VAASAN YLIOPISTO
TEKNIIKAN JA INNOVAATIOJOHTAMISEN YKSIKKÖ
TIETOTEKNIIKKA

Simo Niemelä

TÄSMÄKIELEN TOTEUTUS TUOTEKONFIGURAATTORIIN

Pro gradu -tutkielma

VAASA 2019

SISÄLLYSLUETTELO

1 JOHDANTO 7
1.1 Tutkimuskysymykset 9
1.2 Käytettävä teoria ja tutkimusmenetelmä 9
1.3 Keskeiset käsitteet 10
2 AIKAISEMMAT TUTKIMUKSET 11
2.1 Oracle Graal ja Truffle 11
2.2 IRON-täsmäkieli 12
3 TUOTTEEN KONFIGUROINTI 14
3.1 Konfiguroitavat tuotteet 15
3.2 Tuotekonfiguraattorit 17
3.2.1 Sääntöpohjaiset konfiguraattorit 18
3.2.2 Rajoitepohjaiset konfiguraattorit 19
3.2.3 Funktiopohjaiset konfiguraattorit 19
4 TÄSMÄKIELI JA TULKKI 21
4.1 Täsmäkielen hyödyt ja haitat 23
4.2 Täsmäkielen suunnittelumallit 24
4.2.1 Kielen hyödyntäminen 25
4.2.2 Kielen keksiminen 26
4.3 Täsmäkielen toteuttaminen 26
4.4 XSLT 28
5 JÄRJESTELMÄ 31
5.1 Tuotemalli 33
5.2 Funktiokielen vaatimukset 38
6 TOTEUTUS 39

6.1 Täsmäkieli ... 39
6.1.1 Numerot, merkkijonot ja muuttujat ... 40
6.1.2 Listat ... 41
6.1.3 Binäärioperaatiot ... 42
6.1.4 Loogiset operaatiot ... 43
6.1.5 Sijoitusoperaatio ... 43
6.1.6 Hakuoperaatio ... 44
6.1.7 Funktiot ... 45
6.1.8 Objektit ... 46
6.2 Tulkki ... 48
6.2.1 Suoritettavan puumallin muodostaminen ... 51
6.2.2 Tulkin liittäminen konfiguraattoriin ... 53
7 EVALUOINTI ... 57
7.1 Suorituskykytesti: Summa kahdella jaollisista luvuista ... 58
7.2 Suorituskykytesti: Toisen asteen yhtälö ... 59
7.3 Suorituskykytesti: Alikysely ... 60
8 JOHTOPÄÄTÖKSET ... 63
9 JATKOTUTKIMUS ... 65
LÄHDELUETTELO ... 67

KUVALUETTELO

Kuva 1. Eri tuotetyyppien vertailua (Tiihonen & Soininen 1997). 16
Kuva 2. Suoritettavan ja tulkattavan täsmäkielen prosessi yleisesti. 28
Kuva 3. XSLT-muunnoksen eri vaiheet. 29
Kuva 4. Konfiguraattorin rakenne ja toimintaperiaate. 32
Kuva 5. Esimerkki tuotemallin rakenteesta. 34
Kuva 6. Tulkki-suunnittelumalli..... 49
Kuva 7. UML-esitys kontekstiluokan näkyvyysalueista. 50
Kuva 8. Muuttujan tai symbolin arvon etsintäjärjestys näkyvyysalueista..... 50
Kuva 9. Jäsennysluokan generointi. 51
Kuva 10. Funktio-ohjelman muuntaminen suoritettavaksi puumalliksi..... 52
Kuva 11. Ulkopuolisen metodisuorittajan liittäminen tulkkiin ja sen käyttäminen. 55
Kuva 12. Ulkoisten funktioiden liittäminen tulkkiin ja niiden suorittaminen. 56
Kuva 13. Suorituskykytulokset summa kahdella jaollisista luvuista -testissä. 59
Kuva 14. Suorituskykytulokset toisen asteen yhtälö -testissä. 60
Kuva 15. Suorituskykytulokset alikyselytestissä. 62

TAULUKKOLUETTELO

Taulukko 1. Yleisiä käytössä olevia täsmäkieliä (Jones 1996). 22
Taulukko 2. Kielen tason suhde tuottavuuteen (Jones 1996). 23
Taulukko 3. Täsmäkielen suoritustekniikoita. 27
Taulukko 4. Komponentin perusrakenne. 35
Taulukko 5. Esimerkki oikeasta komponentista. 35
Taulukko 6. Esimerkki tekstikenttäkomponentista. 36
Taulukko 7. Esimerkki komponentista ja laskentaviitteestä. 37

VAASAN YLIOPISTO
Tekniikan ja
innovaatiojohtamisen yksikkö
Tekijä: Simo Niemelä
Tutkielman nimi: Täsmäkielen toteutus tuotekonfiguraattoriin
Ohjaajan nimi: Timo Mantere
Tutkinto: Kauppatieteiden maisteri
Pääaine: Tietotekniikka
Opintojen aloitusvuosi: 2013
Tutkielman valmistumisvuosi: 2019

TIIVISTELMÄ:

Tässä työssä suunnitellaan ja toteutetaan täsmäkieli kaupalliseen tuotekonfiguraattoriin. Täsmäkieli on tietylle sovellusalueelle erikoistunut ohjelmointikieli, joka helpottaa ja tehostaa toimintaa kohdeympäristössä riippumatta tuotemallista ja sen rakenteesta. Täsmäkieli toteutetaan tuotekonfiguraattoriin uutena komponenttina, johon sisältyy täsmäkielen syntaksin määrittely sekä tulkin suunnittelu ja toteutus. Kääntäjää, jäsennintä ja selaajaa ei toteuteta, vaan käännösvaihe hoidetaan kolmannen osapuolen ohjelmakirjastolla. Tavoitteena on tuottaa täsmäkieli, joka on suorituskykyinen, helposti lähestyttävä ja nopeasti opittava ohjelmointikieli, jonka kirjoittaminen ei edellytä ohjelmointikokemusta tai teknistä koulutusta. Täsmäkielen inspiraatioita ovat taulukkolaskentaohjelmien makrot ja kaavat.

Työn alussa esitellään teoriaa liittyen tuotekonfiguraattoriin, täsmäkieleen ja tulkkiin. Lisäksi tarkastellaan tuotemallien rakennetta ja roolia tuotekonfiguraattorissa. Teorian jälkeen esitellään järjestelmä, johon täsmäkieli tullaan toteuttamaan. Toteutusosiossa määritellään täsmäkielen syntaksi, suunnitellaan ja toteutetaan täsmäkielen tulkki. Työn lopussa analysoidaan täsmäkielen suorituskykyä.

Työn tuloksena on tuotekonfiguraattoriin toteutettu tuotantovalmis täsmäkieli, jolla on vaivatonta määritellä ja ohjelmoida matemaattisia kaavoja erilaisiin ongelmiin. Täsmäkieli on myös suorituskykyinen, koska tulkki on suunniteltu ja toteutettu ajettavaksi vain tämän työn kohteena olevassa tuotekonfiguraattorissa.

AVAINSANAT: täsmäkieli, tulkki, konfiguraattori

UNIVERSITY OF VAASA
School of Technology and Innovations
Author: Simo Niemelä
Topic of the Master's Thesis: Implementation of a domain-specific language to a product configurator
Instructor: Timo Mantere
Degree: Master of Science in Economics and Business Administration
Major: Computer Science
Year of Entering the University: 2013
Year of Completing the Master's Thesis: 2019

ABSTRACT:

The purpose of this thesis is to design and implement a domain-specific language to a commercial product configurator. A domain-specific language is a programming language that is designed to a specific problem domain. Domain-specific languages enhance productivity and efficiency of users who don't have prior experience in programming.

The domain-specific language in this thesis is designed and implemented as a new component to the commercial product configurator. This includes defining of the syntax of the domain-specific language, design and implementation of the interpreter. Third-party library is used to generate a lexer and a parser for the domain-specific language. The goal of this thesis is to produce a domain-specific language to the commercial product configurator that is efficient and easy to learn.

The theoretical section of this thesis describes the theory of product configurators, domain-specific languages and interpreters. The system – the commercial product configurator – is presented after a theoretical section. Design and implementation of the domain-specific language and the interpreter are described in the implementation section. Benchmarks of the interpreter are presented and discussed in the evaluation section.

The result of this thesis is a production-ready domain-specific language that is efficient, easy to learn and easy to write mathematical equations and solutions to various problems.

KEYWORDS: domain-specific language, interpreter, configurator

1 JOHDANTO

Tässä työssä suunnitellaan ja toteutetaan täsmäkieli kaupalliseen tuotekonfiguraattorijärjestelmään. Täsmäkielen tarve on lähtöisin tuotekonfiguraattorin konsulteilta ja tuotemallintajilta, jotka tarvitsevat helppokäyttöisen ja suorituskyvyltään tehokkaan ohjelmointikielen, jolla voidaan mallintaa ja ohjelmoida matemaattisia kaavoja tuotekonfiguraattorin tuotemalleihin. Tuotemallintajat ovat toteuttaneet aikaisemmin matemaattisia kaavoja XSLT-kielellä, mutta se on todettu olevan vaikeasti lähestyttävä ja suorituskyvyltään hidas. Täsmäkielen halutaan olevan helposti opittava ja suorituskykyinen, joka muistuttaa syntaksiltaan taulukkolaskentaohjelmien kaavoja. Tämä sen vuoksi, että tuotemallintajat ja toimialueen asiantuntijat ovat usein kokeneita taulukkolaskentaohjelmien käyttäjiä, joten näiden ohjelmien kaavoja ja makroja muistuttava täsmäkieli on helpompi omaksua. Täsmäkielen helppokäyttöisyys korostuu entisestään, sillä tuotemallintaja voi olla kaupallisen tuotekonfiguraattorin asiakas ja tuotekonfiguraattorin konsultti. Asiakkaalla on mahdollisuus itse mallintaa tuotteita tuotekonfiguraattoriin tai ostaa mallinnus konsultaationa.

Valmiin täsmäkielen on tarkoitus korvata kaupallisen tuotekonfiguraattorin tuotemalleissa käytetty XSLT-kieli, jota on käytetty aikaisemmin algoritmien tai matemaattisten kaavojen ohjelmointiin. XSLT-pohjaisten kaavojen tekeminen on ollut haastavaa ja hidasta konfiguraattorin asiantuntijoille, sillä XSLT on suhteellisen tekninen ja tarpeettoman monisanainen kieli. Korvaamalla XSLT-kielen uudella täsmäkielellä oletetaan tuovan massiivisia parannuksia tuotekonfiguraattorin suorituskykyyn, asiantuntijoiden tuotemallinnustehokkuuteen ja sitä kautta mahdollisesti nopeimpiin toimitusaikoihin ja korkeimpiin tuottoihin.

Täsmäkieli on tietylle sovellusalueelle erikoistunut ohjelmointikieli, joka helpottaa ja tehostaa toimintaa kohdeympäristössä, johon kieli on kehitetty. Sovellusalueella tarkoitetaan jotain alaa tai toimialuetta, ja se voi olla myös liiketoiminta-ala. Sen tarkoituksena ei ole olla yleinen ohjelmointikieli, joka mahdollistaisi eri tyyppisten ohjelmistojen toteutuksen esimerkiksi Java-kielellä. Täsmäkielen tavoitteena on tehdä

ongelmanratkaisu kohdejärjestelmässä helpommaksi kuin perinteisellä ohjelmointikielellä, koska sen loppukäyttäjät ovat usein jonkin alan asiantuntijoita, joilla ei välttämättä ole teknistä koulutusta tai ohjelmointikokemusta. Hyvin suunniteltu täsmäkieli on helposti lähestyttävä ja intuitiivinen, joka mahdollisesti parantaa asiantuntijoiden tehokkuutta kohdejärjestelmässä.

Tuotteen massaräätälöinti yhdistää massa- ja yksilötuotannon yhdeksi kokonaisuudeksi. Massatuotannolla tarkoitetaan standardisoitua prosessia, jolla pyritään valmistamaan tai tuottamaan tuotteita nopeasti ja kustannustehokkaasti. Yksilötuotannolla tai käsityönä valmistettu räätälöity tuote on yksittäisen asiakkaan tarpeiden mukaan valmistettu. Usein yksilötuotanto tulee asiakkaalle kalliimmaksi kuin massatuotanto (Blecker, ym. 2004). Massaräätälöinnissä tavoitteena on saavuttaa massatuotannon tehokkuus yksilötuotannolle.

Massaräätälöintiä voidaan harjoittaa, kun tuote on konfiguroitavissa. *Konfiguroitava tuote* tarkoittaa, että tuotteesta voidaan rakentaa erilaisia versioita, vrt. legot. Konfiguroitava tuote on rakenteeltaan hierarkkinen, ja se sisältää kaikki tiedot tuotteen materiaaleista, alikomponenteista, osista ja niiden lukumääristä, joita vaaditaan toimivan lopputuotteen valmistamiseksi. *Tuotekonfiguraattorit* ovat järjestelmiä tai ohjelmistoja, joissa tuotteiden massaräätälöinti on keskeinen ominaisuus. Tuotekonfiguraattori auttaa ja ohjaa rakentamaan tuotteen vastaamaan käyttäjän tai asiakkaan tarpeita. Tyypillisesti tuotteen konfigurointi on interaktiivista käyttäjän ja tuotekonfiguraattorin välillä.

Tuotemallintaja on asiantuntija, joka tuntee teknisesti tuotteen tai palvelun, ja sen toiminta-alueen. Tuotemallintaja tyypillisesti luo tuotekonfiguraattoriin konfiguroitavia tuotteita, joita voidaan konfiguroida jälkeenpäin interaktiivisesti. Tuotemallintajan tehtävä on luoda tuoterakenteeseen sääntöjä, jotka estävät käyttäjää rakentamasta tuotetta, jota ei voi valmistaa. Säännöt voivat olla myös ohjaavia.

1.1 Tutkimuskysymykset

Uuden täsmäkielen toteuttaminen kaupalliseen tuotekonfiguraattoriin on erittäin mielenkiintoista, koska se on keskeisessä roolissa tuotemalleissa ja tuotemallinnusprosessissa. Koko tuotemallin liiketoimintalogiikka ja matemaattiset kaavat perustuvat nykyisin XSLT-pohjaisiin kaavoihin, jotka tullaan korvaamaan täsmäkielellä. Liiketoimintalogiikka ja kaavat voivat sisältää esimerkiksi hinnanlaskentaa ja rakenteiden lujuuslaskentaa, mutta periaatteessa ne voivat sisältää mitä tahansa laskentaa, joita pystyisi tekemään taulukkolaskentaohjelmalla. Tässä työssä käsiteltävä kaupallinen tuotekonfiguraattori on ohjelmisto, joka suorittaa liiketoimintalogiikkaa ja XSLT-pohjaisia kaavoja. XSLT-kaavojen suurin ongelma on huono suorituskyky ja käytettävyys. Tämän työn tutkimuskysymykset ovat seuraavat:

- *Kuinka täsmäkieli toteutetaan ja otetaan käyttöön tuotekonfiguraattorissa?*
- *Mikä on täsmäkielen suorituskyky verrattuna XSLT-pohjaisiin kaavoihin?*

1.2 Käytettävä teoria ja tutkimusmenetelmä

Työ toteutetaan toimeksiantona vaasalaiselle ohjelmistoyritykselle. Kyseessä on konstruktiivinen tutkimus, koska työssä kehitetään täysin uusi täsmäkieli vain tuotekonfiguraattorin käyttöön. Täsmäkielen tulkki ohjelmoidaan Java-kielellä, jotta täsmäkielellä tehtyjä kaavoja ja algoritmeja voidaan suorittaa ajonaikaisesti Javan virtuaalikoneessa, koska kohteena oleva tuotekonfiguraattori on ohjelmoitu Java-kielellä. Vastaavia tutkimuksia tai töitä löytyy yllättävän vähän liittyen täsmäkielen liittämiseen kaupalliseen tuotekonfiguraattoriin, mutta täsmäkieliä on kuitenkin kehitetty moneen muuhun eri tarkoitukseen ja ympäristöön. Kappaleessa 2 esitellään aikaisempia tutkimuksia täsmäkielistä ja niiden nykytilannetta.

Täsmäkielen suunnittelun ja toteutuksen tukena käytetään ohjelmistoalan artikkeleita ja kirjallisuutta. Varsinaisen työn kannalta oleellinen teoria liittyy täsmäkieleen ja tulkkiin. Tarkemmin sanottuna teoriaa esitellään liittyen abstrakteihin syntaksipuihin ja niiden

tulkkeihin. Tuotekonfiguraattoreiden ja konfiguroitavien tuotteiden teoriaa esitellään myös.

1.3 Keskeiset käsitteet

Työn keskeiset käsitteet ovat täsmäkieli, tulkki, abstrakti syntaksipuu, tuotekonfiguraattori, tuotemalli, Java ja XSLT.

2 AIKAISEMMAT TUTKIMUKSET

Tässä työssä käsiteltävä täsmäkieli ja tuotekonfiguraattori ovat yhdessä hyvin ainutlaatuinen yhdistelmä, joten tutkimuksia on hyvin vaikea löytää, joissa molempia aiheita on käsitelty. Erillisiä tutkimuksia kuitenkin löytyy tuotekonfiguraattoreista ja täsmäkielistä. Tämän kappaleen tarkoituksena on esitellä nykyaikaisia tutkimuksia täsmäkielistä ja niiden käyttötarkoituksia erilaisissa ympäristöissä.

2.1 Oracle Graal ja Truffle

Graal on moderni ja kokeellinen Java-kielellä kirjoitettu kääntäjä, joka voidaan integroida Java-virtuaalikoneeseen korvaamaan alkuperäisen C++-kielellä kirjoitetun C2-kääntäjän. Truffle on täsmäkielen kehittämiseen ja suoritettavan puumallin (abstrakti syntaksipuu) määrittämiseen suunniteltu sovelluskehys, joka toimii yhdessä Graal-kääntäjän kanssa. (Wimmer & Würthinger 2012.)

Truffle tarjoaa Java-ympäristössä täsmäkielten kehittäjille työkalut suorituskykyisen kielen kehittämiseen. Truffle-sovelluskehystä käyttämällä täsmäkieli toteutetaan yksinkertaisella AST-rakenteeseen perustuvalla tulkilla, jossa jokainen täsmäkielessä oleva operaatio tai operandi kuuluu omaan suoritettavaan solmuun. AST-rakenteeseen tai solmujen suoritusmetodeihin lisätään Truffle:n tarjoamia erikoistettuja Java-annotaatioita ja muita Trufflea ja Graal-kääntäjää ohjaavia komentoja. Truffle:n ja Graal-kääntäjän yhteistoiminnan ansiosta Java:n virtuaalikoneessa oleva Graal-kääntäjä osaa käsitellä täsmäkieltä kieliriippumattomasti, ikään kuin se olisi Java-ohjelma.

