Niko Heikkinen

Koodikloonien havaitseminen ohjelman riippuvuuskaavioiden avulla

Tietotekniikan kandidaatintutkielma

1. kesäkuuta 2016

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Niko Heikkinen

Yhteystiedot: niko.h.t.heikkinen@student.jyu.fi

Työn nimi: Koodikloonien havaitseminen ohjelman riippuvuuskaavioiden avulla

Title in English: Code Clone Detection with Program Dependency Graph

Työ: Kandidaatintutkielma

Sivumäärä: 26+0

Tiivistelmä: Koodiklooni on toisteisena ohjelmassa esiintyvä koodinpätkä. Toisteista ohjelmakoodia on yleisesti pidetty huonona ohjelmointitapana. Tutkielmassa käydään läpi miten koodiklooneja havaitaan ohjelman riippuvuuskaavioiden avulla. Ohjelman riippuvuuskaavio esittää ohjelmassa lauseiden välillä olevia riippuvuuksia. PDG-pohjaisissa menetelmissä käytetään hyväksi näitä riippuvuuksia kloonien havaitsemiseksi. Tämän ansiosta menetelmällä voidaan havaita tyyppi-3:n koodiklooneja.

Avainsanat: koodiklooni, ohjelman riippuvuuskaavio, koodikloonien havaitseminen

Abstract: Code clone is a recurring code fragment in a source code. Code clones are generally considered to be code smell. This thesis goes through how clones can be detected with program dependence graphs. Program dependence graph brings up dependencies between statements. In PDG-based clone detection these dependences are used to find clones. Thanks to this it can detect type-3 code clones.

Keywords: code clone, program dependence graph, code clone detection

Kuviot

Sisältö

1 Johdanto

Ohjelmoitaessa tulee usein vastaan tilanteita, joissa huomaa ohjelmassa olevan jo koodinpätkän, joka sopisi toiseenkin kohtaan. Tässä tilanteessa kyseinen pätkä usein kopioidaan ja liitetään toiseen kohtaan. Tämän jälkeen tarkistetaan, että ohjelma toimii vielä toistaiseksi ja jatketaan ohjelmakoodin kirjoittamista ajatellen miten näppärä ratkaisu ongelmaan löytyikään. Kopioiminen ja liittäminen eivät vaadi paljoa vaivaa ja koodia tulee lisää useita rivejä kerralla ja näitä saatetaan toistaa useamman kerran. Lopulta ohjelma alkaa näyttää valmiilta, mutta sen alussa huomataan pieni virhe, joka saattaisi aiheuttaa ongelmia jatkossa. Virhe korjataan ja ollaan tyytyväisiä, sillä se huomattiin ajoissa. Enää ei kuitenkaan muisteta, että tätä virheellistä koodia oli kopioitu ja liitetty useamman kerran ohjelmoinnin aikana.

Koodiklooneihin ei tule välttämättä kiinnitettyä huomiota ohjelmointivaiheessa. Sen sijaan jos ylläpitovaiheessa joutuu käsittelemään kloonipitoista koodia, niin voi se aiheuttaa melko suurtakin vaivaa, jos ohjelmisto on suuri (Baker 1995). Näin voi ohjelmoitaessa aloittaa suurenkin ongelman projektilleen olemalla tietämätön, jos kloonattavasssa koodinpätkässä on alun alkaen ollut virhe. Voisiko virheen korjaaminen ja estäminen olla helpompaa, jos tämän kloonirakenteen havaitsee ajoissa? Olisiko lähdekoodi ja sen rakenne helpommin ymmärrettävää ilman turhia klooneja? Voiko havaituista klooneista hyötyä muilla tavoin?

Tutkimuksen aiheena on koodikloonien havaitseminen käyttäen ohjelman riippuvuuskaaviota (eng.*program dependence graph*) eli lyhennettynä PDG:ta. Koodikloonien havaitsemiseen on useita algoritmeja ja menetelmiä ja jokaisella havaitsemismenetelmällä on omat vahvuutensa ja heikkoutensa (Bellon et al. 2007). Aiheen rajaamiseksi keskitytään tutkimuksessa vain yhteen näistä menetelmistä eli PDG:n avulla tehtävään kloonien etsintään.

Tutkimuskysymyksenä on miten koodiklooneja havaitaan PDG:n avulla ja miten menetelmällä havaituilla klooneilla voidaan parantaa ohjelman toimintaa ja ylläpitoa. PDG:ssä kiinnostavaa on varsinkin miten se havaitsee semanttisia klooneja ja

miten tämän avulla voidaan parantaa ohjelman laatua ja estää mahdollisia klooneista syntyviä ongelmia.

