin niitä voidaan käyttää ohjelman modularisointiin. Mikäli kutsuttu funktio saisi parametrin viitteen perusteella, pystyisi se muuttamaan kutsujan näkemää arvoa ja näin ollen kutsuja olisi riippuvaisempi funktion toiminnasta. Puhtaasti funktionaaliseen paradigmaan tällainen menettely ei sovellu, koska funktion suorituksella olisi tällöin sivuvaikutuksia.

Funktiolle välitettävä parametri ei välttämättä ole arvo, vaan se voi olla lauseke, jonka evaluointi eli laskeminen tuottaa konkreettisen parametrin arvon. Esimerkiksi lausekkeessa $(+ 1 2 (* 2 2))$ annetaan yhteenlaskufunktiolle kolmantena parametrina lauseke, kutsu kertolaskufunktioon. Lausekkeen laskeminen voidaan tehdä joko sovellusjärjestyksessä eli funktion kutsuhetkellä tai normaalijärjestyksessä eli vasta kun parametria käytetään [Sco00, s. 604 -609]. Xfunc-kielessä laskeminen tehdään normaalijärjestyksessä, koska tällä tavoin funktioiden avulla voidaan luoda uusia ohjausrakenteita – ohjausrakenteet on toteutettu kielen sisäisinä funktioina. Esimerkiksi Xfunc-kielen ehtorakenne *if* on itse asiassa funktio, joka tutkii ensimmäisen parametrin totuusarvon ja palauttaa toisen tai kolmannen parametrin arvon riippuen ensimmäisen parametrin arvosta. Lausekkeen $if(true), 1, (f x)$ arvoksi tulee 1, lausekkeen $(f x)$ arvosta huolimatta. Parametrien laskentajärjestyksellä voi olla merkitystä ohjelman oikeellisuudelle. Alla oleva esimerkki ei toimi, mikäli käytetään sovellusjärjestyksessä tapahtuvaa parametrien laskemista. Kaikkien parametrien laskeminen ennen *if*-funktion rungon suoritusta aiheuttaisi nollalla jaon.

```
(define $value 0)
(if (equals $value 0)
    0
    (/ 2 $value))
```

Normaalijärjestyksellä voidaan lisäksi välttää tarpeettoman arvon laskeminen. Ylimääräisen laskemisen välttäminen on hyödyllistä erityisesti siksi, että monet väliohjelmistojen operaatiot – kuten SOAP-pyyntöt – ovat hyvin raskaita, jolloin turhien operaatioiden optimointi pois nopeuttaa ohjelman suoritusta merkittävästi. Vastaa-avat optimoinnit joudutaan imperatiivisella kielellä toteuttamaan itse, mikä monimutkaistaa ohjelmia.

Ohjausrakenteiden toteuttamiseen tarvitaan lisäksi toinen funktionaalisille kielille tyypillinen piirre, häntärekursion optimointi. Häntärekursiolla tarkoitetaan rekursiota, jossa funktio kutsuu suorituksen päätteeksi itseään. Tällöin funktiokutsut eivät haaraudu, eikä funktion paikallisia muuttujia enää tarvitse säilyttää. Näin ollen funktion kutsuessaan itseään voidaan uusi aktivaatietietue kirjoittaa pinon päällimmäisen tietueen tilalle. Optimoitu häntärekursio suoritetaan silmukkana, joka ei kasvata aktivaatietuepinoa. Rekursio on funktionaalisten kielten tärkein kontrollirakenne [McC60]. Niinpä funktionaalisissa kielissä on tyypillistä käyttää häntärekursiota silloin, kun imperatiivisissa kielissä käytettäisiin silmukkarakennetta [Kam90, s. 25]. Häntärekursio olisi perusteltu ominaisuus myös Xfunc-kieleen, mutta sitä ei tämän tutkielman puitteissa ehditty toteuttamaan.

3.1.3 Kielen syntaksi

Xfuncin yksinkertainen syntaksi on esitetty alla kontekstittoman kieliopin uudelleenkirjoitussääntöinä [Par66]. Aloitussymboli on *Program* ja päätesymbolivakiot (Xfunc-ohjelmassa olevat merkit) on merkitty paksunnettuna. Symboli *string* tarkoittaa vapaata merkkijonoa ja *number* numeroarvoa.

```

Program → Statement+
Statement → Definition | Expression
Definition → ( define Variable Expression )
Expression → Lambda | FunctionApplication | Variable | Value
FunctionApplication → ( Function Expression* )
Function → Variable | BuiltinFunction
Variable → $ string
BuiltinFunction → string
Value → number | " string "
Lambda → ( function Arglist Expression )
Arglist → ( Variable* )

```

Xfunc-ohjelma koostuu yhdestä tai useammasta lauseesta, jotka joko määrittelevät uuden muuttujan tai ovat arvon palauttavia lausekkeita. Lausekkeita ovat funktion luova lambdalauske, funktion kutsu, viittaus muuttujaan tai vakioarvo.

Xfuncin ohjausrakenteet ovat yksinkertaiset. Muuttujat määritellään *define*-avainsanan avulla. Kaikki muuttujien nimet tulee aloittaa \$-merkillä. Funktioita

voidaan luoda *function*-avainsanalla, joka vastaa useiden kielten lambdarakennetta. Funktion luonnin yhteydessä annetaan lista muodollisista parametreista sekä funktion runko. Jotta luotua funktioita voidaan käyttää, se tulee sitoa nimeen joko antamalla se funktiokutsun parametriksi tai määrittelemällä muuttujaksi. Alla on esimerkki funktion luonnista, muuttujaksi määrittelystä ja käytöstä muuttujan kautta.

```
(define $tuplaa
  (function ($luku)
    (* 2 $luku)))
($tuplaa 4)
```

Funktio luodaan *function*-rakenteella, sidotaan nimeen *\$tuplaa* ja kutsutaan nimeä käyttäen. Käyttäjän funktion kutsu eroaa primitiivifunktion kutsusta siinä, että viite käyttäjän funktioon on muuttujaviite, joka aloitetaan *\$*-merkillä.

Silmukkarakenteita Xfunc-kielessä ei ole, vaan silmukat toteutetaan rekursiona. Ehdollisia rakenteita on yksi: suoritusjärjestyksen yhteydessä mainittu *if*. Se on funktio, joka ottaa parametriksi totuusarvon, ehdollisen lohkon sekä mahdollisen vaihtoehdoisen lohkon. Koska kaikki funktiot ovat laiskasti laskettuja, ainoastaan valitun lohkon arvo lasketaan. Laiskan laskennan ja ensimmäisen luokan funktioiden avulla monipuolisempia rakenteita voidaan toteuttaa tarpeen mukaan. Toistorakenteiden luominen vaatisi kuitenkin tukea häntärekursion optimoinnille. Normaalisti rekursiota käytettäessä jokainen kutsu kasvattaa aktivaatitietuepinon kokoa, joten optimoimaton rekursio ei sovellu käytettäväksi pitkien toistosarjojen toteuttamiseen.

Xfunc-kieli sisältää yhden normaalista funktionaalisista rakenteista poikkeavan piirteen, nimittäin XML-dokumenttien käsittelyyn käytettävän heijastuksen. Heijastus ei ole syntaktinen rakenne, vaan sitä käytetään tavallisten funktiokutsujen avulla. Heijastusmenetelmä esiteltiin alla olevan dokumentin yhteydessä.

```
<?xml version="1.0" ?>
<root>
  <element name="element1">
  </element>
  <element name="element2">
    <element name="inner-element">
```

```
    </element>
  </element>
</root>
```

Oletetaan, että dokumentti on ladattu muuttujaan *\$xml*. Juurielementin lapsielementit voidaan poimia *select-elements*-funktion avulla.

```
(define $nodes
  (select-elements $document
    (function ($node) (is-root (parent-of $node))))))
```

Funktiolle *select-elements* välitetään parametrina käsiteltävä dokumentti sekä valintaehto funktiona, joka ottaa syötteen solmun ja palauttaa totuusarvon. *select-elements* tukee heijastusta: palautetut XML-dokumentin solmut säilyttävät viittauksen alkuperäiseen solmuun. Yksinkertaisuuden vuoksi Xfunc ei tue listoja, joka olisi funktionaalisessa kielessä luonnollinen tapa palauttaa joukko solmuja. Sen sijaan solmut sijoitetaan keinotekoisin juurielementin alle. Funktio palauttaa siis yhden dokumentin, mutta kaikki XML-käsittelyfunktiot tunnistavat nimettömän juurielementin ja käsittelevät solmuja kuten ne olisivat listassa.

Valitut elementit voidaan käsitellä, esimerkiksi uudelleennimetä.

```
(define $renamed-nodes
  (map-elements
    (function ($element)
      (rename-element $element "renamed-element")))
    $nodes))
```

rename-element palauttaa uudelleennimetyin kopion parametrina annetusta solmusta. *map-elements* suorittaa ensimmäisenä parametrina annetun funktion jokaiselle toisena parametrina annetulle solmulle, ja palauttaa funktiokutsujen palauttamien solmut.

Muuttuja *\$renamed-nodes* sisältää kaksi solmua lapsisolmuineen. Vaikka solmut on käsittelyn aikana kopioitu kahdesti, ne sisältävät yhä viittauksen alkuperäiseen dokumenttiin. Niinpä ne voidaan antaa parametriksi *apply-changes*-funktiolle, joka palauttaa alkuperäisen dokumentin muutettuna siten, että muokatut solmut ovat korvanneet niitä vastaavat alkuperäiset solmut.

```
(apply-changes $renamed-nodes)
```

Heijastusfunktio palauttaa muutetun dokumentin, siis kopion alkuperäisestä dokumentista, jossa kahden juurisolmun lapsielementin nimi on vaihdettu. Koska Xfunc on funktionaalinen kieli, ovat funktiot ohjelman modularisoinnin tärkein väline. Alla on esitetty koko esimerkki muutettuna siten, että solmut poimitaan oman valitsijafunktion avulla eli valintaehto on erotettu solmujen käsittelystä.

```
(define $node-selector
  (function ($node) (is-root (parent-of $node))))
(define $nodes
  (select-elements $document $node-selector))
(define $node-renamer
  (function ($element)
    (rename-element $element "renamed-element")))
(define $renamed-nodes
  (map-elements $node-renamer $nodes))
(apply-changes $renamed-nodes)
```

Ohjelma toimii kuten äsken esitetty. Valitsijafunktio tutkii, onko solmun vanhempisolmu juurisolmu. Nyt kuitenkin solmun valitseminen on kapseloitu erilliseksi funktioksi, joka voidaan vaihtaa. Huomattavaa on, että ohjelma on yhä melko lyhyt ja hyvin itseään dokumentoiva. Silti sekä solmujen valintaperuste että tehtävä muutosoperaatio ovat vaihdettavissa ilman, että muita ohjelman osia joudutaan muuttamaan.

3.1.4 Tyypijärjestelmä

Ohjelmointikielissä tyyppien avulla voidaan toisaalta välttää yhteensopimattoman datan käyttö ristiin ja toisaalta valita oikea operaatio datan tyyppin mukaan. Xfuncielen tyypijärjestelmä sisältää kahdenlaisia tyyppejä: funktiotyyppejä sekä skaalarityyppejä. Funktiotyypit ovat funktioiden tyyppejä, jotka muodostuvat funktion parametrien tyypeistä sekä funktion paluuarvon tyyppistä. Esimerkiksi mikäli XML-elementtejä poimiva funktio ottaa parametriksen valitsijafunktion, joka ottaa yhden XML-muotoisen parametrin ja palauttaa totuusarvon, aiheuttaa merkkijonoja katoavan funktion antaminen parametriksi tyyppivirheen valitsijafunktiota käytettäessä.

Skalaarityyppejä ovat *string*, *boolean*, *number* ja *xml*. Lisäksi Xfunc sisältää erikoisarvon *null*, joka on kaikkia yllä mainittuja tyyppejä. *null* saa tyypistä riippuen arvot: "" (*string*), *false* (*boolean*), *0* (*number*), tyhjä dokumentti (*xml*) ja *f(...)=null* (funktio-tyyppi). Xfuncin *null* ei siis vastaa semantiikaltaan Javan *null*-arvoa, jolla suoritettut viittaukset ovat virheellisiä. Löyhemmän käytännön perusteena on se, että Xfuncin kaltaisella yksinkertaisella kielellä suoritettavissa tehtävissä käytön yksinkertaisuus on tärkeämpää kuin mahdollisten väärinkäyttöjen systemaattinen estäminen.

Lähdekoodissa numerovakiot merkitään suoraan ilman erityistä notaatiota ja merkijonot ympäröidään lainausmerkein, yleisesti käytetyn notaation mukaisesti. Aluksi kaikki vakiot merkittiin ilman lisänotaatiota ja tyyppi pääteltiin sisällön mukaan: mikäli merkkijono oli muutettavissa lukuarvoksi, sen pääteltiin olevan lukuarvovakio, muussa tapauksessa merkkijonovakio. Syynä oli syntaksin yksinkertaisuuteen pyrkiminen sekä se, että tällä tavoin pystyttiin kirjoittamaan XML-muotoisia vakioita suoraan XML-syntaksilla, kuten alla on esitetty.

```
(parse-xml
  <dokumentti>
    <elementti attribuutti='arvo' />
  </dokumentti>)
```

Erikoismerkkien ja välilyöntien käyttö vaati kuitenkin apufunktioiden käyttöä, joka osoittautui liian työlääksi. Esimerkiksi välilyönti jouduttiin tuottamaan erillisen apufunktion avulla, koska kaksi erillistä sanaa tulkittiin kahdeksi merkkijonovakioksi, eikä yhdeksi merkkijonoksi, jossa on välilyönti. XML-muotoisten vakioiden käsittelyssä tämä ei ollut yleensä ongelma, mutta yleisten merkkijonojen ja osoitteiden – kuten URL-osoitteiden ja tiedostonimien – käsittelyssä alkuperäinen käytäntö osoittautui liian työlääksi. Alla on esitetty yksinkertainen tulostus vanhalla ja uudella notaatiolla.

```
(print Press (space) any (space) key (space) to (space) continue ...)
(print "Press any key to continue...")
```

Xfuncin nykyversiolla ei siis voida upottaa ohjelmaan XML-vakioita suoraan XML-syntaksia käyttäen. XML-syntaksi kaikkine yksityiskohtineen on monimutkainen, mutta Xfunc-kielen käytön kannalta riittävä olisi ollut XML-kielen perusominaisuuksia käyttävien dokumenttien upottaminen. Niinpä aiheeseen voidaan palata kielen jatkokehityksessä.

xml on keskeinen tyyppi, koska Xfunc on tarkoitettu XML-dokumenttien käsittelyyn. *xml* ei ole rakenteinen, vaan jakamaton primitiivityyppi. Funktionaaliseen paradigmaan olisi sopinut luontevasti XML:n käsittely listojen avulla. XML-dokumentit ovat tavallisesti hyvin epätasapainoisia puurakenteita, ja niissä olevat listarakenteet – sisäsolmujen listat ja esivanhempisolmujen listat – eivät ole erityisen hyödyllisiä. Käsittely kohdistuu yleensä vaihtelevilla syvyyksillä oleviin tietyn ehdoin valikoi-tuihin solmuihin. Tällöin käyttökelpoisempia ovat rekursiivisesti toimivat funktiot, joille voidaan antaa parametrina solmujen valintakriteeri.

XML-tyyppi on siis abstrakti tietotyyppi. Tietotyypin rakenne, eli sen toteutus, on piilotettu ohjelmoijalta käsittelyfunktioiden avulla. Xfunc ei sisällä perinteisiä tietorakenteita, kuten listoja. Ratkaisuun päädyttiin kielen yksinkertaisuuden vuoksi.

Xfuncissa tyyppitarkistukset tehdään vasta ohjelman suorituksen aikana, käyttäen muuttujan osoittaman arvon tyyppiä. Tässä suhteessa Xfunc muistuttaa jossain määrin skriptikieliä kuten Perl tai PHP. Xfunc-kielessä tyyppien merkitys on pääasiassa siinä, että niiden avulla estetään operaatioiden soveltaminen sellaisiin arvoihin, joille niitä ei ole määritelty. Virheiden havaitsemisen lisäksi tyyppejä käytetään polymorfismiin: primitiivifunktiot voivat toimia eri tavalla, kun niitä sovelletaan eri tyyppisiin arvoihin. Näin ollen myös ohjelmoijan luomat funktiot voivat olla polymorfisia. Xfunc-kielessä ohjelmoijan luomat funktiot saavat määrittelemättömän tyyppisiä arvoja parametreina ja voivat antaa ne eteenpäin primitiivifunktioille, jotka toteuttavat polymorfisen toiminnallisuuden. Mikäli funktio kutsuu primitiivifunktioita väärän tyyppisellä parametrilla, se aiheuttaa tyyppivirheen.

Selkeyden vuoksi Xfunc-kirjastoissa polymorfisuutta on vältetty: esimerkiksi joissain kielissä matemaattinen yhteenlasku on määritelty myös merkkijonoille, jolloin yhteenlasku tuottaa merkkijonon katenaation – Xfunc-kirjastossa on sen sijaan erillinen *concat*-funktio merkkijonon katenaatiota varten. Xfunc ei tue eksplisiittistä tyyppitystä, joten muuttujien tyyppien lukeminen lähdekoodista vaatii päättelyä. Polymorfisten funktioiden runsas käyttö vaikeuttaisi päättelyä ja näin ollen heikentäisi ohjelman luettavuutta.

3.1.5 Xfunc ja ohjelmointikielten laadulliset piirteet

Korkean tason sovellusaluekohtaisen kielen etuna voidaan pitää sitä, että kielen rajatun käyttötarkoituksen vuoksi tekniset rajoitteet ovat löyhemmät ja kielen suunnittelussa voidaan keskittyä enemmän kielen laadukkuuteen. Xfuncin suunnittelu

pohjautuikin ohjelmointikielten laatua käsittelevään kirjallisuuteen. Robert Sebesta esittää hyvän ohjelmointikielen tunnusmerkeiksi luettavuuden, kirjoitettavuuden, luotettavuuden ja hinnan [Seb01, s. 8 - 20]. Xfunc on tarkoitettu upotetuksi skriptikieleksi, joten siinä pyrittiin pienten ohjelmien hyvään luettavuuteen ja kirjoitettavuuteen sekä kohtuulliseen luotettavuuteen. Luettavuuteen vaikuttaa kielen yksinkertaisuus, rakenteiden ortogonaalisuus ja syntaksin havainnollisuus. Ohjelmointikielten suunnittelun pioneeri C. A. R. Hoare pitää syntaksin yksinkertaisuutta ohjelmointikielten suunnittelun tärkeimpänä tavoitteena [Hoa89]. Yksinkertaiselle syntaksille on myös helpompi tehdä kääntäjiä ja muita työkaluja. Ohjelmointikieli ei ole pelkästään tietokoneen käskyttämisen väline, vaan myös dokumentti ohjelman toiminnasta – tästä näkökulmasta koodin luettavuus on sen tärkein laadullinen piirre. Xfunc-kielen syntaksi on hyvin yksinkertainen ja jäsentimen toteuttaminen olikin helppoa. Yksinkertaisen syntaksin vuoksi Xfunc-ohjelmat ovat pääasiassa funktio-kutsuja. Niinpä funktioiden suunnittelu ja hyvä nimeäminen vaikuttaa ratkaisevasti Xfunc-ohjelmien luettavuuteen.

Merkittävä luettavuuteen vaikuttava tekijä on myös kielen piirteiden ortogonaalisuus eli se, että toisiinsa liittymättömät asiat ovat toisistaan riippumattomia myös kielessä⁷. Xfuncissa ortogonaalisuus on hyvä: funktioilla ei ole sivuvaikutuksia, joten niitä voidaan yhdistellä vapaasti. Koska funktiot voivat olla polymorfisia, voidaan sama funktio toteuttaa tukemaan useita eri parametrityyppejä, mikäli se funktion suorittamaan tehtävään soveltuu. Xfuncissa heijastusmenetelmä rikkoo hieman ortogonaalisuutta: *xml* tietotyyppi toimii ulkoisesti eri tavoin kuin muut tietotyypit, koska heijastamista voidaan käyttää vain sen tyyppisiin arvoihin. Itse menetelmä on kuitenkin kapseloitu heijastusfunktion sisään, joka käyttäytyy kuten muutkin kielen funktiot.

Kirjoitettavuus tarkoittaa kielen yksinkertaisuutta ja ortogonaalisuutta, mutta myös sen ilmaisuvoimaa ja abstraktioiden tasoa. XML-suuntautuneet abstraktiot – *xml*-tietotyyppi, XML-käsittelyfunktiot ja heijastusmenetelmä – ovat ilmaisuvoimaisia, josta todisteena Xfunc-kieliset ohjelmat ovat merkittävästi lyhyempiä kuin vastaavat yleiskäyttöisellä kielellä toteutetut. Kirjoitettavuus on myös pitkälti riippuvainen funktiokirjaston muistettavuudesta ja rakenteesta: mikäli funktiot soveltuvat tehtävään hyvin ja muodostavat helposti hahmotettavan kokonaisuuden, on itse kielenkin kirjoittaminen helppoa.

⁷Ortogonaalisuutta voitaisiin kuvailla vapaamuotoisemmin "pienimmän yllätyksen periaatteeksi". Ohjelmointikielten piirteiden ei tulisi olla toisistaan riippuvaisia, mikäli ohjelmoijan ei voida uskoa näin intuitiivisesti olettaen.

Kielen luotettavuuteen vaikuttavat kielessä olevat tyyppitarkistukset ja virheiden hallinta. Tyyppitarkistusten avulla voidaan välttää monia ohjelmointivirheitä, joten Xfuncin vahva tyyppijärjestelmä onkin merkittävä kielen laatua parantava seikka. Staattisten tarkistusten avulla osa tyyppivirheistä voitaisiin havaita jo suorituksen aikana. Tehokkaat staattiset tarkistukset vaatisivat kuitenkin tyyppitiedon kirjoittamista lähdekoodiin, mikä hankaloittaa kielen käyttöä ja on vastoin Xfuncin tarkoitusta toimia nopeana skriptaustyökaluna.

