Lähdekoodin symbolinen analysointi tekoälyn näkökulmasta

Erkki Laitila

Department of Mathematical Information Technology, University of Jyväskylä
Oy P.O. Box 35, FIN-40014 Jyväskylä, Finland

SwMaster Oy
Sääksmäentie 14, 40520 Jyväskylä

erkki.laitila@swmaster.fi

Tiivistelmä

Tämä artikkeli käsittelee lähdekoodin automaattisen käsittelyn menetelmiä ja tuo esille uutena konstruktiona symbolisen analyysin, joka on tunnettuja menetelmiä korkeatasoisempi, koska siinä tulee esille semioottinen, tulkitseva näkökulma. Esitämme sen tärkeimmät toiminnot ja piirteet tekoälyn näkökulmasta. Artikkelissa kuvataan analysointiprosessi ja sen hyödyntämiseen soveltuva symbolinen tiedonhakumenetelmä, jonka avulla käyttäjä voi saada symboliseen malliin perustuen tarvitsemaansa pragmaattista informaatiota tarkoituksena helpottaa nykyisen Java-kielisen ohjelmaversion ymmärtämistä haluttaessa kehittää uusi versio. Ratkaisu on ohjelmoitu ja koeajettu Visual Prolog-kehittimellä, joka hybridikielenä ja symbolista ohjelmointia tukien tarjoaa erinomaiset menetelmät vastaavan työkalun toteutukseen.

1 Johdanto

Tuskin mikään ohjelmistotoimitus valmistuu jo ensimmäisellä kerralla lopulliseen muotoonsa. Pikemminkin voidaan sanoa, että ohjelmistojen kehittäminen on lähes poikkeuksetta jatkuvaa tieto- ja informaatiovirtaa, siis prosessi, joka sisältää lukuisia vaiheita ja toimituksia. Kaikissa tilanteissa uudelleenkäytettävän koodin määrä tulisi, luotettavuus– ja kustannussyistä, olla mahdollisimman suuri. Lyhimmillään toimitussykli on päivittäistä koostamista sovellettaessa päivän luokkaa. Tietoliikennealalla toimitusten väli saattaa olla kireimmillään vain muutamia viikkoja, missä ajassa kaikki mahdollinen hyöty entisestä versioista olisi pystyttävä siirtämään uuteen versioon. Olemassa olevan koodin hyödyntäminen on siten merkittävä haaste. Sitä kuvataan seuraavassa metodologian näkökulmasta.

1.1 Takaisinmallintaminen

Takaisinmallintaminen (software reverse engineering) tarkoittaa olemassaolevan koodin hyödyntämistä. Erityisesti uudelleenmallintaminen (reengineering) tähtää saadun informaation käyttöön uudessa ohjelmaversiossa.

Selvästi todettu ongelma ohjelmistotyössä on, että ohjelmistoprosessi on luonteeltaan vain yksisuuntainen, sillä sen tuotoksesta, kuten työn laadusta, määrästä, monimutkaisuudesta ja lopullisen koodin käyttäytymisestä on ollut vaikeaa saada palautetta. Koska palaute on puuttunut, ei prosessia ole voitu optimoida samalla tapaa kuin miten teollisen tuotannon prosessit on aikanaan saatu hallintaan. Juuri ohjelmistoprosessien optimoitavuus on todettu korkeimmaksi alan kehityksen tasoksi, mutta vain aniharvat yritykset ovat päässeet korkeimmalle CMMI-tasolle (CMMI). Ilmeistä kuitenkin on, että jos ohjelmistoprosessista saataisiin yhtä hyvä ja ajanmukainen palaute kuin parhaimmissa tuotantoprosesseissa asianmukaisella mittaustekniikalla saadaan, myös ohjelmisto-organisaatiot saisivat merkittävän parannuksen toiminnan laatuunsa, millä olisi positiivisia vaikutuksia luotettavuuteen, läpimenoaikoihin, töiden delegoitavuuteen ja ennenkaikkea mahdollisuuteen saada uutta ydintietämystä jatkokehitykseen.

Tässä artikkelissa kuvattu symbolinen menetelmä tähtää juuri takaisinmallintamispalautteen parantamiseen.

1.2 Semioottinen näkökulma ohjelmistotyöhön

Semiotiikka tarkoittaa merkitysten ja kommunikaation tutkimusta. Se on tämä artikkelin kannalta kiinnostava aihe, koska ohjelmoijahan on aikanaan jättänyt koodiin jälkensä, joiden vaikutuksia ja tarkoituksia ylläpitäjä haluaa tutkia jatkokehityksen kannalta.

Peircen tulkintojen mukaan (1998) semiotiikka jakaantuu kolmeen pääkäsitteeseen, joita ovat ovat merkki (*sign*), kohde eli objekti ja tulkinta (*interpretant*). Merkki viittaa vastaavaan objektiin ja tulkinta yhdistää niitä toisiinsa. Esimerkiksi nähdessään jäniksen jäljet, ihminen tekee tulkinnan, joka yhdistää jäljet oletettuu jänikseen, jolloin jänis on objekti ja jäljet ovat vastaava

merkki. Toisin kuin merkit, tulkinnat ovat aina sisäisiä esityksiä. Merkit voidaan jakaa osoittimiin (indice, index), ikoneihin ja symboleihin riippuen siitä minkälaisesta objektista on kysymys. Esimerkiksi kirja on tietämyksen osoitus (tunnusluku, indeksi), samoin savu on osoitus tulesta.

Vieläpä vaikka kausaalista yhteyttä tai mitään attribuutteja olioon ei olisikaan, silti voi löytyä jokin indikaatio (merkki) luomaan yhteyksiä objektiinsa. Symbolin käyttö soveltuu sellaiseen tarkoitukseen, ja juuri tällaista epäsuoraa kytkentää käyttää hyväkseen tässä artikkelissa kuvattu symbolinen menetelmä. Koska kaikki ohjelman sisäiset artifaktat ovat näkymättömiä, tulee analysointiohjelmiston muodostaa riittävän tarkka ja yleinen virtuaalimalli viittauksineen ja symboleineen käyttäjän näkösälle kuvaruudulle. Näytön koko on kuitenkin rajattu ja työkalun on siten mahdotonta tuottaa kerralla riittävää informaatiota yhtenä kokonaisuutena. Siksi analysoinnin tulee olla interaktiivista navigointia, joka mahdollistaa uusia tiedonhakuja eri näkökohdista.

1.2.1 Tietojärjestelmien semiotiikka

Morris (1971) jakaa semiotiikan kolmeen dimensioon, joita ovat syntaksi, semantiikka ja pragmaattinen tarkastelu, jolloin pragmaattinen osuus viittaa Peircen semioottiseen tulkintaan. Seuraavassa kuvataan, kuinka Frisco (1998) on määritellyt nämä kolme dimensiota tietojärjestelmien kehittämisen suhteen.

Frisco luokittelee informaatiojärjestelmän semioottiset tasot seuraavasti. Järjestelmän rakentaminen aloitetaan fyysisen maailman tasolta, joka sisältää valtaisan määrän signaaleja, merkityksiä ja luonnonlakeja. Järjestelmän suunnittelu tehdään empiirisen kokemuksen avulla luoden kokonaisuudesta malleja ja uusia käsityksiä. Tulos ohjelmoidaan, jolloin saadaan syntaksia, käytännössä kiinteä ohjelmisto tiedostoineen, kääntäjineen ja toimintaympäristöineen. Kun syntaksia ja koodia tulkitaan semantiikan kautta, saadaan tietoa koodin rakenteiden merkityksistä, väittämistä ja viittauksista ulkomaailmaan. Silloin kun järjestelmää tutkitaan nimenomaan eri roolien tarpeista lähtien, tavoitetellaan pragmaattista tietoa. Sillä on Friscon mukaan selvä yhteys järjestelmän tuottamaan hyötyyn. Kokonaisvaikutuksen tulee näkyä tietojärjestelmää ympäröivässä sosiaalisessa maailmassa, jonka tarpeisiin tietojärjestelmän tulisi tuottaa lisäarvoa.

Friscon määrittelykonseptia on kritisoitu esimerkiksi siitä, että se pyrkii formalisoimaan lähes epäformaaleja asioita, mutta takaisinmallintamisen kannalta tätä edellä kuvattua ketjua voidaan pitää kelvollisena lähtökohtana, koska takaisinmallintamisessa keskitytään nimenomaan formaalin koodin hyödyntämiseen.

1.3 Koodin analysoinnin teknologiat

Lähdekoodin analysoiminen liittyy ohjelmistojen takaisinmallintamisen (software reverse engineering) alueeseen, jonka tarkoituksena on juuri helpottaa uusien ohjelmaversioiden tuottamista. Koodin analysoinnissa periaatevaihtoehtoina on tutkia koodia käsin perinteisillä editoreilla ja kehitystyökaluilla tai käyttää parhaiten koodimassan käsittelyyn soveltuvia erikoismenetelmiä, joita on varsin vähän.

Kielioppien taustateoriat on tunnettu 50-luvulta asti ja koodin jäsentämisen metodologiat kehitettiin 70-luvulla Unixin mukana. Ylläpidon kannalta haitallista on, että analysointimenetelmät ovat kehittyneet kääntäjäteknologian ja koodin testauksen menetelmien sivutuotteina siten, että suurtakaan huomioita ei ole kiinnitetty juuri analysointituloksen hyödynnettävyyteen koko prosessin kannalta, informaatiojärjestelmän kokonaishyödystä puhumattakaan.

1.3.1 Abstract Syntax Tree (AST)

Vallitseva käytäntö koodin käsittelystä analysointiohjelmistoissa perustuu Abstract Syntax Tree-teknologiaan (AST). Siinä työkaluun tallentuu kieliopin mukaisia abstrakteja rakenteita, jotka muodostavat keskenään tiiviin hierarkian. Vaikka AST on hyödyllinen kääntäjiä suunniteltaessa, se ei kuitenkaan anna mahdollisuuksia syvälliseen tulkintaan, koska kielen semantiikka ei kokonaisuudessaan sisälly kielioppiin. Esimerkiksi muuttujaviittausten ja metodiviittausten logiikka ja oliosidosten toiminta sekä rakenteiden välinen semantiikka puuttuvat AST-määrittelystä.

1.3.2 UML ja round-trip-engineering

Käytännön tasolla oliopohjaisen koodin analysoinnissa on viime aikoina selvästi yleistynyt *round trip engineering* - niminen menetelmä, joka auttaa luokkarakenteiden poimimista koodista. Valitettavasti näin syntyvien UML-notaatioiden tarkkuus ei riitä pitkälle. Lisäksi eri työkalut saattavat tuottaa erilaisia kaavioita samalle lähdekoodille.