Tutkimus on systemaattinen kirjallisuuskatsaus. Koodiklooneja on tutkittu melko paljon ja aiheesta löytyy paljon lähdekirjallisuutta. Aiheen käsittelylukuja ovat luvut 2-4. Luvussa 2 käsitellään koodiklooneja yleisesti ja millaisia vaikutuksia niillä on havaittu olevan ja miten niihin suhtaudutaan. Luvussa 3 käsitellään ohjelman riippuvuuskaaviota ja katsastetaan millainen se on. Luvussa 4 käsitellään koodikloonien havaitsemista ohjelman riippuvuuskaavioiden avulla ja muutamia PDG-pohjaisia työkaluja. Lisäksi luvussa käsitellään, millaisia klooneja menetelmällä havaitaan ja miten niillä voi hyötyä.

2 Koodiklooni

Luvussa käsitellään käsitettä koodiklooni ja tuodaan esille sen määritelmää, syntymisen syitä ja aiheuttamia seurauksia. Lisäksi tarkastellaan hieman millaisia menetelmiä kloonien havaitsemiseksi on kehitelty.

2.1 Koodikloonin määritelmä

Ohjelmistoja kehittäessä kehittäjät turvautuvat usein turvalliseen kopioi ja liitä-periaatteeseen toistuvan ongelman tullessa vastaan. Vanha jo aikaisemmassa ongelmassa hyväksi todettu ja kerran keksitty ratkaisu pistetään uusiokäyttöön. Toisinaan ratkaisua saatetaan hieman muokata tehtävään sopivaksi, mutta usein lopulta alunperin kopioitu ja liitetty koodinpätkä jää suurilta osin toiminnallisuudeltaan ja ulkonäöltään samankaltaiseksi kuin alkuperäinen.

Koodikloonit ovat samankaltaisia koodinpätkiä tietyssä ohjelmassa, jotka tietyn määritelmän mukaan ovat samankaltaisia. Samankaltaisuuden määrittelyn perusteita ovat muun muassa yhdenmukaisuus merkeiltään, toisiaan muistuttava leksikaalinen tai syntaktinen rakenne sekä koodin toiminnallinen samankaltaisuus. Samankaltaisuutena voidaan myös pitää sitä, että koodissa on käytetty eri ohjelmakoodin sisältämissä instansseissa samaa kaavaa tai tiettyä tyyliä, joka ilmenee samankaltaisina rakenteina ja ratkaisuina (Koschke 2007).

Ohjelmistoissa esiintyvät koodikloonit luokitellaan yleensä neljään eri tyyppiluokkaan (Patil et al. 2015). Tyyppi-1:n kloonit ovat täysin identtisiä klooneja, joita ei ole muokattu lainkaan alkuperäisestä. Tyyppi-2 on toiminnaltaan identtinen ja syntaktisesti samanlainen kuin alkuperäinen. Tyyppi-2:n erot tyyppi-1:een voivat olla esim. uudelleen nimetyt muuttujat. Tyyppi-3:n kloonit ovat kuin tyyppi-1:n tai -2:n, mutta ne sisältävät uusia tai poistettuja lauseita. Tyyppi-3 klooneja kutsutaan usein aukollisiksi klooneiksi, jotka kattavat myös ei-peräkkäiset kloonit (engl. *non-contiguous clone*). Tyyppi-4:n kloonit on koodinpätkiä, jotka omaavat samanlaisen toiminnallisuuden, mutta ovat syntaktisesti poikkeavia toisistaan. Näitä kutsutaan

yleensä semanttisiksi klooneiksi.

Tarkaa määritelmää koodiklooneille ei varsinaisesti ole ja koodinpätkien pitäminen kloonikanditaatteina riippuu yleensä arvioijasta. Koodikloonien määritelmästä ei ole saavutettu yhteisymmärrystä, edes tehtäväkohtaisella tasolla (Koschke 2007). Klooneja on määritelty ulkoisten ja toiminnallisten ominaisuuksien mukaan useaan eri ryhmään, mutta yksinkertaista määritystä pelkälle koodikloonille ei ole (Roy ja Cordy 2007).

2.2 Kloonien syntymisen syitä

Klooneja syntyy usein, kun olemassa olevaa koodia kopioidaan ja liitetään ohjelmassa toistuvasti tulevien ongelmien ratkaisuksi. Monesti ohjelmoinnissa käytetty ohjelmointikieli tuottaa myös omia rajoitteitaan, jotka voivat johtaa ohjelmassa esiintyvän koodin kopiointiin. Usein ohjelmoijat kuitenkin viivyttelevät koodin uudelleen muotoilua viimeiseen asti ja tähän mennessä on ehditty kopioida ja liittää useita pätkiä, eikä toistoja huomata. Monesti ohjelmoijat käyttävät tiettyä koodinpätkää kehyksenä, jota muokataan tilanteen vaatimalla tavalla. Usein myöskään ei haluta käyttää aikaa mahdollisten seurausten ja ongelmien miettimiseen (Koschke 2007).