Mahdollisuus rakentaa abstraktioita ja kehittää toiminnallisuutta niiden avulla on keskeinen piirre kaikissa ohjelmointiparadigmoissa. Sovellusaluekohtaisissa kielissä abstraktiot perustuvat pitkälti sovellusalueen käsitteisiin ja idiomiin. Wirth ja Hoare esittävät, että ohjelmointikielen rakenteiden tulisi olla samalla abstraktiotasolla keskenään [Wir74, Hoa89]. Mikäli kieli sisältää matalan tason ominaisuuksia, joita käyttämällä voidaan rikkoa korkeamman tason abstraktioita, menettävät abstraktiot hyödyllisyyttään. Esimerkiksi joissain kielissä ohjelmoija voi käyttää oman valintansa mukaan viitteen tai arvon perusteella tapahtuvaa parametrin välitystä. XML-käsittelyn yhteydessä piirrettä voidaan pitää matalamman tason ominaisuutena, ja se olisi haitallinen Xfunc-kieleen toteutettuna. Tällöin nimittäin viitteen perusteella parametrin välittävällä funktiokutsulla voisi olla sivuvaikutuksia, koska kutsuttu funktio voisi muuttaa sille annettujen parametrien arvoa myös kutsuvassa funktiossa. Funktion toiminnan ymmärtäminen vaatisi sen kutsumien funktioiden toteutuksen yksityiskohtien tuntemista. Parametrien kopioiminen takaa myös sen, että samaa solmua ei voida syöttää XML-dokumenttiin kahdesti. Viiteparametrin avulla se olisi mahdollista, jolloin puumuotoiset dokumentit muuttuisivat verkoiksi, eli olisivat virheellisessä muodossa, ja niillä tehtävät käsittelyoperaatiot johtaisivat virheisiin. XML-käsittelyn oikeellisuuden varmistaminen jäisi tältä osin ohjelmoijan vastuulle. Sovellusaluekohtaisella kielellä samantasoisuus on helppo saavuttaa, koska rajatun sovellusalueen abstraktiot ovat luonnollisesti samantasoisia. Xfunc-kieleen ei ole sisällytetty merkittäviä XML-käsittelyn ulkopuolisia abstraktioita, kuten teknisiä rakenteita, koska ne ovat varsinaista sovellusaluetta matalammalla tasolla. Korkean tason XML-abstraktioita käyttäessään Xfunc-ohjelmoija voi olettaa niiden pätevän aina.

3.2 Cclang-metakielen käyttö

Flow-alustan metakielellä on kolme käyttötarkoitusta: alustan ylläpito, komponenttien toisiinsa kytkentä ja aspektiohjelmointi. Alustan ylläpidolla tarkoitetaan alus-

tan ja siinä toimivien tulkkien monitorointia ja hallintatoimia. Komponentteja eli komponenttikielillä kirjoitettuja ohjelmia voidaan kytkeä (putkittaa) toisiinsa ja laitteistorajapintoihin. Aspektiohjelmoinnin avulla voidaan rajattuun tehtävään tarkoitettulla sovellusaluekohtaisella kielellä toteutettuihin skripteihin lisätä sovellusalueeseen kuulumattomia piirteitä, kuten lokitoimintoja.

Cclang ei sisällä erillisiä ominaisuuksia edellä mainittujen käyttötarkoitusten toteuttamiseen, vaan kielen logiikkaohjelmointiin perustuvan päättelymenetelmän avulla voidaan suorittaa kaikki tehtävät. Alustan tila esitetään loogisina faktoina, joita kyselemällä voidaan muun muassa selvittää tulkkien sen hetkinen tila – kieli siis tarjoaa näkymän alustan toteuttavaan oliomalliin. Luomalla uusia päättelysääntöjä saadaan alusta tekemään muutoksia tilaansa, kuten kytkemään komponentteja toisiinsa ja käynnistämään uusia tulkkeja. Muutokset voivat olla ehdottomia, jolloin ne suoritetaan välittömästi, tai ne voidaan sitoa alustan tilaan, jolloin ne suoritetaan, kun alustan tila vastaa annettuja ehtoja.

Cclang-tulkin pohjalla käytetään ulkopuolista Java-pohjaista logiikkaohjelmointikielen toteutusta. Ulkopuolisen toteutuksen käyttö on mahdollista, koska alusta ei määrää metakielen tulkille erityistä suoritustyyliä, toisin kuin komponenttikielten tulkeille. Työssä käytettiin tuProlog-tulkkiä, joka on erityisesti Java-sovellukseen upottamista varten suunniteltu Prolog-toteutus [DOR01]. Niinpä itse Cclang-kieli on Prolog-kielen pohjalta luotu deklaratiiivinen kieli.

3.2.1 Alustan tilan esittäminen logiikkaohjelmointikielissä ja tulkkien monitorointi

Cclang-tulkki esittää alustan tilan loogisina rakenteina, joita voidaan kysellä ja joiden pohjalta voidaan suorittaa päättelyjä. Tulkki tarjoaa siis korkean abstraktiotason menetelmän tilatiedon käsittelyyn: perinteisesti ohjelmointikielten tulkit tarjoavat hyvin niukasti tilatietoa toiminnastaan, tai tilatieto on prosessoimatonta tietoa (kuten pinovedoksia), jonka käsittely vaatii erillisen analyysityökalun. Cclang-tulkki tarjoaa logiikkaohjelmointiin perustuvan analyysimenetelmän sisäänrakennettuna.

Cclang-kieli on rakennettu Prolog-ohjelmointikielen päälle. Prolog- ja Cclang-ohjelmat muodostuvat faktoista, säännöistä ja kyselyistä [WMW95, s. 118 - 132]. Faktat ovat tosia lauseita, joita tulkki käyttää pohjatietona suorittaessaan päättelyjä sääntöjen avulla. Faktat ja säännöt muodostavat siis tulkin tietämuskannan. Cclang-tulkin toiminnan käynnistää se, että ohjelmoija esittää sille kyselyn. Tulkki

pyrkii todistamaan väittämän hakumenetelmää käyttäen. Todistuksen hakumenetelmän voidaan siis ajatella toimivan abstraktina kielen tulkkina [EK76].

Cclang-kieli perustuu predikaattilogiikkaan. Predikaattilogiikan mukaiset lauseet rakentuvat symboleista, joita on viittä eri tyyppiä: atomit, predikaatit, funktiot, operaattorit ja muuttujat. Atomit ovat logiikkajärjestelmän kannalta jakamattomia objekteja, jotka vastaavat mallinnettavan maailman käsitteitä – Cclangin tapauksessa alustaan liittyviä komponentteja sekä yleisiä vakioita, kuten *interpreter* tai 120. Cclang-kielessä, kuten Prologissakin, atomit aloitetaan pienellä alkukirjaimella tai numerolla.

Predikaatit ovat vakioita, joista yksipaikkaiset kuvaavat atomien ominaisuuksia ja monipaikkaiset atomien välisiä suhteita. Esimerkiksi tulkin käynnissä oloa voitaisiin kuvata yksipaikkaisella predikaatilla *is_running(interpreter)* ja lukujen eroja kaksipaikkaisella predikaatilla *greater_than(120, 100)*. Ensimmäisen kertaluvun predikaatit eivät voi kuvata predikaattien ominaisuuksia tai suhteita. Funktiot ovat vakioita, jotka ottavat parametreina atomeja ja palauttavat atomin. Esimerkiksi itseisarvofunktion kutsu *abs(-120)* palauttaa atomin 120. Cclang-ohjelmissa voidaan käyttää Prologiin kuuluvia funktioita, mutta Cclang ei sisällä omia funktioita.

Monimutkaisempia loogisia lauseita rakennetaan käyttäen loogisia operaattoreita. Niistä yleisimmin käytetyt ovat kaksipaikkaiset \wedge (ja), \vee (tai), \neg (ei), \rightarrow (implikaatio) ja \leftrightarrow (yhtäpitävyys). Cclang-kielessä näitä vastaavat `,` (pilkku), joka on \wedge , `;` (puolipiste), joka on \vee , *not*, joka on \neg ja *if*, joka on \rightarrow . Cclang ja Prolog poikkeavat matemaattisesta logiikasta kahdessa suhteessa. Implikaatiossa ehto merkitään oikealle puolelle ja seuraus vasemmalle puolelle: *if* on siis oikeammin \leftarrow . Negatio rinnastetaan päättelyn epäonnistumiseen. Tällöin väittämän päätellään olevan epätosi, mikäli todistusta ei löydetä. Kuten tavallisessa aritmetiikassa, operaattorit sidotaan vasemmalta oikealle ja suoritusjärjestyistä voidaan muuttaa sulkujen avulla.

Muuttujat ovat symboleja, jotka toimivat paikanpitäjinä atomeille – niitä voidaan siis käyttää samoissa yhteyksissä kuin atomejakin. Muuttujat sidotaan eli kvantifoidaan erityisten loogisten operaattoreiden, kvanttorien, avulla. Esimerkiksi predikaattilogiikan lause *nopea(x)*, jossa *x* on muuttuja, ei ole hyvin muodostettu, koska *x* ei ole kvantifioitu. Sen sijaan esimerkiksi $\forall x : (\text{maksinopeus}(x) > 120) \rightarrow \text{nopea}(x)$ on hyvin muodostettu, koska *x* on universaalisti kvantifioitu eli lause pätee kaikille muuttujan arvoille. Lauseen tulkinta on, että kaikki objektit, joiden maksiminopeus on yli 120, ovat nopeita.

Cclang noudattaa Prologin käyttämää muuttujien kvantifointia. Kielessä ei ole eksplisiittisiä kvanttoreita, vaan säännöissä ja faktoissa kaikilla muuttujilla on implisiittinen universaalikvanttori, paitsi jos ne esiintyvät vain säännön ehto-osassa, jolloin niillä on eksistentiaaliquanttori. Tällöin esimerkiksi sääntö $nopea(X) \text{ if } auto(X)$ tarkoittaa, että kaikki autot ovat nopeita, ja sääntö $isoisa(X,Z) \text{ if } isa(X,Y), isa(Y,Z)$ tarkoittaa, että on olemassa isä, jonka isä isoisä on. Faktoissa muuttujilla on eksistentiaaliquanttori.

Kun kysely syötetään, tulkki päättelee annettuja faktoja käyttäen, pitääkö väittämä paikkansa, ja mikäli pitää, niin millä väittämässä esiintyvien vapaiden muuttujien sidonnoilla näin on. Esimerkiksi jos on ladattu sääntö $nopea(X) :- auto(X)$ ja fakta $auto(mazda)$, tulkki päättelee väittämän $nopea(mazda)$ olevan tosi. Mikäli halutaan selvittää, että mitkä autot ovat nopeita, voidaan esittää väittämä $nopea(Auto)$. Tällöin muuttuja $Auto$ sidotaan arvoon $mazda$. Mikäli muuttujan varsinaista arvoa ei haluta selvittää, voidaan muuttuja korvata alaviivalla – kyselyn $nopea(_)$ avulla voidaan selvittää, onko mikään auto nopea.

Cclang-tulkin todistusmenetelmä perustuu taaksetjetutukseen (engl. backward chaining) [JS96]. Todistusmenetelmä toimii siten, että se unifioi väittämän eli sitoo vapaille muuttujille jotkin mahdolliset arvot⁸ ja etenee sääntöjä käyttäen, kunnes vastaan tulee ehto, joka on tosi sen hetkisillä muuttujien sidonnoilla. Tällöin kyselyyn on löytynyt vastaus. Mikäli sen sijaan mitään sääntöä ei voida enää soveltaa, tulkki peruuttaa päättelyketjussa taaksepäin ja haarautuu uudelleen, sitoen muuttujalle seuraavan mahdollisen arvon.

Prologissa tietämuskantaan kuuluvat faktat ja säännöt esitetään eri liittymän kautta kuin kyselyt – usein tietämuskanta ladataan käynnistyksessä ja käyttäjältä otetaan vastaan kyselyjä interaktiivisen liittymän kautta. Cclang on interaktiivinen hallintatyökalu, joten oli luontevinta, että kaikki kieleen kuuluvat rakenteet syötetään saman liittymän kautta. Niinpä syötettävät lausekkeet tulee erotella: faktat ja säännöt päätetään huutomerkillä ja kyselyt pisteellä.

Predikaattien avulla voidaan suorittaa kyselyjä alustan olioiden tilasta. Tässä kuvataan niistä tärkeimmät. Alustan oliot tunnistavat predikaatit, eli tyypittävät predikaatit, on annettu alla.

⁸Jos predikaatit samaistetaan funktioihin, voidaan unifikaatio samaistaa parametrien välitykseen.

```
flow_object(FlowObject).
identifiable_object(IdentifiableObject).
language(Language).
interpreter(Interpreter).
interface(Interface).
endpoint(EndPoint).
connection(Connection).
```

Predikaatti *flow_object* tunnistaa kaikki alustan oliot. Useimpiin niistä viitataan tunnistenumeron perusteella. Käytäntö muistuttaa käyttöjärjestelmissä käytettävää tapaa viitata prosesseihin tunnistenumeron perusteella. Viitattavia olioita ovat itse alusta (jonka tunnistenumero on aina 1), käynnissä olevat tai keskeytetyt tulkit, yhteydet ja laitteistorajapinnat. Oliotyypit voidaan tunnistaa predikaattien *interpreter*, *connection* ja *interface* avulla. *identifiable_object* tunnistaa kaikki viitattavat oliot. Kielten ja lähdekoodimoduulien kanssa käytetään yksinkertaisempaa käytäntöä: ne tunnistetaan nimen perusteella, siis esimerkiksi merkkijonovakio *“xfunc”* vastaa Xfunc-kieltä ja *“my_source.xfunc”* kyseisestä tiedostosta ladattua lähdekoodimoduulia.

Alustan olioiden välillä vallitsee suhteita. Suhteita voidaan kysellä alla annetuilla predikaateilla.

```
are_connected(SourceEndPoint, Connection, DestinationEndPoint).
is_executing(Interpreter, Frame).
is_executing(Interpreter, SourceFileName).
has_name(Frame, FrameName).
has_language(Interpreter, LanguageName).
has_tag(FlowObject, TagName).
```

Predikaattien *are_connected*, *is_executing*, *has_name* ja *has_language* avulla voidaan kysellä olioiden välisiä suhteita. Esimerkiksi *are_connected* kertoo, mikä yhteysolio sitoo toisiinsa kaksi pääteoliota. Pääteolio voi olla tulkin instanssi tai laitteistorajapinta, ja se voidaan tunnistaa predikaatilla *endpoint*. Tulkkien tilaa voidaan tutkia myös tarkemmin, kuten esimerkiksi selvittää *is_executing*-predikaatilla, mitä metodia tulkki suorittaa. Tähän palataan luvussa 3.2.4.

Koska kielen tulkitseminen perustuu loogisten lauseiden todistamiseen, pätee Cclang-ohjelmointiin niin kutsuttu syöte-tulos -suhde [EK76]. Alla olevat kaksi Cclang-kyselyä havainnollistavat suhdetta.

```
has_language(X, "xfunc").
```

```
has_language(123, X).
```

Ensimmäisessä lauseessa todistusmenetelmää käytetään sen selvittämiseen, että mikä tulkki-instanssit kuuluvat kieleen *“xfunc”*, ja toisaalta toisessa lauseessa menetelmää käytetään sen selvittämiseen, että mihinkä kieleen tulkki-instanssi numero 123 kuuluu. Cclang sopiikin siis hyvin kyselytyyppiseen ongelmanratkaisuun, mutta tästä poikkeavasta lähestymistavasta johtuen sen integrointi perinteisiin imperatiivisiin ohjelmointityökaluihin on haastavaa, kuten luvussa 3.2.2 havaitaan.

3.2.2 Työkalun tilan muuttaminen Cclang-kielellä

Cclang-kielen logiikkaohjelmointimenetelmän avulla alustan tilan tutkiminen on suoraviivaista, onhan paradigma suunnattu erityisesti kyselytarkoituksiin. Alustan tilan muokkaaminen vaatii kuitenkin suoritusmallin laajentamista. Tilallisen alustan ja komponenttikielten tulkkien sekä ohjelmoijan näkökulmasta tilattoman metakielen yhdistäminen ei ole suoraviivaista – toisin sanoen imperatiivisen toiminnallisuuden käsittely puhtaasti deklarativisen kielen avulla on hankalaa. Ongelmallisuutta voidaan havainnollistaa esimerkiksi yhden komponenttikielen tulkin pysäyttämistä. Ensimmäisessä kokeiluversiossa kaikki alustan tulkit voitiin pysäyttää seuraavalla komennolla.

```
stop(_)!
```

Predikaattiin *stop* oli kytketty sivuvaikutus, jolloin predikaatin unifikaatio aiheutti sivuvaikutuksena muuttujaan sidotun tulkin pysäyttämisen. Lähestymistavassa on kuitenkin kaksi ongelmaa: ajallinen rajoittaminen ja riippuvuus käyttäjän interaktiosta. Kun alusta on pysäyttänyt kaikki tulkit, tulisi komennon raueta – tai muuten alustaan ei voida enää käynnistää uusia tulkkeja. Deklaraatiolla tulisi siis olla ajallinen rajoite. Toinen ongelma on, että predikaatin unifikaatio tapahtuu vain kyselyn seurauksena. Koska vain käyttäjä voi syöttää kyselyjä, vain käyttäjän toimenpiteet voivat vaikuttaa alustan tilaan. Aspektin kehoitteen suorittaminen pitäisi kuitenkin olla mahdollista milloin tahansa. Niinpä tulkin pitäisi itse esittää alustan tilaan vaikuttavien predikaattien kyselyt.

Interaktio-ongelma on ratkaisu niin, että Cclang-tulkki esittää jokaisella suoritusaskeleella alustan tilaan vaikuttavien predikaattien kyselyt. Muutosten toteuttaminen

sivuvaikutusten avulla ei ole kuitenkaan järkevää, koska tällöin vuorovaikutus on hyvin hankalasti hallittavissa. Philip Wadler on tutkinut imperatiivisen toiminnallisuuden käsittelyä deklaratiivisen kielen avulla funktionaalisten ohjelmointikielten syöte- ja tulosteominaisuuksien yhteydessä [Wad97]. Hänen mukaansa yksinkertaisin ratkaisu – jota myös tuProlog-tulkki suoraan tukee – on lisätä deklaratiivisiin rakenteisiin sivuvaikutuksia, eli kytkeä predikaatteihin metodeja, jotka suoritetaan kun predikaatti unifioidaan. Wadler kuitenkin esittää, että deklaratiivisen ja imperatiivisen paradigman integrointia voidaan selkeyttää hallittavan solmupisteen avulla. Sivuvaikutusten tapauksessa solmupiste on predikaatin kysely, ja koska sivuvaikutuksellisia predikaatteja on useita, on solmupiste hajonnut useisiin osiin. Niinpä Cclang-tulkki käyttää sen sijaan erityistä vuorovaikutuspredikaattia, johon liittyvät kyselyt se esittää itse ja suorittaa muutokset alustan tilaan kyselyjen tulosten perusteella. Tällöin tulkki itse toimii solmupisteenä, ja vuorovaikutus on helpommin hallittavissa.

Alustan tilan muuttaminen perustuu *action*-vuorovaikutuspredikaattiin. Predikaatti sitoo yhteen toiminnan subjektin, tyyppin sekä kohteen. Esimerkiksi *action(123, connect, 234)* kytkee komponentin numero 123 komponenttiin numero 234, kuten HTTP-rajapinnan web-pyyntöjä käsittelevään skriptiin. Predikaatista on myös kaksipaikkainen versio, jota käytetään silloin kun kohdetta ei tarvita. Toiminnan tyyppi voi Cclang-kielen nykyversiossa olla *start*, *stop*, *connect*, *create* tai *tag*. *start* ja *stop* käynnistävät ja pysäyttävät subjektina olevan tulkin. *connect* kytkee kaksi oliota toisiinsa. *create* luo uuden päätepisteen, eli tulkin tai laitteistorajapinnan. Luotava olio sidotaan subjektina annettuun nimeen ja olion tyyppi sekä luontiparametrit luetaan kohteena annettavasta merkkijonosta. *tag* lisää olioon merkinnän Cclang-tulkin sisäisessä kirjanpidossa. Mekanismi esitellään tarkemmin luvussa 3.2.4. *action*-toiminnallisuus on vielä kokeiluluontoinen, eikä täysin kata alustan ohjelmointirajapinnan mahdollistamia toimenpiteitä.

Kuvattu ratkaisu ei kuitenkaan vielä ratkaise deklaraatioiden ajattomuuteen liittyvää ongelmaa. Kun pyydetty toimenpide on suoritettu, muuttuu siihen johtanut sääntö tarpeettomaksi, mahdollisesti myös haitalliseksi. Esimerkiksi jos kaikki tulkit sammutetaan ja sääntö jää voimaan, sammutetaan myös kaikki jatkossa käynnistettävät tulkit.

Kun Cclang-tulkki suorittaa toimenpiteen, se poistaa tietämuskannastaan säännön, jonka seurauksena toimenpidettä vastaava *action*-predikaatti on. Koska palvelinsovellukset voivat olla käytössä hyvinkin pitkiä aikoja, tulee käytettyjen al-

goritmien olla sellaisia, että käytöstä poistettu tieto ei jää muistiin. Poistamismenetelmä ei ole luonteeltaan deklaratiiivinen, mutta se on helppo käyttää ja on toteutettavissa tehokkaasti ja yksinkertaisesti. Poistamismenetelmää voidaan myös ohjata: *invariant_action*-predikaatin avulla suoritetaan toimenpiteitä samoin kuin *action*-predikaatin avulla, mutta niitä vastaavia sääntöjä ei poisteta. *invariant_action* soveltuu siis erityisesti aspektiohjelmointiin. Predikaatin käyttö vaatii tarkkuutta, mutta vastaavuuden vuoksi sitä on helppo testata aluksi *action*-predikaattina. Cclang ei tällä hetkellä sisällä mahdollisuutta eksplisiittisesti poistaa *invariant_action*-predikaatteja. Taustalla olevan Prolog-tulkin ominaisuuksia käyttäen sääntöjä voidaan poistaa, mutta helppokäyttöisyyden ja hallittavuuden vuoksi kieleen tarvittaisiin rakenteet *invariant_action*-sääntöjen hallintaan, eli poistamiseen ja pois kytkemiseen.