Truffle:n ja Graal:n toiminta Java-virtuaalikoneessa perustuu ajonaikaiseen käännökseen, jossa täsmäkielen AST-rakenteen suorituksen aikana tavukoodi monistetaan välittömästi konekoodiksi. Täsmäkieli hyötyy samoista käännösvaiheen optimoinneista kuin Java-kieli Java:n virtuaalikoneessa, mutta Truffle ja Graal-kääntäjä mahdollistavat saman suorituskyvyn kieliriippumattomasti. Javascript-kielelle on tehty prototyyppitoteutus

Truffle-sovelluskehyksellä, jota voidaan suorittaa Java:n virtuaalikoneessa, jossa Java-virtuaalikoneen C2-kääntäjä on korvattu Graal-kääntäjällä. (Wimmer & Würthinger 2012.)

Truffle:n ja Graal:n kehittäjien mukaan AST-rakenteeseen perustuvan tulkin toteuttaminen on yksinkertaisempaa ja sen ylläpitäminen on vaivattomampaa kuin kielikohtaisen tavukoodin tuottaminen ja ylläpitäminen. AST-rakenne luodaan kerran ohjelmoijan toimesta ja sen jälkeen voidaan olettaa, että AST-rakenne on vakio tai muuttumaton. Graal käsittelee AST-rakennetta yhtenä loogisena yksikkönä ja muokkaa rakennetta suorituksen aikana tehokkaammaksi. Graal-kääntäjä käyttää tähän erilaisia optimointimenetelmiä, kuten puun uudelleenkirjoittamista (*tree rewriting*) ja osittaista evaluointia (*partial evaluation*). Graal-kääntäjän suorittamat optimoinnit ovat riippuvaisia AST-rakenteesta ja sen tulkin tuottamista ajonaikaisista profilointitiedoista käyttäen Truffle:n rajapintoja. (Würthinger, Wöß, Stadler, Duboscq, Simon & Wimmer 2012.)

2.2 IRON-täsmäkieli

IRON on täsmäkieli, joka on kehitetty esineiden internet (*Internet of Things, IoT*) -sovellusalueelle, jolla voidaan luoda helposti tapahtuma-ehto-toiminta -tyyppisiä sääntöjä ja ohjata IoT-objekteja. IRON-täsmäkieli on toteutettu yksinkertaisella tulkkitoteutuksella, jonka isäntäkielenä on LUA. IoT tai älykkäät ympäristöt ovat järjestelmiä, jotka reagoivat ympäristössä tapahtuviin muutoksiin. IoT-järjestelmät ovat näin ollen reaktiivisia, johon IRON-täsmäkieli on suunniteltu helppokäyttöiseksi kieleksi reaktiivisen järjestelmän mallintamiseksi. Helppokäyttöisyyden lisäksi IRON-täsmäkielen tulkin tavoitteena on ehkäistä tilanteita, joissa sääntöjen välille muodostuu syklejä tai sääntöjen lopputoiminnot ovat epämääräisiä. Tulkki pyrkii tuottamaan yksityiskohtaisia raportteja tai viestejä ajonaikana ilmenevistä ongelmista, jotka auttavat kielen käyttäjiä selvittämään ongelmatilanteita. (Cacciagrano & Culmone 2018.)

IRON-tulkissa on kuitenkin se heikkous, että virhetilanteet esiintyvät ohjelman suorituksen aikana eikä käännöksen tai sääntöjen luonnin aikana ohjelmointityökalussa. Cacciagranon ja Culmonen mukaan on kuitenkin mahdollista tehdä tulevaisuudessa komponentti, joka tarkistaa käännöksen aikana tai ennen sitä mahdolliset virhetilanteet. Tällöin tulkille saadaan tuotettua virheetöntä tavukoodia ja sen ei tarvitse tarkastella mahdollisia virhetilanteita suorituksen aikana. (Cacciagrano & Culmone 2018.)

3 TUOTTEEN KONFIGUROINTI

McDermott (1982) määrittelee konfiguraation seuraavalla tavalla:

- Sisältää kiinteän, ennalta määritetyn joukon komponentteja, joiden ominaisuuksia on kuvattu attribuuteilla, yhteyksillä muihin komponentteihin ja muilla rakenteellisilla rajoitteilla
- kuvaus toivotusta konfiguraatiosta
- ja sisältäen mahdollisesti kriteerit optimaalisten valintojen tekemiseksi.

Konfiguraation tarkoitus on olla yksi tai useampi konfiguraatio, jotka toteuttavat annetut vaatimukset, missä konfiguraatio on joukko eri komponentteja ja kuvaus yhteyksistä joukossa olevien komponenttien välillä, tai havainto vaatimuksissa esiintyvistä epäjohdonmukaisuuksista.

Ylläolevassa määritelmässä on kolme tärkeää huomioitavaa asiaa:

1. komponenttijoukko on kiinteä, eli joukkoon ei voi lisätä uusia komponentteja
2. komponentit voidaan yhdistää keskenään ennalta määritetyllä tavalla eikä yhteyksiä voi muuttaa jälkeenpäin
3. ratkaisu määrittelee komponentit ja kuinka ne voidaan yhdistää keskenään.

Tuotekonfiguraatio tuottaa yksilöllisiä, asiakaskohtaisia tuotteita. Päämääränä on luoda tuotteita, jotka täyttävät asiakkaan tarpeet. Samaan aikaan tuotteen konfigurointi pyritään pitämään kustannustehokkaana ja pienentämään myyntivaiheen läpimenoaikaa. (Soininen 2000.)

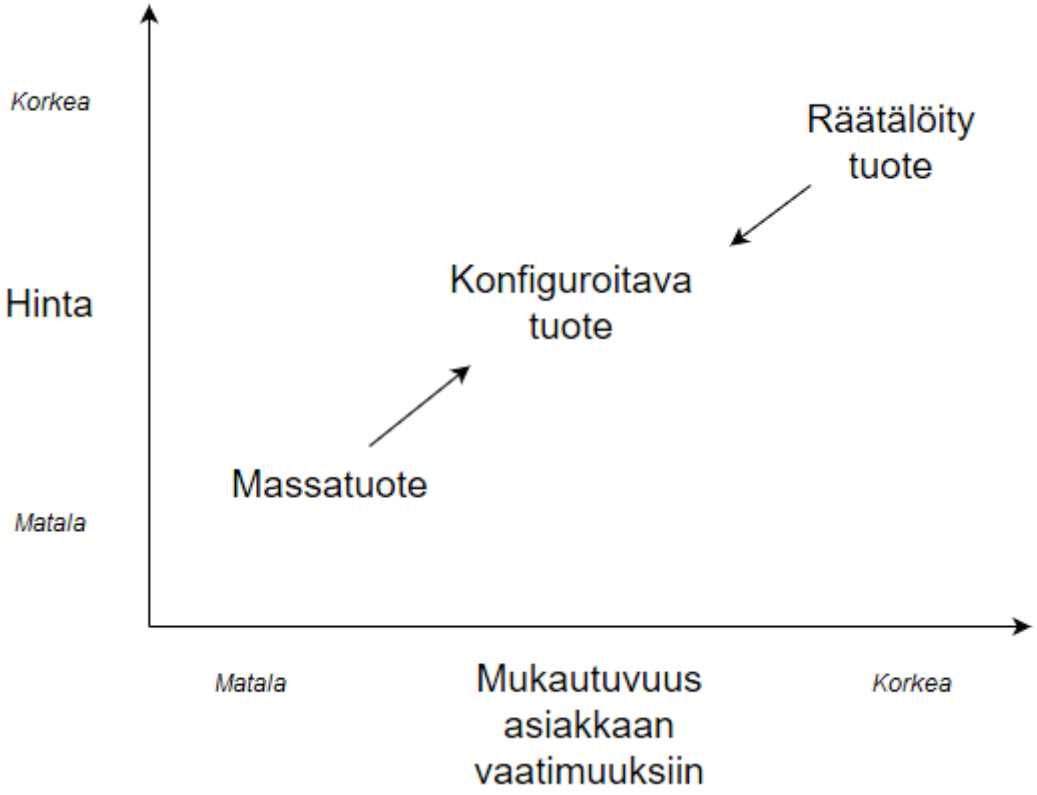
3.1 Konfiguroitavat tuotteet

Konfiguroitava tuote (usein kutsuttu nimillä ”tuoteperhe” tai ”tuotemalli”) on määritelmä tai suunnitelma itsenäisestä kokonaisuudesta, jota yritys myy. Tuoteinstanssi (yksilö) on yksittäinen muunnelma tai variaatio konfiguroitavasta tuotteesta, joka on valmis toimitettavaksi asiakkaalle. Konfiguroitava tuote sisältää joukon komponentteja. Komponentti on erillinen osa tuotteesta, joka voi viitata tuotteen fyysiseen osaan. (Soininen 2000.)

Soininen (2000) ja Tiihonen (1997) määrittelevät konfiguroitavalle tuotteelle seuraavat ominaisuudet:

1. Jokainen tuote on räätälöity sopimaan yksittäiselle asiakkaalle
2. Tuote vastaa asiakkaan antamiin vaatimuksiin
3. Jokainen tuote koostuu joukosta ennalta määriteltyjä komponentteja, jotka eivät muutu myynti- ja toimitusprosessin aikana
4. Tuotteella on esisuunniteltu yleinen rakenne
5. Muokkaus ja suunnittelu voidaan hoitaa järjestelmällisesti myynti- ja toimitusprosessien aikana.

Konfiguroitava tuote eroaa selkeästi räätälöitävistä- ja massatuotteista. Kuten kuvasta 1 nähdään niin yritys voi vaihtaa konfiguroitaviin tuotteisiin räätälöitävistä tuotteista tai massatuotteista. Massatuotteet ovat usein erittäin halpoja ja mukautuvat heikosti asiakkaan tarpeisiin. Kun yritys aloittaa tuottamaan konfiguroitavia tuotteita, tarkoituksena on tehdä tuotteita, jotka vastaavat asiakkaan vaatimuksiin paremmin kuin massatuotteet, mutta samaan aikaan hinta nousee. Yleensä räätälöity tuote sopii asiakkaalle täydellisesti, mutta ovat kalliita. Tuotteiden hinnat laskevat ja mukautumiskynnys asiakkaan tarpeisiin myös laskee, kun vaihdetaan konfiguroitaviin tuotteisiin. (Sarinko 1999.)



Kuva 1. Eri tuotetyyppien vertailua (Tiihonen & Soininen 1997).

Konfiguroitava malli sisältää kaiken tarpeellisen informaation, jota tarvitaan oikeiden tai pätevien tuoteinstanssien konfiguroimiseen. Yleensä malli edustaa erittäin suurta joukkoa eri tuoteinstansseja. Malli myös edustaa kaikkia valideja komponenttien kombinaatioita tuotteesta. Tämä on yleensä saatu aikaan muun muassa rajoitteilla ja tuotanto-kulutus-suhteilla. (Soininen 2000.)

Yleinen esimerkki konfiguroitavasta tuotteesta on auto. Autossa on suuri joukko erilaisia komponentteja. Eräitä mahdollisia komponentteja voisi olla auton malli (viisto- tai porrasperä), moottori (1 tai 2 litraa, bensiini tai diesel), vanteet, ajovalot jne. Kaikki komponentit eivät ole kuitenkaan yhteensopivia keskenään, joten niiden välille on yleensä määritelty rajoitteita. Erilaisten tuotteiden määrä kasvaa valtavasti, kun komponenttien määrä nousee. Esimerkiksi on arvioitu, että C-luokan Mercedes-Benz autosta on olemassa erilaisia variaatioita 10^21 kappaletta (Kübler, Zengler & Küchlin 2010).

3.2 Tuotekonfiguraattorit

Tuotekonfiguraattorin kokonaisvaltaisena määritelmänä voidaan pitää ohjelmistoa, jolla voidaan luoda, ylläpitää ja käyttää elektronisia tuotemalleja, jotka mahdollistavat täydellisen kuvauksen kaikista mahdollisista tuoteoptioista ja tuotevariaatioiden kombinaatioista minimaalisella tietomäärällä ja ylläpidolla (Bourke 2000). Yksi keskeisimmistä ideoista on se, että tuotekonfiguraattorilla on kaikki oleellinen informaatio, miten rakentaa ”hyväksyttäviä” tuotteita (Skjevdal & Idsoe 2005).

Tuotekonfiguraattoreiden tavoitteena on tehdä massakonfiguroinnista helpompaa ja tehokkaampaa. Asiakas tai myyjä osaa tehdä virheettömän tuotteen tuotekonfiguraattorin avulla eikä asiakkaan tarvitse olla tuotteen asiantuntija. Tuotteen kelpoisuus tai hyväksyttävyys voidaan tarkistaa välittömästi tuotekonfiguraattorilta eikä siihen tehtävään tarvita ulkopuolista henkilöä.

Myynti- ja toimitusprosessissa on viisi vaihetta: 1) myyntimäärittely 2) valmistusmäärittely 3) valmistus 4) kokoonpano 5) toimitus. Tuotekonfiguraattori hoitaa kaksi ensimmäistä vaihetta ja tarjoaa tuotannolle määrittelyt ja valmistusohjeet. Ensimmäisessä vaiheessa asiakkaan vaatimukset ovat sovitettu vastaamaan myytäviä tuotteita ja niiden välisiä vaatimuksia. Valmistusmäärittelyn tekoa varten etsitään vastaavat tekniset tuotteet. (Heiskanen 2012.)

Varhaiset tuotekonfiguraattorit olivat järjestelmiä, joihin käyttäjä antoi vaatimukset syötteenä ja sen jälkeen järjestelmä etsi sopivan konfiguraation. Käyttäjä ei kuitenkaan nähnyt varsinaisia komponentteja ja interaktiivinen toiminta tuotekonfiguraattorin kanssa konfiguroinnin aikana oli hyvin rajoitettua. (Axling & Haridi 1996.)

Konfiguraattoreista tuli myöhemmin interaktiivisia. Perustavanlaatuisena ideana oli, että konfiguraattori avustaisi ja ohjaisi konfiguraation tekemisessä sen sijaan, että se tuottaisi automaattisesti valmiin konfiguraation. Käyttäjä antaisi vaatimukset askel kerrallaan ja

jokaisen askeleen jälkeen konfiguraattori näyttäisi, mitkä valinnat tai optiot ovat sallittuja. Axling (1996) kutsuu tätä valintoihin perustuvaksi interaktiiviseksi konfiguroinniksi (eng. *interactive configuration by selection*).

Tuotekonfiguraattoreiden yleinen ongelma on ollut järjestelmän ylläpidon haastavuus. Tämä johtuu siitä, että tuotemallit muuttuvat ajan myötä, jolloin malleihin lisätään uusia tuotteita ja komponentteja, vanhoja poistetaan ja muutetaan.

Eräs ensimmäisistä kaupallisessa käytössä olevista myyntikonfiguraattoreista oli XCON, jota DEC käytti tietokonejärjestelmien konfigurointiin (McDermott 1982). XCON otettiin käyttöön vuonna 1980. Kuten kaikki ensimmäiset konfiguraattorit, XCON toimi käytännössä vain säännöillä, joka puolestaan johti ylläpito-ongelmiin valtavien sääntökokonaisuuksien vuoksi (McDermott 1993).

3.2.1 Sääntöpohjaiset konfiguraattorit

Sääntö on looginen seuraus A => B, jossa A ja B ovat lausekkeita, jotka voidaan lukea ”jos A niin B”. Looginen seuraus toimii vain yhteen suuntaan, jolloin lausekkeella A ei ole vaikutusta lausekkeeseen B. Koska säännöt ovat yksisuuntaisia, sääntöpohjaiset konfiguraattorit edellyttävät käyttäjää antamaan syötteet tietyssä järjestyksessä. Vanhemmat konfiguraattorit perustuivat pelkästään sääntöihin, koska sääntöjen toteuttaminen on vaivatonta ja tehokasta.

Käyttäjät kokevat joskus helpommaksi käyttää sääntöpohjaisia tuotekonfiguraattoreita kuin rajoitteisiin perustuvia konfiguraattoreita. Tämä johtuu yksinkertaisesti siitä, että sääntöpohjaiset konfiguraattorit ovat suunniteltu kysymään käyttäjältä syötteitä yksi kerrallaan tietyssä järjestyksessä, mikä tekee järjestelmän käyttämisestä suoraviivaista ja järjestelmällistä.

Säännöt ovat todella hyödyllisiä tilanteissa, joissa halutaan kieltää tai sallia yksittäisiä valintoja tai joissa tehtyjen valintojen seurauksena mahdottomia valintoja ja

tuotekomponentteja halutaan piilottaa käyttöliittymästä. Tämä saattaa olla tarpeellinen esimerkiksi tuotevalitsin-tyyppisissä sovelluksissa, joissa käyttöliittymässä näkyvien valintojen tai tuotteiden suurta määrää halutaan rajata tehtyjen valintojen perusteella.

3.2.2 Rajoitepohjaiset konfiguraattorit

Rajoitteet toimivat molempiin suuntiin toisin kuin säännöt. Rajoitteet asettavat raja-arvoja, joiden rajoissa kyseisen valinnan vaihtoehdot ovat mahdollisia. Tällöin esimerkiksi kaikki vaihtoehdot ovat mahdollisia, jotka ovat suurempia kuin alaraja, mutta pienempiä kuin yläraja. Rajoitteita käytettäessä tehtyjen valintojen seurauksena on mahdottomat valinnat ja tuotekomponentit näkyvät käyttöliittymässä, mutta niiden valitsemista on rajoitettu. Jos käyttäjä haluaa valita kielletyn vaihtoehdon, järjestelmä ilmoittaa usein mitkä valinnat täytyy muuttaa, jotta kyseinen vaihtoehto on mahdollinen.

Rajoitteet ovat hyödyllisiä tilanteissa, joissa valinnoille halutaan asettaa raja-arvoja tai kiellettyjen arvojen halutaan näkyvän käyttöliittymässä. Tällöin käyttäjä on tietoinen siitä, jos jonkin kiinnostavan valinnan tekeminen onkin estetty aikaisemmin tehdyn valinnan seurauksena. Koska rajoitteet toimivat molempiin suuntiin, yksi valinta saattaa vaikuttaa ketjureaktion tavoin myös muihin valintoihin. Tällöin käyttäjä usein pääsee nopeasti haluamaansa lopputulokseen jo muutamalla valinnalla.

3.2.3 Funktiopohjaiset konfiguraattorit

Funktioilla voidaan toteuttaa monia sellaisia toimintoja, joita ei voi tehdä säännöillä tai rajoitteilla. Tällaisia toimintoja ovat erilaiset matemaattiset ja ehdolliset funktiot, joita käytetään muun muassa matemaattisissa laskutoimituksissa. Lisäksi kaikkein vaativimmissa toiminnoissa voidaan yhdistää useita eri funktioita yhdeksi funktioksi. Perussäännöstöä toteutetaan harvemmin pelkästään funktioilla, vaan usein ne toimivat sääntöjen ja rajoitteiden lisänä. Tässä työssä toteutetaan täsmäkieli, josta käytetään nimitystä funktiokieli kappaleesta 6 alkaen.

Funktiot ovat todella hyödyllisiä erilaisissa matemaattisissa laskutoimituksissa, kuten useista eri tekijöistä koostuvista hintojen, kustannusten, katteiden ja lukumäärien laskemisessa. Funktioilla voidaan myös vaikuttaa sääntöjen ja rajoitteiden aktivoitumiseen ja valintojen näkymiseen käyttöliittymässä. Lisäksi funktioita voidaan hyödyntää esimerkiksi haettaessa suuresta tietomäärästä halutuilla ehdoilla sopiva arvo.