Ohjelmistoprojekteissa on usein kiireellinen aikataulu ja resursseista saattaa olla pulaa (Roy ja Cordy 2007). Aikataulujen luoman paineen alla kehittäjät saattavat joutua kopioimaan vanhaa koodiaan ja käyttämään sitä suoraan muuttumattomana. Lisäksi ohjelmistokehityksessä tehtyä työtä arvioidaan usein tehtyjen koodirivien perusteella. Kopioimalla ja käyttämällä samaa koodia uudestaan saadaan nopeasti luotua lisärivejä ja näin yleisesti käytettyjen edistymistä mittaavien mittarien mukaan edistystä.

Suurissa ohjelmistoissa on löydetty toistuvia kloonauksen malleja. Näitä ovat haarautuminen (engl *forking*), mallintaminen (engl. *templating*) ja kustomointi (engl *customization*) (Kapser ja Godfrey 2006). Haarautumisessa koodia kopioidaan siinä uskossa, että se tulee kehittymään erilaiseksi, mutta usean muokkauksenkaan jälkeen

ei omaa mitään alkuperäisestä poikkeavaa. Mallintamisessa kopioidaan suoraan ja abstraktio ei ole kyseisessä pätkässä millään tasolla mahdollista. Kustomoinnissa olemassa olevaa koodia käytetään ratkaisemaan uusia ongelmia.

Toisinaan koodin kloonausta käytetään tarkoituksellisesti kehittämisstrategiana (Roy ja Cordy 2007). Vanhan koodin hyvänä puolena on, että se on usein testattu ja todettu toimivaksi. Näin voidaan pitää riskinä luoda uutta koodia. Kriittisissä ohjelmissa, joissa ohjelman on toimittava tilanteessa kuin tilanteessa saatetaan suosia kloonien käyttöä. Näin voi olla varma siitä, että vaikka ohjelmisto osittain pettäisi, voisivat yksittäiset komponentit toimia itsenäisesti ilman riippuvuutta alkuperäiseen koodin instanssiin.

2.3 Kloonien seurauksia

Koodiklooneista ollaan yleisesti oltu melko kiinnostuneita. Niitä on luokiteltu ja määritelty ja niiden havaitsemiseksi on kehitetty useita menetelmiä. Mutta miksi kaikki tämä vaiva on nähty niiden takia? Onko jokin tarve havaita koodikloone-ja ohjelmistoissa? Halutaanko ne havaita, koska niistä on haittaa ohjelmistoille vai onko tähän jotain muita syitä?

Ohjelmistojen ylläpidon kannalta koodiklooneista on hieman vaihtelevia mielipiteitä. Epäjohdonmukaiset koodikloonit aiheuttavat suurta vaivaa ohjelmistoissa, jos niitä ei huomioida tarpeeksi kehitysvaiheessa, sekä aiheuttavat paljon virheitä, kun niitä muokataan tahattoman epäjohdonmukaisesti (Juergens et al. 2009). Sen sijaan Göde ja Koschke ovat tutkimuksessaan huomanneet, että erittäin harva koodiklooni joutuu tahattoman epäjohdonmukaisesti muokatuksi ja, että erittäin harva koodiklooneista joutuu elämänsä aikana edes muokatuksi (Göde ja Koschke 2011). Eli klooneihin ei pääse usein syntymään virheitä tahattomien muokkauksien johdosta. Sen sijaan Göde ja Koschke (2011) myöntävät, että kloonit haittaavat ylläpitoa siltä osin, että ne kasvattavat ohjelmiston kokoa ja tekevät siitä vaikeaselkoisemman.

2.4 Koodikloonien havaitseminen

Koodikloonien havaitsemisen tutkimiseen on käytetty jo useampia vuosia aikaa ja tähän tarkoitukseen on kehitetty useampia menetelmiä (Roy ja Cordy 2007). Tämä osoittaa sen, että yleisesti kiinnostusta klooneihin löytyy ja on nähty tarpeelliseksi kehitellä menetelmiä niiden havaitsemiseksi. Menetelmät eroavat toisistaan toimintaperiaatteiltaan ja havaittavien kloonien osalta. Tekniikoita joita käytetään koodikloonien havaitsemisessa ovat teksti-, token-, puu-, mittari- ja PDG-pohjaiset menetelmät.

Tekstipohjaisissa menetelmissä havaitseminen perustuu puhtaasti tekstin ja merkkijonojen tutkimiseen. Tämä johtaa siihen, että menetelmässä on useita ongelmia. Muun muassa yksittäisten merkkien ja rivinvaihtojen erot koodinpätkissä aiheuttavat menetelmälle jo ongelmia kloonien havaitsemisessa (Roy ja Cordy 2007). Näin ollen havaittavat kloonit ovat syntaktisesti samankaltaisia ja ovat niin kutsuttuja tyyppi-1 klooneja.