Toisaalta UML:n eri esitystavat eivät skaalaudu hyvin riittävän isojen näyttöjen luomiseen, minkä takia käyttäjä joutuu monesti tutkimaan lukuisia erilaisia kaavioita voidakseen tehdä johtopäätöksiä nykyisen koodin toiminnasta. Toistuva erillisten kaavioiden lukeminen on työlästä ja se kuormittaa käyttäjän lyhytkestoista muistia merkittävästi, jonka takia varsinainen ongelma saattaa jäädä ratkaisematta.

Olio-ohjelmoinnin kehittäjäjärjestö OMG keskittyy pääasiallisesti uusien sovellusten kehittämiseen ja sitä kautta uuden koodin luomiseen. Ehkä sen takia heidän

mallinsa eivät tarjoa tarpeeksi tarkkoja esityksiä, jotka mahdollistaisivat koodin tarkimpien lauseiden tallentamisen tai suorittamisen, mikä olisi tarkan analyysin perusedellytys. Siten käsitteellä executable UML ei tarkoiteta vastaavan koodin suorittamista vaan lähinnä vastaavan mallin tietojen evaluointia nimenomaisen mallin kannalta, ei koodin kannalta.

1.3.3 Koodin ymmärtämisen peruskäsitteet

Kuten edellä todettiin, takaisinmallintamisen luoma palaute on arvokasta kehittäjäorganisaatiolleen, ja että syntaksi tarjoaa erinomaisen pohjan uudelle automaatiolle, koska käsiteltävä koodin informaatio on luonteeltaan formaalia. Esille tulee kuitenkin kysymys, mikä osa koodista on merkityksellistä ja arvokasta? Vas onko koodi vain tasapaksua massaa, jonka lukeminen siltikin onnistuu parhaiten editorilla?

Ohjelman ymmärtämisen teorioita on kehitetty 80-luvulta lähtien, jolloin Pennigton (1987) kehitti tarkastelun alhaaltapäin alkavan tarkastelun viisikkomallinsa (function, control flow, data flow, state & operation) ja Brooks (1983) kehitti yleiset teoriat tarkastelun hypoteeseille. 90-luvulla von Mayrhauser (1992) loi integroidun metamallin, johon liitettiin osana ylhäältäpäin alkava osittava tarkastelutapa sekä tilannemalli, joka on kumulatiivista ohjelmasta saatua informaatiota toimialansa tietoon yhdistettynä.

Wiedenbeck (2002) on edelleen laajentanut aiempia teorioita 2000-luvulla kattamaan olio-ohjelmien ymmärtämisen tarpeet. On esitetty myös abstraktimpia käsitteitä kuten suunnannäyttäjä (beacon) ja lohko (chunk) kuvaamaan koodin ymmärtämistä esimerkiksi lajittelualgoritmien osalta, mutta niiden merkitys oliomaisen koodin tarkastelussa ei ole enää kovin merkittävä, koska oliot ja metodit rajaavat tehokkaasti toimintoja sisäänsä muodostaen uuden helpottavan abstraktiotason.

Sen sijaan uutena asiana olio-ohjelmointi on tuonut suunnittelumallit ja kerrosarkkitehtuurit uusiksi tarkastelutavoiksi. Niiden jäljittäminen koodista on paljon tutkittu alue ja koodin rakenteen uudelleen suunnittelu (*refactoring*) on siihen soveltuva konkreettinen käyttökohde.

1.4 Tekoälyn ja koodin analysoinnin suhde

Tekoälyyn liittyvät keskeisesti tietämyksen hankinta, palautus ja käsittely, ihmisen ja koneen välinen vuorovaikutus, päättelymekanismit ja älykkäät käyttöliittymät. Lähdekoodin tutkimuksen alueelta voidaan siihen liittyen määritellä, että ohjelman ja sen toiminnan ymmärtäminen on oppimisprosessi, jossa lukija pyrkii kehittämään tietämystään tietojärjestelmän nykyarvon kasvattamiseksi. Ohjelma sisältää paljon sumeaa tietoa, jota ohjelmoijat ovat tallentaneet koodiin omalta kokemustaustaltaan optimoiden. Sen sisältö on usein osittain virheellistä. Tietoa on jopa miljoonia rivejä, joten sitä on mahdotonta esittää näytössä kerrallaan, tarvitaan siis erilaisia haku- ja suodatusmekanismeja.

Parhaille formaalien kielten analysoinnin sovelluksille on ominaista, että koodin tieto saadaan poimittua tarkkaan semantiikan mukaan. Penningtonin viisikkomalli muodostaa tämän tiedon perustan. Se määrittää mikä on kiinnostavaa dataa. Käyttäjä on liikkeellä yleensä joko perehtymistarkoituksessa tai paikantamistarkoituksessa. Molemmissa tapauksissa hän haluaa evaluoida kriittisimpiä koodiosuuksia apukysymysten ja hypoteesien kautta. Yhteinen nimittäjä kaikille näille tarpeille on koodin ymmärtämisen tehostaminen työkalun avulla (program comprehension, PC).

1.5 Tutkimuksen rajaus

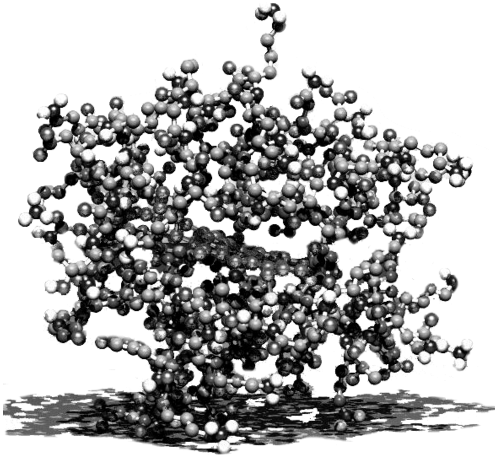
Erilaisia ohjelmointikieliä tunnetaan suuri joukko. Painopiste teollisuudessa on siirtynyt oliopohjaisiin kieliin, mutta silti alan tutkimus huomattavan suurelta osin käsittelee proseduraalisia kieliä kuten C, Pascal tai Cobol. Oliopohjaisista pääkielistä C++ on laajin, monimutkaisin ja ehkä jopa tärkein, mutta sen analysointiin on varsin rajatusti menetelmiä. Sen sijaan Java-kieltä ja sen käsittelyä on tutkittu paljon, koska se on kielenä selväpiirteinen.

Tässä artikkelissa käsittelemme etupäässä Javaa, koska sen selväpiirteisen kieliopin ansiosta on mahdollista päästä nopeasti hyvin tarkkaan käsittelytulokseen. Lisäksi edistysaskeleiden osoittaminen paljon tutkitulle Javalle voi olla selvä osoitus uuden tutkimuslähestymisen paremmuudesta perinteisempiin tutkimuslähestymisiin nähden.

Symbolinen analyysi on SwMaster Oyn toimesta käynnistynyt innovaatio, jota Erkki Laitila tutkii väitöstutkimuksessaan Jyväskylän Yliopistossa. Sen tekninen osuus keskittyy koodin symbolien ja niihin liittyvän logiikan kuvaamiseen mahdollisimman hyvin Visual Prolog- kehittimellä, joka on hybridikieli sisältäen erinomaisen oliojärjestelmän mallintamistarkoituksiin ja logiikkaohjelmoinnin perustan koodin logiikan, assosiaatioiden ja riippuuvuksien käsittelyyn.

Symbolinen koodin malli on uusi atomistinen konstruktio, joka mahdollistaa koodin tutkimisen kaikista lähtökohdista, jotka liittyvät Java-koodin semantiikkaan ja simuloidun koodin käyttäytymiseen ja ajovaikutuksiin. Kuva 1 (seuraava sivu) esittää molekyylääristä rakennetta sidoksineen. Se muistuttaa ohjelmakoodia, koska molemmissa on sidoksia. Tärkein asia ohjelmistoja hahmotettaessa on ymmärtää kuinka eri lähestymistavat kuten käyttötapaukset, ohjausvuo ja ohjelmasuunnitelmat liittyvät toisiinsa. Olio-ohjelmointi aiheuttaa omia periaatteellisia ongelmiaan, koska vastaavat kutsut sijoittuvat näkymättömiin eri metodien sisälle koodiin. Syntyviä olioita ei voida myöskään tarkkailla millään

analysoinnin perusmenetelmillä. Siten näin havainnollista kuvaa ei tänä päivänä ole saatavissa, mutta mahdollisuuksia on, mikäli tässä tutkimuksessa kuvattu teknologia pääsee aikanaan käytäntöön asti.



Kuva 1. Symbolinen, atomistinen malli.

Voimakkaasti yksinkertaistaen voidaan sanoa, että koodin ymmärtäminen on siinä olevien symbolien ja rakenteiden välisten suhteiden ymmärtämistä. Nämä suhteet muodostavat keskenään symbolivuon, joka tuottaa hyödyllisintä informaatiota silloin kun se on jäsentyneenä ohjelman suoritusjärjestykseen. Siten kuvassa 1 ylinnä voisi olla *main*-metodi, joka haarautuisi alaspäin suorituslogiikan mukaisesti.

2 Analysointimenetelmät

Perinteisiä koodin analysointimenetelmiä ovat staattinen ja dynaaminen analyysi. Staattinen analyysi ei sovellu hyvin oliomaisille kielille kuten Javalle, koska vain osa toiminnoista on tulkittavissa suoraan koodista. Toisaalta dynaaminen analyysi vaatii lähes aina monimutkaisia ajojärjestelyjä.

Staattisella analyysillä voidaan tuottaa riippuvuuskaavioita ja kutsupuita, jotka toimivatkin hyvin proseduraalisille kielille jopa niin, että niiden avulla voidaan saada täsmällinen tieto mitä koodinosia mikäkin proseduuri käyttää ja missä järjestyksessä ohjelmavuo etenee, mikä on arvokasta tietoa mm. vianpaikannustilanteissa. Vaikeita haasteita staattiselle analyysille ovat osoittimet (pointer) ja pysähtymättömyyden analysointi.

Dynaaminen analyysi perustuu pitkälle ohjelman ajamiseen debuggerilla ja sitä kautta koodin toimintojen jäljittämiseen valittujen pysäytysehtojen, testitapausten ja jäljityskomentojen mukaisesti. Näin saadaan talteen jälki, trace, joka voi sisältää erittäin paljon aivan turhaakin tietoa. Jopa 98 % jäljestä voi olla epäkuranttia.