3.2.3 Komponenttien kytkeminen

Flow-työkalun komponenttikielten avulla toteutettavat komponentit on tarkoitettu rajattujen tehtäväalueiden hoitamiseen. Alustan kytkentämekanismiin avulla syötteiden ja tulosteiden käsittely voidaan hoitaa komponenttien ulkopuolella, jolloin käsittely ei sotke niiden rakennetta ja komponentit ovat riippumattomampia siitä, mistä niiden syötteet tulevat tai mihin niiden tulosteet menevät – tässä mielessä kytkentä siis muistuttaa Unix-ohjelmien putkitusta. Flow-alustan kytkentämekanismiin nykyversio on hyvin yksinkertainen: kytkennässä on yksi lähde ja yksi kohde ja tieto siirretään merkkijonojen muodostamana virtana. Vaikka tiedon siirtomuoto onkin yksinkertainen, on sen todettu soveltuvan hyvin haastaviinkin tehtäviin hieman Flow-työkalua vastaavan Taverna-sovellusintegraatiotyökalun [OAF⁺04] yhteydessä. Yhteydet kuuluvat olioihin, joita alusta hallinnoi ja joita muokataan metakielen avulla. Esimerkiksi HTTP-porttia kuunteleva Web Service -komponentti kytketään Cclang-kielellä seuraavasti.

```
action(Component, start, "my_web_module.xfunc"),  
    action(http, connect, Component)!
```

Ensimmäinen *action*-predikaatin kutsu lataa moduulin ja muuttujan *Component* arvo unifioidaan ladatun komponentin tunnistenumeroon. Seuraava kutsu kytkee ennalta määritellyn vakion *http* osoittaman HTTP-rajapinnan ladattuun komponenttiin. Tiedostossa *my_web_module.xfunc* oleva ohjelma voisi olla seuraavanlainen.

```
(define $xml (parse-xml (input)))
(define $processed-xml
  (do-required-xml-processing $xml))
(output (xml-to-string $xml))
```

Ohjelma on aiemmin esitelty XML-käsittelyesimerkki. Kytkenät näkyvät komponenttikielissä kyseiseen kieleen ja paradigmaan parhaiten sopivalla tavalla, Xfuncin tapauksessa funktioina. Funktio *input* lukee yhden merkkijonon syötteestä ja palauttaa sen. Merkkijono jäsennetään, saatu dokumentti käsitellään halutulla tavalla, muutetaan takaisin merkkijonoksi ja annetaan parametriksi funktiolle *output*, joka kirjoittaa sen tulostevirtaan yhtenä merkkijonona. Tulee huomata, että komponentti käynnistetään ennen kuin kytkentä on luotu, mutta *input*-funktion kutsu jää odottamaan, kunnes kytkentä on olemassa ja merkkijono on luettu.

3.2.4 Aspektiohjelmointi Cclang-kielellä

Aspektiohjelmoinnin avulla voidaan rajattuun tehtävään tarkoitettulla sovellusaluekohtaisella kielellä toteutettuihin skripteihin lisätä sovellusalueeseen kuulumattomia piirteitä, kuten lokitoimintoja. Komponenttien kytkemisen lisäksi aspektiohjelmointi on tärkeä väline, jolla komponenttien sovellusaluekohtaiseen logiikkaan kuulumattomat piirteet voidaan kytkeä komponentteihin ulkopuolelta.

Aspektiohjelmoinnin perusidea on risteävien aiheiden modularisointi. Aihe⁹ on ohjelmaan liittyvä tavoite, käsite tai kiinnostuksen kohde [KLM⁺97]. Toisiinsa liittymättömiä aiheita kutsutaan risteäviksi aiheiksi. Risteävät aiheet aiheuttavat ohjelmakoodin sotkeutumista ja hajautumista [KLM⁺97]. Koodin sotkeutuminen tarkoittaa sitä, että yhdessä lähdekoodin kohdassa on nähtävissä monia toisiinsa liittymättömiä aiheita, jolloin sen ymmärtäminen ja muuttaminen on hankalampaa. Koodin hajautuminen taas viittaa siihen, että aiheiden risteävyyden vuoksi jotkin aiheista, yleensä varsinaisen sovelluslogiikan kanssa risteävät aiheet, ovat hajautuneet ympäri ohjelmakoodia. Näin ollen risteäviin aiheisiin liittyvästä ohjelmakoodista tulee vaikeasti hallittavaa ja myös muun ohjelmakoodin selkeys kärsii siihen sotkeutuneesta koodista. Esimerkiksi ei-toiminnallinen vaatimus siitä, että ohjelma kirjaa lokiin kaikki tiettyyn ohjelman osaan ulkopuolelta suoritettut funktiokutsut, joudutaan to-

⁹Tässä tutkielmassa käytetään soveltaen Juha-Pekka Laineen pro gradu -työssä luotua suomenkielistä aspektisanastoa [Lai01].

teuttamaan perinteisessä imperatiivisessa kielessä lokimetodin kutsuna välittömästi ennen jokaista metodikutsua. Niinpä vaatimus johtaa lokiinkirjoitusten hajautumiseen ympäri ohjelman ja toisaalta sotkee lokitoiminnallisuuden moniin koodin osiin, joihin se ei suoranaisesti liity.

Sotkeutuminen ja hajautuminen huonontavat koodin jäljitettävyyttä, uudelleenkäytettävyyttä ja teknistä laatua, ja näin ollen heikentävät ohjelmoijien tuottavuutta ja vaikeuttavat ohjelman jatkokehittämistä [KLM⁺97]. Aspektimenetelmä jakaa ohjelman aspektikoodiin, joka on toteutettu aspektikielellä, ja komponenttikoodiin, joka on toteutettu perinteisellä ohjelmointikielellä. Komponenttikoodi sisältää vain komponenttikielellä helposti modularisoitavia toiminnallisuuksia, ja aspektikoodi taas valittuun erotteluun sopimattomia toiminnallisuuksia. Aspektityökalu punoo yhteen aspektikoodin ja komponenttikoodin. Punonnan avulla aspektikoodi vaikuttaa komponenttikoodin suoritukseen – aspektiohjelmointi siis koostuu kiinnostavien liitoskohtien valinnasta ja datan sekä suorituksen käsittelystä noissa pisteissä [WZL03].

Aspektityökalu voi suorittaa punonnan staattisesti ennen suoritusta tai dynaamisesti suorituksen aikana. Monissa tilanteissa dynaaminen aspektien punonta on parempi vaihtoehto palvelinsovellusten punomiseen. Koska tuotantokäytössä olevia järjestelmiä ei voida sammuttaa kuin harvoin ja koska kehityskäytössäkin olevien järjestelmien sammuttaminen on usein työlästä, ei staattisesti punottuja aspekteja voida muuttaa helposti, ainakaan ilman monimutkaista ohjelmakoodin uudelleenlatausjärjestelmää [CST00, SCT03]. Palvelinsovellusten yhteydessä on perusteltuja tarpeita aspektien ajonaikaiseen muuttamiseen. Esimerkiksi kriittisen tietoturva-aukon paikkaaminen voidaan tehdä ajonaikaisesti, mikäli sovellukseen punotaan dynaamisesti aspekti, joka estää tietoturva-aukon hyväksikäyttämisen [SCT03]. Samoin suorituskykyoptimointien teko tarpeen mukaan, virheiden paikkaaminen tai muutosten tekeminen on mahdollista silloinkin, kun järjestelmää ei voida sammuttaa muutostöitä varten [SCT03].

Aspektiohjelmointi on vielä melko uusi tutkimuksen ala ja ohjelmointiin käytettävät menetelmät ja työkalut eivät ole vielä vakiintuneita. Kuitenkin kaksi pääsuuntausta voidaan erottaa: ohjelmamuunnokset ja koostesuotimet [Paw00, BP00]. Menetelmien jako perustuu siihen, millä menetelmällä punominen eli komponenttikoodin ja aspektikoodin yhdistäminen on toteutettu. Ohjelmamuuntimet ovat aspektityökaluja, jotka instrumentoivat komponenttikoodin aspektikoodilla, eli lisäävät ennen käännöstä tai tulkkauksta komponenttikoodiin koukkuja, jotka kutsuvat aspektikoodia.

Koostesuotimet ovat vanhempi käsite kuin aspektiohjelmointi. Niiden tunnuksenomainen piirre on se, että työkalut eivät muokkaa lähdekoodia, vaan ohjelman osien toisilleen lähettämiä viestejä, yleisimmin metodikutsuja [AWB⁺94, Ber94]. Koostesuotimissa punonta on siis dynaamista: punoja seuraa ajonaikaisesti viestejä ja kutsuu aspektikoodia, mikäli viesti kuuluu aspektissa määriteltyyn liitoskohtaan. Dynaamisuuden vuoksi koostesuodinmenetelmä valittiin Flow-työkaluun.

Dynaamisella aspektiohjelmoinnilla ja logiikkaohjelmoinnilla on yhteisiä piirteitä. Sekä logiikkaohjelmoinnin säännöt että aspektiohjelmoinnin liitoskohtamäärittelyt ovat deklarativisia rakenteita. Voidaan ajatella, että implikaatioon liittyvä ehto vastaa aspektin liitoskohtamäärittelyä ja faktaan liittyvä seuraus taas aspektin kehoitetta. Mikäli liitoskohtamäärittelyn ehdot täyttyvät, punoja kutsuu aspektin kehoitetta, ja vastaavasti mikäli faktan ehdot täyttyvät, logiikkakielen tulkki päättää seurauksen olevan tosi. Cclang-kielen aspektiohjelmointiominaisuudet perustuvat tähän yhtäläisyyteen. Menetelmissä on kuitenkin ero kontrollin toteutuksessa: aspekti on aktiivinen rakenne, joka suoritetaan aina, kun ehdot täyttyvät, kun taas faktan päättely vaatii, että seuraukseen totuusarvon tutkimisen vaativa väittäjä on esitetty päättelykoneistolle. Cclang-tulkin tapa käsitellä *action*-predikaatteja korjaa tämän puutteen, mahdollistaen Cclang-kielen käytön aspektiohjelmointiin.

Käytännön esimerkki kokonaisesta aspektista olisi seuraava komento, joka sammuttaa kaikki kiellettyä komponenttikielistä ohjelmaa suorittavat tulkit.

```
action(Interpreter, stop) if
  is_executing(Interpreter, "forbidden_source.xfunc")!
```

Säännön seuraukset toteutetaan, kun metakielen tulkki suorittaa *action*-predikaattiin liittyvät päättelyt komponenttitulkin seuraavassa suoritusaskeleessa. Komento käskee sammuttamaan tulkin, mikäli tulkki suorittaa kiellettyä ohjelmaa. Vastaavasti aspekti: “jos joku komponenttikielen Foobar tulkki-instanssi suorittaa metodin *authenticate*, niin käynnistä uusi tulkin instanssi suorittamaan moduulia *log*”, määriteltäisiin seuraavasti.

```
action(LogComponent, start, "log.xfunc") if
  frame(Frame),
  is_executing(Interpreter, Frame),
  has_name(Frame, "authenticate"),
  has_language(Interpreter, "Foobar")!
```

Suoritettavan metodin tunnistaminen perustuu yksinkertaiseen kieliriippumattomaan malliin tulkin toiminnasta: tulkin tila koostuu pinosta kehyksiä (engl. frame), joilla on nimi ja joukko sidontoja eli muuttujia. Kehyspino muistuttaa siis aktivaatietietuepinoa. Kehyksen nimi eli useimmissa tapauksissa kutsutun funktion nimi voidaan selvittää predikaatilla *has_name*. Kehyspinoja ei esitellä yksityiskohdaisemmin tässä tutkielmassa.

Monissa aspektimenetelmissä tuetaan kolmea vihjetoiminnallisuustyyppiä: etukäteistä, jälkikäteistä ja ympäröivää vihjetoiminnallisuutta [KHH⁺01]. Kaikki nämä tyypit voidaan toteuttaa myös Cclang-kielillä. Edellä esitellyt aspektit ovat etukäteisiä, koska komponenttikielen tulkki päivittää tilansa ja palauttaa kontrollin alustalle, joka voi tehdä tilaan muutoksia. Esimerkiksi Xfunc asettaa seuraavan funktiokutsun pinoon ja kutsuu alustaa, jolloin aspektitoiminnallisuus havaitsee funktiokutsun ennen kuin tulkki on aloittanut sen suorittamisen. Tällöin esimerkiksi edellä ollut tulkin pysäyttävä aspekti pysäyttäisi tulkin, ennen kuin kiellettyä funktiota on ehditty suorittaa. Ympäröivän vihjetoiminnallisuuden toteuttaminen on myös suoraviivaista, koska Cclang-tulkki voi muokata kehyspinoa ja näin ollen vaikuttaa etenemiseen määritellyssä liitoskohdassa.

Koska tieto funktiosta poistumisesta tai vastaavasta tilasiirtymästä katoaa, kun funktiokutsua vastaava kehys poistetaan kehyspinosta, jälkikäteisen vihjetoiminnallisuuden käyttö vaatii ylimääräistä kirjanpitoa. Alusta tukee yksinkertaista kirjanpitoa menetelmää, jolla alustassa toimiviin objekteihin voidaan kiinnittää merkintä. Tällä tavoin voidaan merkitä liitoskohdassa oleva tulkki, jotta kehote voidaan suorittaa kun tulkki poistuu liitoskohdasta.

```
action(Interpreter, tag, "stop_me") if
  is_executing(Interpreter, "forbidden_source.xfunc")!
action(Interpreter, stop) if
  has_tag(Interpreter, "stop_me"),
  not is_executing(Interpreter, "forbidden_source.xfunc")!
```

Esimerkissä *stop_me* on merkintä, jolla kiellettyä ohjelmaa suorittava tulkki merkitään. Tulkki sammutetaan, kun se on merkitty, eikä enää suorita kiellettyä ohjelmaa.

Liitoskohtamäärittelyissä ja vihjetoteutuksissa voidaan viitata myös useampiin tulkkieihin. Näin ollen olisi esimerkiksi mahdollista määritellä aspekti, joka tunnistaa lukkiutumisen eli tilanteen, jossa kaksi tulkkia odottaa toistensa varaamia lukkoja.

4 Flow-työkalun toteutus

Xfunc-työkalu poikkeaa toteutukseltaan merkittävästi yleisemmistä ohjelmointityökaluista. Se on rakennettu täysin korkean tason ohjelmointikielen, Javan, päälle ja perustuu oliopohjaiseen malliin. Kielten toteutukset ovat itsenäisiä kokonaisuuksia, jotka kiinnittyvät alustaan toteuttamalla kieliriippumattomia rajapintoja.

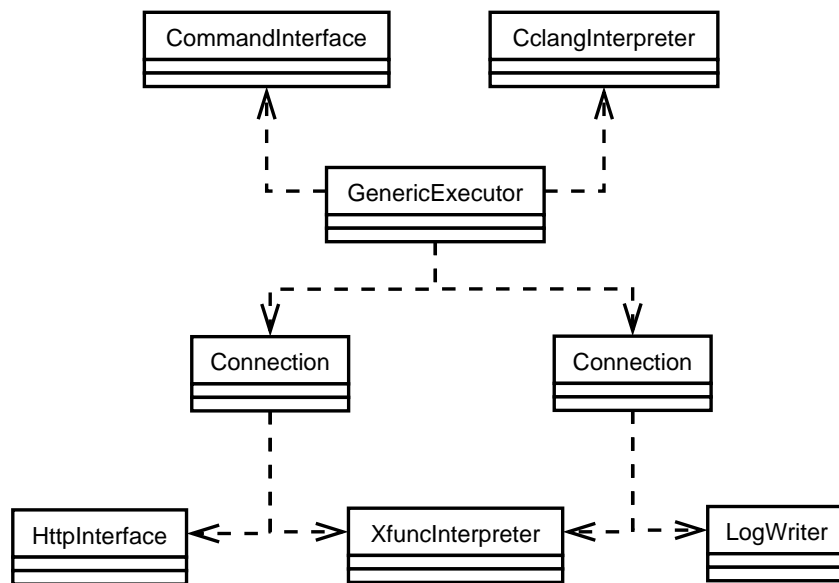
4.1 Flow-alustan toteutus

Flow-työkalun avulla voidaan upottaa Java-pohjaiseen sovellukseen sovellusaluekohtaisilla kielillä toteutettuja komponentteja. Koko työkalu on toteutettu Java-ohjelmointikielellä ja se toimii kaikissa Javan versiota 5 tukevissa ympäristöissä. Flow-työkalu ei tue muita isäntäkieliä. Mahdollisuus upottaa Flow-komponentteja muihin isäntäkieliin olisi monimutkaistanut työkalua huomattavasti, joten sitä ei huomioitu työkalun suunnittelussa.

Flow-työkalun ja sen alustakomponentin ydin on *GenericExecutor*-olio, joka omistaa kaikki muut työkalun osat: komponenttikielten tulkit, Flow-komponentit, meta-kielen tulkin, laiteistorajapinnat ja komponenttitulkkien väliset kytkennät. *GenericExecutor* hallinnoi tulkkien kommunikaatiota ja yhteisiä olioita. *GenericExecutor* hoitaa myös tulkkien suorituksen rinnakkaistamisen. Flow-alustan elinkaari on sama kuin *GenericExecutor*-olion elinkaari. Tarvittaessa yhdessä Java-sovelluksessa voi olla käytössä useita Flow-alustoja eli *GenericExecutor*-olioita – tällä tavoin voidaan esimerkiksi eristää eri käyttäjien Flow-komponentit toisistaan.

Kuvassa 4.1 on esitetty esimerkki mahdollisesta Flow-työkalun kokoonpanosta. Siinä on kytketty laiterajapinta *HttpInterface* *XfuncInterpreter*-tulkin syötteeksi ja laiterajapinta *LogWriter* tulkin tulosteeksi. *GenericExecutor* käsittelee kaikkia olioita, mutta yhteydet laiterajapintoihin ja Xfunc-tulkkiin on jätetty selkeyden vuoksi piirättä.

Esimerkkikokoonpanossa alustaan on kytketty Cclang-metakielen tulkki *CclangInterpreter*. Kuvaan on piirretty myös toinen ulkopuolinen liittymä: ohjelmoija tai järjestelmän ylläpitäjä voi kytkeytyä työkaluun komentokielisen liittymän kautta, jonka toteuttaa luokka *CommandInterface*. Komentoliittymä on suora yhteys metakielen tulkkiin. Ilman työkaluun kytkettyä metakieltä komentoliittymä ei ole käytössä, mutta työkalu on yhä käytettävissä ohjelmointirajapinnan kautta. Aikaisemmissa versioissa alusta tuki suoraan yksinkertaista komentokieltä, mutta se poistettiin,



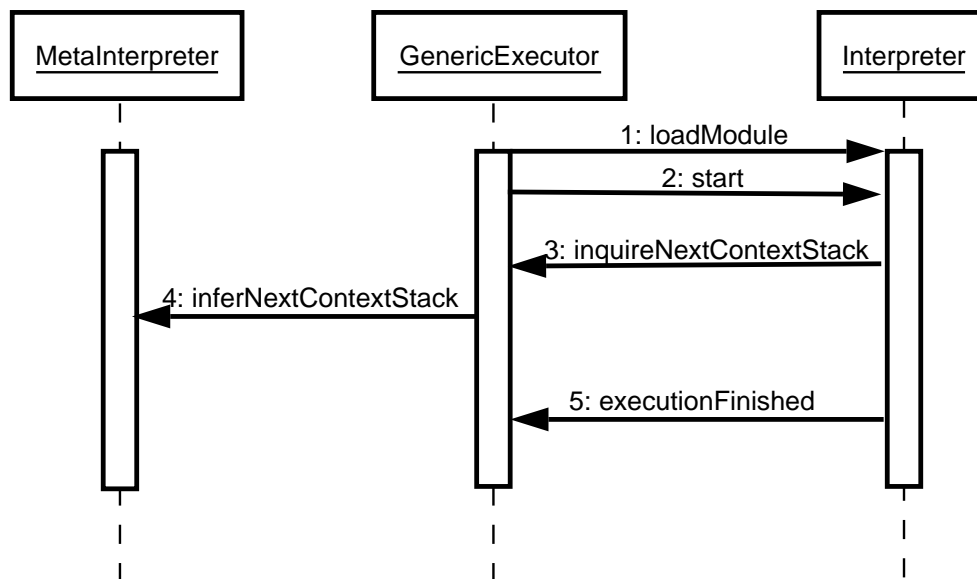
Kuva 4.1: Esimerkki työkalun kokoonpanosta.

koska rajanveto metakielen ja komentokielen välillä oli hankalaa. Hyvin yksinkertainen kieli oli hyödytön, kun taas monimutkaisempi komentokieli olisi sisältänyt metakielen kanssa päällekkäisiä toiminnallisuuksia.

Työkalun nykyversiossa Flow-komponenttien lähdekoodia ei voi muokata liittymien kautta, vaan ne ovat latauksen jälkeen muuttumattomia. Tarkoitus on jatkossa mahdollistaa myös komponenttien luominen ja muokkaaminen ulkoisten liittymien kautta. Tällöin Javaa voidaan käyttää esimerkiksi generoimaan komponentteja Flow-työkalun sisälle ja komentoliittymää voidaan käyttää interaktiivisena ohjelmointiympäristönä.

GenericExecutor luo uuden säikeen jokaiselle suoritettavalle tulkille ja kutsuu tulkia säikeessä, antaen näin kontrollin tulkille. Vuorovaikutus perustuu siihen, että tulkki palauttaa kontrollin vapaaehtoisesti alustalle jokaisen suoritusaskeleen jälkeen. Javan avulla ei olisi ollut mahdollista keskeyttää tulkisäikeen toimintaa pakotetusti, mutta se ei olisi ollut myöskään tarkoituksenmukaista. Tulkin tulee raportoida tilansa alustalle ja muuttaa sitä alustalta tulevien ohjeiden mukaisesti, joten vuorovaikutus voi toimia vain, jos tulkki noudattaa rajapintaan liittyviä sääntöjä.