Token-pohjaisissa menetelmissä lähdekoodi muutetaaan sarjaksi tokeneita ja tästä lähdetään etsimään samankaltaisia tokenien muodostamia alasarjoja (Roy ja Cordy 2007). Menetelmä on varmempi ja kyvykkäämpi kuin tekstipohjaiset menetelmät, sillä se ei välitä pienistä merkkien muutoksista.

Puupohjaisisa menetelmissä ohjelma jäsennetään jäsennyspuuksi tai abstraktiksi syntaksipuuksi (engl. *abstract syntax tree*) halutulla kielen jäsentimellä (Patil et al. 2015). Puupohjaisilla menetelmillä lähdetään etsimään puusta samanlaisia alipuita ja näin havaitsemaan klooneja. Puupohjaisen menetelmän etuja on se, että kloonien havaitsemiseen eivät pääse vaikuttamaan eri tunnisteiden nimet, vaan tutkitaan enemmänkin toiminnallisuutta ja syntaksia.

Mittari- eli metrispohjaiset menetelmät ovat laajalti käytetty kloonikoodien havaitsemiseen. Algoritmi perustuu siihen, että ohjelmakoodin kappaleista tuotetaan mittoja ja vertaillaan niitä keskenään. Menetelmä on suorituskykyisempi kuin puu- ja PDG-pohjaiset menetelmät, mutta epätarkempi havaitsemaan klooneja (Avetisyan et al. 2015).

PDG-pohjaisissa menetelmissä käytetään nimensä mukaisesti PDG:tä havaitsemisen apuna. Ohjelmasta tehdään PDG:tä ja klooneja lähdetään havaitsemaan näiden avulla. Tästä kerrotaan lisää ja tarkemmin tulevissa luvuissa.

3 Ohjelman riippuvuuskaavio

Ohjelmien esittämiseksi on kehitelty useita erilaisia kuvauksia ja malleja. Ohjelman riippuvuuskaavio (engl. *program dependence graph*) eli PDG on eräs ohjelman esittämistapa. PDG:n avulla esitetään ohjelmassa esiintyviä riippuvuuksia. Riippuvuuksia joita PDG tuo näkyville, ovat data- ja kontrolliriippuvuudet, jotka syntyvät ja ilmenevät ohjelman operaatioiden, kuten lauseiden ja ehdollisten predikaattien, välillä (Ferrante, Ottenstein ja Warren 1987).

Datariippuvuuksia syntyy ohjelmien sisällä lauseiden välille muun muassa muuttujien välityksellä. Näin käy, kun kahdessa lauseessa käytetään samaa muuttujaa, ja jos lauseet käännettäisiin toisinpäin suoritusjärjestykseltään, niin tuloksesta tulisi väärä (Ferrante, Ottenstein ja Warren 1987).

Listaus 3.1: Datariippuvuustilanne.

```
x = v * z #L1
y = z + x #L2
```

Listauksessa 3.1 olevassa tilanteessa lause L2 on riippuva lauseesta L1, sillä muuttujan x arvoa muutetaan L1:ssä ja tätä arvoa käytetään L2:ssa. Jos L2 suoritettaisiin ennen L1:tä, niin L2 ei saisi käyttöönsä muuttujan x oikeaa arvoa ja muuttujan arvosta ei tulisi sitä, mitä oli alunperin ajateltu.

Kontrolliriippuvuus syntyy, kun toinen lause kontrolloi arvollaan toisen lauseen suoritusta (Ferrante, Ottenstein ja Warren 1987).

Listaus 3.2: Kontrolliriippuvuustilanne.

```
if(x): #L3
  y = v #L4
```

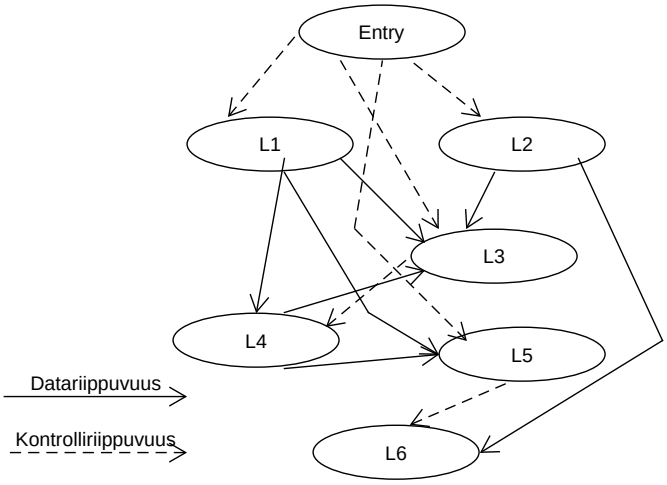
Listauksessa 3.2 olevassa tilanteeessa lauseen L4 suoritus on täysin riippuvainen lauseessa L3 esiintyvästä predikaatista. Tämä tulee määräämään, tullaanko lausetta

L4 koskaan suorittamaan. Eli lauseiden välillä on kontrolliriippuvuus.