Edellä mainittujen menetelmien rajoitteet tuntien on aiheellista kysyä, voidaanko muodostaa sellainen uusi analysointitapa, jonka painopisteenä olisi koodi-informaation jalostaminen ja muokkaaminen sellaiseen muotoon, että syntyvästä ratkaisusta voitaisiin selektiivisesti kyselyin poimia haluttu informaatio? Voisiko syntynyt menetelmä esimerkiksi toimia kuten tietokanta, jolloin käyttäjä saisi vapaasti valita minkä asioiden välisiä suhteita hän haluaisi tarkastella suunnitellessaan muutoksiaan?

Mietittäessä vastauksia edellisiin kysymyksiin päädytään ideaalisen analyysin määritelmään: Millainen olisi ideaalinen analysointimenetelmä, jos sen voisi vapaasti määritellä puhtaalta pöydältä?

Vertauskohtia voimme tietysti löytää muiden tietojärjestelmien ja teknologioiden parhaista toteutuksista, joita ovat mm. relaatiomalli, informaatioteoria, diagnostiikan perustietous, asiantuntijajärjestelmien perustyypit. Näiden kautta voimme päätyä synteesinomaisesti seuraaviin vaatimuksiin:

- Analyysi voidaan tehdä milloin vain mistä tahansa koodin osasta.
- Analyyysi poimii käsittelyyn kaikki tarvittavat asiat kuten Penningtonin ja Wiedenbeckin määrittelemät tiedot.
- Tuotoksella tulee olla ideaalinen tiedonsiirtosuhde ilman turhaa tietoa.
- Vastaava analysointiprosessi tukee tiedonhankintaa, mahdollistaen uuden tietämyksen hankinnan.
- Vastaus staattiseen kysymykseen saadaan nopeasti. Silti ratkaisu vaativaan ongelmaan saisi kestää kauemminkin, jos vertailukohtana käytetään käsin tehtävää perushakua, mikäli haku vaatii runsaasti syvällistä laskentaa..
- Menetelmän tulee tukea käyttäjän pohdintaa syntaksin ja semantiikan suhteista, mieluummin myös pragmaattiselta kannalta katsottuna.
- Menetelmän tulisi voida tunnistaa arkkitehtuurin kaikki keskeiset piirteet.

Ohjelmakoodi sisältää erilaisia tasoja, kerroksia ja menetelmiä, joten käytännön tutkimuksessa on tehtävä rajauksia. Tässä tutkimuksessa keskitytään

a) ”kovoteoriaan”, joka tuottaa mahdollisimman ytimekkään, mutta ilmaisuvoimaisen mallin
b) ”pehmoteoriaan”, joka määrittelee ihmisen tyypillisen käyttäytymismallin ylläpito-ongelmia ratkottaessa.

Siten abstraktimmat käsitteet, jotka ovat etäällä kieliopin määrittelyistä, jätetään huomioimatta. Esimerkiksi kerrosrakenteen poimimista koodista ei tutkita eikä myöskään suunnittelumallien tunnistamista.

3 Koodin käsitehierarkia

Koodin tutkiminen muistuttaa monin tavoin tutkimusta, koska sen lukeminen vailla tarkoitusta ei ole mielekästä. Aina tarvitaan jokin tavoite, jolloin lukija joutuu miettimään tavoitteen ja koodin välistä suhdetta (von Mayrhauser 1992). Se onnistuu vain olettamuksia eli hypoteesejä tehden. Näin koodin tarkasteluun voidaan soveltaa tieteen peruskäsitteistöjä alla olevaan tapaan.

Ontologia kuvaa koodin olemuksen ja sen keskeisen rakenteen. Epistemologia tunnistaa koodista olennaista tietoa pyrkien määrittelemään sen tietämystä. Metodologia kuvaa tavan, jolla selektiivinen tiedonkeruu ja tietämyksen johtaminen siitä voi tapahtua. Lisäksi tarvitaan vielä menetelmä, joka soveltuu arkielämän tarpeisiin ja erityisesti työkalu, joka helpottaa käyttäjän toimia.

3.1 Peruskäsitteet

Seuraavassa esitetään lyhyesti tutkimuksen keskeiset termit, jotka selostetaan myöhemmin tarkemmin.

3.1.1 Sanasto ja lyhenteet

G kieliopin symboli
X koodin syntaksin rakenne kieliopissa G
N rakenteen X tyyppi (non-terminal)
Y X:n vaikutus koodissa, semantiikka
Z X:n merkitys lukijalle, pragmatiikka

B X:n käyttäytymismalli (input/output-tarkastelu)
S koodista saatu symboli
O symbolista S luotu olio
L logiikka, joka olioon O sisältyy (interpretant)
M malli, joka koostuu olioista
A analyysi, joka hyödyntää mallia M
Q kysely, jolla analyysi A käynnistyy
H hypoteesi, jonka perusteella kysely Q voidaan tehdä
P päättelyprosessi, josta käsin luodaan hypoteeseja H
T ylläpitotehtävä päättelyprosessin P aktivoimiseen
I pragmaattinen informaatio ylläpitoa tukemaan
K tietämys (knowledge), jota prosessissa P kehitetään

Symbolinen analyysimenetelmä toimii ideaalisesti, jos se pystyy yhdistämään edellä mainitut käsitteet ideaalisesti.

3.1.2 Määritelmiä

Informaatio on uutta, käyttäjäkohtaista merkittävää tietoa, jonka määrä on minimoitu tarpeeseensa.
Tietämys on sellaista tietoa, jota voidaan käyttää ratkaistaessa muita samantyyppisiä ongelmia.

Tietämyksen hankintaan liittyy siten oppimista ja sisäistämistä. Arvokkain tieto, jota metodologia voi tuottaa, on juuri tietämystä, taitoa ratkaista tulevia ongelmia yhä järjestyneemmin verrattuna tapauskohtaiseen käsittelyyn, jossa ratkaiseminen tapahtuu yhä uudelleen entiseen tapaan. Uuden tietämyksen avulla nykyistä menetelmää voidaan kehittää edelleen, jolloin päästään positiiviseen kierteeseen ja optimoimaan menetelmän parametreja sekä eliminoimaan virhemahdollisuuksia. Walenstein (1998) esittää, että näin tietämys muuttuu säännöiksi ja taidoiksi.

3.1.3 Ohjelmaesimerkki

Seuraavassa esitettyä lyhyttä esimerkkiä käytetään jatkossa havainnollistamaan symbolisen analyysin ideaa. Siinä oleva luokka Server käynnistää muutamaa lausetta käyttäen (4-10) Server-olion.

```
   // Start the server up, listening on an
   // optionally specified port
1    class Server extends Thread {
2     {
3     protected int port;

4a     public static void main(String[]
4b             args) {
5          int port = 0;
6          if (args.length == 1) {
7a              try { port =
7b              Integer.parseInt(args[0]); }
8a              catch (NumberFormatException
8b          e) { port = 0; }
9          }
10         new Server(port);
11    }
12   }
```

Javan koodi on selkeä. Se sisältää vain varattuja sanoja ja joko viittauksia Javan kirjastoon tai käyttäjän antamia symboleita viittauksineen. Varatut sanat rajaavat rakenteita (*nonterminal*), esimerkkeinä luokka Server ja sen sisältämä metodi main.

Javaa tuntematon lukija ei voi päätellä esimerkkikoodin (X) aiheuttamasta toiminnasta (B) paljoakaan, koska hän ei tunne kielen semantiikkaa (Y), riippuvuussääntöjä (N) eikä käsitteitä (G). Se sijaan kokenut Java-ohjelmoija ymmärtää yllä olevan toiminnan täydellisesti (Z) luettuaan jokaisen rivin huolellisesti.

3.2 Koodin ontologia

Formaalien kielten jäsentäminen ei ole ongelmallista, koska kielen määrittely on alkuvaiheissa suunniteltu säännönmukaiseksi. Seuraavassa kuvataan kielten tulkinnan mahdollisuuksia tietosisällöistä lähtien.

3.3 Kieliopin määritelmä

Kieliopin määritelmät ovat peräisin jo puolen vuosisadan takaa (Chomsky 1956). Seuraavassa formaalin kielen kieliopin määritelmä:

$G = \langle N, \sum, R, S0 \rangle$

Symboli N edellä tarkoittaa jakokelpoista termiä (non-terminal), jolla on vähintään kaksitasoinen hierarkinen rakenne. ∑ on kielen kaikkien varattujen sanojen luettelo sisältäen välimerkit ja avainsanat (terminal). R on joukko sääntöjä (production rule), jotka kertovat kaikki sallitut rakenteet, joita kukin N voi saada. S0 on kielen käännöksen aloitussymboli, jolla kuitenkaan ei ole analysoinnissa juuri merkitystä.

Tärkeimpiä rakenteita Javan osalta ovat luokan määrittely *ClassDeclaration*, metodi *MethodDeclaration*, lause *Statement*, lauseke *Expression* ja muuttuja *Identifier*.

Ontologian ja kapseloinnin takia luokan määrittely on ensiarvoisen tärkeä. Se on erittäin selkeä. Luokka sisältää määrittelynsä mukaan luokan jäseniä (metodi ja attribuutti), ja sillä on perintämäärittely. Yhdessä nämä ulkoiset määrittelyt muodostavat luokan allekirjoituksen, luokkasopimuksen.

Javan sanastossa erityisesti jokainen viittaus käsitteeseen *Identifier* viittaa joko JDK-kirjastoon tai käyttäjän antamaan symboliin, joka voi olla joko paketin, luokan, metodin, attribuutin tai muuttujan nimi. Kaikki muut kielen rakenteet tulee tunnistaa suhteessa *Identifier*-viittauksiin.

3.3.2 AST:n rajoitukset analysoinnissa

Yleinen tapa käsitellä kielen rakenteita on tallentaa ne AST-muotoon. Siihen liittyy kuitenkin paljonkin ongelmia analysointitarkoituksessa. AST-rakenteilla ei nimittäin ole yksiselitteistä viittaussemantiikkaa, jonka mukaan voitaisiin poimia itsenäisesti evaluoitavaksi yksilöllisiä koodin rakenteita esiin kuten if-lause metodista main. Siten sisempiä AST-rakenteita ei ole mahdollista käsitellä ja evaluoida itsenäisesti vaan vain osana laajempaa kokonaisuutta. Siksi tarvitaan kehittyneempi menetelmä, jota kuvaamme seuraavassa.