Komponenttikielen tulkin ja alustan välinen kutsusekvenssi on esitetty kuvassa 4.2. Askeleessa 1 *GenericExecutor* lataa moduulin syöttämällä ladatun lähdekoodin tulkille, joka jäsentää sen ja palauttaa jäsennetyn moduulin. Moduuli tallennetaan alustan moduulivarastoon, joten sitä ei tarvitse jäsentää uudelleen. Alusta hallinnoi



Kuva 4.2: Alustan, komponenttikielen ja metakielen välinen kutsusekvenssi tulkinan aikana.

jäsennettyjä moduuleja, kutsuen jäsentäjää vain silloin, kun moduulia ei ole vielä jäsennetty tai kun sen lähdekoodi on muuttunut.

Askeleessa 2 alusta käynnistää tulkin omassa säikeessään. Käynnistyksen yhteydessä tulkille välitetään *ExecutorCallback*-rajapinta. Komponenttikielen tulkki kutsuu rajapinnan kautta *GenericExecutor*-ydintä askeleessa 3, jotta ydin voi seurata komponenttien tilaa ja tarvittaessa tehdä siihen muutoksia. *GenericExecutor* välittää askeleessa 4 tiedon komponentin tilasta metakielen tulkille, joka muuttaa tilaa, mikäli tulkkiin ladattu metaohjelma niin käskee. Tämän jälkeen kutsu palautuu komponenttikielen tulkkiin, joka siirtyy suorittamaan seuraavaa suoritusaskelta. Lopulta askeleessa 5 komponenttitulkki ilmoittaa alustalle suorituksen päättyneen. Tällöin alusta tekee tarvittavat päivitykset kirjanpitoonsa ja päättää säikeen.

4.1.1 Alustan ohjelmointirajapinnan kuvaus

Isäntäsovellus voi käsitellä Flow-alustaa *ExecutorProgrammingInterface*-rajapinnan kautta, jonka *GenericExecutor*-olio toteuttaa. Myös metakielen tulkki käyttää rajapintaa alustan käsittelyyn.

```

public interface ExecutorProgrammingInterface {
    public String executeMetaCommand(String command)
        throws FlowException;
}
  
```

```

public void addInterpreter(String name, Interpreter interpreter);
public Interpreter getInterpreter(String name);
public boolean isRunning(Interpreter interpreter);
public Iterable<Interpreter> interpreters();
public Connection addConnection(Endpoint from, Endpoint to);
public void removeConnection(Connection connection);
public Iterable<Connection> connections();
public void attachDevice(String name, Device device);
public void detachDevice(String name);
public Iterable<Device> devices();
public Value execute(Interpreter interpreter, Module module)
    throws FlowException;
}

```

Ohjelmointirajapinta sisältää metodeja neljään käyttötarkoitukseen: metakielen käyttöön, komponenttitulkkien käyttöön, yhteyksien käyttöön ja laiterajapintojen käyttöön. Metakieltä kutsutaan *executeMetaCommand*-metodin avulla, joka suorittaa annetun käskyn metakielen tulkissa ja palauttaa tulkin palauttaman merkijonon. Mikäli metakielen tulkkia ei ole kytketty, aiheuttaa metodin kutsuminen poikkeuksen.

Tulkkeja voidaan lisätä alustaan *addInterpreter*-metodin avulla. Tulkin lisäksi parametrina välitetään nimi, jonka avulla tulkki voidaan hakea *getInterpreter*-metodilla. Käynnissä olevat tulkit voi käydä lävitse *interpreters*-metodin palauttaman iteroitavan olion avulla¹⁰.

Alustassa olevia päätepisteitä, eli tulkkeja ja laiterajapintoja, voidaan kytkeä *addConnection*-metodin avulla. Metodi palauttaa viitteen yhteyteen, jota käyttäen se voidaan poistaa *removeConnection*-metodilla. Luodut yhteydet voi käydä lävitse *connections*-metodin palauttaman olion avulla.

Päätepisteet toteuttavat rajapinnan *Endpoint*.

```

public interface Endpoint {
    public String read();
    public void write(String string);
}

```

¹⁰Palautetut arvot ovat tyyppiä *Iterable*, jotta niitä voidaan käyttää Javaan versiossa 5.0 lisätyn laajennetun *for*-rakenteen kanssa.

Yhteydet ovat yksisuuntaisia. Niiden toiminta perustuu yksinkertaiseen *String*-olioiden välittämiseen. Alusta kutsuu yhteyden lähteen *read*-metodia ja välittää sen palauttaman arvon yhteyden kohteen *write*-metodille. Yhteyden päätepisteistä toisen tulee olla tulkki ja toisen laiterajapinta. *read*-metodia kutsutaan jokaisella tulkin suoritusaskeleella. Mikäli *read* palauttaa arvon *null*, sitä ei välitetä. Tällä tavoin alustan ei tarvitse ylläpitää puskuria välitettäviä arvoja varten. Ratkaisu on toimiva vain yksinkertaisimmissa tapauksissa, joten yhteyksien kehittyneempään toteutukseen palataan työkalun jatkokehityksessä.

Laiterajapinta kytketään alustaan *attachDevice*-metodin avulla. Rajapinnalle annetaan nimi, jota käytetään, kun rajapinta poistetaan *detachDevice*-metodilla, ja jonka avulla rajapintaan voidaan viitata metakielessä. Kytkeytyt laitteet voi käydä lävitse *devices*-metodin palauttaman olion avulla.

Tärkein rajapinnan kautta suoritettava kutsu on *execute*. Sen avulla voidaan käynnistää annettu moduuli annetussa tulkissa. Moduuli ladataan tulkin *loadModule*-metodia käyttäen. Tulkin tulee olla lisätty alustan alaisuuteen ennen suorituksen käynnistämistä. Kutsu palauttaa suorituksen tuottaman arvon, jonka tyyppi on *Value*. Se on rajapinta, joka esittää komponenttikielessä esiintyvää arvoa.

```
public interface Value {
    public String asString()
        throws FlowException;
    public boolean valueEquals(Value anotherValue)
        throws FlowException;
    public boolean typeEquals(Value anotherValue)
        throws FlowException;
    public Object asJavaObject()
        throws FlowException;
}
```

Arvoja ja niiden tyyppien yhtäsuuruutta voidaan vertailla, lisäksi arvo voidaan muuttaa merkkijonoksi. Yleisten rajapintojen kautta arvoja voidaan siis käsitellä vain rajatusti. *asString*-metodi palauttaa arvoa vastaavan merkkijonon käyttäjälle esittämiseen soveltuvassa muodossa. *asJavaObject*-metodi palauttaa arvoa vastaavan Java-olion, jos sellainen on. Esimerkiksi Xfuncin funktiotyypisille arvoille ei löydy vastaavaa Java-arvoa, joten tällöin palautetaan *null*.

Ohjelmointirajapinta saadaan käyttöön luomalla *GenericExecutor*-olio. Yksinkertaisimmillaan Xfunc-komponentti voidaan suorittaa komentorajapintaa käyttäen alla esitetyllä tavalla. Metatulkkia ei oteta käyttöön, joten *GenericExecutor*-luontimetodille välitetään parametriksi *null*.

```
ExecutorProgrammingInterface ge = new GenericExecutor(null);
Interpreter interpreter = new XfuncInterpreter();
StringBuffer sourceCode = loadSourceCode();
ge.execute(interpreter, interpreter.loadModule(sourceCode));
```

Alla on esitetty monimutkaisempi esimerkki rajapinnan käytöstä. Se vastaa kuvassa 4.1 esiteltyä kokoonpanoa.

```
ExecutorProgrammingInterface api
    = new GenericExecutor(new CclangInterpreter());
Device httpInterface = new HttpListenerDevice("/my_service");
api.attachDevice("http", httpInterface);
Interpreter xmlProcessor = new XfuncInterpreter();
api.addInterpreter("xml-processor", xmlProcessor);
Device xmlFile = new FileIODevice("my.log");
api.attachDevice("xml-file", xmlFile);
api.addConnection(httpInterface, xmlProcessor);
api.addConnection(xmlProcessor, xmlFile);
StringBuffer sourceCode = loadSourceCode();
Module module = xmlProcessor.loadModule(sourceCode);
api.execute(xmlProcessor, module);
```

Aluksi ladataan moduulin lähdekoodi ja luodaan *GenericExecutor*-olio, johon kytketään Cclang-metakielen tulkki. Sen jälkeen luodaan HTTP- ja tiedostolaiterajapinnat ja Xfunc-tulkki sekä kytketään ne alustaan. HTTP-rajapinta kytketään tulostamaan Xfunc-tulkkiin ja tulkki taas tulostamaan lokitiedostoon. Lopuksi lähdekoodi jäsennetään moduuliksi ja käynnistetään sen suoritus.

4.1.2 Alustan ja tulkkien rajapintojen toteutus

Alustalla on kaksi tulkkirajapintaa: toinen metakielen tulkkiin ja toinen komponenttikielten tulkkeihin. Alustan ja komponenttitulkin välisen rajapinnan ensimmäinen

tapahtuma on se, että alustaa on pyydetty suorittamaan Flow-komponentti. Alusta luo komponenttikielen tulkin instanssin, lataa komponentin ja kutsuu tulkkia.

Alusta käsittelee kaikkia komponenttikielten tulkkeja *Interpreter*-rajapinnan kautta.

```
public interface Interpreter extends Endpoint {
    public Module loadModule(StringBuffer sourceCode)
        throws FlowException;
    public void start(Module module, ExecutorCallback callback)
        throws FlowException;
}
```

Interpreter-rajapinta sisältää kaksi metodia: *loadModule*-metodin avulla alusta pyytää tulkkia jäsentämään moduulin lähdekoodin ja *start*-metodia kutsumalla alusta käynnistää tulkkauksen. Käynnistyskutsun yhteydessä välitetään *ExecutorCallback*-rajapinta, jonka kautta tulkki kutsuu alustaa suorituksensa edetessä. Alustan ja komponenttitulkin vuorovaikutus perustuu tulkin *ExecutorCallback*-rajapinnan kautta suorittamiin kutsuihin.

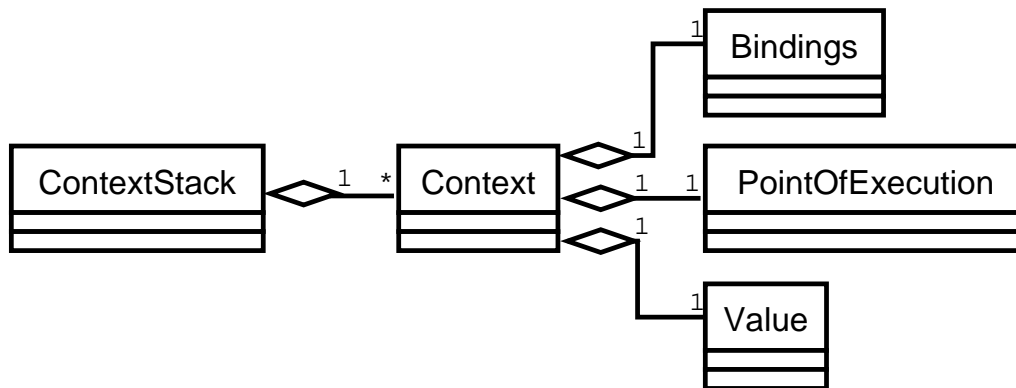
```
public interface ExecutorCallback {
    public ContextStack inquireNextContextStack(Interpreter inquirer,
        ContextStack suggestion);
    public void executionFinished(Interpreter interpreter,
        Value returnValue);
}
```

inquireNextContextStack on keskeinen metodi tulkin toiminnalle. Jokaisen suoritusaskelen jälkeen tulkki päivittää tietorakenteensa vastaamaan seuraavaa suoritusaskelta, kuten funktiokutsua. Kuitenkin ennen kuin tulkki siirtyy suorittamaan seuraava suoritusaskelta, se välittää kontekstipinonsa alustalle *inquireNextContextStack*-metodin kutsun avulla ja saa takaisin uuden kontekstipinon, jolla se korvaa aiemman. Tällä tavoin alusta ja sen alaisuudessa toimiva metakielen tulkki voivat vaikuttaa komponenttitulkin toimintaan. Useimmiten komponentin suoritukseen ei ole tarvetta vaikuttaa, jolloin alusta palauttaa saman kontekstipinon kuin komponenttitulkki on sille välittänyt.

ContextStack-rajapinta esittää kontekstipinoa, jonka avulla tulkki esittää sen hetkisen suorituksen tilan. Kontekstipinon tehtävä on toimia ohjelmointikieli- ja paradigmarippumattomana tapana käsitellä tulkin tilaa. Kontekstipino on rakennettu

kielille yhteisten käsitteiden pohjalta. Vaikkakin kaikille paradigmoille yhteisiä käsitteitä on vaikea löytää, ovat käsitteet nimi, sidonta ja näkyvyysalue sovellettavissa lähes kaikkiin paradigmoihin. Nimi on ohjelmointikielessä esiintyvä nimi, kuten esimerkiksi muuttujan nimi. Se on sidottu johonkin arvoon, kuten kokonaislukuun tai muistiosoitteeseen [Seb01, s. 179-211]. Näkyvyysalue on se ohjelman osa, josta nimi on saavutettavissa [ASU85, s. 394 - 396]. Näkyvyysalue voi olla koko ohjelma, jolloin nimi on määritelty globaalisti, tai jokin rajattu ohjelman osa. Esimerkiksi muuttuja on nimi, joka on sidottu tiettyyn muistissa olevaan tietoalkioon. Sidonnassa lähes aina kohteena olevalla arvolla ja useimmiten myös sitovalla nimellä on tyyppi. Yhteistä eri kielten tyyppienkäsittelysäännöille eli tyyppijärjestelmille on kuitenkin vain se, että tyypit toimivat eräänlaisina merkkeinä, joita arvot ja kielen rakenteet kantavat mukanaan. Tyyppien käytännöllinen merkitys eri kielissä vaihtelee suuresti, joten kaikille kielille yhteistä tyyppijärjestelmää ei Flow-työkaluun pyritty rakentamaan.

Kontekstipino on tavallinen pinotietorakenne, joka koostuu suorituksen konteksteista. Kontekstin tulkinta on kieliriippuvainen. Useimmiten se on aktivaatitietuepino, mutta esimerkiksi Prologin kaltaisten logiikkaohjelmointikielen yhteydessä näin ei olisi. Logiikkaohjelman eteneminen voitaisiin esittää esimerkiksi siten, että kontekstipino on todistuksen haun sen hetkinen hakupolku. Kontekstipinon ei ole tarkoitus kuvata tulkin tila täysin kattavasti, vaan siten, että tieto riittää tulkin tilan seuraamiseen.



Kuva 4.3: Kontekstipino ja siihen kuuluvat osat.

Kuvassa 4.3 on esitetty kontekstipinon rakenne. Kontekstipinon yksittäinen osa on konteksti, joka koostuu nimisidonnoista, suorituspisteestä ja arvosta. *Bindings*-rajapinta esittää kontekstiin kuuluvat nimisidonnat, kuten funktion paikalliset muuttujat. *PointOfExecution* esittää suorituspistettä, kohtaa modulissa, jossa kon-

tekstin suoritus on menossa. Xfuncissa se on viite syntaksipuun solmuun. Kontekstiin liittyvän arvon merkitys riippuu kielestä, Xfuncissa se on funktion paluuarvo.

Metakielen tulkin toteuttama rajapinta poikkeaa merkittävästi komponenttitulkin rajapinnasta.

```
public interface MetaInterpreter {
    public void setMetaExecutorCallback(GenericExecutor executor);
    public List<String> inferInterpretersToBeStarted()
        throws MetaException;
    public String processCommand(String command)
        throws MetaException;
    public ContextStack inferNextContextStack(Interpreter interpreter,
        ContextStack currentStack) throws MetaException;
}
```

Alusta kutsuu jokaisella komponenttitulkin suoritusaskeleella metatulkin metodia *inferNextContextStack*. Palautettu kontekstipino välitetään takaisin komponenttitulkille. Komento- ja ohjelmointirajapintojen kautta välitetyt metakielen komennot välitetään metatulkille *processCommand*-metodin avulla. Lisäksi alusta pyytää metatulkia päättämään käynnistettävät tulkit aina, kun metakielen tulkin tila on muuttunut, eli *inferNextContextStack*- ja *processCommand*-kutsujen jälkeen. Metatulkilla ei ole omaa rajapintaa alustaan, vaan se käyttää yleistä ohjelmointirajapintaa *ExecutorProgrammingInterface*.

4.1.3 Tulkkien rinnakkaistus ja synkronointi

Flow-komponentit suoritetaan rinnakkain. Java-ohjelmoinnissa rinnakkaisuuden toteuttamiseen käytetään säikeitä [GJS96], joten alustasta riippuen komponentteja suoritetaan usealla prosessorilla yhtä aikaa tai yhdellä prosessorilla suoritusvuoroa vaihdellen. Molemmissa tapauksissa komponenttien suorittamat komennot voivat limittyä täysin sattumanvaraisella tavalla, mikä tulee huomioida silloin, kun käsitellään komponenteille yhteisiä tietorakenteita.

Rinnakkaiseen suoritukseen liittyvät ongelmat voidaan jakaa kahteen luokkaan: turvallisuus- ja eloisuusongelmiin [SA86]. Turvallisuusongelmat johtuvat siitä, että yhteisiä tietorakenteita käsitellessä ei oteta huomioon muiden komponenttien toimintaa. Esimerkiksi luettaessa yhteisen tietorakenteen alkuosaa, voi toinen komponentti saada suoritusvuoron ja muuttaa loppuosaa, jolloin alkuperäinen komponentti

lukee tietorakenteen virheellisenä. Turvallisuusongelmat voidaan välttää sarjallistamalla kriittisten osioiden suoritus tarvittavalta osin synkronointirakenteiden avulla. Väärin toteutettu synkronointi voi kuitenkin aiheuttaa eloisuusongelmia. Tällöin komponenttien suoritus sarjallistuu niin, että muut komponentit eivät saa suoritusvuoroa riittävän usein tai ollenkaan. Rinnakkaisen suorituksen toteutus on yksi haastavimmista hajautettuihin järjestelmiin liittyvistä ohjelmointitehtävistä. Onkin perusteltua, että alustaohjelmisto tukee rinnakkaisuuden toteutusta.

Flow-alusta hoitaa komponenttien käynnistämisen erillisiin säikeisiin ja alustaan kuuluvien tietorakenteiden käsittelyn synkronoinnin. Koska alusta hoitaa rinnakkaistuksen, komponenttitulkit eivät saa käynnistää uusia säikeitä. Mikäli tulkeilla on jaettuja tietorakenteita, niiden tulee hoitaa tietorakenteiden käsittelyn synkronointi.

Väärin toteutettu komponenttitulkki voi aiheuttaa vakavia rinnakkaisuusvirheitä. Komponenttitulkkien rajapinnan lähtökohtana on kuitenkin ollut, että alustaa ei pyritä suojaamaan komponenttitulkkien virheelliseltä toiminnalta, koska tällöin olisi jouduttu rajapinnasta tekemään monimutkaisempi ja hitaampi. Komponenttitulkit ovat siis tiukasti kytkettyjä alustaan, ja niissä olevat toimintavirheet aiheuttavat myös alustan virheellisen toiminnan.

Komponenttikielten tulkit toimivat omissa säikeissään, mutta ne palauttavat kontrollin jokaisella askeleella alustalle, joka vuorostaan kutsuu metakielen tulkkia. Alusta ei synkronoi metakielen tulkkiin suoritettavia kutsuja, koska alusta käsittelee tulkkia toteutusriippumattoman rajapinnan kautta. Ainoa tapa hoitaa synkronointi alustassa olisi sarjallistaa metakielen tulkin toiminta kokonaan, joka heikentäisi työkalun suorituskykyä ja skaalautuvuutta merkittävästi – metakielen tulkista tulisi hajautetun järjestelmän pullonkaula. Niinpä säieturvallisuus on toteutettava metakielen tulkissa. Cclang-tulkki perustuu tuProlog-tulkkiin, joka on tehty säieturvalliseksi [DOR01].

4.2 Xfunc-kielen toteutus

Xfunc-tulkki sisältää kaksi toimintoa: lähdekielisten ohjelmien jäsentämisen ja jäsenettyjen ohjelmien tulkitsemisen. Jäsentäminen Xfunc-alustassa ei poikkea jäsentäjien toiminnasta yleensä, kun taas tulkkauksen toteutuksessa joudutaan huomioimaan alustan asettamat rajoitukset ja vaatimukset.

Xfunc-jäsentäjä¹¹ perustuu rekursiivisesti laskeutuvaan algoritmiin (engl. recursive descent). Jäsentäminen tuottaa abstraktin¹² syntaksipuun [ASU85, s. 287 - 289]. Tarkkaan ottaen yksi moduuli voi sisältää useita syntaksipuita, koska jokainen moduulin päätasolla oleva lauseke jäsennetään omaksi puukseen.

Jäsennyksen tuottama syntaksipuun on tietorakenne, joka sisältää kaiken lähdekoodista luettavissa olevan merkityksellisen informaation. Syntaksipuun ohella muita yleisesti käytettyjä tulkattavan ohjelman esittämisen tapoja ovat lähdekieli ja välikieli: jotkut tulkit käsittelevät lähdekielistä ohjelmaa suoraan, kun taas jotkut muuttavat syntaksipuun erityiseen välikieleen ennen tulkkausta. Xfuncissa suoraan lähdekieliseen esitykseen perustuva tulkitseminen olisi ollut hankala toteuttaa, koska tulkitsemisen toteutukseen olisi jouduttu sotkemaan matalamman tason jäsennykseen liittyviä yksityiskohtia – lähestymistapa onkin perusteltu vain hyvin yksinkertaisissa kielissä, kuten komentokieliissä. Toisaalta tarvetta välikielelle tai muulle abstraktia syntaksipuuta korkeamman tason esitykselle ei ollut. Välikielen avulla voitaisiin ohjelmat ja laitteisto eristää paremmin toisistaan. Kuitenkin Flow-työkalun alla oleva Java-virtuaalikone mahdollistaa jo alustariippumattomuuden, joten yksittäisen Xfunc-kielen tulkin toteutuksessa välikielestä ei olisi ollut merkittäviä hyötyjä.