4 TÄSMÄKIELI JA TULKKI

Täsmäkieli (Domain-specific language) on ohjelmointikieli, joka on suunniteltu tietylle sovellusalueelle. Ohjelmointikielet ovat monipuolisia ja yleiskäyttöisiä, mutta monet niistä ovat kuitenkin täsmäkieliä, koska ne ilmentävät tarkasti tietyllä sovellusalueella käytettävää semantiikkaa. Täsmäkieltä kutsutaan myös sovellus- ja tehtäväkohtaiseksi kieleksi ja erikoiskieleksi (Mernik, Heering & Sloane 2005). Jotkut täsmäkielistä ovat suunniteltu niin, että ne ovat yleiskäyttöisiä ohjelmointikieliä, joita voi käyttää miten ja missä tahansa (Wirth 1974). On myös täsmäkieliä, jotka ovat toteutettu itsenäisinä tulkkeina tai kääntäjinä käyttäen tavallisia ohjelmointikieliä (General-purpose Language, GPL), suunnittelumalleja ja työkaluja (Aho, Sethi & Ullman 1985).

Esimerkkejä yleisesti käytössä olevista täsmäkielistä:

- HTML (Berners-Lee & Connolly 1995), käytetty dokumenttien merkintäkielenä
- SQL (Chamberlin & Boyce 1974), standardisoitu kyselykieli relaatiotietokantoihin.

Täsmäkieliä käytetään usein suhteellisen rajoitetuissa ja suppeissa ympäristöissä ja sovellusalueilla, jolloin kielen ilmaisukyky korostuu yleiskäyttöisyyttä enemmän. Täsmäkielen käyttäminen on tehokkaampaa kuin tavallisen tai yleiskäyttöisen ohjelmointikielen, kun siihen lisätään sovellusalueelle ominainen notaatio tai merkintätapa. Näin ollen se parantaa käyttäjien tuottavuutta ja vähentää ylläpitokustannuksia. (Mernik ym. 2005.)

Perinteisessä ohjelmistokehityksessä käytetään tyypillisesti ohjelmointikieltä, joka soveltuu monipuoliseen käyttöön, kuten Java- tai C++-kieltä. Täsmäkieliä käytetään osana ohjelmistoa, mutta harvoin koko ohjelmistoa toteutetaan pelkästään täsmäkielellä. Täsmäkielen suunnittelu ja kehitys edellyttää taloudellisia ja teknisiä resursseja yritykseltä, jos kieltä kehitetään yrityksen sisällä. Tämän lisäksi yrityksen täytyy sitoutua ylläpitämään ja mahdollisesti laajentamaan täsmäkieltä myös tulevaisuudessa. Tämä voidaan kokea negatiivisena asiana, mutta on kehitetty erilaisia suunnittelumalleja, joita

voidaan hyödyntää täsmäkielten suunnittelussa ja toteutuksessa välttäen polkupyörän uudelleensuunnittelua ja korkeita kehitykseen kuluvia kustannuksia (Spinellis 2001). Taulukossa 1 esitellään yleisiä käytössä olevia täsmäkieliä.

Taulukko 1. Yleisiä käytössä olevia täsmäkieliä (Jones 1996).

Täsmäkieli	Sovellusalue	Taso
Excel	Taulukkolaskenta	57
HTML	Hypertekstin merkintäkieli	22
Latex	Ladontajärjestelmä	-
MATLAB	Tekninen laskenta	-
SQL	Tietokantakyselyt	25
VHDL	Laitteistokuvauskieli	17
Java	Yleisohjelmointikieli	6

Taulukossa 2. kuvataan ohjelmointikielten eri tasoja ja niiden suhdetta tuottavuuteen henkilökuukautta kohti. Tuottavuus on mitattu funktiopisteinä. Funktiopiste mittaa sitä, montako ohjelmariviä vaaditaan yhteen funktiopisteeseen (Jones & Bonsignour 2012). Se tarkoittaa käytännössä sitä, että matalan tason kieli vaatii enemmän ohjelmalauseita kuin korkean tason kieli vastaavan toiminnallisuuden toteuttamiseen, joka tällöin pienentää matalan tason kielen funktiopisteitä.

Kielen tason suhde kehityksen aikaiseen tuottavuuteen ei ole kuitenkaan täysin yksiselitteistä, sillä ohjelmoinnin osuus ohjelmistoprojekteissa on usein vain 30 prosenttia (Jones 1996). Täsmäkieliä on tarkkailtu usein myös käytännössä ja havainnoista on olemassa kvantitatiivisia tuloksia, mutta tulosten vahvistaminen on vaikeaa yleisellä tasolla ja tietyissä tapauksissa (Mernik ym. 2005).

Taulukko 2. Kielen tason suhde tuottavuuteen (Jones 1996).

Taso	Tuottavuus funktiopisteinä keskimäärin henkilökuukautta kohden
1-3	5-10
4-8	10-20
9-15	16-23
16-23	15-30
24-55	30-50
> 55	40-100

4.1 Täsmäkielen hyödyt ja haitat

Täsmäkielet mahdollistavat tavan ilmaista ratkaisutapoja tietyllä sovellusalueella. Tämä auttaa erityisesti sovellusalueen asiantuntijoita, jotka ymmärtävät sovellusalueen ja pystyvät hyödyntämään täsmäkieltä sovellusaluekohtaisiin tilanteisiin eli käytännössä ohjelmoimaan täsmäkielellä ratkaisuja. Täsmäkielet ovat myös helppo omaksua, koska ne ovat rajoitettuja ja suppeita sekä suunniteltu semantiikaltaan ja käyttötavaltaan sovellusaluetta silmällä pitäen. (Mernik ym. 2005; Spinellis 2011.)

Tyypillisesti tekniset henkilöt, kuten ohjelmistosuunnittelijat ovat toteuttaneet algoritmeja tai ratkaisuja sovellusalueelle, mutta täsmäkielet mahdollistavat sen, että sovellusalueen tietotaidon omaavat asiantuntijat pystyvät itse toteuttamaan tehokkaasti samat asiat täsmäkielellä (Aram & Neumann 2015). Toisaalta, Freudentahlin (2010) mukaan asiantuntijat, joilla ei ole teknistä taustaa tai aikaisempaa ohjelmointikokemusta voivat kokea täsmäkielen kirjoittamisen ja muokkaamisen haasteelliseksi.

Täsmäkieliä on kallista kehittää ja ylläpitää. Sen lisäksi täsmäkielen kehittäminen edellyttää työkalut tai ympäristön, jossa täsmäkielen ohjelmia voi toteuttaa. Täsmäkielet ovat usein suorituskyvyltään hitaampia kuin yleisillä ohjelmointikielillä toteutetut

vastaavat ohjelmat. Täsmäkielestä voi olla myös hankalaa löytää esimerkkejä toimivista ohjelmista, jos se on kehitetty yrityksen sisällä ja se ei ole yleisessä käytössä. Tämän lisäksi työvoimakustannukset voivat nousta, jos täsmäkielen osaajia ei ole tarpeeksi. (Mernik ym. 2005; Spinellis 2011.)

4.2 Täsmäkielen suunnittelumallit

Suunnittelumallit (*Design Patterns*) (Gamma, Helm, Johnson & Vlissides 1995) ovat ohjelmistojen suunnittelussa käytettyjä tekniikoita, joita on hyödynnetty runsaasti 1990-luvulta alkaen erityyppisissä ohjelmistoissa ja ohjelmointikielissä. Suunnittelumalleja on helppo ottaa käyttöön, koska ne eivät vaadi tiettyä teknologiaa tai ohjelmointikieltä ja sen lisäksi ne ovat suosittuja, koska suunnitteluideoita on mahdollista muuttaa suhteellisen vaivattomasti suunnittelumalleiksi (Koskimies & Mikkonen 2005).

Suunnittelumallit ovat testattuja ja hyväksi havaittuja yleisiä ratkaisumalleja tai kuvauksia tiettyihin ohjelmistojen suunnittelua koskeviin ongelmiin (Gamma ym. 1995). Koskimies ja Mikkonen (2005) kuvaavat suunnittelumallin keskeisimpiä osia seuraavasti:

- *Ongelma*. Suunnittelumallilla yritetään ratkaista jokin yleinen suunnitteluongelma, joka on riippumaton käytetystä ohjelmointikielestä.
- *Ongelmayhteys*. Kuvaa tilannetta, jossa suunnittelumalli on sovellettavissa ja millaisia vaatimuksia se asettaa ratkaisulle.
- *Ratkaisu.* Ratkaisun tulee olla samalla tavalla yleinen kuin ongelmankin, sillä ratkaisun täytyy pystyä esittämään visuaalisesti formaalimmalla kielellä, kuten UML:llä.

Suunnittelumallit koostuvat ohjelmointikielestä riippuen tyypillisesti eri ohjelmayksiköistä, kuten komponenteista, luokista, rajapinnoista ja metodeista.

Suunnittelumalli määrittelee yksiöiden väliset suhteet ja kuinka ne kommunikoivat keskenään. (Koskimies & Mikkonen 2005.)

Suunnittelumalleja ei pidä sekoittaa algoritmeihin tai tietorakenteisiin, koska suunnittelumalleja ei voi ohjelmoida siten, että ne olisivat valmiita yleiskäyttöisiä komponentteja tai metodeja, joita voisi käyttää missä tahansa ohjelmistossa uudelleen (Spinellis 2001).

4.2.1 Kielen hyödyntäminen

Täsmäkieltä voidaan lähteä kehittämään suunnittelumallilla, jossa käytetään hyväksi olemassa olevan yleisohjelmointikielen tai täsmäkielen osia. Tämä lähestymistapa on toteutettavuudeltaan helpoin. Mernik (ym. 2005) tunnistaa kolme suunnittelualimallia:

- *Reppuselkä* (Piggyback). Käyttää olemassa olevaa ohjelmointikieltä, mutta vain osittain. Ohjelmointikieli toimii täsmäkielen isäntäkielenä, jolloin täsmäkieli pystyy käyttämään isäntäkielen ominaisuuksia. Eräitä reppuselkämallia hyödyntäviä täsmäkieliä ovat Yacc ja Lex.
- *Erikoistaminen* (Specialization). Muistuttaa hieman reppuselkämallia, mutta tässä mallissa isäntäkielen ominaisuuksia karsitaan ja supistetaan, jotta täsmäkieli saadaan muodostettua. Täsmäkieli muistuttaa kuitenkin isäntäkieltä, mutta siitä on voitu poistaa ylimääräisiä syntaktisia ja semanttisia ominaisuuksia sekä muut komponentit, kuten säikeet ja IO-operaatiot. Erikoistettuja täsmäkieliä ovat esimerkiksi MicroC/OS ja JavaLight.
- *Laajennus* (Extension). Täsmäkieli laajentaa olemassa olevan ohjelmointi- tai täsmäkielen syntaktisia tai semanttisia ominaisuuksia. Täsmäkieli voi myös täydentää kohteena olevan kielen elementtejä kuten lisäämällä uusia tietotyyppejä. Täsmäkieli on yhteensopiva isäntäkielen kanssa niin syntaksiltaan kuin ominaisuuksiltaan, mutta täsmäkieli tarjoaa sen lisäksi laajennettuja ominaisuuksia, joita isäntäkieli ei tarjoa. Esimerkiksi alkuperäinen C++-kääntäjä käytti tätä tekniikkaa (Spinallis 2001).

4.2.2 Kielen keksiminen

Joskus tilanne edellyttää, että täsmäkieli suunnitellaan täysin alusta. Kielen suunnittelu alusta on hyvin haastavaa, jos suunniteltava täsmäkieli ei perustu mihinkään olemassa olevaan ohjelmointikieleen. Tämän kaltainen tilanne vaatii resursseja ja päteviä teknisiä henkilöitä. Brooksin (1996) ja Tennentin (1977) mukaan luettavuus, yksinkertaisuus ja ortogonaalisuus ovat tärkeitä yleisohjelmointikielen suunnitteluperiaatteita, mutta ne pätevät myös täsmäkieliin. Ortogonaalisuus on tärkeä käsite ohjelmointikielessä, koska se kertoo kielen yksinkertaisuudesta siinä määrin, että kuinka vähän komponentteja ja kuinka vähin tavoin niitä voidaan yhdistellä, joilla haluttu lopputulos saavutetaan. Mitä vähemmän poikkeuksia ohjelmointikielessä ja syntaksissa, sitä ortogonaalisempi kieli on.

Vaikka täsmäkielen suunnittelija pystyisi noudattamaan tarkasti suunnitteluperiaatteita, on silti otettava huomioon, että täsmäkielen käyttäjät eivät aina ole välttämättä ohjelmoijia tai teknisiä henkilöitä. Täsmäkielen suunnittelijan tulisi pyrkiä pitämään sovellusalueen ja loppukäyttäjien tunteman notaation tai merkintätavan alkuperäisenä ja välttää kielen liiallista jalostamista yleiskäyttöiseksi (Wile 2004).

4.3 Täsmäkielen toteuttaminen

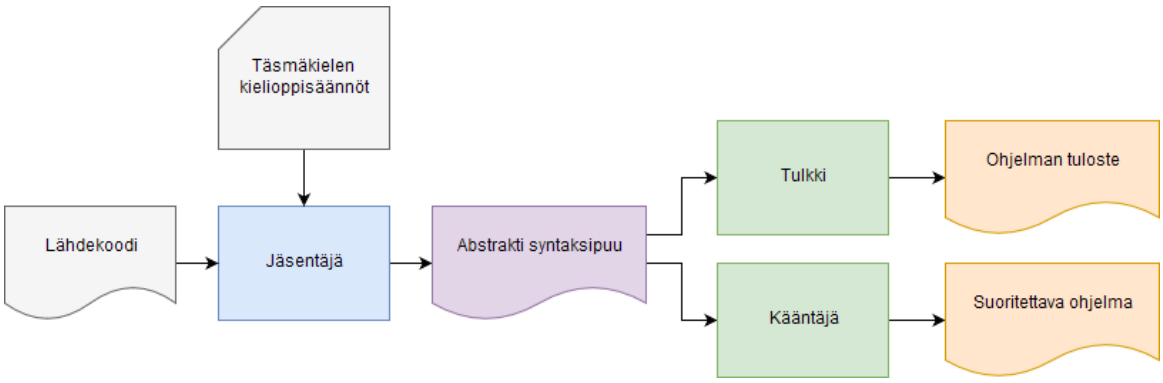
Täsmäkielet ovat pääsääntöisesti ajettavia tai suoritettavia. Toteutusprosessi lähtee liikkeelle sopivan suoritustekniikan valinnasta. Täsmäkielen suoritusmalli voi perustua tavanomaisiin tekniikoihin, joita on käytetty yleisohjelmointikielissä. Siitä huolimatta käytetyt suoritustekniikat täsmäkielissä eroavat usein tavallisista ohjelmointikielistä, koska täsmäkielissä käytetyillä suoritustekniikoilla pyritään vähentämään toteuttamiseen kuluvaa aikaa ja helpottamaan ylläpitoa. (Mernik ym. 2005.) Taulukossa 3 on esitelty erilaisia suoritustekniikoita tai arkkitehtuureja (Mernik ym. 2005).

Taulukko 3. Täsmäkielen suoritustekniikoita.

Suoritustekniikka/arkkitehtuuri	Kuvaus
Tulkki	Käytetään kun ohjelmointikieli on luonteeltaan dynaaminen ja suoritusnopeudella ei ole suurta merkitystä. Yksinkertaisempi toteuttaa kuin kääntäjä ja mahdollistaa paremman ylläpidettävyyden.
Kääntäjä / sovellusgeneraattori	Täsmäkielen rakenteet muunnetaan kantakielen rakenteiksi (Assembly, C/C++) ja kirjastokutsuiksi. Täsmäkielen kääntäjää kutsutaan usein myös sovellusgeneraattoriksi.
Upottaminen	Täsmäkielen rakenteet ovat upotettu tai sisällytetty isäntäkieleen. Tämä voi tarkoittaa uusien tietotyyppien ja operaattoreiden lisäämistä isäntäkieleen.
Esiprosessointi	Samankaltainen kuin kääntäjä, jossa täsmäkielen rakenteet käännetään kantakielen rakenteiksi. Esimerkiksi täsmäkielen lähdekoodi voidaan esiprosessoinnissa muuttaa toisen kielen lähdekoodiksi ilman välimuotoa.

Tulkkiarkkitehtuurissa tulkki suorittaa ennalta määritettyjä toiminnallisia kuvauksia olemassa olevassa ohjelmistoalustassa käyttäen hyväksi sen tarjoamia palveluja. Esimerkiksi SQL-kielen toiminnallista kuvausta ja standardisoitua notaatiota voidaan siirtää ja käyttää eri tietokantahallintajärjestelmissä, koska näissä järjestelmissä on usein tulkit, jotka osaavat suorittaa yleistä SQL-kieltä. (Koskimies & Mikkonen 2005.)

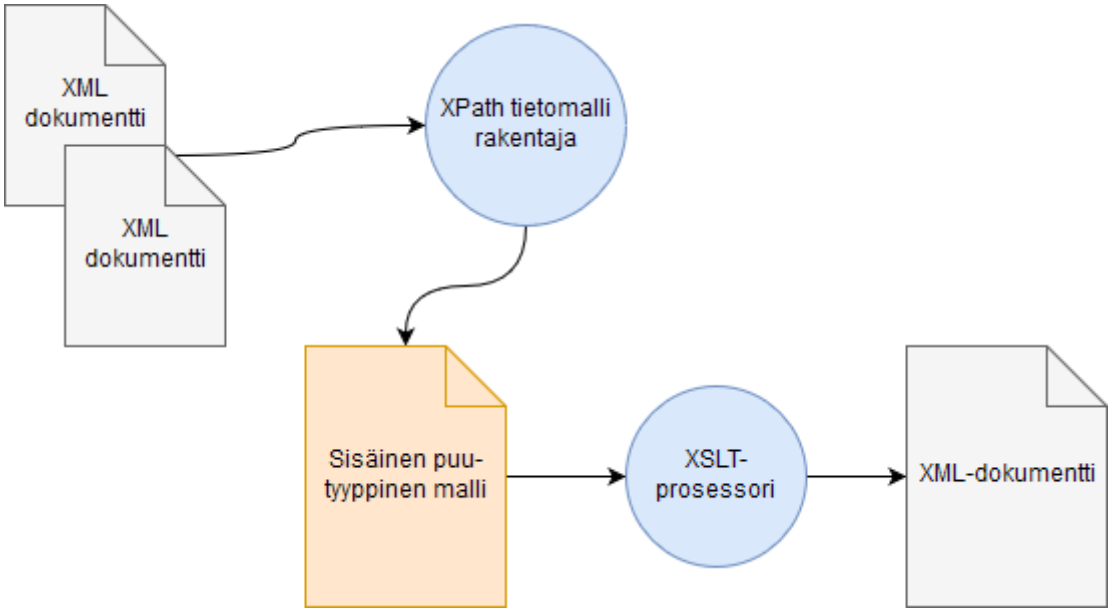
Tässä työssä tullaan toteuttamaan tulkkipohjainen ratkaisu täsmäkielelle. Toteutusalustana voi toimia myös toteutuskieli, jolloin erillistä toteutusalustaa ei tarvita, esimerkiksi Java-kielellä toteutettu täsmäkielen tulkki voi toimia myös toteutusalustana. Suoritettavan täsmäkielen prosessia yhdessä tulkin ja kääntäjän kanssa on havainnollistettu kuvassa 2.