3.3.2 Kielioppien saatavuus ja ongelmat

Kielioppeja eri kielille ja myös Javalle on saatavissa internetistä. Ongelmana on kuitenkin kielioppitiedostojen laatu, erilaiset kirjoitusvariaatiot ja niiden taltiointitarkkuus yleisesti, koska osa niistä on pelkästään periaatteellisia ja vain osa riittävän tarkkoja, että ne mahdollistaisivat sellaisenaan parserin rakentamisen.

Esimerkiksi Sunin julkaisema version 3 (Java1.5) kielioppi, joka vastaa uusimman kääntäjän tilannetta ja julkaistaan myös Java 3rd Edition-kirjan mukana, on ongelmallinen. Siinä on useita virheitä ja lisäksi moniosaisten termien semantiikkaa ei ole kuvattu tarkoituksenmukaisesti. Hankalinta on, että Sunin ilmoittama kielioppi ei esitä tarkkaan lausekerakenteita, koska siinä kaikilla laskentatoiminnoilla yhteenlasku, kertolasku, jakolasku ja loogiset vertailut olisi virheellisesti sama prioriteetti ja siten sama laskentajärjestys.

Siksi jotta riittävään tarkkuuteen päästäisiin, tulee Sunin toimittama kielioppi unohtaa kokonaan ja organisoida kielen rakenteet itse uudelleen ja muuntaa ne ristiriidattomaan, hierarkisempaan muotoon.

Prolog-kielelle on kehitetty edistyneitä kieliopin käsittelyn menetelmiä kuten definite clause grammar (DCG), jossa yksittäisen lausekkeen prioriteetti voidaan ilmaista lukuna. Tämä menetelmä soveltuu tulkkaavalle ja ISO-standardin mukaiselle Prologille. Tutkimuksen mukainen Visual Prolog-kehitin sisältää oman menetelmän, jota kutsutaan semanttiseksi kielioppirakenteeksi, minkä on alkujaan kehittänyt Leo Jensen, Prolog Development Center. Alla if-lauseen kuvaus tällä menetelmällä:

```
1 Statement =
2  "if" Condition "then" Statement ->
3    iff(Condition, Statement).
```

Termin nimenä on Statement, jonka yksi mahdollinen muoto olisi edellä mainittu if-lause, jonka syntaksi sisältää hierakisesti alemmalla tasolla sijaitsevat termit Condition ja Statement. Varattuja sanoja siinä ovat lainausmerkeissä olevat sanat if ja then. Syntaksin oikea järjestys ilmenee ”->” -symbolia edeltävästä osuudesta. Sen oikealla puolella oleva rakenne *iff* argumentteineen on muotoa, joka syntyy jäsennyksen tuloksena vastaavan työkalun muistiin. *Iff*-rakenne kuvaa siis semantiikkaa, mistä käytämme tässä artikkelissa nimitystä Y. Näin saatiin ytimekäs kokonaisuus: rivi 1 kertoo määrittelyn nimen (N), rivi kaksi kertoo syntaksin (X) ja rivi kolme semantiikan (Y).

Jos käytännön tilanteessa asianomainen if-lause ei johda halutulla tavalla sen sisältämän osuuden käynnistymiseen, vika voi olla vain sen Condition-osassa. Tällöin tämä ehto olisi vianpaikantajalle pragmaattista tietoa (Z).

3.3 Koodin mallien epistemologia

Epistemologia pyrkii erottamaan todellisen ja riittävän tietämyksen paikkansa pitämättömästä tietämyksestä. Se tutkii mm. tiedon luonnetta, tietojen välisiä yhteyksiä. OMG ei erikseen puhu epistemologiasta tutkiessaan metodologioita ohjelmistokehityksen näkökulmasta, sehän ei edusta tiedettä vaan käytännön päämääriä. Kuitenkin koodin ja mallien synkronointi on OMG:n

päätavoitteita. OMG:n painopisteenä on mallintamiskokonaisuus nimeltään Model Driven Architecture, MDA (MDA 2006), jossa keskeisimpänä tietosisältönä on MOF-määrittely (MOF 2006). MOF:ssa koodintarkan informaation käsittely osoittautuu mahdottomaksi, koska Javan lause-tason informaatio puuttuu siitä (tosin siinä on OCL-rajoitekieli).

MOF tarjoaa koodista poimitulle informaatiolle seuraavat mallit ja tasot. Taso M0 eli systeemitaso kuvaa koodia sellaisenaan. Taso M1 eli luokkataso viittaa koodista poimittuun luokkamäärittelyjen rajapintaan. Taso M2 viittaa metamalliin, jolla määritellään luokkien yleiset rakenteet. Ylinnä on taso M3 metametamalli, joka määrittelee yleisen transformaatiomallin lähinnä muunnostarkoituksiin.

Koska MOF ei tue lausetarkkaa käsittelyä, sen tarjoama ymmärtämistuki jää suppeaksi. Niinpä alimmalle tarkastelutasolle tarjotaan vaihtoehtona AST-rakenteiden avulla muodostettua mallia, mutta siinähän on muutamia perusongelmia, joita on kuvattu tämän artikkelin alkuosassa.

Seuraavassa jaksossa koodista poimittavissa olevaa tietoa käsitellään erillisten teorioiden valossa tarkoituksena muodostaa looginen ketju sellaisen eheän mallin luomiseksi, joka sisältäisi sekä MOF:n kaltaisen metariippuvuusinformaation että koodin tarkan lauserakenneinformaation, jotta tulosta voitaisiin käyttää yhtenäiseen tarkasteluun.

3.3.1 T1 Koodin jäsentämisen teoria

Koodista saadaan jäsennyksen tuloksena rakenteet (N), jotka parserin jäljiltä ovat yhdessä hierarkisessa rakenteessa, jonka juurena on aloitussymboli (S0), esimerkiksi *TranslationUnit*, joka jakaantuu paketin ja luokan määrittelyihin.

Prologilla jäsennetty parsintapuu (hierarkinen kokonaisuus) on predikaattilogiikan (1. kertaluvun logiikka) mukaisesti täydellisesti aksiomatisoitavissa. Parsinta on täydellinen ja todistettavissa, jos kukin rakenne säilyttää alkuperäisen muodon ja riittävän tarkan informaation (koherenttius ja konsistenssius).

3.3.2 T2 Abstraktiotason nosto, symbolinen kieli

Laskentasääntöineen ja sisäkkäisine lauseineen Javasta saadut rakenteet ovat tarpeettoman monimutkaisia sellaisenaan analysoitaviksi. Kun rakenne on kerran jäsennetty oikein, sitä voidaan merkittävästi (noin 90 %) yksinkertaistaa pudottamalla sisäkkäisiä itsestään selviä välitasoja ilman, että uusi sisältö vääristyy tai muuttuu epäloogiseksi.

Jatkoanalysoinnin kannalta tärkeää on ryhmitellä rakenteet mahdollisimman hyvin. Analysointitarpeita ovat mm. seuraavien rakenteiden tunnistaminen, suluissa jatkossa käytetty lyhenne: määrittelylauseet (*def*), konstruktorikomennot (*creator*), dataa muuttavat rakenteet (*set*), dataan viittaavat rakenteet (*ref*), metodiviittaukset (*invoke*, *get*), operaatiot (*op*), silmukkalauseet (*loop*), ehdolliset lauseet (*path*, *condition*), vakiot (*value*) ja muut lauseet. Näin saadaan 11 lauseryhmää.

Tällä ryhmittelyllä päästään seuraavaan tavoitteeseen. Eri analyysit voidaan määritellä abstraktien rakenteiden välisinä yhteyksinä, esimerkiksi data flow – analyysi on saman datan set- ja ref-rakenteiden välinen keruutoiminto. Ohjausvuo on valitusta koodinosasta koottu get- ja creator-lauseiden kokonaisuus, joka käy läpi muiden lauseiden tarvittavat alarakenteet.

Ryhmittelylogiikasta voidaan muodostaa uusi symbolinen kieli, jolle tutkimuksessa on annettu nimi Symbolic. Kuten jokaisella formaalilla kielellä, myös Symbolicilla voi olla oma syntaksi ja semantiikka. Näin ollen sille voidaan luoda jäsennin ja koodigeneraattori, joista on hyötyä muunnettaessa abstrahoituja rakenteita lukijan paremmin ymmärtämään muotoon tai tarvittaessa luoda sujuva testausympäristö analysointikokoonpanolle alkuarvojen syöttömahdollisuuksineen.

Jos Java-kielen keskeinen lause on nimeltään *Statement*, voisii sitä vastaava korkeamman tason rakenne Symbolic-kielessä olla *Clause*. Näin saadaan translaatiotarve *Statement2Clause*, jota kuvataankin seuraavassa.

3.3.3 T3 Translaatio

Formaalien kielten translaatioon on kehitetty erillisiä monimutkaisia menetelmiä, jotka perustuvat useinmin ulkoiseen translaatiomekanismiin (*translation engine*). Mielenkiintoista on, että translaatio voidaan tehdä myös ”suoraan” ilman erillistä sääntölogiikkaa rekursiivisen sijoittamisen avulla. Prolog tukee tällaista vaihtoehtoa tunnistaessaan automaattisesti kielen rakenteet. Tällöin aivan tavallinen sijoituslause käy translaatioksi.

Seuraava rekursiivinen *xslate*-komento muuttaa Java-kielen if-lauseen Symbolisen kielen path-lauseeksi:

```
xlate(iff(Condition,Statement)) =
  path(xlate(Condition),xlate(Statement)).
```

Olennaista ylläolevassa komennossa on se, että vasemman puolen suluissa on muunnettava Java-termi ja yhtäsuuruus-merkin jälkeen Symbolic-kieleen syntyvä rakenne. Koska jokaisen alarakenteen yhteydessä kutsutaan alas laskeutuen uudelleen translaatiokomentoa, kaikki alarakenteet muuntuvat automaattisesti uuteen muotoon. Translaation suorittamiseksi jokaiselle erilaiselle Javan termille tarvitaan yleensä vain yksi edellämainitun kaltainen sijoituslause. On vain muutama poikkeus.

Siirto Javasta symboliseen kieleen (*Statement2Clause*) voidaan validoida analyyttisesti

translaatiolausekkeiden kautta sekä testitulosteilla. Lisäksi Visual Prolog sisältää tyypintarkistustoiminnon, joka tarvittaessa varmistaa, että kukin alalauseke on kattavasti määritelty. Se poistaa oikein käytettynä puutteet muunnoksen kattavuudessa.