Xfunc-tulkkaus on jaettu kahteen tasoon. Tulkkauksen ylätasolla käydään läpi moduulin lausekkeet ja palautetaan moduulin suorituksen paluuarvona viimeisen lausekkeen palauttama arvo.

```
VariableBindings bindings = new VariableBindings(globalVariables);
XfValue ret = null;
for (AstNode node : module.getNodes()) {
    ret = executeAstFragment(node, bindings);
    if (ret == null) {
        ret = XfNull.getInstance();
        break;
    }
}
callback.executionFinished(this, ret);
```

¹¹Toteutus perustuu Matthew Davisin esittämään oliopohjaiseen menetelmään [Dav00].

¹²Määre abstrakti viittaa siihen, että puusta on poistettu tarpeettomat solmut. Useissa jäsentäjissä on tarpeen tuottaa jäsennyksen hallitsemiseksi solmuja, joita ei enää tarvita jatkokäsittelyssä. Xfunc-jäsentäjän ei syntaksin yksinkertaisuuden vuoksi tarvitse tuottaa tarpeettomia solmuja, joten määrettä abstrakti ei käytetä.

Koodissa luodaan aluksi nimisidonnat, jotka alustetaan globaaleilla muuttujilla. Moduulin lausekkeitä vastaavat syntaksipuut käydään läpi ja kutsutaan alatason suorituksen hoitavaa metodia *executeAstFragment*, antaen syntaksipuun juurisolmu ja sen hetkiset sidonnat parametrina. Kutsu *executeAstFragment* palauttaa suorituksen palauttaman arvon, joka on Javan *null*-arvo, jos alusta on pyytännyt tulkin samuttamista. Tällöin palautetaan arvo *XfNull* ja keskeytetään tulkkaus. Muussa tapauksessa moduulin viimeisen lausekkeen palauttama arvo palautetaan moduulin paluuarvona. Tulkkauksen ylätasolla ei palauteta kontrollia alustalle, joten alusta ei pysty vaikuttamaan ylätason tulkkaukseen.

Lausekkeitä suorittaessa muuttujan *bindings* viittaama yhteinen nimisidontaolio on uloin nimien näkyvyysalue, joten mikäli tulkittava lauseke on nimisidonnasta luova *define*-lauseke, nimisidonta luodaan yhteisiin sidontoihin ja se on näkyvissä kaikille lauseketta seuraaville moduulin lausekkeille. Nimisidonnat siirtyvät seuraavaan lausekkeeseen, jotta Xfunc-ohjelmissa voidaan välttää tarpeetonta määrittelyjen sisäkkäisyyttä. Flow-alusta kannustaa interaktiiviseen ohjelmointiin, ja niinpä Xfunc-kieli pyrittiin toteuttamaan siten, että XML-käsittelyoperaatiot voidaan rakentaa askel kerrallaan. Tällöin on hyödyllistä, että välitulokset voidaan helposti tallentaa. Esimerkiksi jo aiemmin esitetty esimerkki heijastuksen käyttämisestä hyödyntää nimisidontojen siirtymistä.

```
(define $xml
  (parse-xml-file "examples/thesis/xml_example.xml"))
(define $node-selector
  (function ($node) (is-root (parent-of $node))))
(define $nodes
  (select-elements $xml $node-selector))
(define $renamed-nodes
  (map-elements
    (function ($element)
      (rename-element $element "renamed-element"))
    $nodes))
(apply-changes $renamed-nodes)
```

Mikäli noudatettaisiin näkyvyyttä, jossa nimet näkyisivät vain niiden omassa näkyvyysalueessa ja sen sisältämissä näkyvyysalueissa, jouduttaisiin *define* määrittelemään uudelleen siten, että se ottaa kolmanneksi parametriksi lohkon, jossa nimi on

näkyvissä ja jonka palauttaman arvon se palauttaa. Muutoksen vaikutus on nähtävissä seuraavassa esimerkissä, jossa *define*-rakenteen sijasta käytetään kuvatun mukaista kuvitteellista *nested-define*-rakennetta.

```
(nested-define $xml
  (parse-xml-file "examples/thesis/xml_example.xml")
  (nested-define $node-selector
    (function ($node) (is-root (parent-of $node))))
  (nested-define $nodes
    (select-elements $xml $node-selector)
    (nested-define $renamed-nodes
      (map-elements
        (function ($element)
          (rename-element $element "renamed-element")))
        $nodes)
    (apply-changes $renamed-nodes))))))
```

Kuten esimerkistä nähdään, ilman poikkeusta päätason näkyvyyssäännöissä olisi Xfunc-ohjelmien selkeys ja muokattavuus merkittävästi heikompi koodin kasvavan sisäkkäisyyden vuoksi.

Varsinainen syntaksipuun tulkkaukseen suoritetaan alemmalla tasolla, metodissa *executeAstFragment*. Tulkkaukseen perustuu syntaksipuun läpikäyntiin. Läpikäynti suoritetaan käyttäen tulkkauksilmukkaa, jossa tunnistetaan suorituksessa oleva abstraktin syntaksipuun solmu ja suoritetaan solmun tyyppin mukainen koodi.

Tulkki toteuttaa *ContextStack*-rajapinnan esittämällä aktivaatitietuepinon rajapinnan läpi. Yhden *Context*-olion suorituskohda viittaa suoraan abstraktin syntaksipuun solmuun. Tulkki käyttää alustan kontekstipinoa myös oman kirjanpitonsa ylläpitoon.

Vaikka rekursio olisi ollut yksinkertaisin tapa toteuttaa tulkki, sitä ei voitu käyttää kahdesta syystä. Jokainen Xfunc-kielen rekursiotaso olisi kasvattanut myös Javan rekursiotasoa, vaikka Xfunc-kielen pitäisi pystyä häntärekursion optimoinnin avulla rajattoman syvään rekursioon. Toinen rekursioon perustuvan tulkkitutetuksen estävä seikka on, että alustan vaatimien rajapintojen toteuttaminen olisi tullut mahdolliseksi. Alusta ja metakielen tulkki voivat muuttaa tulkin aktivaatitietuepinoa, mutta mikäli sitä vastaisi Javan aktivaatitietuepino, ei sen muuttaminen olisi mah-

dollista. Voidaan ajatella, että rekursiossa käytettävä Javan aktivaatitietuepino on korvattu silmukatoteutuksessa alustan *ContextStack*-rakenteella.

Tulkkausmetodin toteutuksen runko on esitetty alla.

```
XfValue executeAstFragment(AstNode astNode, VariableBindings bindings)
    throws XfuncException { // (1)
    ContextStack stack = new ContextStack(); // (2)
    stack.push(new Context(bindings, astNode)); // (2)
    Context currentFrame = null;
    while (!stack.isEmpty()) { // (3)
        currentFrame = stack.pop(); // (4)
        AstNode currentAstNode
            = (AstNode)currentFrame.getPointOfExecution(); // (5)
        VariableBindings currentBindings
            = (VariableBindings)currentFrame.getNameBindings();
        if (currentAstNode instanceof ValueNode) { // (6)
            // ..
        } else {
            throw new IllegalArgumentException("unknown AST node type: "
                + currentAstNode.getClass().getName());
        }
        stack
            = executorCallback.inquireNextContextStack(this, stack); // (7)
        if (stack == null) { // (8)
            return null; // (8)
        }
    }
    XfValue returned = (XfValue)currentFrame.getValue(); // (9)
    return returned; // (9)
}
```

Metodi *executeAstFragment* suorittaa yhden abstraktin syntaksipuun oksan. Suoritettavan oksan juuri sekä suorituksessa käytettävät muuttujasidonnat annetaan parametrina (kohta 1). Aluksi luodaan aktivaatitietuepinoksi uusi *ContextStack*-olio, johon painetaan aktivaatitietue, jonka suorituskohta on suoritettavan oksan juuri-solmu ja muuttujasidonnat ovat metodikutsussa annetut muuttujasidonnat (kohta

2). Tämän jälkeen astutaan silmukkaan, joka päättyy, kun kontekstipino on tyhjä (kohta 3).

Silmukassa haetaan pinon päällimmäinen konteksti, poistaen se pinosta (kohta 4). Kontekstin suorituskohtaviitteen avulla haetaan abstraktin syntaksipuun solmu (kohta 5). Suoritus etenee syntaksipuun solmun tyyppistä riippuen. Tyypin tunnistamiseen käytetään yksinkertaista ehtorakennetta ja solmurajapinnan toteuttavan olion luokkatietoa (kohta 6). Koska erilaisia solmutyyppejä on yleiskäyttöisiin kieliin verrattuna vähän, ei modulaarisemmalle solmutyyppien käsittelylle ollut tarvetta.

Lopuksi kutsutaan alustaa, välittäen parametrina viite tulkin instanssiin ja sen hetkinen aktivaatitietuepino (kohta 7). Aktivaatitietuepino korvataan kutsun palauttamalla pinolla. Mikäli alusta palauttaa uudeksi pinoksi arvon *null*, pysäytetään tulkkaus välittömästi (kohta 8). Ylätason tulkkausmetodille palautettu *null* keskeyttää sen toiminnan ja metodi palauttaa arvon *XfNull*. Tulkkauksen lopettamiseksi alusta voi myös tyhjätä tulkin kontekstipinon. Kun silmukka on päättynyt, palautetaan sen hetkisen eli viimeisenä pinossa ollaan kontekstin arvo (kohta 9). Eli mikäli tulkkaus on pysäytetty kontekstipino tyhjäällä, palautetaan viimeisimmän kontekstin arvo, eikä *XfNull*.

4.2.1 Funktioiden toteutus

Xfunc-kielen funktioiden rungot on toteutettu kahdella tavalla. Kieleen kuuluvat funktiot eli primitiivifunktiot on toteutettu Java-metodeina, kun taas ohjelmoijan *function*-rakenteen avulla luomat funktiot on toteutettu käärinolioina, jotka kutsuvat tulkkia suorittamaan koodin, joka funktioon on sidottu.

Primitiivifunktion toteutus ottaa syötteen parametrilistan Java-standardikirjaston *java.util.List*-listana ja palauttaa paluuarvon *XfValue*-oliona. Seuraava yhteenlaskufunktion toteuttava esimerkkifunktio havainnollistaa primitiivifunktioiden toteuttamista.

```
public class SumFunction extends XfunctionImpl {
    public XfValue evaluate(List<? extends XfValue> parameters)
        throws XfuncException {
        if (parameters.size() == 0) {
            return new XfNumberImpl(0);
        } else {
            double sum = 0;
```

```

    for (XfValue parameter : parameters) {
        if (!(parameter instanceof XfNumber)) {
            throw new XfuncException("parameter "
                + parameter.asString() + " is of illegal type");
        }
        sum += ((XfNumber)parameter).getNumber();
    }
    return new XfNumberImpl(sum);
}
}
}
}

```

Funktion runko on Java-metodissa *evaluate*. Siinä tutkitaan aluksi parametrilistan kokoa. Mikäli parametreja ei ole, palautetaan lukuarvo 0. Muussa tapauksessa parametrilista käydään läpi ja jokaisen parametrin tyyppi tarkistetaan. Läpikäynnin aikana parametrien summa kerätään muuttujaan *sum*. Lopuksi summa palautetaan Xfunc-tyyppisenä numeroarvona.

Suoritusjärjestys ja sen optimointi

Xfunc käyttää normaalijärjestykseen perustuvaa parametrien välitystä. Xfuncin ohjauksen rakenne *if* on toteutettu kielen sisäisenä funktiona, ja se vaatii parametrien laskemista normaalijärjestyksessä, koska toisen haaran laskeminen voi aiheuttaa virheen. Myös itse luotujen funktioiden yhteydessä normaalijärjestyksestä on hyötyä: monet hajautettuihin järjestelmiin liittyvät operaatiot – kuten verkon yli tehtävät pyynnöt tai tietokantahaut – ovat hyvin raskaita, jolloin operaation viivästyttäminen ja suorittaminen vasta, jos ja kun sitä tarvitaan voi olla merkittävä optimointi. Normaalijärjestyksen vaatima monimutkaisempi kirjanpito ei ole merkittävä suorituskykyhaitta, koska Flow-alusta on korkealla abstraktiotasolla toimiva ja muutenkin raskasta kirjanpitoa käyttävä.

Xfunc-ohjelmassa kaikki funktioiden parametrit lasketaan automaattisesti laiskasti, eli jos ja vasta kun niitä käytetään. Laiska laskenta on toteutettu tyyppikohtaisten käärinluokkien avulla, jotka laskevat sisältämänsä arvon vasta, kun sitä yritetään käyttää luokan saantimetodien avulla. Parametrin arvon tuottava lauseke on aina funktiokutsu, joten arvon laskenta merkitsee aina funktion kutsumista. Niinpä

laiskan laskennan toteutuksessa ratkaistava ongelma on, koska funktiokutsu ja sen vaatima tieto kapseloidaan, jotta suoritusta voidaan tarvittaessa jatkaa siitä.

Kapseloitua suorituksen tilaa – kuten käskyosoitinta ja aktivaatitietuepinoa – kutsutaan jatkeeksi [Sco00, s. 270]. Jatkeita yleisempi rakenne ovat sulkeumat, joita voidaan pitää jatkeiden erikoistapauksena. Sulkeuma on kapseloitu funktiokutsu, johon on liitetty kaikki funktiossa näkyvät nimisidonnat [Sco00, s. 141 - 143]. Tulee huomata, että käsitteitä jatke ja sulkeuma käytetään hyvin toisistaan poikkeavilla tavoilla [Dan04], tässä esitetyt tulkinnat eivät ole ainoita käytettyjä.

Merkittävin ongelma tulkin rakenteessa on sisäänrakennettujen funktioiden ja laiskasti laskettujen parametrien yhdistäminen. Sisäänrakennetut funktiot ovat toteutettu Java-funktioina. Parametrien laiska laskeminen on toteutettu siten, että parametrien laskemiseen tarvittava tieto kääritään *XfDelayedValue*-kääreolioon, eli funktiokutsusta luodaan sulkeuma. Mikäli kääreoliota käytetään, se pyytää aktiivista tulkin instanssia laskemaan sulkeuman ja palauttaa saadun arvon Java-funktiolle. Mikäli sulkeumaa laskiessa joudutaan laskemaan sisäkkäinen sulkeuma, se kasvattaa myös Javan aktivaatitietuepinoa.

4.2.2 Muuttujien ja tyyppien toteutus

Xfunc-kielen tyypit on toteutettu Javalla luokkien ja rajapintojen hierarkiana. Toteutuksessa kaikki tyypit toteuttavat Java-rajapinnan *XfValue*. *XfValue*-rajapintaa laajentavat rajapinnat *XfScalar* ja *XfFunction*. Funktiot ovat siis käsiteltävissä kuten muutkin tietotyypit. *XfScalar*-rajapintaa laajentavat rajapinnat *XfString*, *XfBoolean*, *XfNumber* ja *XfXml*. Lisäksi Xfunc sisältää erikoisarvon *XfNull*, joka on kaikkialla mainittuja tyyppiä. Se on toteutettu Ainokainen-suunnittelumallia [GHJV95] noudattavana luokkana. *XfNull* saa tyypistä riippuen arvot: "" (*XfString*), *false* (*XfBoolean*), *0* (*XfNumber*), tyhjä dokumentti (*XfXml*) ja $f(\dots) = XfNull$ (*XfFunction*). *XfNull* ei siis vastaa semantiikaltaan Javan *null*-arvoa, jolla suoritettavat operaatiot ovat virheellisiä. Löyhemmän käytännön perusteena on se, että Xfuncin kaltaisella yksinkertaisella kielellä suoritettavissa tehtävissä käytön yksinkertaisuus on tärkeämpää kuin mahdollisten väärinkäyttöjen systemaattinen estäminen. Etuliitettä *Xf* on käytetty, jotta vältetään nimiristiriidoilta alustan rajapintojen ja Javan luokkakirjaston luokkien, kuten *String*, kanssa.

Jokaista *Xf*-rajapintaa vastaa sen toteuttava luokka, joka tallentaa arvon sisältämäänsä Java-muuttujaan. Esimerkiksi rajapinnan *XfString* toteuttaa *XfStringImpl*,

joka tallentaa merkkijonon Javan merkkijonoksi. Lisäksi viivästettyä laskentaa varten on *XfDelayedValue*-käärinluokka, joka toteuttaa kaikki rajapinnat. Se sisältää laskettavan lausekkeen ja kutsuu tulkkia, kun arvoa yritetään käyttää saantimetodin kautta. Jokaisella tyyppillä on oma saantimetodinsa, joten laskennan palauttaman arvon tyyppi voidaan tarkastaa ja heittää poikkeus, mikäli tyypit eivät ole samat.

Xfunc-ohjelmassa arvot syntyvät kahdella tavalla: tulkin käynnistyksen yhteydessä luotavien globaalien muuttujien arvoina sekä primitiivifunktioiden palauttamina arvoina. Kun funktiokutsun parametriksi annetaan kutsu toiseen funktioon, käärittää kutsua vastaava lauseke *XfDelayedValue*-olioksi. Jos arvoa käytetään, käynnistetään arvon laskeminen.

Mikäli parametri on muuttujan arvo, siitä välitetään kutsuttavalle funktiolle kopio. Arvoja hallitaan *ValueIdentityManager*-oliolla. Olio suorittaa Xfunc-ohjelmissa käytettyjen arvojen kopioinnin, kun ne välitetään funktiokutsun parametrina. Tällä tavoin arvojen kopiointiin voidaan toteuttaa optimointi, joka tunnistaa tilanteita, joissa kopion sijasta voidaan välittää arvo suoraan. Optimointi on merkittävä suurikokoisia XML-dokumentteja käsitellessä, mutta muille tyypeille sen toteuttaminen ei tulle olemaan perusteltua. Kaikki suurikokoinen data Xfunc-ohjelmissa tulisi olla XML-muodossa ja muiden tyyppien tehtävänä on toimia lähinnä aputietona XML-dokumenttien käsittelyssä. Mikäli tarpeettomien kopiointien havaitsemistoiminnallisuus ei ole erittäin nopea, on varmastikin tehokkainta kopioida aina muut tietotyypit. Tyyppien jako muistuttaisi Javan parametrien välitystä, jossa oliot välitetään viitteenä¹³ ja pienikokoisemmat primitiivi-arvot välitetään arvona.

XML-elementtien heijastamisen vuoksi tulkki joutuu pitämään kirjaa siitä, mitä alkuperäistä elementtiä kielen avulla muokattu XML-elementti vastaa. Kirjanpito on hoidettu erillisen *ValueIdentityManager*-luokan avulla. Jotta lähdeviitteet säilyvät, ne tulee huomioida myös arvoja kopioidessa. Aluksi hallinta toteutettiin suoraan XML-tyyppiin eli *XfXmlImpl*-luokkaan, mutta ulkopuolinen hallintaolio osoittautui selkeämmäksi ratkaisuksi.

Tyyppejä ja niihin liittyviä sääntöjä kutsutaan tyyppijärjestelmäksi [ASU85, s. 344 - 348]. Tyyppijärjestelmä määrittelee, mitkä tyypit ovat yhteensopivia, mitkä tyypit ovat toistensa alityyppejä ja mitkä tyypit ovat muunnettavissa miksi tyypeiksi [Sco00, s. 180 - 233]. Tyypin ja alityypin välinen suhde muistuttaa yhteensopivuutta. Tällöin kuitenkin ainoastaan alityypin mukaista tietoalkiota voidaan käyttää ylätyy-

¹³On tosin tulkintakysymys, välitetäänkö Javassa oliot viitteenä tai olioviitteet arvona, koska Javassa oliomuuttujat ovat aina olioviitteitä.

pin alkiona, mutta ei päinvastoin. Xfuncin tyytit eivät ole yhteensopivia, eikä tyyppien välillä ole alityyppisuhteita. Tyyppijärjestelmä on siis tässä suhteessa yksinkertainen. Tyyppien välille voidaan myös määritellä muunnoksia, kuten useimmissa kielissä liukuluku voidaan muuntaa kokonaisluvuksi, jolloin esimerkiksi desimaaliosa tiputetaan luvusta pois. Xfunc tukee muunnoksia kaikkien skalaarityyppien kesken. Useissa kielissä osa muunnoksista on implisiittisiä, eli kääntäjä tai tulkki suorittaa muunnoksen aina tarvittaessa ilman, että ohjelmoija erikseen pyytää sitä [ASU85, s. 359 - 360]. Xfuncin nykyversio ei sisällä implisiittisiä muunnoksia, mutta koska ne helpottaisivat kielen käyttöä, on syytä harkita niiden lisäämistä myöhemmin.

Tyyppijärjestelmät voidaan luokitella tyyppityksen ajankohdan perusteella dynaamisiin ja staattisiin sekä tyyppityksen kattavuuden perusteella heikkoihin ja vahvoihin [Sco00, s. 320 - 322]. Staattinen tyyppijärjestelmä suorittaa kaikki tyyppitarkistuksensa ennen ohjelman käynnistämistä, yleensä käännöksen yhteydessä. Staattisen tyyppityksen etu on, että monet virheet pystytään havaitsemaan ilman ohjelman testaamista. Staattinen tyyppitys kuitenkin monimutkaistaa kielen käyttöä, joten sitä ei pidetty soveltuvana Xfuncin kaltaiseen kieleen. Xfuncin tyyppijärjestelmä on kuitenkin vahva. Se pystyy suorituksen aikana takaamaan, että ohjelmassa ei synny havaitsemattomia tyyppivirheitä eli ohjelmissa ei sovelleta operaatioita arvoihin, joille ne eivät ole määriteltyjä.

Muuttujien näkyvyyden toteutus

Symbolitaulu sisältää ohjelmassa käytetyt symbolit – kuten muuttujien nimet – ja tiedon niiden näkyvyysalueesta. Symbolitaulun rakenne ja toiminta riippuvat kielen näkyvyysäännöistä, kuten esimerkiksi siitä, ovatko ulomman näkyvyysalueen nimet näkyvissä sisemmässä näkyvyysalueessa. Xfunc noudattaa dynaamista näkyvyyttä, eli nimet sidotaan arvoihin ohjelman suorituksen aikana. Näin ollen ei voida käyttää yksinkertaista staattista symbolitaulua, kuten LeBlanc-Cook-symbolitaulua [Sco00, s. 132-139]. LeBlanc-Cook-symbolitaulu on suunniteltu käännöksen aikana käytettäväksi, eikä sen perustoteutuksessa symboleja poisteta taulusta.