Kuva 2. Suoritettavan ja tulkattavan täsmäkielen prosessi yleisesti.

4.4 XSLT

XSLT on W3C-yhteisön kehittämä ja suosittelema tapa määritellä miten XML-dokumentteja esitetään ja muunnetaan eri muotoon. XSLT on periaatteessa ohjelmointikieli, joka on kehitetty ainoastaan XML-dokumentteja varten ja täten osaa toimia vain siinä ympäristössä. XSLT on myös itsessään XML-tyyppinen kieli, jonka syntaksi perustuu XML-merkintäkieleen. XSLT:llä voidaan muuntaa XML-dokumentteja eri muotoon, jolloin lopputuloksena syntyy uusi XML-dokumentti (ks. kuva 3). XML-dokumentti voi olla HTML-sivu, jonka voi esittää WWW-selaimessa. (W3C 2018a.)



Kuva 3. XSLT-muunnoksen eri vaiheet.

XPath on XSLT-kielen tapaan ohjelmointikieli, jolla pystytään tunnistamaan osia XML-dokumentista. XPath ja XSLT toimii yhdessä ja XSLT tyypillisesti käyttää XPath-kieltä kyselykielenä. XPath-kielellä ei voi tuottaa suoritettavaa ohjelmaa ja ajaa sitä sellaisenaan, vaan XPath-kielen suorittaa ja tulkkaa aina kohdeympäristössä toimiva isäntäkieli esimerkiksi XSLT, Java, Python tai joku muu kieli. XPath on hyödyllinen ja tehokas, koska sillä voi esimerkiksi etsiä HTML-dokumentista kaikki div-elementit, joilla on class-attribuutti tietyllä arvolla. (W3C 2018a.)

XSLT-muunnos perustuu erilaisiin sääntöihin millä tavalla lähteen puumalli muutetaan tulospuumalliksi. Puumalli on porrasteinen esitys XML-dokumentin sisällöstä ja rakenteista. Muunnos saadaan aikaiseksi yhdistämällä lähdedokumentista löydetty kaava tai malli XSLT-tyyliohjeessa olevan mallin kanssa. Lähdedokumentin puumalli ja lopputuloksen puumalli on erotettu toisistaan, sillä lopputulos voi olla täysin erilainen ja se voi sisältää eri rakenteita ja uusia elementtejä alkuperäiseen dokumenttiin verrattuna. (W3C 2018b.)

Esimerkki XML-lähdedokumentista:

```
<productfamily>
   <components>
     <component>a</component>
     <component>b</component>
   </components>
</productfamily>
```

Esimerkki XSLT-tyyliohjeesta:

```
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:exsl="http://exslt.org/common">
   <xsl:output method="text" omit-xml-declaration="yes" indent="no" />

   <xsl:template match="/productfamily ">
     <xsl:value-of select="count(components/component)” />
    </xsl:template>
</xsl:stylesheet>
```

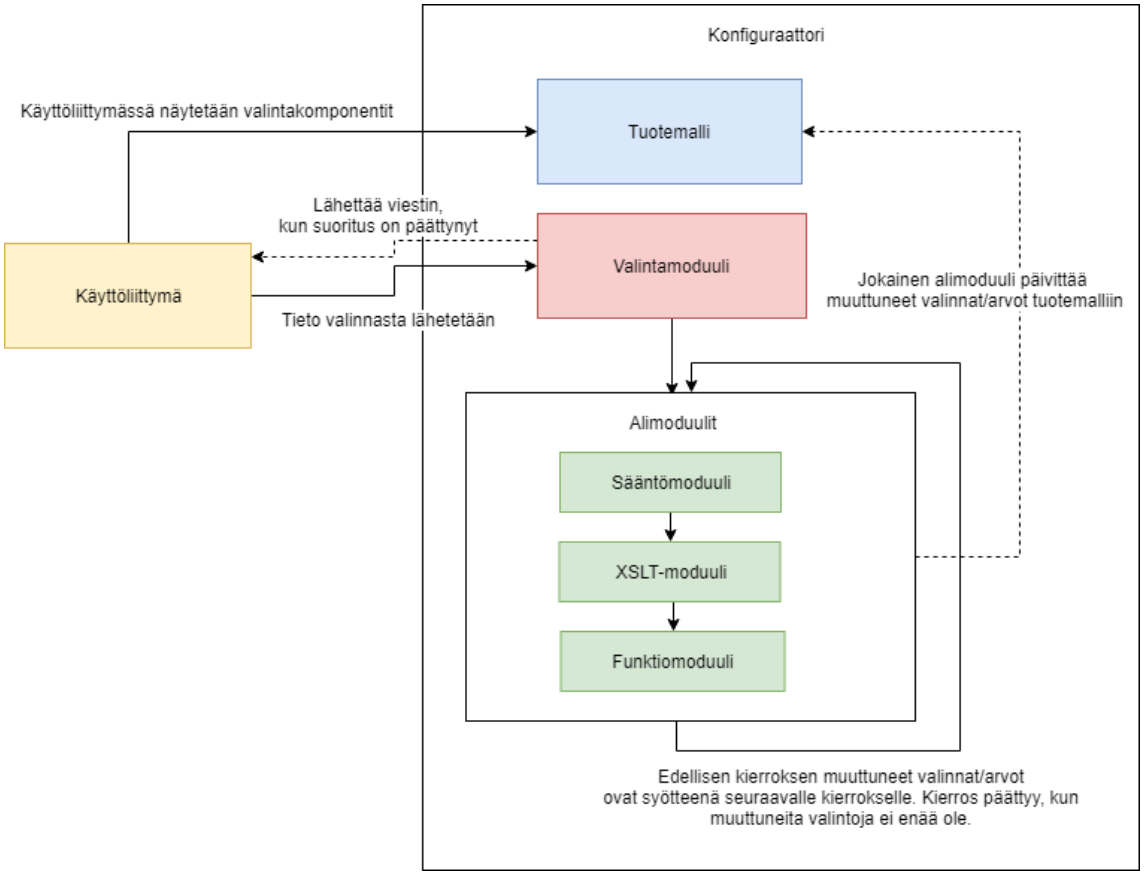
Tämän esimerkkimuunnoksen lopputulos on tekstituloste, jonka arvo on 2. XSLT-tyyliohjeessa on XPath-täsmäyssääntö ”/productfamily”, joka löydetään lähdedokumentin ensimmäisestä XML-elementistä. XSLT-prosessori suorittaa ”template” elementin sisällä olevan muunnoksen ”value-of”, joka palauttaa ”select” attribuutin sisällä suoritetun funktion ”count” ja XPath-kyselyn ”/components/component” yhdessä tuottaman arvon.

5 JÄRJESTELMÄ

Tässä työssä käsiteltävä konfiguraattori on ollut kaupallisessa käytössä jo vuodesta 2001. Konfiguraattorilla on tällä hetkellä yli 7000 käyttäjää ja sillä on tehty 500 000 – 1 000 000 konfiguraatiota ja tilausta vuosittain.

Tähän päivään saakka konfiguraattorin tuotemallien laskentalogiikka on tehty ns. ”variableilla”, jotka ovat perustuneet XSLT-skripteihin. Ne koetaan hitaaksi konfiguraattorissa sekä käytettävyydeltään heikoiksi. Tuotemallien laskentalogiikka vaatii suhteellisen paljon XSLT-skriptejä ja niiden jatkuva käyttö ja kasvava määrä vaikeuttaa ylläpitoa entisestään.

Konfiguraattori kostuu ytimestä, joka pitää sisällään tuotemallin. Konfiguraattori käsittelee valintoja tuotemallissa ja se myös on vastuussa käyttöliittymän päivittämisestä valintojen muuttuessa. Valintojen käsittelemisen lisäksi konfiguraattori ajaa sääntöjä ja suorittaa laskentalogiikkaa. Jokainen kokonaisuus on oma itsenäinen alijärjestelmä, jota konfiguraattori käskyttää tietyssä järjestyksessä valintojen tapahtuessa (ks. kuva 4). Kaikki oleellinen tieto valmiin mahdollisen konfiguraation kannalta sijaitsee tuotemallissa kuten tuotedata, säännöt ja laskentalogiikka. Konfiguraattori on vain ulkopuolinen suorittaja.



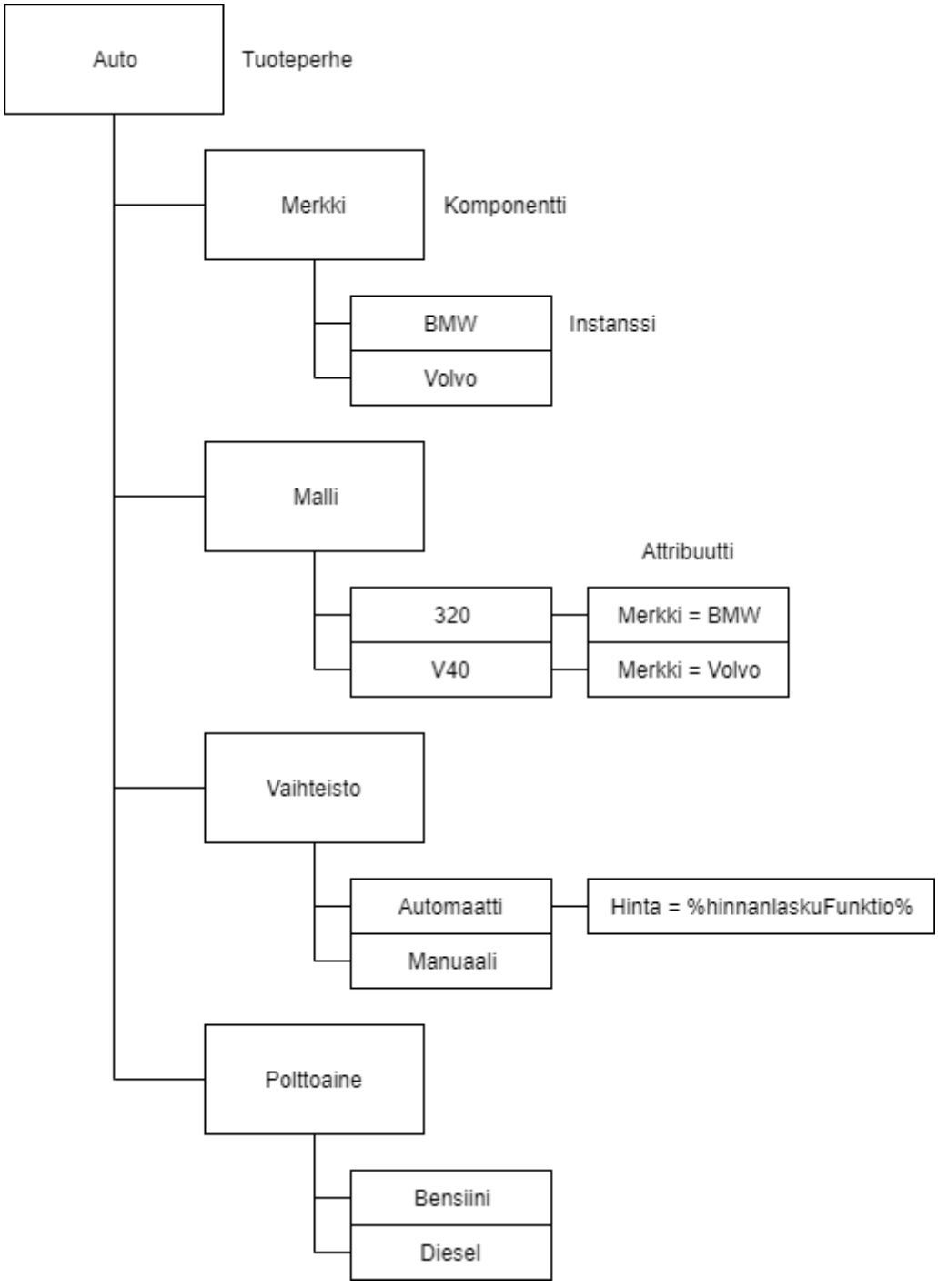
Kuva 4. Konfiguraattorin rakenne ja toimintaperiaate.

Konfiguraattorissa oleva tuotemalli ja sen valinnat yleensä esitetään käyttöliittymässä, jonka kautta käyttäjän tekemät komponenttien valinnat voidaan lähettää konfiguraattorille. Käyttöliittymä voi olla web- tai työpöytäsovellus, joka lukee tuotemallista kaikki komponentit ja visualisoi ne tuotemallissa määrätyllä tavalla, esimerkiksi tekstikenttänä tai pudotusvalikkona. Konfiguraattori suorittaa kaikki alimoduulit tietyssä järjestyksessä, kun käyttöliittymästä vastaanotetaan tieto muuttuneesta valinnasta. Konfiguraattorin toimintaperiaate perustuu siihen, että alimoduuleja suoritetaan toistuvasti niin kauan, kunnes tuotemallissa mikään valinta tai komponentin arvo ei muutu. Ensimmäisellä kierroksella on vain yksi muuttunut valinta, mutta jokin alimoduuleista voi aiheuttaa muutoksen useammassa tuotemallin komponentissa, jolloin seuraavalle kierrokselle annetaan uudet muuttuneet valinnat. Valintamoduuli on tietoinen tuotemallissa tapahtuvista muutoksista ja käynnistää uuden

kierroksen tarvittaessa. Valintamoduuli lähettää käyttöliittymälle viestin, kun konfigurointi on päättynyt ja antaa viestissä tiedon kaikista muuttuneista komponenteista.

5.1 Tuotemalli

Tässä kappaleessa esitellään lyhyesti niitä tuotemallin osia, jotka ovat täsmäkielen toteutuksen kannalta oleellisia. Tuotemalli on hierarkkinen tietorakenne ja se koostuu pääsääntöisesti komponenteista ja niiden ilmentymistä tai instansseista (ks. kuva 5). Komponenttien tieto on tallennettu instanssien attribuutteihin. Taulukossa 4. on kuvattu komponentin perusrakenne.



Kuva 5. Esimerkki tuotemallin rakenteesta.

Tuoteperhe koostuu yhdestä tai useammasta komponentista. Komponentit eivät ole tietoisia toisista komponenteista tuoteperhe rakenteessa. Tuotemalli sisältää kaiken mahdollisen tiedon, joista voidaan muodostaa eri variaatioita. Jotta tuotemalli on

konfiguroitava tai tuottaa realistisia konfiguraatioita, komponenttien välille voidaan rakentaa sääntöjä, jotka rajoittavat tai sallivat komponentin valittavia instansseja. Attribuuteilla ohjataan konfiguraattorin eri moduulien toimintaa, koska ne sisältävät konfiguraation tilan ja toimivat lähtöarvoina eri konfiguraattorin moduuleille. Tämän lisäksi attribuuttien arvoissa tapahtuvat muutokset voivat aiheuttaa sen, että komponenttiin kohdistuvat säännöt suoritetaan uudelleen.

Taulukko 4. Komponentin perusrakenne.

Komponentti 1				
#	Attribuutti 1	Attribuutti 2	Attribuutti 3	…
Instanssi 1	x	y	z	
Instanssi 2	a	b	c	

Taulukko 5. Esimerkki oikeasta komponentista.

Putki			
#	Hinta	Maks. paine (bar)	Halkaisija (mm)
Tyyppi 1	10	10	20
Tyyppi 2	12	8	30
Tyyppi 3	13	8	40
Tyyppi 4	14	6	50

Taulukossa 5. on kuvattu kuvitteellinen komponentti, joka voisi olla todellinen komponentti oikeassa tuotemallissa. Komponentissa on neljä erilaista valittavissa olevaa instanssia. Jokaisella instanssilla on samat attribuutit ja niiden arvot voivat olla mitä tahansa. Oletuksena attribuutin tietotyyppi on merkkijono, mutta sen voi halutessa muuttaa numeeriseksi, jolloin käyttöliittymässä voidaan tarkistaa käyttäjän antamien syötteiden oikeellisuus.

Usein attribuuttien arvot ovat staattisia ja pysyvät muuttumattomina koko konfiguroinnin ajan, jos attribuutissa ei ole viitattu XSLT-skriptiin. Toinen poikkeus tähän on tekstikenttä-tyyppinen käyttöliittymäkomponentti, jolla käyttäjä voi muuttaa komponentissa olevan attribuutin arvoa. Pääsääntöisesti konfiguraattorin käyttöliittymäkomponentit ovat alasputovalikoita, radio- ja valintanappeja. Jokaisella käyttöliittymäkomponentilla on tieto tuotemallin komponentista, jota sen on tarkoitus muokata. Tekstinkentän tapauksessa tuotemallin komponentti sisältää vain yhden vakioinstanssin, koska tekstikentällä ei voi tehdä valintaa. Sillä on mahdollista muuttaa vain yksittäisen attribuutin arvoa. Numeeriselle tekstikentälle voi asettaa ylä- ja alaraja-arvot, jolloin käyttäjä saa palautteen välittömästi käyttöliittymässä virheellisestä syötteestä. Taulukossa 6. on esimerkki tekstikenttäkomponentista. Tässä komponentissa on yksi attribuutti, jonka oletusarvoksi on asetettu ”x”. Käyttäjä näkee käyttöliittymässä ”x”-arvon tekstikentässä. Jos käyttäjä muuttaa tekstikentän arvoa käyttöliittymässä niin konfiguraattori asettaa uuden arvon tuotemallin komponentin attribuuttiin.

Taulukko 6. Esimerkki tekstikenttäkomponentista.

Tekstikenttäkomponentti 1	
#	Attribuutti 1
Vakioinstanssi	x

Tuotemallissa voi myös olla sääntöjä ja laskentaa. Säännöt ovat esitetty osiossa 3.2.1, JOS – NIIN -tyyppiset loogiset syy-seuraustapahtumat. Tässä työssä käsiteltävässä konfiguraattorissa säännöt voidaan liittää komponenttien attribuutteihin. Laskentaa voidaan tehdä ns. ”variableilla”, jotka ovat matemaattisia funktiota ja ne lasketaan annetun kaavan mukaan. Variableilla voidaan myös suorittaa kyselyjä tuotemallin komponentteihin, joilla saadaan haettua tietoa halutuista attribuuteista. Variable on konfiguraattorin sisäinen nimitys XSLT:llä toteutetusta skriptistä. Säännöt voivat toimia yhdessä XSLT-skriptien kanssa (skriptin suoritustulos voidaan asettaa sääntöön), jolloin säännöistä saadaan entistä dynaamisempia. Tässä työssä ei tulla käsittelemään sääntöjä,

joten niitä ei esitellä tarkemmin. XSLT-skriptiin voidaan viitata komponentin attribuuteissa, jolloin konfiguraattori osaa sijoittaa skriptin paluuarvon oikeaan attribuuttiin ja instanssiin. Taulukossa 7. on esimerkki komponentista, jossa hinnanlasku on hoidettu XSLT-skriptillä.