PDG tuo esille nämä edellä mainitut riippuvuudet. Tämän ansiosta siitä pystyy näkemään ohjelman rakenteen ja datan virtauksen kätevästi (Krinke 2001). PDG:n graafisessa esityksessä lauseet merkitään solmuina ja riippuvuudet kaarina (Gabel, Jiang ja Su 2008). Kuviossa 1 oleva PDG on esimerkki kaaviomaisesta PDG esityksestä. Kaavioon merkitään yleensä sisäänkäyntisolmu, joka vastaa metodin alkua. Näin ollen kaikki muut lauseet ovat kontrolliriippuvuussuhteessa tämän kanssa, sillä muiden lauseiden suoritus riippuu siitä tullaanko koko metodia suorittamaan (Hotta, Higo ja Kusumoto 2012).

Listaus 3.3: Koodiesimerkki, jossa esiintyy riippuvuuksia.

```
x = 0 #L1
y= 5 #L2
while(x<y): #L3
   x= x+3 #L4
if(x == 0):#L5
   print(x>y) #L6
```



Kuvio 1: Listaus 3.3 koodia vastaava PDG.

PDG käyttämisessä ohjelmien esittämiseksi on monia syitä. Se antaa ohjelmista abstraktimman kuvan. Lisäksi PDG:llä on olemassa monenlaisia sovelluksia, joissa

niitä voidaan hyödyntää. PDG:n avulla voidaan ohjelmista havaita toisteisuutta ja mahdollisia solmun halkaisemistilanteita, nähdä ohjelman liike ja muokata sitä hallitusti, yhdistellä silmukoita ja käyttää viipaloinnissa (Ferrante, Ottenstein ja Warren 1987). Näiden sovellusten takia PDG:tä voikin käyttää hyväksi optimoinnissa ja ohjelmistokehityksessä.

4 Koodikloonien havaitseminen ohjelman riippuvuuskaavioiden avulla

Ohjelman riippuvuuskaavioiden hyväksikäyttäminen soveltuu hyvin koodikloonien tunnistukseen. PDG tuo esille ohjelmassa olevat riippuvuudet operaatioiden välillä, joten se antaa ohjelmasta abstraktin kuvan ja tuo koodista esille sen semanttiset yksityiskohdat (Patil et al. 2015). Tämän ansiosta menetelmällä voi myös havaita semanttisia klooneja.

Luvussa käydään läpi miten PDG-pohjaisen kloonien havaitsemismenetelmien perusalgoritmit toimivat. Tämän jälkeen tutustutaan hieman uudempaan ja jalostetumpaan työkaluun, jonka jälkeen arvioidaan, millaisia klooneja loppujen lopuksi menetelmillä havaitaan ja miten näillä hyödytään.

4.1 Perusalgoritmit

Krinken ja Komondoorin ja Horwitzin kehittelemiä algoritmeja pidetään yleisesti PDG-pohjaisina perusalgoritmeina. Useissa vertailussa ja tutkimuksissa on käytetty esimerkkeinä näitä tekniikoita. Vaikka tekniikoilla on hieman ikää ja parannuksiakin on kehitelty, niin perusperiaatteet ovat pysyneet samoina kuin mitä näissäkin algoritmeissa.

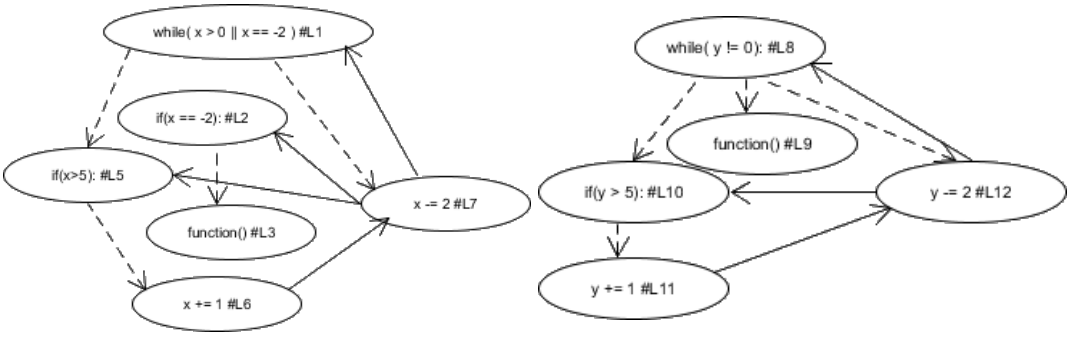
4.1.1 Komondoorin ja Horwitzin algoritmi

Komondoor ja Horwitz (2001) selittävät tekniikkaansa, jota pidetään yleisesti PDG-pohjaisen koodikloonien havaitsemisen perustekniikkana. Tässä algoritmissa ensin jokainen ohjelman sisältämä proseduuri esitetään PDG:een avulla. Tämän jälkeen aletaan toistamaan kolmea askelta, jotka ovat klooniparien etsintä, sisällytettyjen kloonien poisto ja klooniparien yhdistäminen suuremmiksi ryhmiksi.