Suorituksen aikaista dynaamista näkyvyyttä käytettäessä joudutaan huomioimaan myös symbolien poistaminen taulusta. Tietorakenteen valinnassa keskeistä on löytää kielen luonteeseen sopivat suorituskykypiirteet: joillakin toteutuksilla symboliviitteiden ratkaiseminen on nopeaa, kun taas toisilla näkyvyysalueisiin siirtyminen ja niistä poistuminen on nopeaa. Yleisimmät dynaamisen näkyvyyden toteutukset

ovat assosiaatiolista ja keskitetty viitetaulu [Sco00, s. 132-139]. Assosiaatiolistassa nimisidonnat muodostavat pinorakenteen, jossa sisimmässä näkyvyysalueessa olevat sidonnat ovat päällimmäisenä. Näkyvyysalueeseen tullessa uudet nimet sijoitetaan pinon päälle ja poistetaan sieltä, kun näkyvyysalueesta poistutaan. Uloimmassa näkyvyysalueessa määritellyn nimen hakeminen vaatii koko listan läpikäyntiä, joten viitteiden hakeminen on hidasta. Haku voidaan tehdä nopeammin käyttämällä keskitettyä viitetaulua. Siinä jokaisella nimellä on oma alkionsa, joka viittaa pinoon nimisidontoja, joista sisimmäisessä näkyvyysalueessa määritelty on päällimmäisenä. Näkyvyysalueeseen tuleminen tai sieltä poistuminen vaatii jokaisen siinä alueessa määritellyn nimen kohdalla olevien nimisidontapinojen päällimmäisen alkion lisäämistä tai poistamista. Koska näkyvyysalueet ovat Xfuncissa pieniä ja ne vaihtuvat usein, ei keskitetty viitetaulu ole tehokas ratkaisu.

Xfunc-kielessä muuttuja on näkyvissä määrittelylohkossa, sen alilohkoissa, näkyvyysalueella luoduissa sulkeumissa ja saman lähdekoodimoduulin seuraavissa lohkoissa. Symbolitaulun pinototeutus pohjautuu assosiaatiolistaan. Jokainen pinossa oleva näkyvyysalue on *VariableBindings*-olio. Kun muuttujaviittausta selvitetään, kutsutaan sisimmäisen näkyvyysalueen *get*-metodia. Se hakee muuttujaa nimen perusteella kyseisen näkyvyysalueen sidonnat sisältävästä hajautustaulusta. Mikäli muuttujaa ei löydy, kutsutaan vanhempisolmun vastaavaa metodia ja palautetaan sen palauttama arvo. Näin päädytään puussa ylöspäin, kunnes sidonta löytyy tai vanhempaa ei ole, jolloin heitetään poikkeus, joka kertoo tulkille, että nimeä ei ole määritelty.

Jotta funktio voidaan välittää parametrina tai paluuarvona, joudutaan välittämään myös sen ympäristö eli luontihetkellä voimassa olleet nimisidonnat. Käyttäjän määrittelemän funktion kutsuhetkellä haetaan funktion sisältävästä sulkeumasta funktion luontihetken nimisidonnat ja painetaan myös ne pinon päälle, ennen parametreja. Näin ollen vaikka näkyvyysalueet on toteutettu pinona, vastaavat ne toiminnallisuudeltaan puuta – pinossa olevat alkiot voivat sisältää uuden pinon. Tätä korkeamman kertaluvun funktioiden näkyvyyden toteuttamiseen liittyvää ongelmaa kutsutaan funarg-ongelmaksi. Vaikka kyseessä on merkittävä ohjelmointikielten toteutukseen liittyvä kysymys, ongelma on kuitenkin pitkälti suorituskykyongelma: pinoon perustuva symbolitaulu voidaan usein toteuttaa merkittävästi tehokkaammin matalan tason optimointeja käyttäen. Koska Flow-alusta on muutenkin korkealla abstraktiotasolla toimiva, yksinkertainen puutyypin toteutus on parhaiten soveltuva.

5 Flow-työkalun arviointi

Tässä luvussa arvioidaan tutkielmassa saatujen kokemusten pohjalta Flow-työkalun kehittämisprosessia, teknistä toteutusta sekä sopivuutta tehtävänsä. Työkalua ja sen osia verrataan muihin ratkaisuihin, tosin aihepiirin uutuuden vuoksi suoria vertailukohteita ei kuitenkaan juuri ole.

5.1 Työkalun kehittämisprosessi

Suoraan Flow-työkalua vastaavia ohjelmistoja ei ole, joten sen kehityksessä jouduttiin soveltamaan muun tyyppisten ohjelmointityökalujen yhteydessä kehitettyjä menetelmiä. Koska tuotettu työkalu oli pitkälti kokeellinen, pidettiin sopivimpana kehitysprosessina iteratiivista prosessia. Työkaluun luotiin uusia ominaisuuksia asetteittain ja niiden toimivuutta tutkittiin ohjelmointikokeiluilla. Kehitys alkoi Xfunc-kielen ensimmäisen version kehittämisellä, minkä jälkeen kehitettiin alustaohjelmisto ja metakieli. Kaikkia osia kuitenkin refaktoroitiin koko kehityksen ajan [Fow99].

Kehitysprosessin kannalta ongelmallista oli työkalun osien keskinäinen riippuvuus, mikä esti täysin iteratiivisen prosessin käytön. Esimerkiksi kun ohjelmointikokeiluja tehdessä havaittiin kielessä puute, vaati sen korjaaminen kielen toteutuksen, kirjastofunktioiden ja myös mahdollisesti alustan muuttamista. Mikäli jatkossa alustan ominaisuuksia lisätään, vaarana on, että työkalun osien kytkentä tulee vieläkin vahvemmaksi. Niinpä jatkokehityksen kannalta on tärkeää, että alustan ydinosat voidaan stabiloida, jotta niissä tapahtuvat muutokset eivät häiritse liikaa muiden osien kehitystä.

Flow-työkalua kehitettäessä käytettiin yksikkötestausta, eli luotiin joukko testikomponentteja ja tutkittiin, että ne palauttavat halutun arvon. Testit ajettiin aina, kun toteutukseen tehtiin merkittäviä muutoksia. Ilman testausta muutosten tekeminen alustan rakenteeseen olisi ollut hyvin vaikeaa, koska pienet virheet olisivat helposti jääneet huomaamatta ja niiden selvittäminen myöhemmin olisi ollut vaikeaa. Java-ohjelmien refaktorointia tukevan Eclipse-kehitysympäristön avulla voitiin tehdä muutoksia työkalun toteutukseen helposti ja muutosten toimivuus voitiin varmistaa yksikkötestien avulla. Sen sijaan kielen muuttaminen osoittautui hankalaksi, koska kielen piirteiden muuttamisen kaikkia vaikutuksia ja mahdollisia ristiriitoja oli vaikea arvioida ennalta. Kokemusten perusteella ohjelmointikielten rakenteiden valinta on vaikeaa ilman niiden testaamista käytännössä, joten kieltä kehitettäessä kannattaa panostaa kielen toteutukseen, joka mahdollistaa kielen refaktoroinnin mah-

dollisimman helposti. Esimerkiksi mikäli tulkin toteuttamiseen olisi ainakin osittain käytetty generaattoria, olisi muutosten tekeminen ollut helpompaa.

Suunnittelun kannalta haaste oli myös löytää oikea taso, jolla muutokset toteutettiin. Kokeiluissa havaittuihin muutostarpeisiin voitiin vastata muuttamalla kieltä, kirjastoja tai alustaa – aina ei ollut ilmeistä, mikäli oli oikea valinta. Esimerkiksi heijastusta varten oli tarkoitus luoda oma syntaktinen rakenne. Se kuitenkin toteutettiin aluksi yksinkertaisuuden vuoksi funktiona. Kuitenkin osoittautui, että tarvetta oman rakenteen tekoon ei ollut, vaan funktioiden toteutusta käyttävät ohjelmat olivat riittävän yksinkertaisia. Generaattorin avulla olisi erilaisia ratkaisumalleja voitu kokeilla helpommin myös muita kielen piirteitä kehitettäessä.

5.2 Työkalun käyttö

Flow-työkalu kehitettiin Sysbio-projektissa ilmenneisiin tarpeisiin. Sen avulla kokeneemmat ohjelmoijat voivat toteuttaa yksinkertaisia sovellusaluekohtaisia kieliä, joiden avulla sovellusalueiden asiantuntijat voivat rakentaa heidän vastuualaansa kuuluvia komponentteja. Lisäksi yhtenäisen alustan avulla voidaan toteuttaa alustan tilaa valvovia metaohjelmia.

Periaatteessa alusta täyttää tärkeimmän sille asetetun tavoitteen, eli mahdollistaa hyvinkin erityyppisten sovellusaluekohtaisten kielten integroinnin. Tätä ei kuitenkaan tutkielman puitteissa kokeiltu käytännössä, koska alkuperäisen rajauksen mukaisesti työkaluun toteutettiin vain yksi komponenttikieli.

5.2.1 Xfunc-kieli ja muut XML-työkalut

Xfunc-ohjelmat ovat helposti kirjoitettavia, koska syntaksi on yksinkertainen. Yksinkertaisuuden aiheuttama ongelma on heikompi luettavuus. Tavallinen Xfunc-ohjelma koostuu peräkkäisistä funktiokutsuista, jotka voivat sisältää sisäkkäisiä funktiokutsuja, kuten alla on esitetty.

```
(funktio1 (funktio2 "p1") (funktio3 "p2" "p3" (funktio4 1 2)))  
(funktio5 "p4")
```

Funktiot suoritetaan järjestyksessä *funktio2*, *funktio4*, *funktio3*, *funktio1* ja *funktio5*. Ohjelman lähdekoodia luettaessa peräkkäiset funktiokutsut suoritetaan lukujärjestyksessä, kun taas sisäkkäiset funktiokutsut suoritetaan päinvastaisessa järjestyksessä.

sä – sisempi kutsu suoritetaan ennen ulompaa, vaikka ulompi on lähdekoodissa aina ensimmäisenä. Tällöin ohjelman staattinen ja dynaaminen muoto eivät vastaa toisiaan. Ohjelman toiminnan päättelemisen sen lähdekoodista on työläämpää, mikä heikentää ohjelman luettavuutta [Dij68].

Xfunc-ohjelmat tukevat pienen mittakaavan modularisointia. Korkeamman asteen funktioiden avulla eri XML-käsittelyn osat voidaan erottaa toisistaan. Xfunc-funktioita tai komponentteja ei voida kuitenkaan yhdistää suuremmiksi kokonaisuuksiksi, esimerkiksi yhdistää nimiavaruuksiksi. Suuren mittakaavan modularisointiominaisuuksia ei pidetty perusteltuina yksinkertaisessa skriptikielessä, vaikka alustaan niitä olisikin syytä jatkossa kehittää.

Xfunc-kielen tulisi mahdollistaa XML-käsittely Javaa helpommin. Tarkastellaan aiemmin esiteltyä XML-dokumenttia, josta haluamme muuttaa tiettyjen solmujen nimet.

```
<?xml version="1.0" ?>
<root>
  <element name="element1">
  </element>
  <element name="element2">
    <element name="inner-element">
    </element>
  </element>
</root>
```

Muutettavat solmut ovat juurisolmun lapsisolmut, joiden nimeksi vaihdetaan *renamed-element*. Muutos tuottaa seuraavanlaisen XML-dokumentin.

```
<?xml version="1.0" ?>
<root>
  <renamed-element name="element1">
  </renamed-element>
  <renamed-element name="element2">
    <element name="inner-element">
    </element>
  </renamed-element>
</root>
```

Xfunc-kielen avulla muutos on melko yksinkertainen toteuttaa heijastusta käyttäen. Esimerkissä luodaan solmut valitseva *node-selector*-funktio. Sen jälkeen kutsutaan heijastuksen suorittavaa *apply-changes*-funktioita, joka ottaa parametriksi muutettujen solmujen joukon.

```
(define $xml
  (parse-xml-file "examples/thesis/xml_example.xml"))
(define $node-selector
  (function ($node) (is-root (parent-of $node))))
(apply-changes
  (map-elements
    (function ($element)
      (rename-element $element "renamed-element"))
    (select-elements $xml $node-selector)))
```

Vastaava Java-ohjelma on sen sijaan hieman monimutkaisempi. Jotta Java-ohjelma olisi verrattavissa Xfunc-ohjelmaan, tulee solmujen valintaehdon olla kapseloitu. Se voidaan toteuttaa DOM-rajapintaan kuuluvan *NodeFilter*-olion avulla. Esimerkissä oletetaan, että DOM-dokumentti on sijoitettu muuttujaan *xml*, jolla on määre *final*, jotta siihen voidaan viitata anonyymissä sisäluokassa.

```
NodeFilter filter = new NodeFilter() {
  public short acceptNode(Node n) {
    if (n.getParentNode() == xml.getDocumentElement()) {
      return FILTER_ACCEPT;
    } else {
      return FILTER_REJECT;
    }
  }
};
NodeIterator iterator = ((DocumentTraversal)xml).createNodeIterator(
  xml.getDocumentElement(), NodeFilter.SHOW_ELEMENT,
  filter, false);
for (Node node = iterator.nextNode(); node != null;
  node = iterator.nextNode()) {
  xml.renameNode(node, "", "renamed-element");
}
```

Luotu *NodeFilter*-olio tarkistaa, että onko solmun vanhempi sama kuin dokumentin juuri. Halutut solmut kerätään iteroitavaksi kokonaisuudeksi, ja solmujen nimet vaihdetaan silmukassa. Ohjelma on hyvä esimerkki siitä, kuinka oliokirjastoilla voidaan mallintaa muita paradigmoja: ratkaisu muistuttaa pitkälti funktionaalisten XML-työkalujen toimintaa. Kuitenkin ylimääräisen kerroksen ja Java-syntaksin vuoksi ohjelma on pidempi ja vaikeaselkoisempi. Java-ratkaisun ymmärrettävyydessä keskeistä onkin olio-ohjelmoinnin, suunnittelumallien, DOM-rajapinnan sekä Javan idiomien hallitseminen. Ohjelmoijan kyky hallita joukkoa yhtäaikaista käsitteitä on hyvin rajallinen. Pelkästään Javan perusteet hallitsevalle ohjelmassa on paljon yksityiskohtia: *NodeFilter*-yläluokka, *filter*-olio, *acceptNode*-metodin toteutus, *if*-kontrollirakenne ja sen kaksi haaraa, *DocumentTraversal*-rajapinta, *createNodeIterator*-metodin kutsu parametreineen, *iterator*-olio, *for*-toistorakenne ja *renameNode*-kutsu. Kaikkien näiden käsitteiden hallinta yhtäaikaaisesti on hankalaa, ellei niitä kykene aiemman kokemuksen pohjalta hahmottamaan suurempina kokonaisuuksina, kuten *filter*-oliota vakiintuneena tapana suodattaa joukko XML-dokumentin solmuja pois. Xfunc-ohjelman ymmärtäminen vaatii vähemmän esitietoja.

Yleinen ratkaisu XML-muotoisen tiedon käsittelyyn on siihen tarvittavien rakenteiden integrointi olemassa olevaan yleiskäyttöiseen ohjelmointikielen. Tähän lähestymistapaan perustuu esimerkiksi Xen-ohjelmointikieli, jossa C#-kieleen on lisätty rikkaampi tyyppijärjestelmä, jonka avulla voidaan käsitellä luontevasti XML-dokumentteja ja relaatiotietokantoja [MS03]. Lähestymistavan etu Flow-työkaluun verrattuna on, että lisätty käsittelylogiikka tulee osaksi alkuperäistä kieltä. Flow-työkalun tavoitteena kuitenkin oli mahdollistaa yksinkertaisten kielten käyttö, jotta vähäisen ohjelmointikokemuksen omaavatkin henkilöt voivat toteuttaa rajattuun tehtävään tarkoitettuja komponentteja. Xen-kielen hallitseminen ei vaadi pelkästään monimutkaisen C#-kielen osaamista, vaan sen lisäksi myös varsin abstraktien tyyppijärjestelmän laajennusten sisäistämistä. Xen onkin lähinnä kokeneemmille ohjelmoijille tarkoitettu työkalu. Toinen Flow-työkalun etu on, että siihen voidaan lisätä uusia kieliä tarpeen mukaan – esimerkiksi matriisienkäsittelyrakenteiden lisääminen Xen-kielen olisi paljon työläämpää kuin matriisienkäsittelykielen lisääminen Flow-alustaan.

5.2.2 Cclang-metakieli alustan hallinnan ja monitoroinnin välineenä

Metakielen tehtävä on toimia alustan ylläpidon, komponenttien kytkennän ja aspektiohjelmoinnin välineenä. Metaohjelmointikieli täyttää tärkeimmän sille asete-

tun tavoitteen, eli mahdollistaa komponenttien monitoroinnin. Alustan ylläpito on nykyversiossa verrattain yksinkertaista ja metakieli soveltuukin myös siihen hyvin. Mikäli alustaa kehitetään komponenttien hallinnan suuntaan, esimerkiksi mahdollistamalla versionhallintajärjestelmän käsittely tai komponenttien muokkaus alustan kautta, joudutaan metakielen alustan hallinnan menetelmiä kehittämään.

Myös komponenttien kytkentään metakielen ominaisuudet ovat riittävät. Sen sijaan aspektiohjelmointia voidaan pitää vielä kohtuullisen hankalakäyttöisenä. Sitä voitaisiin kehittää joko luomalla Cclang-kirjasto aspektiohjelmoinnin avuksi tai kehittämällä itse kielen aspektipiirteitä. Koska kieli pohjautuu valmiiseen tulkkito-teutukseen, vaatisi kielen muuttaminen esikäsitteilyominaisuuksien lisäämistä, mikä monimutkaistaisi sen toteutusta.

Metakieli toteutettiin ulkopuolisen tuProlog-tulkkitoteutuksen [DOR01] avulla, jota täydennettiin omalla esikäsitteilyllä. Ratkaisu osoittautui toimivaksi. Metakielen toteutuksen merkittävin haaste oli deklaratiiivisen metakielen ja imperatiivisten alustan sekä komponenttitulkkiin yhdistäminen. Ongelmallisuus ei kuitenkaan johnutun valitusta tuProlog-toteutuksesta tai Prolog-kielestä, vaan kyseessä on yleinen paradigmojen välinen ongelma. Toimenpiteiden suorittaminen aikajärjestyksessä ei sovellu hyvin deklaratiiiviseen kieleen, jossa suoritusjärjestys on piilotettu ohjelmoiljalta.

Deklaratiivisten rakenteiden ja aikaan sidottujen toimenpiteiden yhdistämiseen liittyvää ongelmaa on tutkittu erityisesti funktionaalisten kielten syöttö- ja tulostusominaisuuksien yhteydessä. Yksinkertainen ratkaisu on kytkeä deklaratiiivisten rakenteiden laskentaan sivuvaikutuksia, kuten myös tuProlog-tulkissa on tehty. Ratkaisua ei kuitenkaan pidetä hyvänä vaikean hallittavuutensa vuoksi [Wad97]. Niinpä sitä ei käytetty Cclang-kielessä, vaan tuProlog-tulkin päälle rakennettiin helpommin hallittava vuorovaikutuksen toteutus. Vaihtoehtoisesti ratkaisu olisi voinut perustua eteenketjutukseen (engl. forward chaining) [JS96]. Tällöin päättely ei perustuisi annettuun väittämään, jonka totuusarvoa pyritään selvittämään, vaan tunnettujen faktojen pohjalta päätettäisiin kaikki toiminnan kannalta tärkeät seuraukset. Menetelmää on sovellettu menestyksellä hajautettuihin järjestelmään integroitaviksi tarkoitetuissa sääntöpohjaisissa järjestelmissä kuten Drools¹⁴. Prolog perustuu taaksetjutukseen, joten sitä ei voitaisi käyttää suoraan kielen pohjalla.

Metakieli on erityisen merkityksellinen alustan suorituskyvyn kannalta. Ohjelmointikielten kehityksen historiassa uusia ohjelmointityökaluja on usein kritisoitu siitä,

¹⁴<http://drools.org/>

että niillä tuotetut ohjelmat ovat hitaampia ja ne kuluttavat enemmän muistia kuin aikaisemmillä työkaluilla tuotetut, koska korkean tasot ominaisuudet ovat usein raskaampia toteuttaa ja vaativat monimutkaisempaa kirjanpitoa. Erityisen paljon kritisoidaan sitä, että sellaiset ominaisuudet, joita ohjelmassa ei käytetä, aiheuttavat ajonaikaista kuormaa tai monimutkaistavat ohjelman lähdekoodia. Niinpä onkin hyvä pyrkiä siihen, että kielen kehittyneemmät ominaisuudet tuovat ylimääräistä kuormaa vain, kun niitä käytetään. C++-kielen kehittäjä Bjarne Stroustrup esitti tämän ajatuksen muodossa: “what you don’t use, you don’t have to pay for.” Sääntö oli ehkä yleisin syy hylätä C++-kieleen ehdotettu uusi ominaisuus [Str94, s. 120 - 122].

Koska alusta kutsuu metakielen tulkkia jokaisella komponenttikielten suoritusaskeleella, myös metakielen tulkin toteutuksella on vaikutus komponenttien suorituskykyyn. Työkalun korkeamman tason ominaisuudet pyrittiin suunnittelemaan niin, että ne eivät hidasta ohjelmia silloin, kun ne eivät ole käytössä. Niinpä metakielen ei pitäisi hidastaa komponenttiohjelmien toimintaa silloin, kun komponentteihin vaikuttavia sääntöjä eli aspekteja ei ole ladattu. Näin on, koska aspektien toiminta perustuu siihen, että alusta suorittaa todistuksen haun predikaateille *action* ja *invariant_action*. Aspektien seurauksessa on aina predikaattina *action*, eli aspektit ovat seuraavan muotoisia.

```
action(Interpreter, some_action) if
    some_condition(Interpreter)!
```

Mikäli Cclang-metakieleen ei ole ladattu yhtään sääntöä, jonka seurauksena on *action*- tai *invariant_action*-predikaatti, päättyy todistuksen haku välittömästi. Eli ilman aspekteja ainoa merkittävä metakielen aiheuttama lisätyö on alustan tilaa vastaavien tietorakenteiden päivitys. Päivitettävät tietorakenteet ovat kuitenkin hitaasti päivittyviä, kuten käynnissä olevien tulkkien lista. Jokaisella suoritusaskeleella päivittyviä rakenteita, kuten tulkin nimisidontoja, käsitellään laiskojen predikaattien avulla. Ne hakevat tilatiedon vain, jos sitä tarvitaan todistuksen haussa.