Taulukko 7. Esimerkki komponentista ja laskentaviitteestä.

Putki			
#	Yksikköhinta	Kappale	Hinta
Tyyppi 1	10	2	$hinta$
Tyyppi 2	20	3	$hinta$

Jokaisella tuotemallin komponentin instanssilla on sisäinen tilamuuttuja, joka kertoo konfiguroinnin aikana käyttäjälle ja konfiguraattorille, onko instanssi sallittu vai ei. Jos instanssi on sallittu, se voidaan valita vapaasti säännöistä piittaamatta. Käyttöliittymässä ei näytetä komponenttien instansseja, jotka ovat kiellettyjä, koska säännöt toimivat vain yhteen suuntaan.

Kuten jo aikaisemmin mainittiin, konfiguraattorin käyttöliittymäkomponentilla on kaksisuuntainen yhteys tuotemallin komponenttiin. Eli komponentti ei välttämättä muutu tuotemallissa käyttäjän tekemän valinnan johdosta, vaan muutos voi tapahtua osittain sisäisten muutosten seurauksena, esimerkiksi uuden instanssin valinta komponentissa säännön vaikutuksesta. Lappukomponenttia (eng. label) käytetään tiedon esittämiseen eikä tietoa voi muokata käyttöliittymässä. Tämä on hyödyllinen laskentatulosten esittämiseen.

5.2 Funktiokielen vaatimukset

Tämän tutkimuksen tavoitteena on suunnitella ja toteuttaa vaihtoehtoinen täsmäkieli korvaamaan XSLT-täsmäkielen, jotta tuotemallien rakentaminen olisi kustannustehokasta ja ketterää sekä ylläpidollisesti vaivatonta. Korvaavaa täsmäkieltä kutsutaan tässä työssä funktiokieleksi. Funktiokielen syntaksin määrittelyssä ja jäsentämisessä käytetään apuna kolmannen osapuolen ohjelmakirjastoa.

Funktiokieltä käytetään laskemaan ja hakemaan lisätietoa komponenteista. Tulokset voidaan sijoittaa komponentin attribuutteihin. Funktiokielen tulkki liitetään konfiguraattoriin niin, että funktioilla laskettuja arvoja voidaan hyödyntää ajonaikana muissa konfiguraattorin moduuleissa esim. säännöissä. Funktiokielen toiminnallisuuden ja käyttötavan halutaan ilmentävän Excelin kaavoja, jotta funktiokielen oppimiskynnys olisi mahdollisimman matala ja nopea omaksua, jos henkilö on aikaisemmin käyttänyt Excelin kaavoja.

Funktiokielen täytyy myös tukea ns. arvokohtaisia funktiota, mikä tarkoittaa sitä, että funktion suorituksen aikana kontekstina on komponentin instanssi. Samaa funktiota voidaan siis käyttää laskemaan useammalle komponentin instanssille arvoja käyttäen lähtötietona laskettavan komponentin instanssia. Excelissä on vastaava toiminnallisuus, jossa soluosoitetta käyttämällä samaa kaavaa voidaan toistaa useammalle riville.

6 TOTEUTUS

Tässä osiossa käsitellään täsmäkielen syntaksin suunnittelua ja tulkin implementointia. Täsmäkielen syntaksin ja semantiikan määrittelyssä hyödynnetään yleisesti käytettyä Backus-Naur-muotoa (BNF), joka on metakieli, jolla kuvataan ohjelmointikielien kielioppien esitysmuotoa (Backus 1958; Naur 1961). Funktiokielen syntaksi dokumentoidaan täten BNF-muotoa käyttäen. Syntaksin BNF-esitys voidaan muuntaa AST-rakenteeksi käyttäen kolmannen osapuolen ANTLR-kirjastoa. ANTLR-kirjasto on erikoistunut jäsentäjien automaattiseen generointiin BNF-esitysten pohjalta (Parr & Quong 1995). ANTLR-kirjasto on implementoitu useammalle eri kielelle ja sen on saatavilla mm. Java-kielelle (ANTLR 2018). Tulkki integroidaan konfiguraattoriin ja se implementoidaan Java-kielellä, jotta integroinnista saadaan tehokas ja helposti ylläpidettävä samassa ohjelmistossa ja teknologiaympäristössä.

ANTLR-kirjastolla luotu AST-oliorakenne ei ole vielä täydellinen suorituksen kannalta, koska rakenteista puuttuu tarkempi olioiden tyyppitieto. Tästä syystä ANTLR AST-rakenne erikoistetaan funktiokielen omaksi AST-rakenteeksi, johon lisätään suorituksen kannalta tarkempaa tietoa tulkin tehokkuuden ja yksinkertaisen toteutuksen mahdollistamiseksi. AST-rakenteista poistetaan lisäksi kaikki viitteet ANTLR-kirjastoon, jotta funktiokieltä käyttävät osapuolet eivät joudu esittelemään riippuvuutta ANTLR-ohjelmakirjastoon ohjelmistoissaan.

6.1 Täsmäkieli

Täsmäkieltä kutsutaan tässä työssä funktiokieleksi. Kyseessä ei ole kuitenkaan funktionaalinen kieli, vaikka nimi siihen viittaakin. Funktiokielen syntaksin inspiraationa ovat Excel-kaavat ja C#-kielen LINQ-lause. Excel-kaavojen tunnettuus ja helppo lähestyminen mahdollisesti edesauttavat funktiokielen onnistumista käyttöönotossa ja kouluttamisessa. LINQ on C#-kieleen sisäänrakennettu täsmäkieli, jolla voidaan esimerkiksi tehdä hakuja datajoukkoon ja suodattaa dataa tiettyjen ehtojen perusteella.

LINQ muistuttaa hyvin paljon SQL-kyselyä. Funktiokielellä täytyy pystyä hakemaan dataa tuotemallista helposti ja tehokkaasti, joten funktiokieleen lisätään LINQ-lausetta muistuttava hakuoperaatio.

Funktiokielen ohjelmalla ei saa tehdä suoria muutoksia elävään dataan tuotemallissa, koska se voi vääristää konfiguraatiota, jos muutos tehdään konfiguraattorin ulkopuolella konfiguroinnin aikana. Konfiguraattori ja funktiokielen liityntärajapinta on vastuussa funktiokielen ohjelman tulosteen päivittämisestä oikeaan paikkaan tuotemallissa.

Funktiokielen ohjelma kostuu hakuoperaatioista ja laskutoimituksista, jotka palauttavat arvoja eli kieli perustuu kokonaisvaltaisesti lausekkeisiin. Ohjelmassa pitää kuitenkin pystyä sijoittamaan väliaikaisia arvoja paikallisiin muuttujiin. Alla on kuvitteellinen esimerkki ohjelmasta, jossa sijoitetaan lista järjestettyjä kokonaislukuja paikalliseen muuttujaan ja palautetaan summa kaikista kahdella jaollisista luvuista:

```
$dataJoukko := [1, 2, 3, 4, 5, 6]
sum($dataJoukko as n where n % 2 == 0 select n)
```

6.1.1 Numerot, merkkijonot ja muuttujat

Funktiokielessä numerot ovat tärkein tietotyyppi, mutta myös merkkijonoilla on merkityksensä. Niitä voidaan käyttää funktiokielessä tekstin manipulointiin samoin kuten Excelissä. Numeroiden ja merkkijonojen BNF-muoto seuraava:

```
string ::= ’”’ (~’”’)* ’”’
boolean ::= ’true’ | ’false’
number ::= [0-9]+ (’.’ [0-9]+)?
symbol ::= [a-zA-Z_][a-zA-Z0-9_]*
localVar ::= ’$’ symbol
literal ::= string | boolean | number | symbol | localVar
```

Funktiokielessä numeroita käsitellään aina liukulukuina, vaikka kokonaislukuja voidaan käyttää ohjelmassa vapaasti. Liukuluvuilla saadaan tarkempia laskutuloksia suorituksen aikana. Tämän lisäksi funktiokieleen sisäänrakennetut funktiot hyödyntävät isäntäkielen aritmetiikkakirjastoa, joissa moni Java-metodi vaatii liukulukuja parametreiksi. Tulkki pyrkii aina muuntamaan syötetyt kokonaisluvut liukuluvuiksi, mutta funktiokieleen lisätään apufunktioita, joilla tyyppimuunnoksia voidaan tehdä manuaalisesti tilanteesta riippuen.

6.1.2 Listat

Funktiokielessä on listatietotyyppi, jolla voidaan esittää järjestettyjä arvoja, jossa sama arvo voi esiintyä useamman kerran. Lista on keskeinen tietotyyppi funktiokielessä, koska hakuoperaation toiminta perustuu listoihin ja niiden iterointiin (ks. 6.1.6). Lista on myös lauseke ja se määritellään BNF-esityksenä seuraavalla tavalla:

```
argumentsList ::= expr (’,’, expr)*
arrayExpr ::= ’[’ argumentList? ’]’
```

Listan voi myös määritellä tyhjänä tai listaan voi lisätä määrittelyvaiheessa muita lausekkeita pilkulla erotettuna. Lista voi sisältää myös muita listoja. Listaan lisätään ominaisuus, jolla listasta voidaan hakea yksittäinen arvo tietystä indeksistä tai muodostaa osajoukko alkuperäisestä listasta antamalla listalle parametrina alkuindeksi ja loppuindeksi. Alla oleva esimerkki kuvaa listojen käyttämistä funktiokielessä (#-merkki on kommentti):

```
# Listan määritteleminen muuttujaan.
$list := [1, 2, 3]

# Listasta voidaan noutaa tietty arvo halutusta indeksistä.
$firstValue := $list[0]

# Osajoukon tai listan muodostaminen.
```

```
# Tämä palauttaa uuden listan [2, 3].
$subList := $list[1:3]

# Listaa ei välttämättä tarvitse sijoittaa muuttujaan,
# jotta sitä voidaan käyttää.
$subList2 := [1,2,3,4][0:3] # Uusi osajoukko [1,2,3]
```

BNF-esitys listasta hakemiseen määritellään niin, että operaattorin kohteena eli operandina voi olla mikä tahansa lauseke ja parametreina korkeintaan kaksi lauseketta. Operandi voi palauttaa ajonaikana mitä tahansa arvoja, joten ohjelman oikeellisuutta ei voida täysin varmistaa jäsennysvaiheessa ja virheet ilmenevät vasta suorituksen aikana. Esimerkiksi muuttujan kautta palautuva arvo voi olla jokin muu arvo kuin lista ja virhe ilmenee suorituksen aikana ohjelmaa ajettaessa. Listan hakemisen BNF-esitys:

```
arrayAccessExpr ::= expr ’[’ expr (’:’, expr)? ’]’
```

6.1.3 Binäärioperaatiot

Funktiokielessä binäärioperaatioita ovat kaikki yhteen-, vähennys-, potenssi-, jako- ja kertolasku sekä jakojäännös. Binäärioperaatio nimenä ei viittaa binaarien käsittelemiseen, vaan kyseessä on suorituksen aikainen operaatio, joka käsittelee kahta lauseketta, jotka ovat operandeja. Vertailuoperaatiot ovat myös binäärioperaatioita funktiokielessä. Binäärioperaatioita voidaan ketjuttaa yhteen lukuun ottamatta vertailuoperaatioita. Vertailuoperaatioiden ketjuttaminen tai yhteen liittäminen täytyy hoitaa loogisilla operaatioilla. Useamman vertailuoperaation käyttäminen ilman loogista operaatiota tai virheellisesti liitettynä osaksi aritmeettista operaattoria ilmenee suorituksen aikaisena virheenä. Sulkujen käyttö on sallittua ja niillä voidaan muuttaa binäärioperaatioiden laskujärjestystä. Alla esimerkki funktiokielestä ja binäärioperaatioista:

```
$tulos := (6-2) * (5 – 3) + 1
$tulos2 := $tulos + sum([2, 3, 4])
```

BNF-muodossa esitettynä binäärioperaatio on rekursiivinen, sillä operandit ovat myös lausekkeita ja ne voivat olla jonkin toisen lausekkeen tuloksia. Näin ollen BNF-muoto täytyy määritellä rekursiiviseksi binäärioperaatioissa käytettävien arvojen kohdalla. Binäärioperaatio on myös lauseke, vaikka sitä ei erikseen mainita alla olevassa BNF-muodossa:

```
highPrecedence ::= expr (’*’|’/’|’%’|’^’) expr
lowPrecedence ::= expr (’+’|’-’) expr
comparison ::= expr (’<’|’>’|’<=’|’>=’|’!=’|’==’) expr
binaryExpr ::= highPrecedence | lowPrecedence | comparison
```

6.1.4 Loogiset operaatiot

Funktiokielessä loogisia operaatioita ovat konjunktio (and), disjunktio (or) ja negaatio (not). Negaatio on unaarinen-operaatio eli sillä on vain yksi operandi. Konjunktio ja disjunktio -operaatiot ovat luonteeltaan binäärioperaatioita suorituksen aikana, koska ne käsittelevät kahta operandia. Esimerkki loogisista operaatioista funktiokielessä:

```
2 > 3 and 5 < 2 and (1 == 0 or 2 == 2) and not (2 > 1)
```

Sulkujen käyttäminen on myös sallittua loogisten ryhmien muodostamiseksi. Loogiset operaatiot suoritetaan funktiokielessä aina vasemmalta oikealle, mutta sulkujen sisällä olevat operaatiot suoritetaan ensin. Loogiset operaatiot ovat myös lausekkeita, sillä ne palauttavat arvoja suorituksen jälkeen. Loogisten operaatioiden BNF-esitys on seuraava:

```
logicExpr ::= expr (’and’|’or’) expr
notExpr ::= ’not’ expr
```

6.1.5 Sijoitusoperaatio

Funktiokielessä on sijoitusoperaatio, jonka tarkoituksena on edistää ohjelman tai funktion luettavuutta ja suorituskykyä. Monimutkaiset ja pitkät ohjelmat voidaan pilkkoa osiin ja

sijoittaa tulokset väliaikaisiin paikallisiin muuttujiin, joita voidaan käyttää myöhemmin muualla samassa funktiossa parantamaan luettavuutta ja suorituskykyä hyödyntämällä valmiiksi laskettuja arvoja. Sijoitusoperaatio on ainoa operaatio, joka ei ole lauseke. Sijoitusoperaatiota ei voi siis käyttää lausekkeiden joukossa, vaan sijoittaminen täytyy hoitaa funktiossa omalla rivillä. Esimerkki sijoitusoperaation käyttämisestä funktiossa:

```
$muuttuja := 1 + 2
$tulos := $muuttuja + $muuttuja
```

Sijoitusoperaation vasemmalla puolella täytyy olla nimetty muuttuja, jonka edessä on dollarimerkki ($). Näitä käsitellään tulkissa paikallisina muuttujina ja ne ovat käytössä vain yhdessä funktiossa tai ohjelmassa. Sijoitusoperaatiot voidaan tehdä vain funktiokielen päälohkossa ja muuttujat ovat globaaleja yhdessä ohjelmassa tai funktiossa. Näin ollen paikallisia muuttujia voidaan käyttää muualla ohjelman osissa, vaikkapa sisäkkäisissä lohkoissa. Sisäkkäiset lohkot tai alilohkot voidaan toteuttaa hakuoperaatioilla (ks. 6.1.6). Paikallisia muuttujia ei voi kuitenkaan määritellä hakuoperaatioissa. Sijoitusoperaation BNF-esitys on seuraava:

```
assignment ::= localVar ’:=’ expr
```

6.1.6 Hakuoperaatio

Funktiokielessä hakuoperaatiolla voidaan etsiä listasta arvoja tietyillä ehdoilla ja muodostaa arvoista uusi lista. Hakuoperaatio palauttaa aina uuden listan eikä hakuoperaatiolla voi muuttaa listaa, jota on käytetty operandina. Operandina voidaan käyttää ainoastaan listatyyppiä, mutta operandi voi olla myös muuttuja, jonka todellinen tyyppi tarkistetaan suorituksen aikana. Esimerkki hakuoperaation käyttämisestä funktiokielessä, jossa alkuperäisestä listasta poimitaan kahdella jaolliset luvut, jotka ovat suurempia kuin kaksi:

```
$lista := [1, 2, 3, 4, 5, 6]
```

```
# $tulos muuttujaan sijoitetaan uusi lista, jossa ovat arvot
# [4, 6]
$tulos := $lista as n where n % 2 == 0 and n > 2 select n
```

Yllä olevassa esimerkissä esitellään symboli n, johon sijoitetaan seuraava arvo alkuperäisestä listasta jokaisella iteraatiolla. Symboleja voi olla korkeintaan yksi. Mikäli symbolia ei ole määritelty, ajon aikana iteroitava arvo sijoitetaan tilapäiseen alaviiva-symboliin:

```
$tulos := $lista where _ % 2 == 0 and _ > 2 select _ limit 2
```

Hakuoperaatiossa on neljä operaattoria, jotka ovat *as*, *where*, *select* ja *limit*, joista vain where-operaattori ja ehdot ovat pakollisia. Where-operaattorilla on neljä operandia, lista muuttuja tai symbolin määritys, ehdot, valinta ja rajoitus. Jos valintaa ei ole erikseen määritelty, niin ajon aikana valitaan iteroitava arvo automaattisesti. Rajoitusoperaattorilla voidaan päättää hakuoperaatio, kun iterointeja on tehty annetun lukumäärän verran. Vaihtoehtoiset operaattorit helpottavat luettavuutta, jos ohjelmassa on paljon yksikertaisia ehtoihin perustuvia hakuja. Valinta-operaattorin yhteydessä voidaan vielä muuttaa palautettavaa arvoa. Hakuoperaation BNF-muoto on esitetty seuraavasti:

```
whereExpr ::= expr (’as’ symbol)? ‘where’ expr (‘select’
expr)? (‘limit’ expr)?
```

6.1.7 Funktiot

Funktiokielessä on paljon sisäänrakennettuja funktioita, joita voidaan käyttää uudelleen kaikissa ohjelmissa. Funktiot toteutetaan Java-kielellä eli isäntäkielellä ja niitä täytyy pystyä kutsumaan funktiokielestä. Ohjelmassa ei kuitenkaan voi toteuttaa omia funktioita. Alla on esitelty esimerkkejä suorasta ja epäsuorasta funktiokutsusta:

```
# Esimerkki suorasta funktiokutsusta,
# jossa summataan kaikki arvot yhteen.
```

```
$lista := [1,2,3]
$tulos := sum($lista)

# Esimerkki epäsuorasta funktiokutsusta.
# Listassa on sum ja product symbolit osoittamassa ko.
# funktioihin.
$lista := [sum, product]
# Poimitaan ensimmäisestä indeksistä sum-funktion osoitin ja
# kutsutaan funktiota asettamalla sulut arvojen 1 ja 2
# ympärille.
$tulos := $lista[0](1, 2)
```