3.3.4 T4 Semantiikan talteenotto

Ohjelmointikielten semantiikan ilmaisemiseen on useita esitystapoja eli notaatioita. Merkittävimpiä ovat toiminnallinen semantiikka (operational semantics). Eräs sen muoto, luonnollinen semantiikka (natural semantics), on Prologin kannalta merkittävä, koska se on Prolog-yhteensopiva. Siten jokainen luonnollisen semantiikan lause voidaan periaatteessa ohjelmoida Prolog-lauseiksi. Myös Prolog-lauseita voidaan tietyin edellytyksin muuntaa luonnollisen semantiikan esitystapaan, jolloin sitä voidaan käyttää kuvaamaan Prolog-koodin käyttäytymistä alkuperäistä korkeammalla tasolla.

Luonnollinen semantiikka tarjoaa seuraavia etuja perinteisempiin semantiikkoihin nähden:

- Kaikki sen oliot ovat äärellisiä termejä, joten rakenteen käsittelyyn ei tarvita monimutkaista tyyppiteoriaa.
- Koska jokaiselle rakenteelle voidaan määritellä useita päättelysääntöjä, sillä voidaan mallintaa myös epädeterminististä hakua, joka on Prologin parhaita ominaisuuksia.
- Epädeterminisitinen määrittely mahdollistaa polymorphististen rakenteiden mallintamisen.

Luonnolisen kielen semantiikan yleinen lausemuoto on seuraava:

$$\frac{H_1 \vdash T_1 : R_1 \;\; .. \;\; H_n \vdash T_n : R_n}{H \vdash T : R} \quad \text{if } \langle cond \rangle$$

Kuva 2. Luonnollisen semantiikan periaate.

Kuvassa 2 alarivillä on todistettava lauseke, joka sisältää hypoteesin H, johon liittyy parametrikokonaisuus T ja tavoitetulos R. Se ratkeaa, jos ylemmällä rivillä oleva hypoteesi H1 argumentein T1 ratkeaa tuottaen tuloksen R1, joka edelleen mahdollistaa muiden hypoteesien H2 … Hn toteutumisen. Lopullinen tulosjoukko R on kombinaatio ehdollisten hypoteesien tuottamasta tulosjoukosta. Tämä lauseke toteutuu vain jos ehto *cond* toteutuu myös.

Prolog sallii predikaatteineen tällaisen lauseen kirjoittamisen. Visual Prolog erityisesti sisältää oliojärjestelmän, joka myös soveltuu sekä ylläolevien hypoteesien ohjelmointiin että koodin muuttamiseen vastaavanlaisiksi hypoteesimäärittelyiksi. Alla pieni esimerkki, jossa hypoteesit on muutettu predikaateiksi:

```
h(T, Cond) = R:-
    ifTrue(Cond),
    R1 = h1(T1),   … Rn=hn(Tn),
    Result = [R1,R2,… Rn].
```

Predikaatti ifTrue tarkistaa aluksi onko ehto Cond voimassa. Jos on, siirrytään h-predikaattien evaluointiin.

Seuraavassa esimerkissä hypoteesit ovat oliota (tässä tutkimuksessa symbolisen mallin elementtejä):

```
evaluate(T, Cond) = R:-
  ifTrue(Cond),
  R1 = H1:evaluate(T1),
  …
  Rn = Hn:evaluate(Tn),
  R = [R1, … Rn].
```

Esimerkin selostus: Visual Prologissa suorat dataviittaukset voivat kohdistua vain ko. olion sisälle. Se estää haitalliset sivuvaikutukset. Jokaisella oliolla tulee olla metodi *evaluate*, joka laskee oman tuloksensa (R_i) rekursiivisesti.

Ohjelmointikielten semantiikassa keskeisiä asioita ovat seuraavat asiat:

- Dataviittausten huomiointi
- Metodikutsujen huomiointi parametreineen
- Polymorphististen metodien huomiointi
- Perintähierarkian huomiointi

Kirjallisuutta semantiikan rajoitteista ja erityisesti Java-koodin semantiikasta on runsaasti. Javan spesifikaatio sisältää täsmälliset määrittelyt kielensä piirteistä edellä mainitut asiat huomioiden.

Symboliseen malliin nämä yllämainitut asiat saadaan toteutettua siten, että alkuperäinen Javan rakenne ja viittaus siihen korvataan joko mallia luotaessa tai mallia läpikäytäessä alkuperäistä laajemmalla rakenteella (tyypillisesti oma haku- metodi *lookUp*), joka suorittaa täsmällisemmän ja yleisemmän identifioinnin rakenteelle.

Esimerkiksi symboli *port* saattaa olla sekä metodin argumenttina, muuttujana ja myös luokan attribuuttina. Siten eri metodeissa sama muuttujanimi viittaa eri asiaan. Tätä kutsutaan alias-ilmiöksi. Sen vuoksi metodi-kutsuissa tuloargumentit ja parametrit on saatava vastaamaan toisiaan.

Semantiikkateorian T4 tarkoituksena on varmistaa, että mallinläpikäyntialgoritmi voi toimia eri tilanteissa samaan tapaan kuin alkuperäinen Java-koodi tunnisten vastaavat rakenteet. Se tulkitsee siten koodin käyttäytymistä ohjelmoijan eduksi. Ohjelmoijan kiusana on ns. jojo-ilmiö, jossa kutsuttua oliorakennetta joudutaan etsimään laajalti olion perintähierarkiasta. Semantiikkateorian tuloksena saadaan kerättyä UML:n sekvenssikaavion mukaiset tiedot, mutta tarkempana sisältäen kaikki lausekkeet ja eri suoritusvariaatiot.

3.3.5 T5 Symbolinen malli ja mallin kutoja

Erilaisia malleja ja mallienluontimekanismeja (model weaver) on kehitetty runsaasti mm. OMG:n toimesta (Bezevin 2005). OMG on verrannut erilaisten malliratkaisujen ominaisuuksia seuraavasti. Tärkeitä vertailtavia piirteitä ovat modulaarisuus, muunnettavuus, jäljitettävyys, formalisoitavuus, koodin suoritettavuus, aspektien poiminnan mahdollisuus ja ratkaisun erikoistamisen mahdollisuudet. MOF-mallien keskeinen heikkous on assosiaatioiden käsittelyssä. Koska oliomallit voivat itsessään käyttää vain olioita, joiden heikkoutena on suljettu sisäinen rakenne, MOF-malleissa joudutaan jokainen assosiaatio pilkkomaan useiksi olioiksi (kuten association, associationEnd, link ja linkEnd). Pilkkomisen takia mallien läpikäynti vaatii paljon ohjelmointia ja monimutkaisia algoritmeja.

Takaisinmallintamisen kannalta tärkeimpiä mallien piirteitä ovat modulaarisuus, muunnettavuus, jäljitettävyys, formalisoitavuus ja suoritettavuus. OMG toteaa, että MDA:lla ei voida tuottaa suoritettavaa mallia, vaikkakin ”executable uml” on paljon käytetty käsite. Sen sijaan äskeisten teorioiden T1-T4 mukaan kehitetty malli on simuloitavissa eli mallinnettavissa niin pitkälle, että myös koodin sivuvaikutukset voidaan analysoida. Samalla simuloinnista saadaan tuloksena jälki (trace), joka vastaa dynaamisen analyysin tulosjoukkoa. Symbolinen malli on formalisoitavissa tarkkaan ja muutettavissa myös MDA-malleiksi ns. xmi-esitystavan avulla (XMI 2005).

Symbolisen mallin keskeisin ja lähes ainoa rakenne on symbolinen elementti (SE), jolla on optimoitu perintähierarkia. Se periytyy Symbolic-nimisestä luokasta, joka sisältää teorian T2 mukaiset määritelmät, parserin ja koodigeneraattorin. Eri tyyppiset elementit, jotka kuvattiin teorian T2 yhteydessä, toteutetaan peruselementin SE erikoistuksina. Kunkin elementin muistiin tallennetaan siihen kuuluva koodi, joka on luontivaiheessa muuntamalla teorian T3 mukaiset T2:n rakenteet omiksi elementeikseen ja prosessoimalla saatu koodi vielä teorian T4 mukaisesti. Näin saadaan aikaan atomistinen malli.

3.3.6 T6 Selektiivinen kysely

Tunnetuin koodin tarkastelun menetelmä on nimeltään viipalointi (slicing). Se tuottaa tietystä koodin osasta siihen suoranaisesti liittyvät muut ohjelmalauseet suoritusjärjestyksessä joko taakse- tai eteenpäin.
Viipalointiin liittyy kuitenkin paljon rajoituksia ja se soveltuu vain koodin matalan tason tarkasteluun. Siksi olemme valinneet paremman perusmenetelmän, joka on nimeltään leikkely eli chopping.

Chopping on tärkeä analysointikeino, joka tarkoittaa haluttujen kohteiden saavutettavuuden tutkimista. Se käy erinomaisesti syy-seuraussuhteiden käsittelyyn, joka on olennaista vianpaikannustehtävissä. Chopping pyrkii analysoimaan kuvan 3 mukaan perättäisten funktiokutsujen joukkoja, jotka alkavat tietystä Start-kohdasta ja etenevät aikajärjestyksessä Target-kohtaan saakka:

```
Start = fk •. fk-1 •. ... •.f  • Target.
```

Chopping-yhtälö voidaan ilmaista monella eri tavalla symbolista notaatiota käyttäen. Ilmaisu lopusta alkuun on muotoa:

```
Target = fk •. fk-1 •. ... •.f (Start).
```

Sisäkkäisessä muodossa se voidaan kirjoittaa:

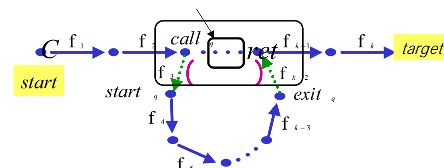
```
Target = f (f (f ( ... f( Start ) ) ) )
```

Tai peräkkäisessä muodossa listana:

```
Target = [f(XK), f(XK-1).. f("Start")].
```

Chopping-analyysin ohjelmointi Prologilla on selväpiirteistä. Seuraava koodi sisältää rekursiivisen säännön, joka kerää informaatiota seuraavilta perättäisiltä solmuilta, jotka on määritelty *f*-rakenteina sisältäen kunkin funktion nimen ja vastaavat argumentit. Tulokset saadaan kumulatiivisesti kerättyä predikaatin toiseen argumenttiin *Sequence*-muuttujan välityksellä:

```
chopping (Target, [f(This)|Sequence]):-
      f(NextId,Arguments),
      NextId:chopping(Target,Sequence).
```



Kuva 3. Chopping ohjausvuon tulkinnassa.