Kun alustaa tarkastellaan osana ohjelmistotuotantoprosessia, on työkalun nykyinen tekninen rakenne ongelmallinen. Työkalun käyttö erilaisissa tuotantotehtävissä vaatii sen kehittämistä ja täydentämistä, mikä kuitenkin on varsin vaativaa. Uusien komponenttikielten toteuttaminen on myös työlästä. Työkalun tarkoitus on tarjota Java-pohjaisia monimutkaisia oliomalleja kevyempiä menetelmiä hajautetun järjestelmän osien toteuttamiseen, joten työkalun tekninen raskaus heikentää sen käytettävyyttä varsinaiseen tehtäväänsä.

5.3 Työkalun toteutus

Flow-työkalun on tarkoitus yksinkertaistaa hajautettujen järjestelmien kehitystä tarjoamalla raskaita oliomalleja helppokäyttöisempiä välineitä järjestelmien rakentamiseen. Työkalu pyrkii yksinkertaistamaan hajautettuun ympäristöön liittyviä ohjelmointiongelmia, kuten suorituksen rinnakkaisuuteen liittyviä ongelmia, ja sen toteutus onkin kohtuullisen monimutkainen. Niinpä alustan suunnittelun tärkein tehtävä on hallita monimutkaisuutta.

Alustan ytimellä eli *GenericExecutor*-oliolla on kolme rajapintaa: ohjelmointirajapinta ja ulkoinen komentoliittymä sekä työkalun sisäinen rajapinta komponenttitulkeille. Useat rajapinnat monimutkaistavat *GenericExecutor*-luokan toteutusta, koska samaan luokkaan on sisällytetty paljon erityyppistä toiminnallisuutta. Jatkokehityksessä *GenericExecutor* tulisi refaktoroida joukoksi pienempiä luokkia. Myös metatulkeilla oli oma rajapinta alustaolion käsittelyyn: kaikille liittymille luotiin omat rajapintamäärittelyt, ja alustan kehittyessä metakielen käyttämä rajapinta kehittyi lähes ohjelmointirajapinnan osajoukoksi. Niinpä rajapinta poistettiin ja metatulkki muutettiin käyttämään yleistä ohjelmointirajapintaa.

Alusta helpottaa yksinkertaisten rinnakkaisohjelmointitehtävien toteuttamista. Esimerkiksi toisistaan riippumattomien komponenttien suorittaminen useassa säikeessä tai komponenttien putkittaminen onnistuu ilman, että komponenttien ohjelmoijan tarvitsee huomioida rinnakkaisuutta tai että komponenttikielen toteutukseen tarvitsee lisätä rinnakkaisuusominaisuuksia. Monimutkaisemmat rinnakkaisohjelmointitehtävät, kuten toisistaan riippuvien tehtävien suorittaminen rinnakkain, vaativat kuitenkin rinnakkaisominaisuuksien toteuttamista komponenttikieleen. Tällöin alusta ei tarjoa apua rinnakkaisohjelmointiongelman ratkaisemisessa.

Alustan nykyversiossa komponenttien välisten yhteyksien kautta kuljetettavia arvoja ei puskuroida, joten komponentit joutuvat odottamaan toisiaan. Puskuroinnin puutteen vuoksi varsinkin useiden putkitettujen komponenttien läpimenoa on huono. Kuljetettavat arvot ovat yksinkertaisesti merkkijonoja. Mikäli käytettäisiin eri tyyppisiä arvoja, voitaisiin yhteyksien luotettavuutta parantaa tyyppitarkistusten avulla ja yhteyksien tehokkuutta tiedonsiirrossa lisätä käyttämällä siirrettävälle tyyppille parhaiten soveltuvaa matalan tason esitysmuotoa.

Alusta lataa komponentit tiedostojärjestelmästä. Alustaan voitaisiin toteuttaa myös muita koodilähteitä, kuten CVS-versionhallintajärjestelmä. Tämä piirre olisi erittäin hyödyllinen, mikäli alustaan toteutetaan mahdollisuus käsitellä ja muokata kompo-

nenttien lähdekoodia. Ilman liityntää versionhallintaan ohjelmoija joutuu viemään kaikki tekemänsä muutokset versionhallintaan käsin. Tyypillisesti Java-koodin toteuttamiseen käytetään kehittyntä sovelluskehittäjä, joka kytkeytyy versionhallintaan suoraan. Ei olisi tarkoituksenmukaista, että Flow-komponenttien kehitys vaatisi käsin tehtäviä versiopäivityksiä.

Flow-alustan monikielisyys perustuu yhteiseen tulkkirajapintaan, jonka kaikki komponenttikielten tulkit toteuttavat. Rajapinnan on sellainen, että tulkkien sisäinen toteutus ei näy alustalle. Rajatun rajapinnan vaihtoehto olisi ollut, että alusta näkee tulkkien tilan, eli tulkit ovat läpinäkyviä. Ratkaisu olisi voinut perustua esimerkiksi abstraktiin tulkkitoteutukseen tai yhteiseen välikieleen. Abstrakti tulkkitoteutus olisi sisältänyt tulkin toiminnan rungon, jota laajentamalla komponenttitulkit olisi toteutettu. Tätä mallia käytetään olio-ohjelmoinnissa melko yleisesti, mutta tunnettuja kääntäjäteknisiä esimerkkejä ei ole. Jäsentäjägeneraattoreita on kehitetty lukuisia, mutta kokonainen tulkki on jäsentäjää paljon monimutkaisempi ohjelmisto ja sen generointiin tarvittaisiin merkittävästi kehittyneempiä generaattoreita. Usean eri paradigman mukaisten tulkkien tuottamista samalla generaattorilla voidaan pitää epärealistisena.

Abstraktiin tulkkitoteutukseen perustuvan ratkaisun huono puoli olisi ollut, että se olisi pakottanut tulkkien toteutukset yhdenmukaisiksi, mikä olisi vaikeuttanut tulkkien toteuttamista merkittävästi. Esimerkiksi jos abstraktin tulkin malliksi olisi valittu Javan tapaan pinopohjainen kone, olisi se ollut liian matalan tason ratkaisu logiikkaohjelmointikielen toteuttamiseen suoraan. Rajoituksen voi kiertää rakentamalla korkean tason tulkin välikielen päälle, jolloin välikieltä tulkitseva virtuaalikone mahdollistaa eri kielten käytön yhdessä. Tällöin kuitenkin virtuaalikoneen tilasta olisi hankala päätellä itse tulkin tilaa, jolloin monipuolisempi tulkkien käsittely, kuten komponenttien monitorointi, on hankalaa. Välikieleen perustuvaan ratkaisuun on päädytty .Net-alustassa, jonka monikielisyys on toteutettu Common Language Runtime (CLR) -välikielen ja -virtuaalikoneen avulla. .Net-alustassa yhteisen välikielen matala abstraktiotaso ei ole ongelma, koska sen avulla ei tehdä tulkkien tilaan perustuvaa komponenttien integrointia. Ratkaisua on kuitenkin kritisoitu siitä, että muiden kuin oliokielten toteuttaminen alustaan on hankalaa [Ham00].

Välikieleen verrattuna Flow-työkalun yhteisten tulkkirajapintojen toinen etu on suorituskyky. Vaikka jokainen tulkitsemisaskel sisältää ylimääräisenä kuormana alustan suorittamat toimet, tehdään esimerkiksi Xfunc-kielessä merkittävä osa laskennasta

primitiivifunktioiden sisällä eli yhden suoritusaskeleen aikana. Tällöin kyseessä on tavallisen Java-koodin suoritus, jota Flow-työkalu ei hidasta.

5.3.1 Xfunc-kielen toteutus

Sysbio-projektissa tarvittiin työkalua, jonka avulla voidaan käsitellä SOAP-pohjaisia geeniannotaatiopalveluja. Xfunc-kieli on tehokas väline XML-käsittelyyn, joten sen avulla annotaatiopalvelut tunteva ohjelmoija voi helposti tuottaa annotaatiokomponentteja. Toisin sanoen Xfunc täyttää tärkeimmän tehtävänsä eli on Javaa yksinkertaisempi ja ilmaisuvoimaisempi ohjelmointityökalu annettuun tehtävään. Xfunc soveltuu myös paremmin skriptaukseen, koska se on Javaa dynaamisempi: sillä kirjoitettuja ohjelmia ei tarvitse kääntää ja siinä on vähemmän staattisia tarkistuksia, joten yksinkertaisten ohjelmien kirjoittaminen on nopeampaa. Kielen dynaamisuudesta olisi saatu merkittävämpiä hyötyjä, mikäli metakieli olisi tukenut komponenttien muokkaamista eli Xfunc-komponentteja olisi voitu muokata komentoliittymän kautta. Tällöin komentoliittymää olisi voitu käyttää interaktiivisena ohjelmointiympäristönä, kuten esimerkiksi Matlab-numeriikkaohjelmaa tai ohjelmoitavaa tietokantaa. Nyt Xfunc-ohjelmointi rajoittui levyllä tallennettujen lähdeohjelmien kirjoittamiseen.

Xfunc-kielen ongelmana voidaan pitää sitä, että sen käyttö vaatii melko hyvää funktionaalisen ohjelmoinnin hallitsemista. Esimerkiksi yksinkertainen heijastusta käyttävä XML-dokumenttia muokkaava ohjelma käyttää funktion välittämistä parametreina toiselle funktiolle. Korkeamman kertaluvun funktiot eivät kuitenkaan ole tuttuja useimmille bioinformaatikoille, koska yleensä heidän melko vähäinen ohjelmointikokemuksensa rajoittuu proseduraalisiin kieliin.

Useimmat nykyään käytetyt ohjelmointikielet on suunniteltu ennen kuin XML-kuvauskieli yleistyi. Harvat kielet tukevat XML-dokumenttien käsittelyä suoraan, vaan käsittelyyn käytetään kieleen kuuluvia tai ulkopuolisia kirjastoja. Xfunc-kielessä XML-käsittely sisällytettiin itse kieleen toteuttamalla siihen XML-tietotyyppi. Tietotyyppiä ei kuitenkaan käsitellä kieleen kuuluvilla rakenteilla, vaan kielen kirjastoihin kuuluvilla funktioilla.

Kirjastoihin perustuva toteutus valittiin yleiskäyttöisyyden ja käytön helppouden vuoksi: kieleen kuuluvien XML-operaatioiden avulla käsittelytoimenpiteitä olisi todennäköisesti voitu yhdistellä helpommin kuin operaatiot kapseloivien funktioiden avulla. Tehtyä ratkaisua voidaan kokeilujen perusteella pitää kuitenkin hyvänä.

Xfuncin avulla funktioiden käsittely on joustavaa: esimerkiksi solmuja valitsevalle funktiolle voidaan antaa valintaehdon toteuttava funktio parametrina. Korkeamman kertaluvun funktiot ovatkin keskeisiä Xfunc-kielen XML-käsittelyominaisuuksien kannalta. Kielen rakenteisiin perustuva käsittely olisi monimutkaistanut kieltä merkittävästi, nyt monimutkaisuus on pilloitettu primitiivifunktioiden toteutukseen.

Xfunc-kieli suunniteltiin XML-dokumenttien käsittelyyn, erityisesti olemassa olevien monimutkaisten dokumenttien muokkaamiseen. Tällöin tärkeitä ominaisuuksia ovat solmujen valinta ja muokkaaminen. Solmujen muokkaaminen perustuu heijastukseen: muokkausfunktiot palauttavat muokatun version solmuista ja saadut solmut voidaan heijastaa alkuperäiseen dokumenttiin, jolloin ne korvaavat alkuperäiset solmut. Menetelmän todettiin soveltuvan hyvin dokumentin valikoitujen osien muuttamiseen. Toisin sanoen heijastuksen avulla pystyttiin toteuttamaan imperatiivisille kielille ominainen hyvä piirre funktionaaliseen kieleen ja vältettiin funktionaalisille kielille tyypillinen triviaalin päivityksen ongelma [Sco00, s. 622 - 623]. Heijastuksen heikkous on sama kuin imperatiivisten XML-käsittelymenetelmien heikkous yleensä: dokumentin rakentaminen uudelleen, esimerkiksi muuttaminen hyvin erilaisen skeeman mukaiseksi, on työlästä. Xfunc ei kuitenkaan pakota käyttämään heijastusta, vaan dokumentti voidaan rakentaa uudelleen pelkästään funktionaalisia piirteitä käyttäen.

Uusien ominaisuuksien toteuttaminen Xfunc-kieleen

Xfunc-kielestä puuttuu häntärekursion optimointi, eli itseään suorituksen päätteeksi kutsuva funktio kasvattaa kutsupinoa jokaisella kutsulla. Tämän vuoksi Xfunc-kielen avulla ei voida toteuttaa mielivaltaisen pitkiä silmukoita. Häntärekursion kannalta keskeinen kohta on Xfunc-tulkkisilmukan kohta, joka käsittelee syntaksipuun funktiokutsusolmun. Primitiivifunktioiden ja käyttäjän määrittelemien funktioiden kutsut käsitellään eri tavoin. Primitiivifunktioissa ei kuitenkaan ole tarvetta optimoituun häntärekursioon, koska funktiot ovat Java-pohjaisia ja niiden toteutuksissa on luontevampaa käyttää toistorakenteita rekursion sijaan. Käyttäjän määrittelemän funktion kutsu on toteutettu seuraavasti.

```
UserDefinedFunctionReference functionRef =  
    (UserDefinedFunctionReference)functionCallNode.getFunctionReference();  
LambdaNode lambdaNode =  
    (LambdaNode)currentBindings.get(functionRef.getName());
```

```

VariableBindings newBindings = new VariableBindings(currentBindings);
Iterator<VariableNode> formalParams =
    lambdaNode.getFormalParameterNodes().iterator();
Iterator<AstNode> params =
    functionCallNode.getParameterNodes().iterator();
while (formalParams.hasNext()) {
    String fp = formalParams.next().getName();
    AstNode p = params.next();
    newBindings.bind(fp, p);
}
stack.push(new Context(newBindings, lambdaNode.getBody()));

```

Koodissa haetaan viite käyttäjän määrittelemään funktioon, jota ollaan kutsumassa. Viite on funktion nimi, jonka avulla funktion toteuttava lambda-rakenne haetaan nimisidonnoista. Kutsua varten nykyisistä nimisidonnoista luodaan kopio, jonne sijoitetaan kutsuun kuuluvat parametrit. Lopuksi osoitin funktion runkoon painetaan pinon päälle.

Kutsuriviin voidaan lisätä tarkistus, joka tutkii, onko kutsuttu funktio sama kuin nykyinen funktio. Yhtäsuuruus voidaan tutkia siten, että haetaan sidonnoista funktioolio sekä kutsutun että nykyisen funktion nimellä. Mikäli palautetut oliot ovat samat, funktio kutsuu itseään. Tällöin ei luoda uusia muuttujasidontoja, vaan kutsun parametrit sijoitetaan nykyisiin sidontoihin. Uuden parametrit ylikirjoittavat edelliset parametrit. Ennen kuin funktion runko asetetaan pinoon, nykyinen runko poistetaan pinosta. Sen jälkeen tulkitseminen voi edetä normaalisti. Alusta näkee optimoidun häntärekursion yhtenä funktiokutsuna, joka vaihtaa parametrien arvoja. Silmukkaan perustuvan tulkkitoiteutuksen ansiosta häntärekursion optimointi on helppo toteuttaa.

Xfunc-kielen luotettavuus

Xfunc ei sisällä implisiittisiä tyyppimuunnoksia, vaikka ne sopisivatkin keveään skriptikieleen. Tyyppimuunnosten toteuttaminen on kuitenkin varsin työlästä. Xfunc-tyypit on mallinnettu Java-tyyppinä, joten Javan tyyppitarkistukset estävät suoraviivaisen toteutuksen. Tyyppien toteutusta joudutaankin muuttamaan merkittävästi. Voidaan siis havaita, että toteutettavan kielen piirteiden mallintaminen niitä vastaavilla toteutettavan kielen piirteillä aiheuttaa tässä tapauksessa merkit-

täviä ongelmia. Vastaava ilmiö havaitaan primitiivisten funktioiden toteutuksessa: laiska laskenta on hankala toteuttaa primitiivifunktioissa, koska ne on toteutettu Javan metodeina, eikä Java tue laiskaa laskentaa. Myös häntärekursion optimoinnissa tilanne on vastaava. Xfunc-tulkin ensimmäisessä versiossa tulkitseminen perustui rekursioon, joka kuitenkin korvattiin silmukalla. Rekursioon perustuvaan tulkkiin ei voitaisi toteuttaa häntärekursion optimointia helposti, koska Java-metodikutsut vastaavat Xfunc-funktiokutsuja, eikä Java tue häntärekursion optimointia.

Xfunc-kielessä *null*-arvo tulkitaan löyhästi, eli esimerkiksi sen kutsuminen funktiona ei ole tyyppivirhe. Lisäksi kun implisiittiset tyyppimuunnokset toteutetaan, voidaan myös niitä pitää löyhempanä tyyppien käsittelyn käytäntönä. Tuotantokäytössä kielen luotettavuuden pitäisi olla hyvällä tasolla, joten jatkokehityksessä onkin tarpeen pohtia, pitäisikö Xfuncin tukea esimerkiksi PHP:n tai Perlin tavoin turvallisempaa suoritustilaa, jossa ei olisi implisiittisiä tyyppimuunnoksia ja *null*-arvon löyhää tulkintaa. Tila kytkettäisiin päälle alustasta. Turvallisessa suoritustilassa *null*-arvon käyttäminen olisi aina virhe ja tyyppimuunnokset pitäisi tehdä eksplisiittisesti. Tiukennus voitaisiin tehdä myös C#-kielen tavoin merkitsemällä turvattomat funktiot. Tällöin turvattomaksi merkittyä kokeiluluontoista koodia ei voitaisi vahingossa kutsua tuotantotasoisesta koodista.

Xfunc ei sisällä poikkeusmekanismia, vaan funktiot ilmoittavat virheistä paluuarvolla, yleisimmin *null*-arvolla. Koska *null*-arvon käyttö ei johda virheeseen, vaatii virheiden havaitseminen paluuarvon tarkistamista. Mikäli käytettäisiin edellä kuvattua turvallisempaa suoritustilaa, ei virheellisesti päättyneen funktion palauttamaa arvoa voitaisi vahingossa käyttää. Tiukempi suoritustila parantaisi siis virhetilanteiden käsittelyn varmuutta ja sitä kautta kielen luotettavuutta.

Yksinkertaisuuteen pyrkiminen Xfunc-kielessä

Xfunc-kieli pyrittiin pitämään yksinkertaisena. Se suunniteltiin pitkälti Scheme-kieltä muistuttavaksi. Tällä tavoin kielen syntaksi ja rakenteet ovat tuttuja useille kokeneemmille ohjelmoijille. Scheme on yleiskäyttöinen kieli, joten monia XML-käsittelyn kannalta tarpeettomia ominaisuuksia jätettiin pois. Tärkein ero Schemeen ja sille tyyppilliseen ohjelmointityyliin oli XML-tietotyyppin toteuttaminen atomaariseksi tietotyyppiksi. Schemen mukainen ratkaisu olisi ollut toteuttaa XML-puu listojen avulla, mutta Xfunc-kielen ei toteutettu listoja ollenkaan. Kun ratkaisua tar-

kastellaan XML-käsittelyyn liittyvien kokeilujen kautta, sitä voidaan pitää onnistuneena: ratkaisu oli yksinkertainen mutta käytännössä toimiva.

Xfunc käyttää dynaamista nimien sidontaa. Sitä pidetään yleisesti ongelmallisena ratkaisuna [GLSG93]. Staattista eli leksikaalista sidontaa käytettäessä nimeen sidottu arvo on pääteltävissä kohtuullisen helposti lähdekoodista, kun taas dynaamista sidontaa käytettäessä sidonnan selvittäminen voi vaatia ohjelman toiminnan tarkkaa tuntemista. Nämä ongelmat ovat kuitenkin merkittäviä vasta suuremmissa ohjelmissa, joten pienikokoisissa Xfunc-komponenteissa dynaaminen sidonta ei aiheuttanut merkittävää epäselvyyttä. Dynaaminen sidonta on helpompi toteuttaa, joten tässä tapauksessa toteutuksen yksinkertaistamista kielen käytön monimutkaistamisen kustannuksella pidettiin perusteltuna.

Alustan rakenne ja toteutus pidettiin yksinkertaisena. Tämä saavutettiin osin siten, että useita tehtäviä jätettiin komponenttitulkeissa hoidettavaksi. Se heijastui Xfunc-kielen toteutukseen, josta tuli varsin monimutkainen. Nykyisen alustan ongelma onkin, että uuden komponenttikielen toteuttaminen on suuri ja vaikea tehtävä.

Xfunc-tulkin toteutus on puhtaasti oliopohjainen. Tulkkien toteutuksissa usein vältetään olio-ohjelmointia siitä aiheutuvan tehohukan vuoksi. Xfunc-tulkin toteutuksessa oliomenetelmä kuitenkin toimi hyvin. Ainoa merkittävä ongelma oli se, että Xfunc-tyyppien esittäminen luokkahierarkiana sotki toteutusta joissain kohden. Ongelma ei kuitenkaan johtunut oliomenetelmästä, vaan sen vääränlaisesta käytöstä: isäntäkielen rakenteilla mallinnettiin upotetun kielen vastaavia rakenteita. Java soveltui oliopohjaisen tulkin toteuttamiseen hyvin. Roskienkeruu oli merkittävä hyöty, koska tulkin monimutkaisen muistin käsittelyn vuoksi manuaalinen muistin vapautus olisi ollut vaativa toteuttaa.