Klooniparien etsinnässä kaikki PDG:een solmut luokitellaan luokkiin syntaktisen rakenteen perusteella (Komondoor ja Horwitz 2001) . Kaksi solmua samassa luo-

kassa ovat niin kutsuttuja yhteensopivia solmuja. Seuraavaksi jokaiselle yhteensopivalle solmuparille (r1,r2) etsitään isomorfiset PDG alikuvaajat, jotka sisältävät r1:n ja r2:n.

Näiden alikuvaajien etsintä aloitetaan taaksepäin suuntautuneella viipaloinnilla (Komondoor ja Horwitz 2001). Aloittaen solmuista r1 ja r2 viipaloidaan taaksepäin sulkuaskeleeseen (engl *lock step*), lisäten edeltäjäsolmu ja yhdistävä kaari viipaleeseen, jos ja vain jos vastaava yhteensopiva edeltäjä on solmun yhteensopivassa parissa. Eteenpäin viipalointia myös käytetään, kun yhteensopivissa silmukkapari- tai if-predikaatti-tilanteissa lisätään niiden kontrolliriippuvuusseuraajat viipaleisiin (Komondoor ja Horwitz 2001). Huomioitavaa tässä on kuitenkin, että eteenpäin viipalointi tehdään vain yhteensopiville predikaateille.



(a) Esimerkki PDG.

(b) Esimerkki PDG.

Kuvio 2: Esimerkki kahdesta samankaltaisuuksia sisältävän koodin PDG:stä.

Kuviossa 2 yhteensopiva solmupari löytyisi solmuista L7 ja L12. Nämä solmut kuvaavat syntaktisesti samanlaisia lauseita. Näistä solmuista aletaan käyttämään taaksepäin suuntautunutta viipalointia yhteensopivien lisäsolmujen etsinnässä. Solmuparin (L7,L12) solmuihin tulee kumpaankin yksi datariippuvuus ja yksi kontrolliriippuvuus, joita voidaan hyväksikäyttää taaksepäin suuntautuneessa viipaloinnissa. Solmuihin saapuvat kontrolliriippuvuudet L1:stä ja L8:sta. Nämä solmut ovat kuitenkin syntaktisesti erilaiset eikä niitä voi lisätä osaviipaleisiin. Sen sijaan L11 ja L6, joista saapuu datariippuvuus ovat syntaktisesti samanlaiset, joten ne voidaan lisätä näihin osaviipaleisiin. Nyt kun kaikki solmuparit on vertailtu aloitussolmuista, niin voidaan lähteä viipaloimaan osaviipaleisiin lisätyistä solmuista. Näistä löyde-

tään yhteensopivat solmut L5 ja L10 tulevan kontrolliriippuvuuden kautta. Koska L5 ja L10 ovat if-predikaatteja, tulisi niissä käyttää myös eteenpäin suuntautunutta viipalointia eli lähtevien riippuvuuksien viipalointia. Tässä tapauksessa solmut L6 ja L11 on jo lisätty osaviipaleeseen, eikä lisää solmuja löydy. Näin ollen kuviossa 2 löydettiin toisiaan vastaavat isomorfiset alikuvaajat (L5,L6,L7) ja (L10,L11,L12).

Kun tämä on ohi, on kaksi isomorfista alikuvaajaa tunnistettu. Näiden alikuvaajien etsimistä jatketaan siihen asti, kunnes kaikki ohjelmassa olevat isomorfiset alikuvaajaparit on löydetty.

Tämän jälkeen joukosta poistetaan sisällytetyt kloonit (Komondoor ja Horwitz 2001). Yhteensopivia solmupareja löytyy myös ymmärrettävästi alikuvaajien sisältä, eikä ole mielekästä esittää niitä omina klooneinaan.

Lopuksi parit yhdistetään kloonijoukoiksi. Jokainen joukko sisältää tietyn samalla tavalla toistuneen koodin toistuman ohjelmasta (Komondoor ja Horwitz 2001). Esimerkiksi, jos on löydetty klooniparit (A1,A2) ja (A2,A3), niin yhdistetään nämä parit isommaksi kloonijoukoksi (A1,A2,A3). Nämä joukot edustavat löydettyjä toisteisia koodiklooneja eli tietyn yhden kloonikanditaatin instansseja.