Funktiokutsu sisältää kutsuttavan funktion tunnisteen tai nimen ja mahdollisen pilkulla erotetun parametrilistan. Jäsentimessä muodostetaan funktion nimestä oma AST-solmu, jonka perusteella tulkki pystyy kutsumaan oikeaa vastaavaa Java-toteutusta. Funktion nimi voi myös olla lauseke, jotta ohjelmaan saadaan dynaamisuutta. Funktion parametreina voi olla mitä tahansa lausekkeita eikä jäsentämisen aikana rajoiteta parametrien määrää. Tulkin vastuulla on päätellä toteutettujen Java-funktioiden salliman parametrien lukumäärän ja käsitellä niihin liittyvät virhetilanteet. Funktiokutsujen BNF-esitys on seuraava:

```
functionExpr ::= expr ( ’(’ argumentList? ‘)’ )+
```

6.1.8 Objektit

Funktiokielessä objektit ovat ilmentymiä isäntäkielen luokista, joiden attribuutteja täytyy pystyä lukemaan funktiokielen ohjelmassa. Uusia objekteja ei voi luoda funktiokielessä. Sen sijaan objektit liitetään ajon aikana integrointirajapinnan välityksellä tulkkiin antamalla objektille jokin kuvaava symboli. Objektit ovat keskeinen tietotyyppi funktiokielessä, koska ne sisältävät tietoa komponenteista ja niiden attribuuteista. Objektit voivat kuitenkin olla mitä tahansa Java-olioita ja attribuuttien lukeminen on integrointirajapinnan vastuulla. Funktiokielessä voidaan kirjoittaa attribuutin nimi, josta

tietoa halutaan lukea ja tulkin tehtävä on välittää lukupyyntö integrointirajapinnalle, joka suorittaa varsinaisen metodikutsun objektiin. Alla on esimerkki, kuinka tietoa luetaan objektien attribuuteista:

```
#  Person.java
class Person {
          private int age = 29;
          private String name = ”John Smith”;

          public String getName() {
                    return name;
          }

          public int getAge() {
                    return age;
          }
}

# Ilmentymä Person-luokasta on liitetty tulkkiin symbolina,
# jonka nimi on person. Nyt voimme lukea person-objektin
# muuttujan arvon syöttämällä muuttujan nimen, joka löytyy
# kyseisestä luokasta.
$personName := person.name

# Objektiin viittaava symboli voidaan liittää
# myös muuttujaan ja kutsua objektin attribuutteja muuttujan
# kautta, koska se viittaa samaan objektiin.
$person := person
$personName := $person.name

# Attribuuttien muuttaminen ei ole sallittua
person.name := ”Korvaava nimi” # VIRHE

# Objekteja voi käsitellä myös listoina ja hakuoperaatioissa
```

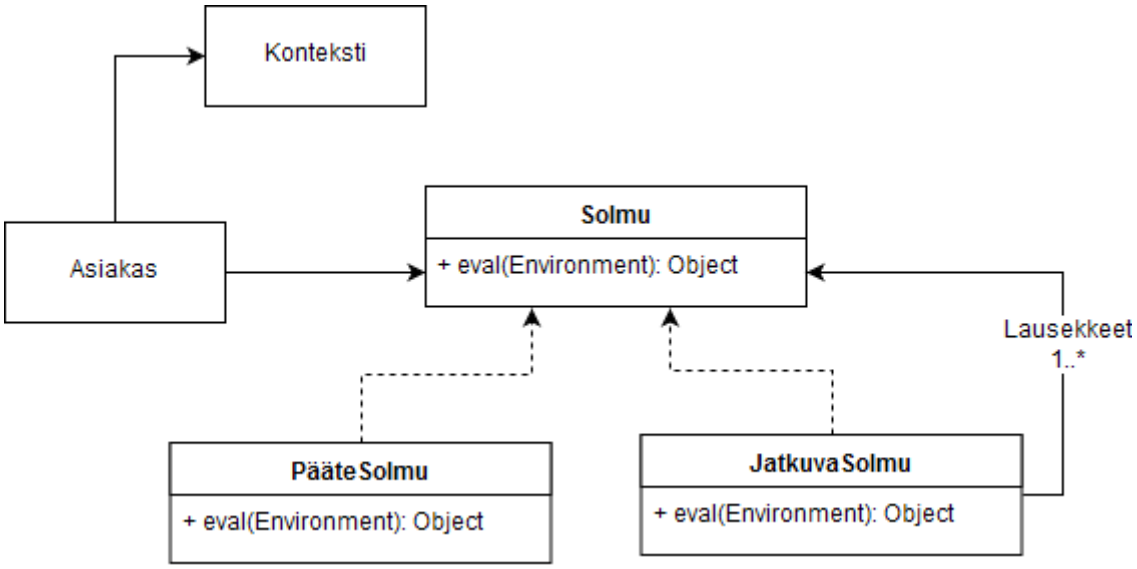
```
$people := [person, person]
$names := $people as p where p.age >= 18 select p.name
```

Objektit määritellään funktiokielen ohjelman julkiseen lohkoon symbolina ja funktiot ohjelmassa ovat myös julkisessa lohkossa. Sisäänrakennettujen funktioiden nimet ovat varattuja, toisin sanoen kahta saman nimistä objektia ja funktiota ei voi olla samaan aikaan ohjelmassa. Tulkki ja liityntä-rajapinta huolehtii, että sisäänrakennettuja funktioita ei saa korvata. Funktiokielen ohjelmassa objekteja käsitellään vain viitteinä ja ainoastaan niiden attribuutteja voidaan lukea. Tulkin vastuulla on ohjata attribuutin lukuoperaatio metodikutsuna Java-olion attribuuttia vastaavaan get-alkuiseen metodiin. Objektiviitteen BNF-esitys on sama kuin symboli, mutta objektin attribuutin luku BNF-esityksenä on seuraava:

```
memberAccessExpr ::= expr '.' symbol
```

6.2 Tulkki

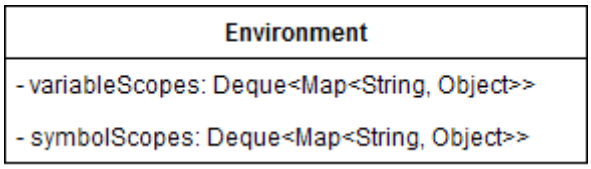
Funktiokielen tulkin arkkitehtuuri perustuu tulkki-suunnittelumalliin, jota on havainnollistettu kuvassa 6. Mallin toteuttaminen edellyttää, että kaikki syntaksipuussa olevat luokat implementoivat saman rajapinnan siten, että luokilla on sama metodi, jota tulkki kutsuu ohjelman suorituksen aikana. Jokainen rakenteessa oleva olio on vastuussa suorituksen jatkamisesta toisiin olioihin kutsumalla samaa rajapintametodia. Metodikutsussa parametrina annetaan ohjelman konteksti, josta löytyy globaalit ja lohkotason muuttujat sekä sisäänrakennettujen funktioiden rekisteri.



Kuva 6. Tulkki-suunnittelumalli.

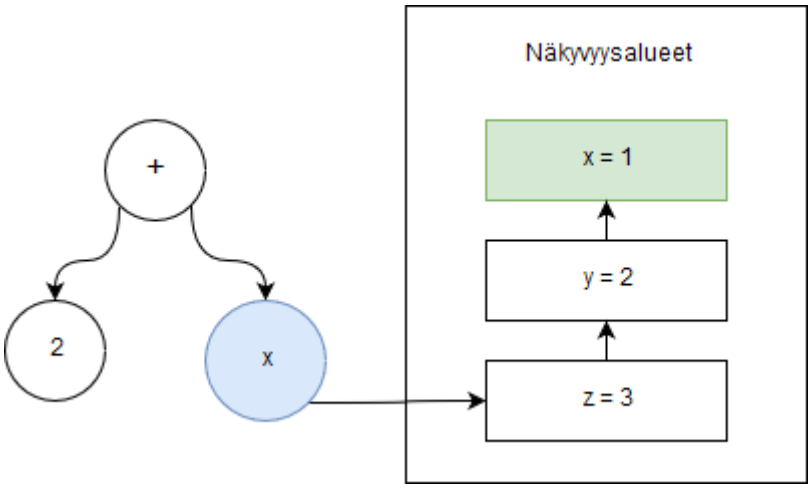
Funktiokielen tulkissa on kolme keskeistä komponenttia, jotka ovat solmu, pääte- ja jatkuvasolmu. Solmu on rajapinta ja se määrittää kaikille muille solmuille pakollisen toteuttavan metodin, jotka solmutyypistä riippuen jatkavat suoritusta muihin solmuihin. Jatkuvasolmu sisältää yhden tai useampia muita solmuja, jotka voidaan suorittaa ennen isäntäsolmua tai sen aikana. Päätesolmulla ei ole muita suoritettavia solmuja ja suoritus sen osalta päättyy siihen. Jatkuvasolmu voi olla esimerkiksi yhteenlaskuoperaatio, jolla on kaksi muuta numerosolmua, jotka ovat puolestaan päätesolmuja. Asiakas on se, joka käynnistää tulkin ja ohjelman eli tässä tapauksessa asiakas on konfiguraattori. Asiakas alustaa ohjelman kontekstin ja rekisteröi kontekstiin ohjelmassa käytettävät objektit ja niiden symbolit.

Konteksti on saatavilla kaikissa AST-rakenteen luokissa ja sitä kautta operaatiot pääsevät noutamaan muuttujien arvoja (ks. kuva 7). Kontekstissa on kaksi eri näkyvyysaluetta, joihin tallennetaan muuttujat ja symbolit. Näkyvyysalue on linkitetty lista assosiatiivisista taulukoista, jonka koko kasvaa, kun hakuoperaatio suoritetaan. Näkyvyysalueella pyritään erottamaan lohkoissa käytetyt symbolit toisistaan, jotta syvin hakuoperaatio-lohko ei korvaa korkeampien näkyvyysalueiden symboleja.



Kuva 7. UML-esitys kontekstiluokan näkyvyysalueista.

Symbolin tai muuttujan arvoja lähdetään etsimään näkyvyysalueiden lopusta annetulla tunnisteella tai nimellä (ks. kuva 8). Jos arvoa ei löydy, etsintää jatketaan aina edelliseen alueeseen, kunnes ensimmäinen tai korkein alue saavutetaan. Tulkki ilmoittaa virheestä ja lopettaa suorituksen, jos haettavaa arvoa ei löydy korkeimmastakaan alueesta. Suoritettavassa abstrakti syntaksipuussa on omat solmut muuttujan ja symbolin arvon lukemista varten.



Kuva 8. Muuttujan tai symbolin arvon etsintäjärjestys näkyvyysalueista.

6.2.1 Suoritettavan puumallin muodostaminen

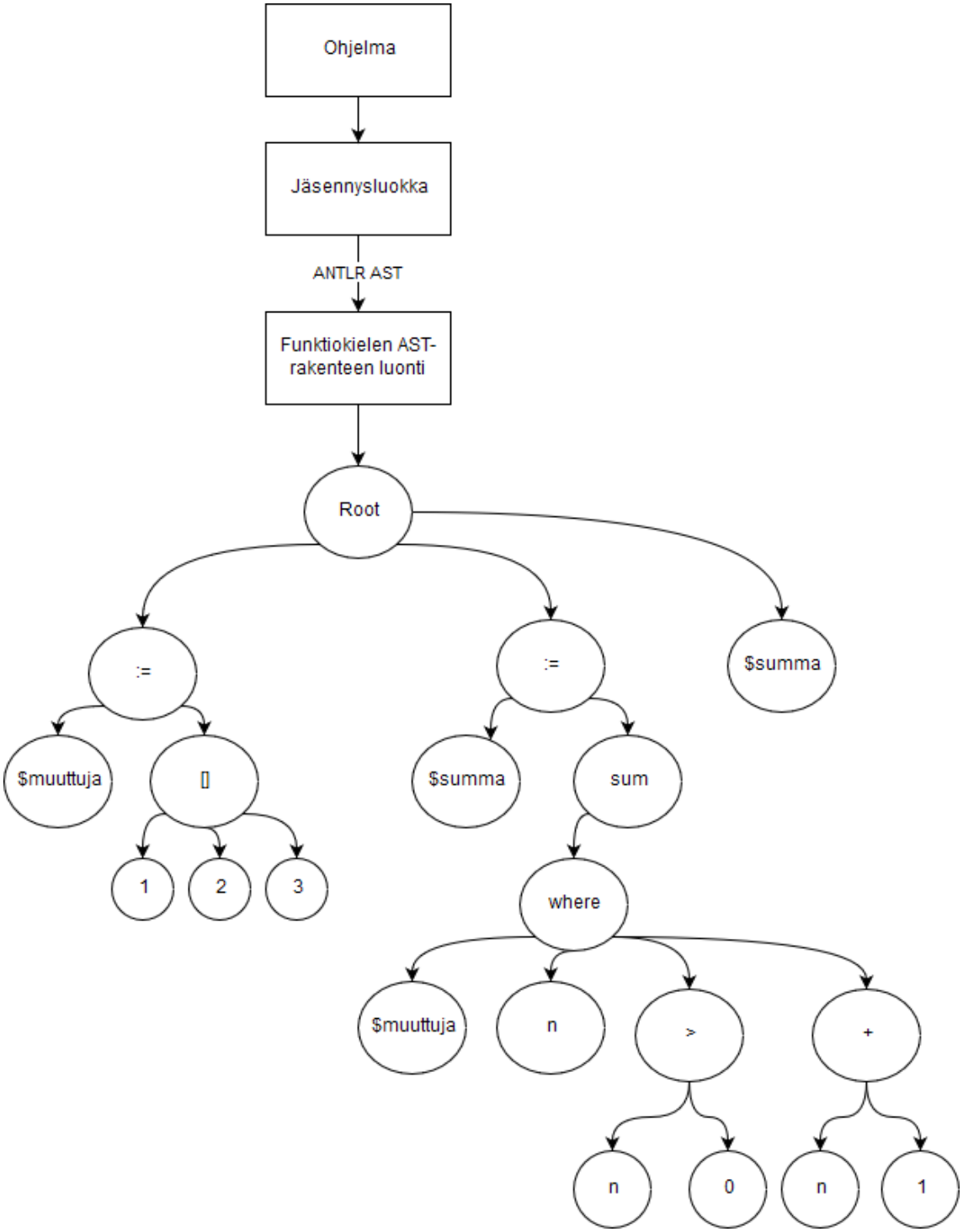
Funktiokielellä on oma erikoistettu AST-rakenne, joka muodostetaan ANTLR-kirjaston generoiman AST-rakenteen pohjalta. ANTLR-kirjaston generoimaa AST-rakennetta voisi periaatteessa käyttää sellaisenaan, mutta se aiheuttaisi vahvan riippuvuuden kyseiseen kirjastoon, koska kirjaston luokkia tarvittaisiin AST-rakenteen luokissa funktiokielen suorituksen aikana.

Funktiokielen BNF-esitys syötetään ANTLR-kirjastolle ja se generoi Java-luokan jäsentämistä varten (ks. kuva 9). Jäsennysluokka sisältää BNF-esityksen mukaiset jäsennyssäännöt ja AST-rakenteiden luontilauseet. Tämä prosessi suoritetaan kerran tai kun funktiokielen syntaksia muutetaan.

Kuva 9. Jäsennysluokan generointi.

Funktiokielen ohjelma voidaan esittää luonnollisesti puumaisena esityksenä, jossa jokainen solmu suorittaa toiminnon tietyssä järjestyksessä (ks. kuva 10). Alla olevassa esimerkissä kuvataan suoritettavan puumallin esitystä kuvitteellisesta ohjelmasta:

```
$muuttuja := [1, 2, 3]
$summa := sum($muuttuja as n where n > 0 select n + 1)
# Viimeinen rivi palauttaa tuloksen ohjelmasta
$summa
```



Kuva 10. Funktio-ohjelman muuntaminen suoritettavaksi puumalliksi.

Suoritettava puumalli pysyy muuttumattomana koko suorituksen ajan. Solmut eivät siis muuta omaa sisäistä toimintaansa suorituksen kannalta edullisemmaksi sitä mukaan, kun

samaa funktio-ohjelmaa suoritetaan. Puun optimointi on käytännössä solmujen sieventämistä yksinkertaisempaan muotoon. Esimerkiksi lauseke 1+2 sisältää kolme solmua suorituksen alussa, mutta koko lauseke voidaan korvata yhdellä vakiosolmulla 3, koska lauseke palauttaa jokaisella suorituskerralla saman arvon. Jotta tulkin toteutus olisi mahdollisimman yksinkertainen, puun tai solmujen muuntaminen tehokkaampaan muotoon ajon aikana ei ole tämän työn tavoitteena. Tämä kompromissi voi kuitenkin vaikuttaa negatiivisesti ohjelman nopeuteen, koska solmuja joudutaan suorittamaan useita kertoja, vaikka siihen ei välttämättä olisi tarvetta.

6.2.2 Tulkin liittäminen konfiguraattoriin

Konfiguraattorissa alimoduulit ovat itsenäisiä toteutuksia ja suorittavat niille kuuluvia toimintoja ja kokonaisuuksia, esimerkiksi sääntömoduuli suorittaa sääntöjä ja XSLT-moduuli suorittaa XSLT-skriptejä. Alimoduulit toteuttavat yhteisen rajapinnan, jotta valintamoduuli pystyy kutsumaan ja lähettämään tiedot muuttuneista valinnoista alimoduuleille. Alimoduuli on Java-luokka, jonka sisällä voi olla monimutkaisia tietorakenteita ja viitteitä muihin Java-luokkiin, mutta konfiguraattori tai valintamoduuli ei ole niistä tietoinen. Alimoduulilla on neljä tehtävää:

1. Muuttuneiden valintojen vastaanottaminen valintamoduulilta
2. Toiminnon suorittaminen
3. Alimoduulissa muuttuneiden valintojen tai arvojen päivittäminen tuotemallin komponentteihin
4. Muuttuneiden valintojen tai arvojen palauttaminen valintamoduulille

Funktiokielen tulkki ei sovellu sellaisenaan alimoduuliksi, koska se ei ole tietoinen konfiguraattorin rajapinnoista tai rakenteista, sen sijaan tulkin täytyy toimia itsenäisen funktiomoduulin sisältä. Funktiomoduuli on myös vastuussa siitä, että oikeat funktiot suoritetaan, kun komponentti muuttuu, valinta tai attribuutin arvo. Tämän lisäksi funktiomoduuli rekisteröi funktiossa käytetyt komponentit kontekstiolioon oikeilla symboleilla ja käynnistään tulkin. Lopuksi funktiomoduuli päivittää tuotemallin viemällä

funktion palauttaman arvon niihin komponenttien attribuutteihin, joissa on viittaus kyseiseen funktioon.