3.3.7 T7 Tyhjentävä haku ongelman ratkaisijana

Perinteinen koodin analysointi tarkastelee tyypillisesti ohjelman lauseiden välisiä suhteita sellaisenaan [Slicing, Reps] ilman korkeamman abstraktion malleja. Tarkka käsittely johtaa usein laskennallisesti liian vaativiin algoritmeihin, jos aluetta ei voida rajata tarpeeksi.
Äsken kuvaamamme selektiivinen kysely, chopping, sen sijaan tuottaa suppean tietoaineiston, joukon mallin elementtejä, jota voidaan käsitellä monin eri tavoin tyhjentävästi mm. tarkastelemalla eri suoritus-polkuvaihtoehtoja, datan kulkua ja sivuvaikutuksia ja mahdollisia aktivointeja eri tilanteissa.

Yhden metodin osalta kaikki Penningtonin tietotarpeet voidaan aina poikkeuksetta poimia täydellisesti siten, että tuntemattomat parametrit käsitellään viittauksina. Mikä tahansa suorituspolku voidaan myös evaluoida yhtenä kokonaisuutena, jolloin voidaan tutkia pienimpiä ilmiöitä koodista.

Ohjelmasilmukoiden käsittely on silti ongelmallista, jos suorituskertoja tulee lukemattomia. Siinä tapauksessa toistokertoja on voitava rajoittaa. Samoin ulkoisten rajapintojen tulkitseminen ehtolausekkeissa vaatii alkujärjestelyjä ennen testausta.

3.3.8 T8 Javan täydellinen relaatiomalli

Javaa ja sen koodista kehitettyä symbolista mallia tarkastellaan järjestelmällisimmin sitä varten suunnitellun, yhtenäisen relaatiorajapinnan kautta. Näin voidaan tunnistaa tärkeimmät käsitteet (entity) ja niiden väliset suhteet (relation), mikä tuottaa kelvollista tietoa vianpaikannukseen. Relaatiomalli sisältää seuraavat perusmääritykset, joiden välisiä relaatioita on mahdollista tutkia:

- Luokan nimi, class
- Luokan jäsenet
- Luokan super-luokka
- Metodin sisä ltämät elementit
- Metodin sisältämät staattiset kutsut
- Metodin suorittamat dynaamiset kutsut
- Kuhunkin elementtiin liittyvät navigointitiedot.
- Kuhunkin elementtiin liittyvät sivuvaikutukset
- Jäljityshistoria

Kun koodin yksittäiset rakenteet saadaan näin yksittäisillä kyselyillä hallintaan, tarkastelua voidaan laajentaa käyttämällä kyselyjen välillä loogisia operaatioita ja alakyselyjä.

3.3.9 T9 Selitysmalli

Tietämysteknisesti tarkasteltuna koodin informaatio on mielenkiintoista. Halutessamme kysyä syy-seuraussuhteita, koodista saadaan läpikäyntialgoritmilla loogiset ehdot ja rakenteet, jotka vaikuttavat kyseiseen tarkasteluväliin. Saatujen symbolisten rakenteiden joukko voidaan muuntaa luonnolliselle kielelle, esimerkiksi englanniksi, jolloin saadaan selväkielinen perusteluketju tarvittavine parametreineen.

Yhteistä kaikelle käsittelylle on se, että saadun tulosinformaation muoto on aina sama kyselystä riippumatta. Se helpottaa ohjelmointia ja toteutusta ja tulosten integrointia. Tulosinformaatiolle voidaan valita erilaisia tulosteita ja tarkastelutapoja kuten what, why tai how, jolloin näytölle syntyvä aineisto saadaan vastaamaan parhaiten käyttäjän tekemää kyselyä.

Tulostuksessa voidaan toki käyttää myös luonnollista semantiikkaa ja sen esitysasua, mutta se on ohjelmoijille vieraampi.

3.3.10 T10 Pragmaattisen tulkinnan teoria

Ylläpito sisältää paljon erilaisia tehtäviä ja alatehtäviä, joissa koodin tarkastelu on tarpeen. Aiheen laajuuden johdosta sitä käsitellään tässä kohdassa vain muutamin esimerkein Penningtonin tietotarvemäärittelyn pohjalta:

- Jokainen työkalun tuotos, joka johdattaa käyttäjää vianpaikannustehtävässä ja supistaa alkuperäistä informaatiojoukkoa, on pragmaattista informaatiota.
- Jokainen perehtymistilanteessa käyttäjälle tuotettu vastaus kysymyksiin what, why ja how, on pragmaattista informaatiota.
- Jokainen syy-seuraussuhdetta erittelevä haku, joka on kohdistettu vian määritykseen, tuottaa pragmaattista vian rajauksen informaatiota.
- Jokainen metodin suorituspolkujen hahmottamiseen liittyvä kokonaisuus palvelee vianhaun tarpeita tutkittaessa ohjelman pysähtyvyysongelmia.
- Jokainen kriittisen kuuntelijaolion tiloihin liittyvä kysely tuottaa pragmaattista informaatiota.

Teknologiamielessä pragmaattisen informaation tarkastelu vaatii käyttäjän tarpeiden tunnistamista ja metodologian sovittamista niihin. Saatu tietämys on kuitenkin arvokasta, sillä se palvelee kiireistä käyttäjää hänen jokapäiväisessä työssään sellaisenaan.

4 Metodologia

Edellä kuvattu ontologia ja tietämyksen määrittely eri teoria-alueineen toimivat lähtötietoina metodologian tarkastelulle.

4.1 Metodologian toteutus

Seuraavassa esityksessä Peircen semioottista käsitteistöä, jonka muodostavat merkki (*sign*), kohde (*object*) ja logiikalla toteutettava tulkinta (*interpretant, logic L*), lähestytään symbolista alkaen. Yhdistämällä nämä käsitteet saadaan hahmoteltua kokonaisuus SOL. Malli (M) määritellään yksinkertaisesti säiliöksi, joka sisältää malliin luodut oliot. Mallin analysointi (A) tapahtuu kyselyiden (Q, Query) avulla oliorajapinnan kautta.

4.1.1 Symbolin toteutus (S)

Kieliopista saadaan suoraan kaikki symbolit, jotka viittaavat lähdekoodin muuttujiin. Niitä ovat luokan nimi, metodi, attribuutti ja muuttujanimet. Koodin tarkasteluun tämä metoditarkkuus ei vielä riitä, koska kun saatua symbolista mallia halutaan tutkia esimerkiksi vianpaikannuksen tarkoituksessa, tarvitaan suorituspolun tarkkuus ja sen toteuttamiseen suorituspolkuelementti.

Jos erityisesti käyttäytymismalli ja elementtien vaikutukset kiinnostavat, tarvitaan lisäksi sivuvaikutuselementti, joka tallentaa kunkin elementin suorittamat ulkoiset muutokset.

Symboli tarkoittaa siis symbolisessa mallissa käsitettä, joka voi olla

- Käyttäjän antama alkuperäinen symboli
- Suorituspolkuun viittava symboli
- Sivuvaikutukseen viittaava symboli

Symboli on käytännössä vastaavaan työkaluun luotu viittaus vastaavaan olioon, joka voidaan tulostaa ja visualisoida eri tavoin sen sisällön ja tulkinnan mukaan. Kun käyttäjä napauttaa symbolin kuvaketta näytöllä hän pääsee kaikkiin sitä vastaavan olion ominaisuuksiin käsiksi, jolloin saadaan aikaan semanttinen käyttöliittymä. Symboli on siis esittämisen, kommunikaation ja ohjauksen väline.

4.1.2 Objektin toteutus (O)

Määrittelynsä mukaan oliojärjestelmän olio ja luokka sisältävät abstrahointia, yleistämistä ja erikoistamista tukevia piirteitä. Ne kaikki ovat hyödyllisiä lähdekoodin analysoinnissa ja soveltuvat Peiren semiotiikkaan seuraavasti:

- Abstrahoinnilla voimme luoda käsiterakenteita, jotka ovat kaikkien analyysien kannalta yhteisiä ja kehittää näin mahdollisimman teoreettista yhteistä mallia.
- Yleistämisellä voimme määritellä yhdenmukaisen olion työkalun luokkana. Tässä sitä kutsutaan symboliseksi elementiksi, *symbolicElement*. Se pystyy mallintamaan kaikkia ohjelman rakenteita, koska kaikilla rakenteilla on merkittävä määrä yhteisiä piirteitä.
- Voimme luoda riittävän tarkkoja erikoistettuja olioita symbolicn alaluokkina siten, että ne jakaantuvat teorian T2 ryhmittelyn mukaan.
- Voimme periyttää kaikki oliot superluokasta *symbolic* , joka sisältää symbolisen kielen määrittelyt. Näin kaikkien olioiden välille saadaan aikaan symbolinen kieli ja sen täydellinen tuki.

Ohjelmistokehitystasolla keskeisin olio, symbolinen elementti, toteuttaa kaikki tarvittavat piirteet rajapintojen ansiosta. Se periytyy *symbolic*-luokasta kuten edellä mainittiin ja sitä käyttävät super-luokkana kaikki symbolisen kielen tyyppirakenteet.

Puhdas oliomalli ei kuitenkaan sellaisenaan riitä olioiden esittämiseen, koska kapselointi rajoittaisi tiedonsaantia olioiden välillä ja tekisi mallin assosiaatioiden läpikäyntialgortmien rakentamisen hyvin vaikeaksi, koettaisiin samat ongelmat kuin xmi-malleissa. Siksi tarvitaan täydentävä ohjelmointiparadigma, logiikka, jota kuvataan seuraavassa.

4.1.3 Tulkinnan toteutus, logiikka (L)

Olion sisäiset tiedon tallennukset suoritetaan logiikan kautta predikaatteina ja faktoina. Oliot muodostavat keskenään navigointia tukevan kaksisuuntaisen verkoston, joka saadaan aikaan *child*- ja *parent*- sekä *next*- ja *previous*-faktoilla. Elementin sisältämä koodi tallentuu contents-nimiseen faktaan, joka on käytännössä symbolisen kielen rakenne, *Clause*.

Mallin läpikäyntialgoritmit käyttävät hyväkseen joko navigointifaktoja tai contents-faktan sisältöä.

Prologin päättelykone tarjoaa tukea läpikäynnin ohjelmoimiseen. Näin saadaan aikaan traverse-toiminto. Mallin suorittamiseen tarvitaan run-toiminto, joka palauttaa oletuksena symbolisen kielen lauseita.