Menetelmällä havaitaan ei-peräkkäisiä, uudelleenjärjesteltyjä ja toisiinsa kietoutuneita klooneja (Komondoor ja Horwitz 2001). Tällaisten kloonien havaitseminen onnistuu, sillä riippuvuudet säilyvät lauseiden välillä, vaikka väleihin lisättäisiin uusia rivejä tai rivejä poistettaisiin. Nämä kloonit kaipaavat yleensä poistoa ja muokkausta, sillä ne tekevät ohjelmasta vaikeaselkoisemman ymmärtää.

Komondoor ja Horwitz (2001) mainitsevat menetelmän heikkoudeksi sen, ettei eteenpäin suuntautunutta viipalointia tehdä aina ja hitauden. Tämän takia kaikkia mahdollisia klooneja ei havaita. Algoritmi havaitsee myös heidän mielestään liikaa klooneja, sillä kaikki toistuvat rakenteet ohjelmissa eivät aina ole klooneja. Algoritmi on raskas suorittaa ja tämän takia melko hidas, eikä sovellu varsinkaan suurien ohjelmien tutkimiseen.

4.1.2 Krinken algoritmi

Krinke (2001) esittelee algoritmin, jossa normaalia PDG:tä on hieman muokattu. Algoritmissa perinteistä PDG:tä on yhdistelty abstraktin syntaksipuun kanssa ja tätä yhdistelmää kutsutaan pienijyväiseksi PDG:ksi (engl. *fine-grained program dependence graph*). Pienijyväinen PDG ei sisällä normaalin PDG:n tapaan solmuja vaan niin kutsuttuja kärkiä. Lisäksi se sisältää hieman erilaisia riippuvuuksia kuin tavallinen.

Pienijyväisen PDG:een kärjet esittävät ohjelman lauseiden sisältämiä komponentteja (Krinke 2001). Erityiskärkiä ovat muuttujat ja proseduurit. Ne ovat yleensä merkittävimpiä ja useimmat riippuvuudet syntyvät niiden pohjalle. Kärjet voivat lisäksi sisältää muitakin asioita, kuten vakioita ja muita lauseiden osia, esimerkiksi viittauksia tai operaatioita.

Kuviossa 3 esitetty yksinkertainen pienijyväinen PDG tuo esille muutamia kärkiä. Formal-in ja formal-out ovat muuttujien alustuksia. Formal-in-muuttujat ovat yleensä ohjelman ulkopuolella määriteltyjä eli esimerkiksi funktion parametreina saamiaan arvoja. Formal-out-muuttujat ja proseduurit on määritelty funktion sisällä. Compound- ja assignment-kärki kuvaavat ohjelmassa olevia proseduureja. Tässä esimerkissä compound-kärjellä viitataan ohjelmassa tapahtuviin muuttujen arvojen käyttämiseen ja assignment-kärjellä suoritettavaan lauseen logiikkaan, joka tässä on a+a ja sijoitus muuttujaan z. Binäärioperaatio + on merkitty omaksi kärjeksi. Kärjiksi merkityt reference:t ovat merkkaamassa kussakin kohtaa käytettyä viittausta alkuperäisestä muuttujasta.

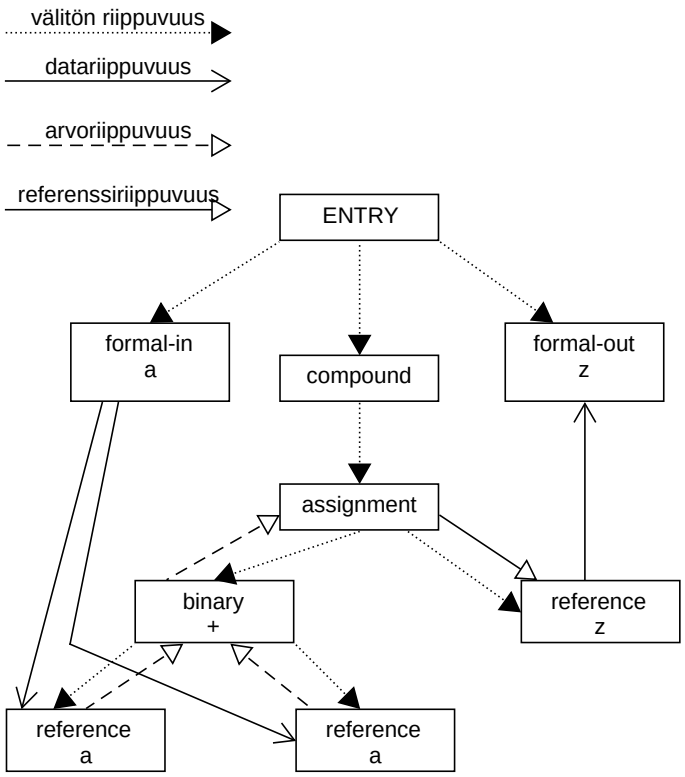
Kärkien välissä on myös muutamia normaalista poikkeavia riippuvuuksia (Krinke 2001). Datariippuvuus on tässä versiossa samankaltainen kuin normaalissa PDG:ssä. Kontrolliriippuvuudet ovat melko pitkälti samankaltaisia kuin normaalissa, mutta niitä kutsutaan välittömiksi riippuvuuksiksi. Välittömällä riippuvuudella tarkoitetaan, että tällaiset riippuvuussuhteet syntyvät jo ennen kuin varsinainen koodi on ajettu. Uusia esille tuotavia riippuvuuksia ovat viittausriippuvuudet ja arvoriippuvuudet. Arvoriippuvuus tuo esille lauseiden komponenttien välistä datariippu-