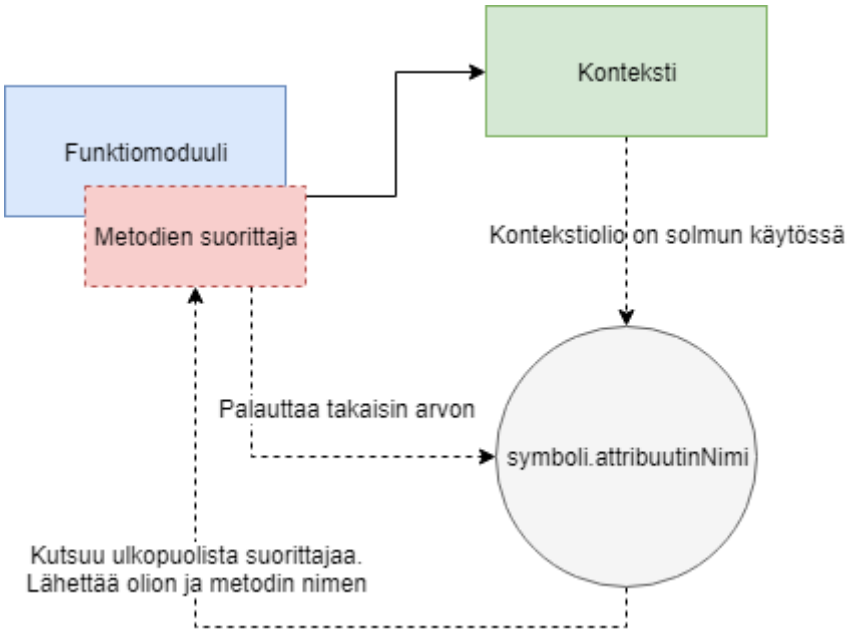
Tulkki ei sisällä viitteitä kaupalliseen konfiguraattoriin, sen rakenteisiin tai luokkiin. Teknisesti tulkki voidaan ottaa käyttöön missä tahansa ohjelmistossa, joka höytyisi funktiokielen tarjoamista ominaisuuksista. Tulkki osaa oletuksena kutsua Java-olion metodeja reflektoinnin avulla, mutta se ei osaa suorittaa olioille muita sovellusaluekohtaisia toimintoja. Esimerkiksi tulkin kontekstiin liitetään symboliksi alikomponentteja sisältävä komponenttiolio ja funktio-ohjelmassa pitäisi pystyä noutamaan kaikki kyseisen komponentin alikomponentit jollain tietyllä ehdolla, mutta siihen tarkoitettua julkista metodia ei ole olemassa komponentilla. Tämä ongelma ratkaistaan kohdeympäristössä laajentamalla funktiotulkkia kahdella eri tavalla:

1. Metodikutsuihin liittyvän oletustoiminnallisuuden ohittaminen tulkissa ja kutsujen suorittaminen kohdeympäristön liityntämoduulissa
2. Sovellusaluekohtaisten funktioiden liittäminen tulkkiin

Tässä työssä tulkki liitetään konfiguraattoriin hyödyntämällä molempia yllä mainittuja tapoja. Konfiguraattorin komponentit sisältävät attribuutteja, joiden arvoja luetaan funktiokielessä viittaamalla symboliin ja attribuutin nimeen, mutta komponentilla ei ole attribuutin nimeä vastaavaa metodia. Attribuutit ovat olioita komponenttiolion sisällä ja attribuuttiolio saadaan haettua nimen perusteella komponentilta siihen tarkoitetulla metodilla. Tätä kyseistä metodia ei voi käyttää sellaisenaan tulkissa, koska se palauttaisi attribuuttiolion ja vaatisi tuen metodiargumenteille funktiokielessä objektikutsujen yhteyteen.

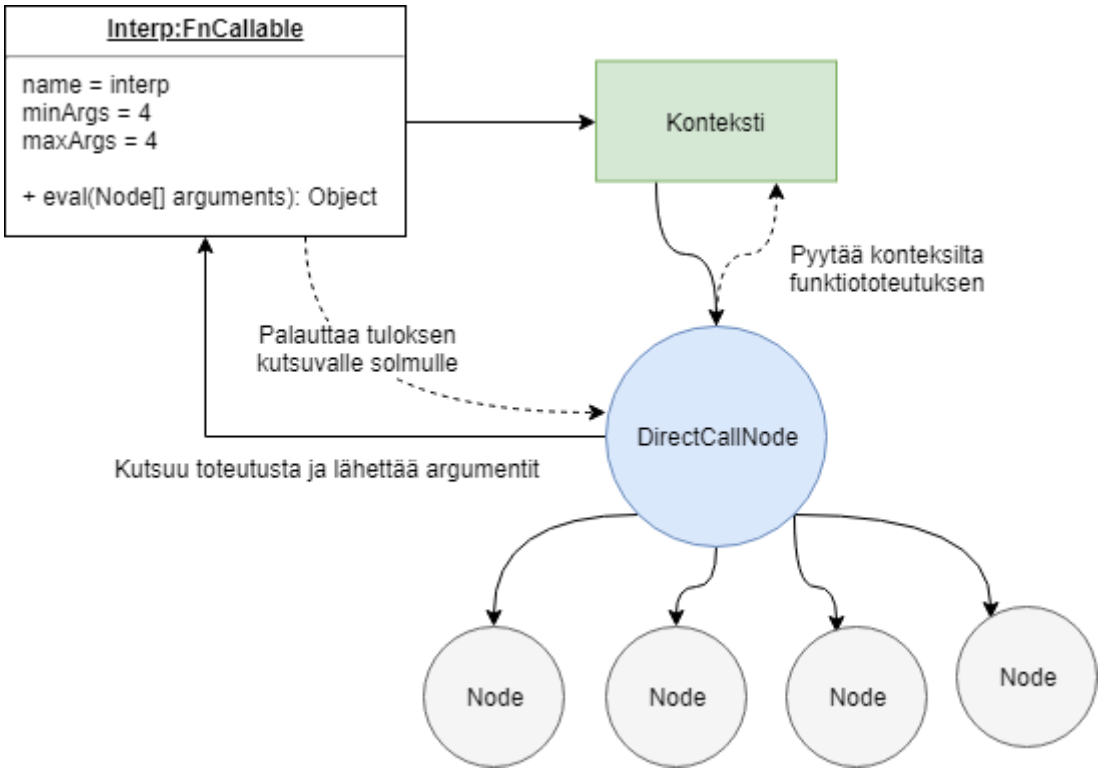
Funktiokielen tulkki olettaa, että symboliin liitetty olio ja siihen kohdistettu metodikutsu palauttaa primitiiviarvon tai merkkijonon. Olioiden metodikutsut suoritetaan normaalisti reflektoinnilla, jos tulkkiin ei ole liitetty ulkopuolista metodikutsujen suorittajaa (eng. *Call site*). Tulkki tarjoaa rajapinnan suorittajaa varten, jonka Java-luokka voi toteuttaa. Näin ollen funktiomoduuli toteuttaa CallSite-rajapinnan ja liittää sen tulkin kontekstiolioon, jotta kaikissa funktio-ohjelmissa tehtävät olioiden metodikutsut

suoritetaan liityntämoduulissa (ks. kuva 11). Tällöin funktiomoduuli osaa hakea oikean attribuuttiolion annetun nimen perusteella ja palauttaa tulkille arvon attribuutista.



Kuva 11. Ulkopuolisen metodisuorittajan liittäminen tulkkiin ja sen käyttäminen.

Toinen tapa laajentaa tulkkia ovat ulkoiset funktiot, joita käyttäjä voi hyödyntää ohjelmissa. Nämä ulkoiset funktiot eivät teknisesti eroa sisään rakennetuista funktiosta, kuten summafunktiosta, koska funktiototeutukset ovat Java-luokkia ja ne toteuttavat yhteisen rajapinnan (FnCallable). Tulkissa on useita sisäänrakennettuja funktiototeutuksia ja ne lisätään kontekstiolion alustuksen yhteydessä funktiorekisteriin. Ennen kuin funktio-ohjelma suoritetaan, kontekstiolioon on mahdollista liittää sovellusaluekohtaisia funktiototeutuksia tulkin ulkopuolelta, jos ne toteuttavat FnCallable-rajapinnan. FnCallable-oliot määrittelevät funktion nimen ja argumenttien minimi- ja maksimimäärän. Rekisteröinnin yhteydessä funktion nimi ja toteuttava olio lisätään rekisteriin eli assosiatiiviseen taulukkoon. Tämä mahdollistaa sen, että sisään rakennettu funktio voidaan korvata vaihtoehtoisella toteutuksella, jos funktion nimi on sama kuin olemassa oleva funktio. Kuvassa 12 havainnollistetaan interpolointifunktion liittämistä tulkkiin ja sen toimintaa suorituksen aikana.



Kuva 12. Ulkoisten funktioiden liittäminen tulkkiin ja niiden suorittaminen.

7 EVALUOINTI

Tässä kappaleessa evaluoidaan funktiokielen suorituskykyä verrattuna XSLT-skripteihin. Suorituskykytesti toistaa samaa ohjelmaa tietyn ajan, kunnes suorituskykytestiä ajava ohjelma palauttaa tulokset. Jokainen testissä ajettava ohjelma toteutetaan funktiokielellä, XSLT-skriptillä ja Java-kielellä. Ohjelmat ovat toiminnaltaan identtiset ja palauttavat samat tulokset testin päättyessä. Java-kielellä toteutettujen ohjelmien suorituskyky antaa tässä evaluoinnissa käsityksen tavoitettavasta tai parhaasta mahdollisesta suorituskyvystä isäntäkielen ympäristössä. Tämän vuoksi on myös mielenkiintoista verrata XSLT-skriptien lisäksi, kuinka lähellä tai kaukana funktiokielen suorituskyky on vastaavasta Java-ohjelmasta.

Suorituskykymittaukset toteutetaan *Java Microbenchmark Harness* (JMH) -kirjastolla, joka on Java-ympäristössä toimiva suorituskykymittauksiin kehitetty sovelluskehys. JMH-kirjaston avulla mittaustuloksista saadaan tarkempia verrattuna manuaalisiin testeihin, koska JMH-kirjasto valmistelee tai optimoi Java-virtuaalikonetta hallitusti ennen varsinaisten testien suorittamista, jotta Java-virtuaalikoneessa tapahtuvat muutokset testien aikana eivät vaikuttaisi negatiivisesti tuloksiin. Käytännössä JMH suorittaa jokaista ohjelmaa useita kertoja ennen varsinaisia testejä.

XSLT-prosessori ja XSLT-skripti alustetaan ennen suorituskykytestiä, jotta niihin kuluva aika ei vaikuta tuloksiin. Näiden lisäksi tuotemallista muodostetaan XML-esitys ennen testiä ja se annetaan syötteenä XSLT-prosessorille jokaisella suorituskerralla. Funktiokielen alustus toimii samalla tavalla eli ohjelma jäsennetään ja siitä muodostetaan suoritettava puumalli, jota voidaan suorittaa toistuvasti. Suorituskykytestin aikana funktiokielen ohjelmaan annetaan syötteenä tarvittavat komponentit tai symbolit. Java-ohjelma kirjoitetaan siihen luokkaan, jossa suorituskykytestiä ajetaan ja se käyttää suorituksen aikana valmiiksi alustettua tuotemallia.

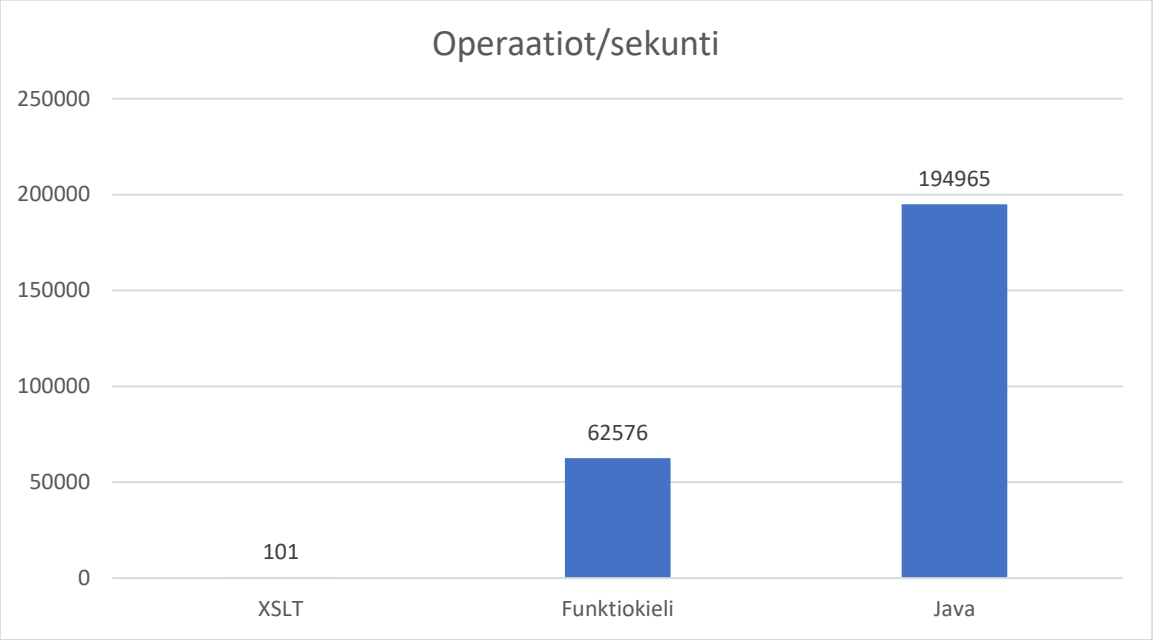
Suorituskykymittaukset suoritetaan pöytätietokoneella, jossa on Windows 10 (versio 1803) -käyttöjärjestelmä ja Intel i5-2500K (3.3GHz) -prosessori. Käytettävä Java-versio on OpenJDK 10.0.2.

7.1 Suorituskykytesti: Summa kahdella jaollisista luvuista

Tässä testissä mitataan iteroinnin ja silmukassa olevan ehdon suorituskykyä. Testissä käytetty tuotemalli kostuu yhdestä komponentista ja 100 komponentin instanssista, jossa jokaisessa instanssissa on yksi attribuutti, jonka arvo on sama kuin instanssin indeksi eli arvo on 1-100 väliltä. Suorituskykytestissä ajettava ohjelma laskee summan kaikista kahdella jaollisista attribuuttien arvoista. Testissä suoritettava funktiokielen ohjelma on seuraava:

```
sum(components as c where c.value % 2 == 0 select c.value)
```

Tulokset ilmoitetaan, kuinka monta kertaa sekunnissa ohjelmaa voidaan suorittaa eli montako operaatiota sekunnissa. Tämän testin tulosten mukaan funktiokielen ohjelma on huomattavasti nopeampi kuin vastaava XSLT-skripti, mutta kuitenkin hitaampi kuin Java-ohjelma (ks. kuva 13). Funktiokieli on 619 kertaa nopeampi kuin XLST-skripti ja Java-ohjelma on 3 kertaa nopeampi kuin funktiokieli. XSLT-ohjelmassa on selkeästi havaittavissa hitautta johtuen syötteen käsittelystä, joka sisältää kaiken tuotemallidatan. Funktiokielessä on myös havaittavissa hitautta verrattuna Java-ohjelmaan johtuen solmujen jatkuvasta läpikäymisestä suorituksen aikana.



Kuva 13. Suorituskykytulokset summa kahdella jaollisista luvuista -testissä.

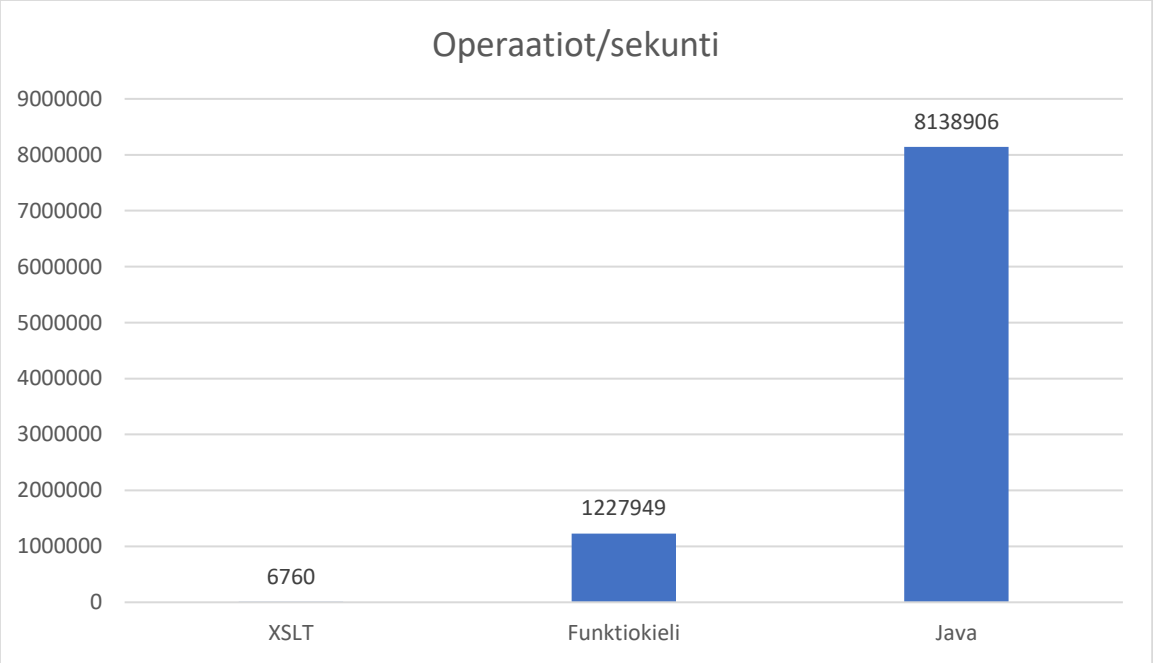
7.2 Suorituskykytesti: Toisen asteen yhtälö

Tässä testissä mitataan ohjelmien aritmeettisten operaatioiden suorituskykyä ilman tuotemallin aiheuttamaa kuormaa. XSLT-skriptin kannalta tämä testi pitäisi olla edullinen verrattuna edelliseen suorituskykytestiin syötteen puuttuessa. Kuten edellisessä testissä, XSLT-skripti ja funktiokielen ohjelma alustetaan ennen testiohjelmien suorittamista, jotta alustusajat eivät vaikuta tuloksiin. Testiohjelmassa lasketaan toisen asteen yhtälön ratkaisukaava kuvitteellisilla arvoilla, jossa käytetään sisäisiä muuttujia välivaiheiden laskentaan. Lopuksi ohjelma tulostaa merkkijonon, jossa muuttujien arvot ovat erotettu välilyönnillä. Funktiokielen testiohjelma on seuraava:

```
$x0 := (-3 + sqrt(3^2 - 4*(-4))) / 2
$x1 := (-3 - sqrt(3^2 - 4*(-4))) / 2
concat($x0, " ", $x1)
```

Suorituskykymittausten mukaan funktiokielen ohjelma on tässäkin testissä suorituskykyisempi kuin vastaava XSLT-skripti (ks. kuva 14). Funktiokieli on testissä

181 kertaa nopeampi kuin XSLT-skripti, kun edellisessä testissä kerroin oli 619. Tästä voidaan kuitenkin päätellä XSLT-skriptin eduksi, että sen suorituskyky on melko hyvä, jos XSLT-prosessorin ei tarvitse käsitellä syötteenä tulevaa XML-dataa ja saa suorittaa valmiiksi alustettua skriptiä. Funktiokielen ja Java-ohjelman ero on tässä testissä suurempi verrattuna edelliseen testiin, kun Java-ohjelma on noin 6 kertaa nopeampi kuin funktiokielen ohjelma. Funktiokielen hitaus johtuu myös tässä testissä solmujen läpikäymisestä, koska jokainen operaatio ja arvo on solmu sisällä ja vaatii metodikutsun solmua vastaavan operaation suorittamiseksi tai numeerisen arvon hakemiseen.