4.1.4 Mallin toteutus (M)

Yleistä symbolista mallia voidaan luonnehtia seuraavalla määrittelyllä:

```
model(L, B, T, F),
```

jossa L (layers) on tarvittavien mallin kerrosten määrä, B (bases) on tarvittavien erilaisten superluokkien määrä, T (types) on ratkaisun erilaisten tietotyyppien määrä ja F (features) on sellaisten funktioiden (piirteiden) määrä, jotka voidaan toteuttaa symbolisella elementillä.

Ylläolevasta tarkastelusta voimme päätellä seuraavaa:

- L on 1, koska tarvitaan vain 1 kerros, joka sisältää kaiken informaation.
- B on 1, koska kaikki oliot periytyvät symbolisesta elementistä, joka edelleen periytyy symbolisen kielen määrittelystä.
- T on 11, koska symbolisen kielen rakenteeseen liittyy 11 perustyyppiä.
- Mallilla toteuttavien piirteiden määrä, F, on periaatteessa ääretön, koska symboliselle mallille voidaan ohjelmoida lukemattomia erilaisia käsittelijöitä eri tarkoituksiin, mm. vastaamaan kyselyihin.

Edelliseen verrattuna MDA-mallit ovat huomattavasti monimutkaisempia, mutta silti heikompia ominaisuuksiltaan ja laajennettavuudeltaan.

Tyypillisimmät mallin toiminnot ovat seuraavat:

- Build rakentaa mallin ja sen oliot
- Traverse käy mallin sisältöä läpi tietyin oletuksin
- Chop hakee kahden metodin väliin kutsuhierarkiassa sijoittuvan metodien joukon.
- Run ajaa (simuloi) koodia halutusta kohdasta lähtien.
- Reset poistaa oliot muistista.

4.1. 5 Analyysin toteutus (A)

Edellä mainittiin mallin ja oliorajapinnan keskeisimpiä piirteitä. Ohjelmistoteknisesti analyysi on kuuntelija

(observer), joka mallin käsittelyn käynnistyttyä jää odottamaan dataa. Analyysin parametrit määrittelevät mistä piirteestä kulloinkin on kysymys. Tulokset saadaan symbolisen kielen muodossa.

4.1.6 Kysely eli Query (Q)

Kuten edellä kerrottiin, tarkastelun tarpeilla on kysymysten kautta yhteys hypoteeseihin, joista edelleen on yhteys kyselyihin. Vastaus kysymyksiin what, why ja how saadaan joukkona symbolisen kielen rakenteita, jotka voidaan edelleen muuntaa kaavioiksi, näytöksi, tekstiksi tai xmi-muotoon siirrettäväksi muihin kehitysvälineisiin.

Tyypillinen kysely liittyy ongelman rajaukseen ja antaa tuloksena joukon elementtejä, joita epäillään. Näitä kutsumme ehdokkaiksi, vikaehdokkaiksi. Kun käyttäjä saa näkyville listan vikaehdokkaista, hän voi valita sopivan etenemistavan navigointiin ongelman ratkaisun täsmentämiseksi.

4.2 Yhteenveto metodologiasta

Symbolisen mallin rakentamiseen ja hyödyntämiseen tarvitaan seuraava teoriakokonaisuus:
T1 Java - koodin lukemiseen parsintapuuksi
T2 Uusi abstraktiotaso parsintapuulle
T3 Translaatio symboliseen kieleen
T4 Javan semantiikan siirto malliin
T5 Mallin kutoja luomaan yhtenäisen aineiston
T6 Suppea kyselyjärjestelmä alkurajaukseen
T7 Laajat kyselyt suorituspolkujen analysointiin
T8 Kohdekielen ja mallin relaatiomalli
T9 Selittämisen teoriat muuntamaan tulokset logiikaksi
T10 Teoriat pragmaattisen informaation tuottamiseen

5 Ylläpitotehtävän suorittaminen

Tutkimusrajauksessa tulotietona toimii ylläpitoon luotu tehtävä, jota kutsutaan muutosvaatimukseksi (Change Request, CR). Tällöin kyseessä on ongelma tai parannusehdotus tai perehtymistarve.

Seuraava proseduuri pyrkii luomaan toimivan yhteyden ylläpitotehtävän ja symbolisen mallin kyselyn välille.

5.1 Ylläpitometodi

Ylläpitotehtävän kulkua kuvataan seuraavassa lyhyen kirjainyhdistelmän TPHQ avulla.

5.1.1 Task (T)

Ylläpitotehtävän käsittelyä hajautetussa ohjelmistotyössä on kuvannut Walenstein (2002) RODS-käsitteistöllään, joka esittää kuinka tehtävän suoritus pyritään minimoimaan, kuinka suoritusalgoritmi halutaan optimoida, kuinka tehtävän eri osat jaetaan useille tekijöille ja kuinka erikoistuneita asiantuntijoita pyritään käyttämään tehtävän osien suorittamiseen. Jos tehtävä ei ole triviaali, se alkuvalmistelujen jälkeen muuttuu prosessiksi, josta kerrotaan seuraavaksi.

5.1.2 Prosessi (P)

Prosessin tarkoituksena on ratkaista monivaiheinen ongelma, joka vaatii useita erillisiä toimenpiteitä. Koodin tarkastelun välivaiheita on tutkinut mm. von Mayrhauser (1992). Ihminen ajattelee tällaisessa tilanteessa kysymysten ja vastausten kautta. Letovsky (1983) on kuvannut ylläpitoon liittyviä kysymyksiä luokittelulla : what, why ja how. Käyttäjä haluaa selvittää mitä jokin koodin osa tekee (what), miksi se suorittaa tiettyjä toimintoja (why) ja kuinka se toimii (how). Koska nämä tiedot eivät selviä lukematta koodia, hän joutuu arvamaan ja tekemään olettamuksia. Näin hän tulee luoneeksi hypoteeseja.

5.1.3 Hypoteesi (H)

Hypoteeseilla käyttäjä todistaa toimiiko esimerkiksi Server-luokka niin kuin hän oletti. Jos se toimii, hän voi kehittää hypoteesin pyrkien todistamaan sen oikeaksi tai vääräksi. Lopullinen päätelmä voidaan tehdä kun hypoteeseista on saatu riittäviä tuloksia. Vaikka tällainen hypoteesien käsittelyn prosessi kuulostaa vaivalloliselta, se on monessa tilanteessa esimerkiksi vianhaussa ainoa vaihtoehto. Jos käyttäjällä on käytössään vain editori, työ on hyvin hankalaa, mutta kehittyneet takaisin-mallintamisen työkalut voivat tässä häntä auttaa.

Peirce esittää ongelman lähestymiseen kolme erillistä tapaa: abduktion, induktion ja deduktion. Seuraavassa muutama esimerkki niistä:

- Abduktion käyttö ohjausvuon käsittelyssä: Kaikki ohjelman sekvenssit, jotka alustavat olion *Server*, ovat palvelimeen liittyviä olioita. Esimerkiksi koska metodi *main* alustaa serverin, metodi *main* on kriittinen palvelimen käynnistymisen kannalta.
- Induktio: Muuttuja *port* saa komponentin sisällä serverin aloituslogiikassa arvoja 80,81 ja 82, jotka kaikki ovat oikeita. Toinen vaihtoehto on se, että komponenttia kutsutaan ulkoa, jolloin portille ei voida asettaa käynnistyarvoa, mikä on riski. Siten on varmistettava nämä haarat ja tarkistuksena lisättävä oletusarvoksi 80. Näin voidaan yleistäen (induktiivisesti) päätellä, että kaikki tapaukset ovat kunnossa.
- Deduktio: Olion *Server* käynnistymiseen vaikuttavat komentoriviltä saatu arvo ja if-lausekkeessa saatu arvo. Käynnistyminen estyy, jos poikkeustilanne laukeaa main-metodissa. Nämä lauseet muodostavat

täydellisen logiikan (and) serverin käynnistymiseen. Jos olio ei käynnisty oikein, vian täytyy olla jossakin näistä kohdista.

Letovskyn kysymyksillä on seuraava yhteys koodin sisältöön:

- What – kysymys viittaa elementin tarkoitukseen ja toiminnallisuuteen (function).
- Why – kysymys viittaa syy-seuraussuhteeseen, joka on vianpaikannuksessa hyvin hyödyllinen piirre.
- How-kysymys viittaa elementin sisäiseen logiikkaan, sen suorituspolkuihin.

5.1.4 Kysely, Query (Q)

Kun kysymys on tunnistettu, se voidaan muuntaa kyselyksi. Jos kysymykseen liittyy kaksi rajausta, se voi olla esimerkiksi muotoa:

```
Query = why(Start,Target,Options)
```

Jos rajauksia on vain yksi, se on muotoa

```
Query = what(Object)
```

tai

```
Query = how(Object)
```

5.1.5 Systemaattinen vai opportunistinen oppiminen

Käyttäjä voi tehdä kyselyjä systemaattisesti tai opportunistisesti optimalla hakujen määrää.
Systemaattinen haku vie usein liian paljon aikaa, joten käyttäjän ammattitaito tulee esille, kun hänen pyrkii selviytymään ratkaisuun mahdollisimman joustavasti, mutta kuitenkin riittävän kattavasti.

Työkalu voi tukea tiedonhakua mm. tarjoamalla joustavan navigoinnin ja minimoimalla kyselyyn kohdistuvan informaation. Näistä piirteistä kerrotaan seuraavassa.

6 Työkalu JavaMaster

Edellä selostettiin karkealla tasolla kohdealueen metodologia ja työskentelymetodi. Seuraavaksi käydään läpi työkalu nimeltä JavaMaster ja sen peruspiirteitä.

Työkalun sisältämän Visual Prolog- kielisen lähdekoodin koko on noin 40.000 riviä. Se sisältää noin 450 luokkaa ja toimii Windows-ympäristössä.

6.1 Työkalun arkkitehtuuri

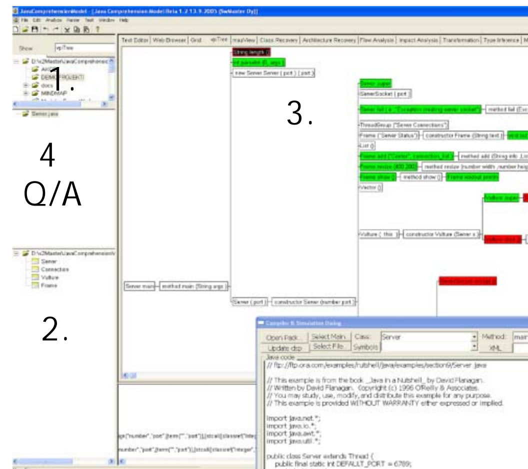
Takaisinmallintamistyökalun arkkitehtuuriksi muodostuu varsin luontevasti Model View Control (MVC) – rakenne, koska lähdekoodista saatu malli muodostaa tärkeimmän osan (Model). Käyttäjä ohjaa työkalua Control-osan kautta saaden aikaan näkymiä (View).
Navigointi tapahtuu näyttöjen kautta tai työkalun päämenuista.