Flow-työkalun kehityksen alkuvaiheessa arvioitiin, että merkittävimmät ongelmat ovat pääasiassa suorituskykyongelmia. Olettamalla työkalun käyttötarkoituksen sellaiseksi, missä puhtaalla laskentanopeudella ei ole suurta merkitystä, uskottiin monet kielten toteutukseen liittyvät ongelmat vältettävän. Useat ongelmat, kuten funarg-ongelma ja vahvan tyyppityksen toteutus, olivatkin suurelta osin suorituskykyongelmia. Kuitenkin muiden ongelmien, kuten laiskan laskennan ja implisiittisten tyyppimuunnosten toteuttamisen, määrä oli odotettua suurempi. Tutkielman kokemusten perusteella voidaan siis todeta, että uusien ohjelmointikielten toteuttaminen on hyvin vaativa tehtävä, vaikka kielet ovat sovellusaluekohtaisia ja niiden toteutuksen tehokkuus ei ole keskeistä. Tämän vuoksi jatkokehityksessä joudutaan arvioimaan uudelleen Xfuncin kaltaisten kielten mielekkyyttä.

Xfunc-kieli olisi voitu toteuttaa myös ilman Flow-työkalua. Tällöin sen toteutus, varsinkin tulkkisilmukka, olisi ollut hieman yksinkertaisempi. Alusta käsittelee tulkkia vain rajatun rajapinnan kautta, joten sen vaikutus tulkin toteutukseen on kuitenkin vähäinen. Mikäli Xfunc olisi itsenäinen kieli ja sen tulkki olisi käsiteltävissä suoraan Java-ohjelmasta – ilman välissä olevaa Flow-alustaa – olisi kielen Java-sidonta voitu toteuttaa paljon monipuolisemmaksi. Flow-työkalussa sidonta hoidetaan geneerisen alustan avulla ja se on näin ollen rajatumpi: esimerkiksi Xfunc-tyypit eivät näy Java-rajapinnassa. Itsenäisenä kielenä Xfuncissa ei olisi voitu käyttää metakieltä, eli Xfunc-komponentteja ei olisi voitu esimerkiksi monitoroida. Mikäli monikielisyys ei ole tärkeä vaatimus, Flow-alustan nykyversion hyödyt verrattuna sen aiheuttamiin haittoihin ovat melko vähäiset.

5.4 Työkalun jatkokehitys

Flow-alusta suunniteltiin siten, että alustan vaikutus komponenttikieliin rajoittuu rajapintoihin, jotka komponenttikielten tulkkien tulee toteuttaa. Rajapintoihin perustuvan ratkaisun heikkous on, että alusta jättää monia tehtäviä, kuten jäsentämisen, komponenttitulkin toteutuksessa hoidettavaksi. Ongelman ratkaisuna voisi toimia joukko työkaluja, jotka tukevat komponenttitulkkien toteuttamista alustan vaatimusten mukaisesti. Esimerkiksi voitaisiin toteuttaa sovellusgeneraattori, joka luo annetun syntaksikuvauksen pohjalta jäsentäjän ja tulkkisilmukkaan perustuvan abstraktin luokan, jotka toteuttavat alustan vaatimat rajapinnat. Tällöin tulkin toteutus tehtäisiin perimällä abstrakti luokka ja täydentämällä se kokonaiseksi tulkkitoteutukseksi. Työkalusta tulisi tällöin myös sovelluskehys ohjelmointikielten tulkeille. Sovelluskehyskiä käsittelevässä kirjallisuudessa on esitetty, että ennen sovelluskehyskehittämistä pitää toteuttaa muutama kyseisen sovellusalueen sovellus [RJ97]. Niinpä tämän tutkielman kokemusten yleistäminen sovelluskehyskeksi onkin hyvä lähestymistapa.

Xfunc-kielen toteuttamisesta opittiin, että tulkkien toteuttaminen Flow-alustan päälle on työlästä. Toteutuksessa törmätään moniin yleiskäyttöisten kielten ongelmiin, joiden ratkaisemisessa alustasta ei ole apua. Alusta tuo myös oman lisänsä toteutuksen monimutkaisuuteen. Voidaankin kysyä, olisiko kieli voitu suunnata vielä tarkemmin pelkästään XML-käsittelyä varten, tai olisiko kielen käyttötarkoitusta voitu rajata vielä tarkemmin, esimerkiksi SOAP-palvelujen kyselyyn. SOAP-palvelujen käsittelykieli olisi hyödyllinen työväline ja riittävä myös alkuperäiseen

Sysbio-projektissa olleeseen tarpeeseen. Jatkokehityksessä voidaan selvittää, olisiko rajatumpi kieli toteutettavissa Xfunc-kieltä merkittävästi yksinkertaisemmin.

Tutkielman havaintojen pohjalta voidaan esittää, että parempi tapa rakentaa upotettu kieli XML-pohjaisten annotaatiopalveluiden yhdistämiseen olisi ollut aluksi luoda korkealaatuinen oliokirjasto. Kirjastosta olisi todennäköisesti tullut kohtuullisen monimutkainen. Kun kirjaston toimivuudesta ja soveltuvuudesta olisi varmistuttu, sille olisi rakennettu käyttöliittymäksi yksinkertainen upotettu kieli. Tällainen nk. edustakieleen perustuva ratkaisumalli olisi ollut ohjelmointiprosessina kevyempi, koska kirjastoa olisi voitu käyttää jo ennen kuin edustakieli on valmis. Kielen toteutuksesta olisi myös tullut yksinkertaisempi, koska se olisi vain hallinnoinut kirjastoon kuuluvia olioita. Samoin kielestä olisi tullut paremmin nimenomaisen sovellusalueen tarpeita palveleva, koska se olisi rakennettu sovellusalueen oliokirjaston päälle. Oliokirjaston avulla on helpompi saavuttaa ongelmanratkaisun kannalta parhaat rakenteet, koska sitä on helpompi refaktoroida kuin ohjelmointikieltä. Niinpä kokeellinen ja iteratiivinen ohjelmointityyli on sovellettavissa paremmin. Ratkaisun huono puoli olisi ollut, että kokonaisuuteen olisi tullut yksi uusi abstraktiotaso lisää.

Oliokirjastoihin verrattuna sovellusaluekohtaisten kielten etu hajautetussa ympäristössä on myös siinä, että ne ovat helpommin siirrettävissä verkon yli tai tallennettavissa tiedostoon. Jatkokehityksessä Flow-työkalusta pyritäänkin tekemään edustakielten toteuttamiseen tarkoitettu sovelluskehys ja edustakielten tulkkien ajamiseen tarkoitettu palvelinohjelmisto. Työkalun ominaisuuksia siis tarkennetaan nimenomaan edustakielten tarpeisiin soveltuviksi ja tulkkien toteuttamista tukevia ominaisuuksia lisätään. Rajapintoihin perustuva rakenne mahdollistaa sen, että nykyiseen työkaluun ei tarvita merkittäviä muutoksia, jotta suunnitellut parannukset olisivat mahdollisia.

6 Yhteenveto

Ohjelmointikieliet ovat ilmaisuvoimaisia abstraktioita. Niiden ongelma on monimutkaisuus, joka estää muita kuin laajan ohjelmointikokemuksen omaavia henkilöitä tuottamasta tuotantotasoisia ohjelmakomponentteja. Tutkielmassa kuvattiin Tieteen tietotekniikan keskus CSC:ssä toteutetun bioinformatiikan alaan kuuluvan Sysbio-projektin tarpeita varten toteutettu Flow-ohjelmointityökalu. Työkalun avulla voidaan upottaa Java-ohjelmointikielillä toteutettuihin hajautettuihin järjestelmiin yksinkertaisilla sovellusaluekohtaisilla ohjelmointikielillä toteutettuja komponentteja.

Flow-työkalu koostuu alustakomponentista, XML-käsittelyyn tarkoitettusta Xfunc-kielestä sekä metaohjelmointiin tarkoitettusta Cclang-kielestä. Xfunc on funktionaalinen kieli, jonka XML-ominaisuudet perustuvat joukkoon XML-solmuja käsitteleviä funktioita. Kieleen on lisäksi yhdistetty imperatiivisia piirteitä niin kutsutun heijastusmenetelmän avulla. Ohjelmointikokeilujen pohjalta voidaan Xfuncin todeta soveltuvan hyvin XML-dokumenttien käsittelyyn.

Metakieltä käytetään Flow-työkalussa alustan hallintaan, sovellusaluekohtaisten kielten avulla toteutettujen komponenttien kytkentään (putkitukseen) sekä aspektiohjelmointiin. Aspektiohjelmointityökalujen käyttämät menetelmät eivät olleet suoraan sovellettavissa Flow-työkalun monikieliseen ympäristöön. Cclang-metakieli poikkeaaakin yleisistä aspektityökaluista: se perustuu Prolog-kieleen ja kolmannen osapuolen tuProlog-toteutukseen. Deklaratiivinen Prolog ei tue operaatioiden suoritusjärjestyksen määrittelyä, mutta alustassa kaikki muutokset ovat aikasidonnaisia. Niinpä tuProlog-toteutuksen päälle on rakennettu lähdekoodin esikäsittelijän avulla täydentäviä ominaisuuksia erityisesti suoritusjärjestyksen hallintaan. Valittu toteutus mahdollistaa ilmaisuvoimaisen deklarativisen kielen käytön rinnakkaisesti suoritettavien komponenttien hallintaan.

Tutkielman kokemusten perusteella voidaan todeta, että metaohjelmointikieli soveltuu hyvin komponenttien monitorointiin. Alustaan perustuvan ratkaisun ongelma on, että se monimutkaistaa kielten toteuttamista huomattavasti. Sovellusaluekohtaiset kielet ovat melko työläs tapa tuottaa ohjelmointitekniisiä apuvälineitä. Kielen rajaaminen tiettyyn sovellusalueeseen ja suorituskykyvaatimusten ohittaminen eivät yksinkertaista kielen toteuttamista merkittävästi, toisin kuin aluksi oletettiin. Kielen kehittämiseen tarvittavan suuren alkupanostuksen vuoksi omien sovellusa-

luekohtaisten kielten käyttöönotto on perusteltua vain, jos niillä voidaan toteuttaa järjestelmään laajoja osakokonaisuuksia.

Flow-työkalun jatkokehityksessä keskitytään tavallisia sovellusaluekohtaisia kieliä yksinkertaisempiin edustakieliin. Varsinainen toimintalogiikka on oliokirjastossa tai ulkopuolisessa järjestelmässä, ja itse kieli toimii vain helppokäyttöisenä liittymänä. Kokonaisia ohjelmointikieliä varten on vaikea toteuttaa pitkälle vietyjä tulkkien kehittämistä helpottavia ominaisuuksia kuten tulkki-generaattoreita. Yksinkertaisempien edustakielten tapauksessa tukiominaisuuksien kehittäminen on helpompaa. Jatkossa alustaa kehitetään siten, että siihen voidaan kytkeä myös tulkkeja, jotka eivät toteuta monimutkaista metaohjelmoinnin vaatimaa rajapintaa. Työkalusta tulee näin ollen sovelluskehys, johon kuuluu metaohjelmointia tukeva apukomponentti.

Sovellusaluekohtaiset ohjelmointikieliset ovat hyödyllisiä työvälineitä myös vähemmän kokeneille ohjelmoijille. Ohjelmointikielen käyttö ei vaadi yhtä laajoja taustatietoja kuin yleiskäyttöisellä oliokielellä toteutettujen oliokirjastojen käyttö, koska sovellusaluekohtaiset kielet perustuvat sovellusalueen merkintöihin ja symboleihin. Sovellusaluekohtaisella kielellä voidaan myös ratkoa sovellusalueeseen kuuluvia ohjelmointiongelmia hyvin tehokkaasti, koska sen rakenteet on suunniteltu nimenomaan kyseessä oleviin ohjelmointitehtäviin soveltuviksi. Ohjelmointikielten toteuttamisen ja kielten integroinnin vaikeus on kuitenkin estänyt kielten yleistymistä. Sovellusaluekohtaisten kielten käyttöönoton kynnyksiä voidaan madaltaa sopivan työkalun avulla. Tässä tutkielmassa havaittiin, että rajattuihinkin tehtäviin suunniteltujen kielten kehittäminen on vaativaa, joten työkalun onnistumisen kannalta on tärkeää panostaa kielten kehittämistä ja toteuttamista tukeviin ominaisuuksiin.

Lähteet

- ASU85 Aho, A. V., Sethi, R. ja Ullman, J. D., *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1985.
- AWB⁺94 Aksit, M., Wakita, K., Bosch, J., Bergmans, L. ja Yonezawa, A., Abstracting object interactions using composition filters. *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, Guerraoui, R., Nierstrasz, O. ja Riveill, M., toimittajat, osa 791. Springer-Verlag, 1994, sivut 152–184.
- BCRP98 Blair, G. S., Coulson, G., Robin, P. ja Papathomas, M., An architecture for next generation middleware. *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Lontoo, Iso-Britannia, 1998, Springer-Verlag.
- Ben86 Bentley, J., Programming pearls: little languages. *Communications of the ACM*, 29,8(1986), sivut 711–721.
- Ber94 Bergmans, L., The composition filters object model. Tekninen raportti, tietojenkäsittelytieteen laitos, Twenten yliopisto, 1994. URL <http://trese.cs.utwente.nl/publications/paperinfo/cf.pi.top.htm>. Tarkistettu 1.1.2006.
- BP00 Bergenti, F. ja Poggi, A., Aspect views as a means to promote reuse in aspect-oriented languages. *ECOOP 2000 Workshop on Aspects and Dimensions of Concerns*, 2000.
- CST00 Chiba, S., Sato, Y. ja Tatsubori, M., Using HotSwap for implementing dynamic AOP systems. *ECOOP 2000 Workshop on Aspects and Dimensions of Concerns*, Lopes, C., Bergmans, L., D'Hondt, M. ja Tarr, P., toimittajat, 2000.
- Dan04 Danvy, O., On evaluation contexts, continuations, and the rest of the computation. *Proceedings of the Fourth ACM SIGPLAN Continuations Workshop*, Thielecke, H., toimittaja, 2004. Julkaistu teknisessä raportissa CSR-04-1, tietojenkäsittelytieteen laitos, Birminghamin yliopisto.
- Dav00 Davis, M. S., An object oriented approach to constructing recursive descent parsers. *ACM SIGPLAN Notices*, 35,2(2000), sivut 29–35.

- Dij68 Dijkstra, E. W., Letters to the editor: Go-to statement considered harmful. *Communications of the ACM*, 11,3(1968).
- DOR01 Denti, E., Omicini, A. ja Ricci, A., tuProlog: A light-weight Prolog for internet applications and infrastructures. Teoksessa *Lecture Notes in Computer Science*, osa 1990, 2001.
- EGK⁺99 Eliassen, F., Goebel, V., Kristensen, T., Plagemann, T., Andersen, A., Rafaelsen, H. O., Yu, W., Blair, G., Costa, F., Coulson, G., Saikoski, K. B. ja Hansen, O., Next generation middleware: Requirements, architecture, and prototypes. *FTDCS '99 Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, Washington DC, Yhdysvallat, 1999, IEEE Computer Society.
- EK76 Emden, M. H. V. ja Kowalski, R. A., The semantics of predicate logic as a programming language. *Journal of the ACM*, 23,4(1976), sivut 733–742.
- FBC⁺98 Fitzpatrick, T., Blair, G., Coulson, G., Davies, N. ja Robin, P., Supporting adaptive multimedia applications through open bindings. *Proceedings of the International Conference on Configurable Distributed Systems*, Washington, DC, USA, 1998, IEEE Computer Society.
- FGM05 Fournet, C., Gordon, A. D. ja Maffeis, S., A type discipline for authorization policies. *European Symposium on Programming*, 2005, sivut 141–156.
- Fow99 Fowler, M., *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- GHJV95 Gamma, E., Helm, R., Johnson, R. ja Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, 1995.
- GJS96 Gosling, J., Joy, B. ja Steele, G., *The Java Language Specification*. Addison-Wesley, 1996.
- GLSG93 Guy L. Steele, J. ja Gabriel, R. P., The evolution of Lisp. *The Second ACM SIGPLAN Conference on History of Programming Languages*, New York, Yhdysvallat, 1993, ACM Press, sivut 231–270.

- Ham00 Hammond, M., Python for .NET: Lessons learned, 2000.
URL <http://starship.python.net/crew/mhammond/dotnet/PythonForDotNetPaper.doc>. Tarkistettu 1.1.2006.
- Hoa89 Hoare, C. A. R., Hints on programming language design. Teoksessa *Essays in Computing Science*, Prentice Hall, 1989, sivut 193–217.
- Hud96 Hudak, P., Building domain-specific embedded languages. *ACM Computing Survey*, 28,4(1996).
- JS96 Jayaraman, V. ja Srivastava, R., Expert systems in production and operations management: Current applications and future prospects. *International Journal of Operations & Production Management*, 16,12(1996), sivut 27–44.
- Kam90 Kamin, S. N., *Programming Languages*. Addison-Wesley, 1990.
- KHH⁺01 Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. ja Griswold, W. G., An overview of AspectJ. *Proceedings of the 15th European Conference on Object-Oriented Programming*, Lontoo, Iso-Britannia, 2001, Springer-Verlag, sivut 327–353.
- KLM⁺97 Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. ja Irwin, J., Aspect-oriented programming. Teoksessa *Proceedings of the European Conference on Object-Oriented Programming*, Akşit, M. ja Matsuoka, S., toimittajat, osa 1241, Springer-Verlag, 1997, sivut 220–242.
- KP05 Kelleher, C. ja Pausch, R., Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys*, 37,2(2005), sivut 83–137.
- Lai01 Laine, J.-P., Aspektiohjelmointi. Tekninen raportti C-2001-15, pro gradu -tutkielma, tietojenkäsittelytieteen laitos, Helsingin yliopisto, 2001.
- Lin03 Linthicum, D. S., *Next Generation Application Integration: From Simple Information to Web Services*. Addison-Wesley, 2003.
- McC60 McCarthy, J. L., Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3,4(1960), sivut 184–195.

- MS03 Meijer, E. ja Schulte, W., Unifying tables, objects and documents. *Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages*, 2003.
- NL05 Neerinx, P. B. ja Leunissen, J. A., Evolution of web services in bioinformatics. *Briefings in Bioinformatics*, 6,2(2005), sivut 178–188.
- OAF⁺04 Oinn, T., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, M., Carver, T., Glover, K., Pocock, M. R., Wipat, A. ja Li, P., Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20,17(2004), sivut 3045–3054.
- Par66 Parikh, R. J., On context-free languages. *Journal of the ACM*, 13,4(1966), sivut 570–581.
- Paw00 Pawlak, R., Nature and benefits of aspect-oriented programming. *ECOOP 2000 Workshop on Aspects and Dimensions of Concerns*, Lopes, C., Bergmans, L., D'Hondt, M. ja Tarr, P., toimittajat, 2000.
- RJ97 Roberts, D. ja Johnson, R. E., Evolving frameworks: A pattern language for developing object-oriented frameworks. Teoksessa *Pattern Languages of Program Design 3*, Addison Wesley, 1997, sivut 471–486.
- RMHB88 R. M. Herndon, J. ja Berzins, V. A., The realizable benefits of a language prototyping language. *IEEE Transactions on Software Engineering*, 14,6(1988), sivut 803–809.
- SA86 Schneider, F. B. ja Andrews, G. R., Concepts for concurrent programming. Teoksessa *Current Trends in Concurrency*, de Bakker, J. W., de Roever, W. P. ja Rozenberg, G., toimittajat, osa 224 sarjasta *Lecture Notes in Computer Science*, Springer, 1986, sivut 669–716.
- Sch03 Schmidmeier, A., Using AspectJ to eliminate tangling code in EAI activities. *Proceedings of the Second International Conference on Aspect-Oriented Software Development*, New York, Yhdysvallat, 2003, ACM Press.
- Sco00 Scott, M. L., *Programming Language Pragmatics*. Morgan Kaufmann Publishers, 2000.

- SCT03 Sato, Y., Chiba, S. ja Tatsubori, M., A selective, just-in-time aspect weaver. *Proceedings of the Second International Conference on Generative Programming and Component Engineering*. Springer-Verlag, 2003, sivut 189–208.
- Seb01 Sebesta, R. W., *Concepts of Programming Languages, Fifth Edition*. Addison-Wesley, 2001.
- Str94 Stroustrup, B., *The Design and Evolution of C++*. Addison-Wesley, 1994.
- vDK98 van Deursen, A. ja Klint, P., Little languages: little maintenance. *Journal of Software Maintenance*, 10,2(1998), sivut 75–92.
- W3Ca W3C, Extensible Markup Language (XML) 1.0 (third edition). URL <http://www.w3.org/TR/2004/REC-xml-20040204/>. Tarkistettu 1.1.2006.
- W3Cb W3C, XML Schema part 1: Structures. URL <http://www.w3.org/TR/xmlschema-1/>. Tarkistettu 1.1.2006.
- W3Cc W3C, XML Schema part 2: Datatypes. URL <http://www.w3.org/TR/xmlschema-2/>. Tarkistettu 1.1.2006.
- W3Cd W3C, XQuery 1.0: An XML query language. URL <http://www.w3.org/TR/xquery/>. Tarkistettu 1.1.2006.
- W3Ce W3C, XSL Transformations (XSLT) version 1.0. URL <http://www.w3.org/TR/xslt>. Tarkistettu 1.1.2006.
- Wad97 Wadler, P., How to declare an imperative. *ACM Computing Surveys*, 29,3(1997), sivut 240–263.
- Wir74 *On the Design of Programming Languages*, Amsterdam, 1974. North-Holland Publishing.
- WMW95 Wilhelm, R., Maurer, D. ja Wilhelm, R., *Compiler Design*. Addison Wesley Longman Publishing, Redwood City, Yhdysvallat, 1995.
- WR99 Wallace, M. ja Runciman, C., Haskell and XML: Generic combinators or type-based translation? *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 1999, sivut 148–159.

WZL03 Walker, D., Zdancewic, S. ja Ligatti, J., A theory of aspects. *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, New York, Yhdysvallat, 2003, ACM Press, sivut 127–139.