Kuva 14. Suorituskykytulokset toisen asteen yhtälö -testissä.

7.3 Suorituskykytesti: Alikysely

Tässä testissä mitataan alikyselyjen suorituskykyä, joka antaa paremman näkemyksen vertailukohteena olevien ohjelmointikielten iteroinnin tehokkuudesta tai heikkoudesta. Tämä testi on käytännössä sama kuin kappaleessa 7.1 poikkeuksena, että select-lausekkeessa suoritetaan toinen alikysely tai tiedon hakulause, jonka suoritus on

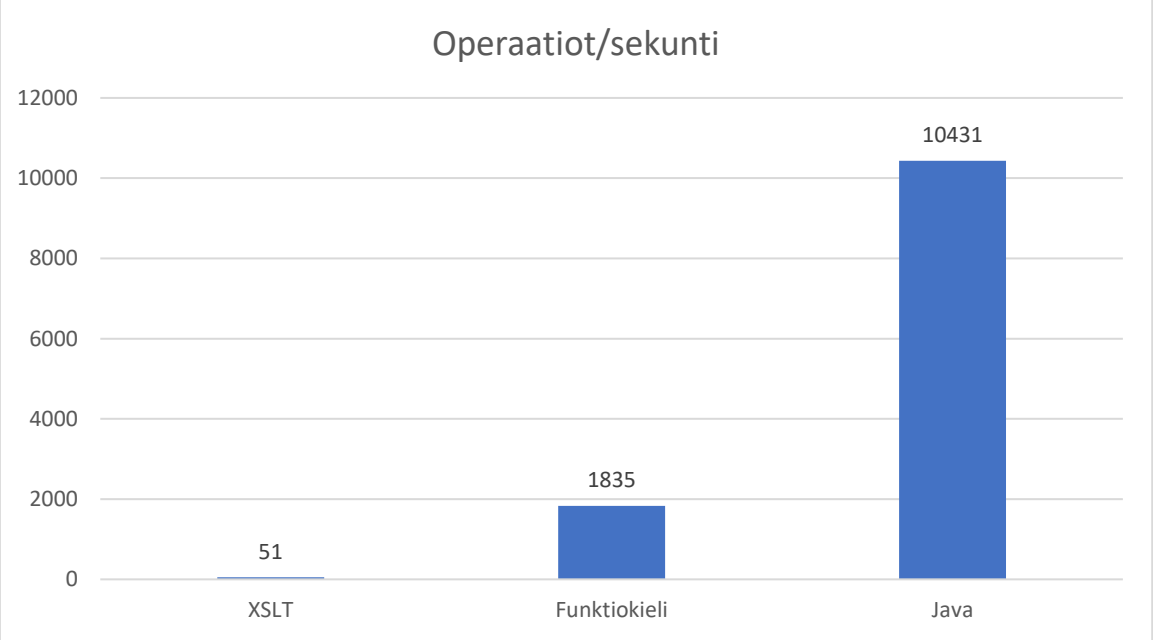
riippuvainen edeltävän hakulauseen tilasta. Funktiokielen kannalta tämä testi on hyödyllinen, koska testiohjelma käyttää symbolien näkyvyysalueita runsaammin verrattuna muihin testeihin. Näin ollen testiohjelman suoritusaikaan vaikuttaa oleellisesti näkyvyysalueiden luontiin ja poistamiseen kuluva aika. Uusi näkyvyysalue tai lohko syntyy funktiokielessä, kun hakulause suoritetaan ja poistuu kun suoritus on päättynyt.

Testiohjelma on kuvitteellinen ja sen tavoitteena on vain suorittaa toinen alikysely, vaikka testissä käytetty tietojoukko on sama molemmissa kyselyissä. Tässäkin testissä Java ja XSLT -ohjelmat ovat kirjoitettu vastaamaan toiminnallisuudeltaan funktiokielen ohjelmaa. Funktiokielen testiohjelma on seuraava:

```
sum(components as c1 where c1.value % 2 == 0 select
(components as c2 where c2.value == c1.value select c2.price
limit 1))
```

Suorituskykymittauksista saatujen tulosten mukaan tässä testissä funktiokielen ohjelman suorituskyky on parempi kuin XSLT-ohjelman suorituskyky (ks. kuva 15). Ero XSLT-ohjelman ja funktiokielen suorituskyvyssä on kuitenkin huomattavasti pienempi verrattuna kappaleessa 7.1 esitettyihin tuloksiin (ks. kuva 13), jossa funktiokielen ohjelma oli 619 kertaa nopeampi, mutta tässä testissä 35 kertaa nopeampi. Kappaleen 7.1 testiohjelma on lähes sama kuin tässä testissä, mutta erona on vain ylimääräinen alikysely ja muutama operaatio. Mielenkiintoinen havainto on, että Javan ja funktiokielen ohjelmien suorituskyky putosi suhteessa enemmän XSLT-ohjelmaan verrattuna, kun tarkastellaan kappaleen 7.1 tuloksia. XSLT-ohjelma on vain 2 kertaa hitaampi, funktiokielen ohjelma 34 kertaa hitaampi ja Java-ohjelma noin 19 kertaa hitaampi.

Tuloksista voidaan päätellä, että funktiokielen suorituskyky alikyselyissä on melko hyvä, mutta suorituskyky on kuitenkin huomattavasti parempi ilman alikyselyjä. Näkyvyysalueiden luomisessa ja poistamisessa on negatiivinen vaikutus suorituskykyyn. Suorituskykyyn vaikuttaa negatiivisesti myös alikyselyistä aiheutuvat uudet solmut, joita tulkin täytyy suorittaa jokaisella iteraatiolla.



Kuva 15. Suorituskykytulokset alikyselytestissä.

8 JOHTOPÄÄTÖKSET

Tutkimuksen tarkoituksena oli kehittää suorituskykyinen ja helppokäyttöinen täsmäkieli kaupalliseen tuotekonfiguraattoriin parantamaan asiantuntijoiden tuotemallinnustehokkuutta ja tuotekonfiguraattorin suorituskykyä. Tutkimuskysymyksiin vastattiin toteutuksen ja testitulosten perusteella eli kuinka täsmäkieli kehitetään ja otetaan käyttöön kaupallisessa tuotekonfiguraattorissa ja onko täsmäkielen suorituskyky parempi verrattuna aikaisemmin käytettyyn XSLT-kieleen.

Tutkimuksessa onnistuttiin toteuttamaan suorituskykyinen ja helppokäyttöinen ohjelmointikieli, joka on suunniteltu asiantuntijoille, joilla ei ole kokemusta ohjelmoinnista tai taulukkolaskentaohjelmista. Täsmäkieli pyrittiin kuitenkin suunnittelemaan niin, että se muistuttaa taulukkolaskentaohjelmissa käytettyjä makroja tai kaavoja, jotta täsmäkieli olisi helpompi omaksua niille, jotka ovat ennen käyttäneet kyseisiä ohjelmistoja.

Täsmäkielen tulkki toteutettiin Java-kielellä ja tulkin rakenteeksi valittiin abstrakti syntaksipuu (AST, Abstract Syntax Tree) -tulkkaaja, koska se oli yksinkertaisempi ja nopeampi toteuttaa. Täsmäkielen syntaksi määriteltiin BNF-notaatiota käyttäen ja sen pohjalta luotiin täsmäkielen jäsentäjä hyödyntäen kolmannen osapuolen ANTLR-kirjastoa.

Tulosten mukaan täsmäkielen suorituskyky on parempi kuin XSLT-kieli ja sen prosessori. Täsmäkieli on nopeampi verrattuna XSLT-kieleen, koska täsmäkielen syntaksi on yksinkertainen ja puolestaan tulkki on rakenteeltaan kevyt ja suoraviivainen. Näiden ominaisuuksien ansiosta täsmäkielen suorittamiseen kuluu vähemmän aikaa verrattuna XSLT-kielen suorittamiseen.

Täsmäkielen suorituskyvyssä havaittiin negatiivisia muutoksia, kun testattavan ohjelman kokoa ja lohkojen määrää kasvatettiin kappaleessa 7.3. Täsmäkielen suorituskykyyn vaikuttaa oleellisesti ohjelman koko ja lohkojen määrä, erityisesti sisäkkäiset lohkot

kuten sisäkkäiset hakulauseet tai alikyselyt. Monimutkaisempi ja suurempi ohjelma tarkoittaa, että suoritettava puumalli on rakenteeltaan syvä ja leveä. Tästä johtuen tulkille aiheutuu lisäkuormaa, koska se joutuu suorittamaan useampia solmuja. Tämän lisäksi solmun suoritusaikaan vaikuttaa Java-kielen metodikutsusta aiheutuva viive sekä solmun sisäinen toteutus. Tässä työssä toteutettu tulkki on yksinkertainen eikä se sisällä kehittyneitä optimointimenetelmiä suorituskyvyn parantamiseen suorituksen aikana. Tulkin suorituskykyä voidaan mahdollisesti parantaa muuntamalla suoritettavan puumallin rakennetta suorituksen aikana, kuten poistamalla tai erikoistamalla solmuja.

Täsmäkieli otetaan käyttöön kaupallisessa tuotekonfiguraattorissa, koska sen suorituskyky ja käytettävyys koetaan hyväksi. XSLT-skriptit tulevat olemaan toistaiseksi käytössä täsmäkielen lisäksi, jotta vanhojen tuotemallien suorittaminen tuotekonfiguraattorin uudessa versiossa toimisi edelleen. Uudet tuotemallit tulevat käyttämään tässä työssä kehitettyä täsmäkieltä laskutoimituksissa ja algoritmeissa.

9 JATKOTUTKIMUS

Kappaleessa 7 huomattiin, että täsmäkielen tulkin suorituskyvyssä on edelleen parannettavaa, vaikka suorituskyky on parempi verrattuna XSLT-kieleen. Jatkotutkimuksen kannalta olisi mielenkiintoista selvittää, onko mahdollista parantaa tulkin suorituskykyä, että se saavuttaisi Java-kielen tehokkuuden, vaikka tulkin isäntäkielenä on Java ja ajoympäristönä Java-virtuaalikone. Tämä voi olla haastavaa, koska tulkin toteutuksessa joudutaan käyttämään Java-kieltä eikä Java-kielellä kirjoitettu tulkki voisi tuottaa tehokkaampaa lopputulosta kuin Java-kielellä kirjoitettu täsmäratkaisu. Abstrakti syntaksipuu -tulkki on todennäköisesti aina hitaampi kuin vastaava isäntäkielen täsmäratkaisu, vaikka tulkissa olisi hyödynnetty suorituksen aikaisia optimointitekniikoita, kuten puun uudelleen kirjoittamista. Java-kielellä kirjoitettu täsmäratkaisu käännetään sellaisenaan optimaaliseksi tavukoodiksi, jota Java-virtuaalikone voi suorittaa välittömästi ilman ylimääräisiä suoritusvaiheita toisin kuin täsmäkielen tulkista tuotettua tavukoodia. Tulkin tavukoodi Java-virtuaalikoneessa sisältää täsmäkielen ja sen ohjelman kannalta epäsuotuisia vaiheita, jotka liittyvät suoritettavan puumallin rakenteeseen ja solmujen sisäisiin toteutuksiin.

Tavoitteena olisi, että täsmäkieli ja Java-kieli olisivat tavukoodina samanarvoiset Java-virtuaalikoneessa, jolloin täsmäkielen ja Java-kielen suoritusajat olisivat teoriassa samaa luokkaa. Eräs ratkaisu olisi toteuttaa erillinen kääntäjä täsmäkielen BNF-notaation pohjalta, joka tuottaisi jäsennyspuusta eli täsmäkielen ohjelmasta erikoistettua Java-virtuaalikoneen tavukoodia ja ohittaisi tulkin kokonaan. Tämä olisi kuitenkin kestämätön ratkaisu, koska optimaalisen tavukoodin tuottaminen manuaalisesti vaatii syvää tuntemusta Java-virtuaalikoneen tavukoodista ja dynaamisessa tuotekonfiguraattorissa ajon aikainen käännös ei ole suorituksen kannalta edullinen toimenpide.

Kappaleessa 2.1 esitelty Graal ja Truffle voisivat olla yhdessä mahdollinen ratkaisuvaihtoehto. Truffle-sovelluskehys vaatii toimiakseen tulkin, joka on kirjoitettu Java-kielellä. Tulkki voisi hyödyntää Truffle-sovelluskehyksen tarjoamia komponentteja abstraktin syntaksipuun rakenteissa suhteellisen pienillä muutoksilla. Graal-kääntäjä

kykenee tuottamaan Truffle-pohjaisesta tulkista suoritettavan ohjelman välitöntä tavukoodia Java-virtuaalikoneeseen ajon aikana. Käytännössä täsmäkielen ohjelman tavukoodi voidaan saada vastaamaan Java-kielellä kirjoitettua täsmäratkaisua. Sama Graal-kääntäjä tuottaa tavukoodia myös tavallisista Java-ohjelmista, jos Java-virtuaalikoneen C2-kääntäjä on korvattu Graal-kääntäjällä. Java-koodi ja täsmäkielen tulkki voisivat toimia samassa ympäristössä yhtä tehokkaasti.

LÄHDELUETTELO

Aho, A. V., R. Sethi & J. D. Ullman (1985). *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA.

Alexander, C., S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King & S. Angel (1977). *A Pattern Language*. Oxford University Press, Oxford.

ANTLR (2018). ANTLR [online]. [23.12.2018]. Saatavissa: https://www.antlr.org/.

Aram, M. & G. Neumann (2015). Multilayered analysis of co-development of business information systems. *Journal of Internet Services and Applications* 6 (1).

Axling, T. & S. Haridi (1996). A Tool for Developing Interactive Configuration Applications. *J. Log. Program.* 26, 2, 147–168.

Backus, J. W. (1959). *The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference*. Proceedings of the International Conference on Information Processing. UNESCO. s. 125–132.

Berners-Lee, T. & D. Connolly (1995). *RFC 1866: Hypertext Markup Language ± 2.0*.

Blecker, T., N. Abdelkafi, G. Kreuter & G. Friedrich (2004). Product Configuration Systems: State-of-the-Art, Conceptualization and Extensions. *Génie logiciel & Intelligence artificielle,* 25-36.

Bourke, R. W. (2000). *Product Configurators: Key Enablers for Mass Customization*. Midrange Enterprise. Bourke Consulting.

Brooks Jr., F. P. (1996). Language design as design. *In History of Programming Languages II. ACM Press,* 4–15.

Cacciagrano, D.R. & R. Culmone (2018). IRON: Reliable domain specific language for programming IoT devices. *Internet of Things* (2018).

Chamberlin, D. D. & R. F. Boyce (1974). SEQUEL: A Structured English Query Language. *ACM SIGMOD Workshop on Data Description, Access and Control*, 249-264. Ann Arbor, Michigan, USA.

Fowler, M. & R. Parsons (2010). *Domain Specific Languages*. 1. painos. Addison-Wesley Professional. ISBN 978-0321712943.

Freudenthal, M. (2010). Using DSLs for developing enterprise systems. *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*, 11: 7. ACM, New York, NY, USA.

Gamma, E., R. Helm, R. Johnson & K. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA.

Jones, C. (1996). *SPR Programming Languages Table Release 8.2* [online]. [7.3.2018]. Saatavissa: http://www.cs.bsu.edu/homepages/dmz/cs697/langtbl.htm.

Jones, C. & O. Bonsignour (2012). *The Economics of Software Quality*. Addison-Wesley, 2012.

Klint, P., T. Storm & V. Jurgen (2010). On the impact of DSL tools on the maintainability of language implementations. *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*, 10: 9. ACM, New York, NY, USA.

Koskimies, K. & T. Mikkonen (2005). *Ohjelmistoarkkitehtuurit*. Jyväskylä: Talentum. ISBN 952-14-0862-6.

Kübler, A., C. Zengler, & W. Küchlin (2010). Model Counting in Product Configuration. *In Workshop on Logics for Component Configuration,* 44–53.

Heiskanen, J. (2012). Sales configurators as means to enhance sales-to-delivery processes of system products. Master's thesis, Aalto University, Industrial Engineering and Management, Espoo, Finland.

Mernik, M., J. Heering, & A. M. Sloane (2005). When and how to develop domain-specific languages. *ACM Comput. Surv.* 37, 4 (December 2005), 316-344.

McDermott, J. (1982). R1, a Rule-based Configurer of Computer Systems. *Artificial Intelligence* 19, 39-88.

McDermott, J. (1993). R1 ("XCON") at age 12: Lessons from an elementary school achiever. *Artificial Intelligence* 59, 1–2 (1993), 241 – 247.

Naur, P. (1961). A Course of Algol 60 Programming. *ALGOL Bull. Sup* 9, 1-38.

Parr, T. J. & R. W. Quong (1995). ANTLR: A Predicated-LL(k) Parser Generator. *Software: Practice and Experience* 25: 7, 789–810.

Sarinko, K. (1999). *Mass Customization, Configuration and Modularization of Customer Specific Products*. Helsingin teknillinen korkeakoulu. Tuotantotalouden laitos. Diplomityö.

Skjevdal, R. & E. A. Idsoe (2005). The Competitive Impact of Product Configurators in Mass Tailoring and Mass Customization Companies. *In World Conference on Mass Customization, Personalization & CoCreation*.

Soininen, T. (2000). *An Approach to Knowledge Representation and Reasoning for Product Configuration Tasks*. Acta Polytechnica Scandinavica: Mathematics and computing series. Finnish Academies of Technology. ISBN 978-951-66656-0-6.

Spinellis, D. (2001). Notable design patterns for domain-specific languages. *The Journal of Systems and Software* 56, 91-99.

Tennent, R. D. (1977). Language design methods based on semantic principles. *Acta Inf.* 8, 97–112.

Tiihonen, J. & T. Soininen (1997). *Product Configurators: Information System Support for Configurable Products*. Helsinki University of Technology. ISBN 978-951-22386-5-1.

W3C (2018a). *Transformation* [online]. [6.6.2018] Saatavissa: https://www.w3.org/standards/xml/transformation.html

W3C (2018b). *XSL Transformation* [online]. [6.6.2018] Saatavissa: https://www.w3.org/TR/xslt-10/#section-Introduction

Wile, D. S. (2004). Lessons learned from real DSL experiments. *Sci. Comput. Program.* 51, 265– 290.

Wirth, N. (1974). On the design of programming languages. In: Information Processing 74: Proceedings of IFIP Congress 74, International Federation for Information Processing. North-Holland, Stockholm, pp. 386-393.

Wimmer, C. & T. Würthinger (2012). Truffle: A Self-Optimizing Runtime System. *SPLASH* '12, 13-14.

Würthinger, T., A. Wöß, L. Stadler, G. Duboscq, D. Simon & C. Wimmer (2012). Self-optimizing AST interpreters. *SIGPLAN Not.* 48: 2, 73-82.