6.2 Työkalun piirteet

JavaMasterin pääpiirteitä ovat: 1) lähdekoodin lataus muistiin ja malliksi, 2) ohjelmamallin tietojen selaus eri välilehtiä hyödyntäen, ja 3) kyselyjen suorittaminen tarkastelun hypoteesien mukaisesti.

Työkalu tarjoaa lukuisia erilaisia tarkastelutapoja. Latausvaiheessa se luo koodista työlistan, joka palvelee lähinnä koodiin perehtyjää, joka sen mukaisesti voi käydä läpi vaativimman aineiston, jota hän voi katsella puiden, taulukkonäyttöjen, riippuvuuskaavioiden, uml-kaavioiden ja xml-dokumenttien muodossa.

Kuva 4 esittää JavaMasterin käyttötilannetta. Koodi ladataan siinä tiedostopuun (kohta 1.) avulla muistiin. Lataus tuottaa puuhun (2.) ohjelmamallin mukaisen hierarkian. Sitten käyttäjä napauttaa Worklist-painiketta, jolloin työkalu näyttää vasteenaan sarjan navigointikysymyksiä (4. Q/A). Navigointitilannetta vastaava informaatio muodostuu eri näyttöihin (3.).



Kuva 4. JavaMasterin käyttöliittymä.

6.3 Työkalun evaluointi

JavaMasterin evaluoimiseksi on useita mahdollisuuksia:

- Koodi ja algoritmit voidaan verifioida analyyttisesti lause lauseelta.
- Työkalun tulokset voidaan evaluoida kokeellisesti tutkien mm. kutsupuiden kattavuus.
- Koodin sisäisiä rakenteita voidaan verrata AST-työkaluihin, joita löytyy mm. Eclipse-työkalusta.
- Käyttötapauksina voidaan tutkia tyypillisiä ongelmanpaikannustilanteita, esimerkiksi tilanteita, joissa tietokantapalvelin toimii väärin. Tälloin selvitetään miten kuvattu ongelmatilanne on purettavissa hypoteeseiksi ja kuinka niitä vastaavat kyselyt on suoritettavissa työkalun avulla.

Työkalun evaluointiprosessi on vielä kesken, mutta sen absoluuttista kapasiteettia on jo evaluoitu syöttämällä työkalun symboliseen malliin miljoonan elementin aineistoja, jotka määrältään vastaavat Microsoftin Word97-ohjelmiston AST-solmujen määrää. Tällöin on todettu, että myös suurten ohjelmistojen analysointi on symbolisella mallilla mahdollista, vaikkakin menetelmän suurimmat edut löytynevät tiettyjen rajattujen ongelmalliseten koodialueiden tarkastelusta monin eri tavoin.

7 Yhteenveto

Esitetty symbolinen malli kuvaa sellaisenaan eräänlaisen virtuaalitoteutuksen, joka abstrahoiduilla rakenteillaan muistuttaa alkuperäistä koodia. Siten se poikkeaa merkittävästi konkreettisista työkaluista, jotka pyrkivät tarjoamaan käyttäjälle aina täydellisen ratkaisun. Abstrahoinnin tueksi Walenstein (2002) on esittänyt, että täydellisen ratkaisun tarjoaminen johtaa käyttäjän ylikuormittumiseen, siksi abstrahointi olisi suositeltavaa. Mutta mikä olisi abstraktin toteutuksen esikuva, onko sellaista esitetty aiemmin?

Kovin monenlaisia erilaisia teorioita abstrakteista simuloitavista koneista ei ole tehty, jos virtaalikoneita ei tässä huomioida. Ehkä tunnetuin niistä on Turingin kone (Kokkarinen 2003), joka käy läpi tulotietona olevaa nauhaa ja ohjausyksiköllään pyrkii reagoimaan nauhan tietoon määrittelyn mukaisesti.

Symbolinen analyysi muistuttaa monin tavoin Turingin konetta ja sen ohjausyksikköä. Symbolinen mallihan sisältää tilakonemaisen ohjausyksikön poimittuaan alkuperäisestä koodista logiikkapolut ja lauseiden semantiikan. Mutta toisin kuin Turingin kone, symbolinen malli sisältää myös formaalin tietoaineiston, joka jäljittelee alkuperäistä koodia.

Suorituksen aikana symbolinen malli tallentaa välitulokset ja lopputulokset perusolioihinsa, joten niiden sisältö palautuu mallin kyselyn kautta käyttäjälle. Siten saatu informaatio kuormittaa ”ohjausnauhaa” eli ihmisen ja koneen välistä kommunikointiväylää mahdollisimman vähän. Sehän oli abstrahoinnin tavoitekin.

Toinen mielenkiintoinen asia on tarkastella, kuinka hyvin saatu konstruktio vastaa semioottista käsitteistöä. Symbolisen mallin pääkäsitteistä voi havaita, että esitetty metodologia sopii suhteellisen hyvin yhteen Peircen käsitteistöön, koska molempiin liittyy kolme pääsuuretta: merkki (sign /symboli), kohde (object) ja tulkinta (interpretator) - sillä tarkennuksella, että tulkintaa symbolisessa mallissa vastaa logiikka, joka on toteutettu mallin elementin sisältämänä predikaattina.

Edellä kuvattu kolmijako muodostui metodologiaan tuntematta Peircen teoriaa vielä suunnitteluvaiheessa, mikä on mielenkiintoinen havainto. Siitä voidaan vetää johtopäätös, että konstruktio pääsi kehittymään oikealla tavalla oikeaan suuntaan jo alusta alkaen.

Kiitokset

SwMaster Oy on mahdollistanut alustavan tutkimuksen ja työkalun kehittämisen vuodesta 2000 alkaen. Sen lisäksi vuoden 2005 alusta Jyväskylän yliopiston COMAS-ohjelma on rahoittanut Erkki Laitilan väitöskirjahankkeeseen liittyvää tutkimusta.

References

Paul Anderson, Thomas Reps, and Tim Teitelbaum. "Design and Implementation of a Fine-Grained Software Inspection Tool", IEEE TSE 29 (8), 2003, pp. 721-733.

Richard Brooks. Towards a Theory of Understanding Computer Programs, IJMS 18 (6), 1983.

Jean-Marie Burkhardt, Francoise Detienne and Susan Wiedenbeck. "Object-Oriented Program Comprehension: Effect of Expertise, Task and Phase", Empirical Softw. Eng. 7 (2), 2002, pp. 115-156.

Gerardo Canfora, A. Cimitile, and U. de Carlini. "A Logic-Based Approach to Reverse Engineering Tools Production", IEEE TSE 18 (12), 1992, pp. 1053-1064.

Thomas Cheatham, Glen Holloway and Judy Townley. "Symbolic Evaluation and the Analysis of Programs", IEEE Trans. Softw. Eng. 5 (4), 1979, pp. 402-417.

Noam Chomsky. Three models for the description of language. IRE Transactions on Information Theory, IT-2:3:113–124, 1956.

CMMI. Capability Maturity Model Integration (CMMI), http://www.sei.cmu.edu/cmmi.

Eclipse. Eclipse Research Community: Eclipse, Open Source Project. http://www.eclipse.org [2003].

David Flanagan. Java Examples in a Nutshell: Example 7-5, ftp://ftp.ora.com/examples/ nutshell/java/examples, O'Reil-ly, 2000.

Frisco. A Framework of information system concepts. http://www.mathematik.uni-marburg.de/~hesse/papers/fri-full.pdf, 1998.

Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst., 12(1):26–60, 1990.

Ilkka Kokkarinen. Tekoäly, laskettavuus ja logiikka, 2003.

Erkki Laitila. Visual Prolog: Teollisuuden Sovellukset, Docendo/Teknolit, Jyväskylä, Finland, 1996, 228 p.

Erkki Laitila. Method for Developing a Translator and a Corresponding System. Patent: W02093371, http://www. espacenet.com, 2001.

Erkki Laitila. "Program Comprehension Theories and Prolog Based Methodologies", Visual Prolog Applic. & Lang. Conf. (VIP-ALC 2006), Prolog Development Center, 2006.

Stan Letovsky. "Cognitive Process in Program Comprehension", Empirical Studies of Programmers: First Workshop, Ablex, 1986, pp. 80-98.

Esko Marjomaa. Theories of Representation, http://cs.joensuu.fi/^marjomaa/tr/tr/doc

Anne-Liese von Mayrhauser and Ann Marie Vans. "Hypothesis-Driven Understanding Processes During Corrective Maintenance of Large Scale Software", Proc. Int. Conf. Softw. Maint. (ICSM 1997), pp. 12-20.

Charles W. Morris. Signs, Language and Behaviour, Prentice-Hall, New York, 1946.

Charles W. Morris. *Writings on the general theory of signs*. The Hague: Mouton (1971).

OMG. UML, Model Driven Architectures, http://omg.org [2005].

Charles S. Peirce. Kotisivut, C.S. Peirce, http://peirce.org, 2006.

Nancy Pennington. "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs", Cognitive Psychology 19 (3), 1987, pp. 295-341.

Prolog Development Center. Visual Prolog 6.3, http://www.visual-prolog.com, 2005.

Prolog Development Center. Visual Prolog Applic. & Lang. Conf. (VIP-ALC 2006), http://www.visual-prolog. com/conference2006, 2006.

Thomas Reps. "Program Analysis via Graph Reachability", ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI'2000).

Tamar Richner and Stand Ducasse. "Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information", Int. Conf. Softw. Maint. (ICSM 1999), IEEE, pp. 13-22.

Spice. Software Process Improvement and Capability Determination. http://www.sqi.gu.edu.au/spice.

Sun. The Grammar of Java Programming Language, http://java.sun.com/docs/books/jls/secondedition/html/syntax.doc.html, 2003.

Turing Machines. http://plato.stanford.edu/entries/turing-machine.

Andrew Walenstein. Cognitive Support in Software Support in Software Engineering tools: A Distributed Cognition Framework, ftp://fas.sfu.ca/pub/cs/theses/2002/AWalensteinPhD.pdf.

Norman Wilde,and Ross Huitt. "Maintenance Support for Object-Oriented Programs", IEEE Trans. Software Eng. 18 (12), 1992, pp. 1038-1044.

Stefano Zacchiroli. Understanding Abstract Syntax Tree, http://www.connettivo.net/article.php3?id_article=49, 2003.