vuutta. Viittausriippuvuus tuo esiin minne ohjelmassa laskettu arvo sijoitetaan.

Kuviossa 3 välittömät kontrolliriippuvuudet ovat syntyneet funktion sisäänkäynnin, muuttujien ja pääproseduurien välille. Datariippuviiksia on muuttujien määrittelyn ja viittausten välillä. Arvoriippuvuuksia on päässyt syntymään lauseiden komponenttien välille, kuten binäärioperaation ja a-viittauksien kanssa. Viittausriippuvuus on syntynyt, koska assignment:n laskema arvo sijoitetaan z:aan.

Listaus 4.1: Yksinkertainen koodinpätkä.

```
def f(a):
   z= a+a
```



Kuvio 3: Listausta 4.1 vastaava pienijyväinen PDG.

Pienijyväinen PDG on attribuuttisuunnattu kaavio, missä kärjen attribuutteja voivat olla luokka, operaattori ja kärjen arvo ja kaaren attribuutteja ovat luokka ja kaaren kuvaama riippuvuus (Krinke 2001). Tämän ansiosta kaaviota voi käydä läpi ver-

taamalla kärkien ja kaarien attribuutteja. Kaksi alikuvaajaa ovat isomorfisia silloin, kun jokainen kaari, joka on bijektiivisesti yhteensopiva toisen kaavion kaaren kanssa ja kaarien ja kärkien attribuutit vastaavat toisiaan (Krinke 2001). Tämän määritelmän avulla voidaan etsiä ohjelmasta isomorfisia alikuvaajia ja samalla toisteista ohjelmakoodia.

Kloonien etsiminen tapahtuu etsimällä isomorfisia alikuvaajia (Krinke 2001). Algoritmin alussa otetaan kaksi yhteensopivaa kärkeä. Päätepisteiden polun pituus on aluksi nolla, eli alikuvaajiin ei ole aluksi otettu vielä yhtään kaaria tai kärkiä mukaan. Lisäämällä kaaria polkujen pituus kasvaa. Aluksi lähdetään tutkimaan kaaria, joissa on samat attribuutit. Jos kummastakin alkukärjestä lähtee kaaret, joissa on yhtenevät attribuutit, niin lisätään nämä alikuvaajiin. Seuraavaksi tutkitaan löytyykö kaarien päistä yhteensopivia kärkiä ja lisätään sopivat kärjet alikuvaajiin. Tätä toistetaan kunnes ei voida enää kasvattaa alikuvaajien kokoa uusilla kaarilla ja kärjillä. Tämän seurauksena on löydetty mahdollisimman suuret isomorfiset alikuvaajat eli mahdollisimman pitkät polut. Tätä etsimistä tehdään koko ohjelmalle, kunnes kaikki yhtenevät kloonikanditaatit on havaittu.

Menetelmällä havaitaan mahdollisimman suuret alikuvaajat, jotka voidaan löytää ohjelmakoodista riippuvuuskaavioiden avulla (Krinke 2001). Tämä poikkeaa Komondoorin ja Horwitzin algoritmista, jossa jokaisesta proseduurista oli tehty oma alikuvaajansa. Tämän ansiosta menetelmällä voi havaita hieman pidempiä ja hieman erilaisempia klooneja.

Tämäkin algoritmi on myös melko hidas. PDG:eiden luominen ja vertailu on raskasta ja aikaa vievää (Krinke 2001). Tämän takia tämäkään menetelmä ei sovellu suuriin ohjelmistoihin. Menetelmä jättää myös havaitsematta paljon peräkkäisiä klooneja (Bellon et al. 2007).

4.2 PDG-pohjainen uudempi työkalu Scorpio

Higo ja Kusumoto (2009) kertovat kehittämästään PDG:tä hyväksikäyttävästä koodikloonien havaitsemistyökalusta Scorpio:sta. Työkalu käyttää hyväkseen kloonien

havaitsemisessa heidän esittämiään tekniikoita, jotka he ovat kehittäneet tehostaakseen PDG-pohjaisen havaitsemismenetelmän tehokkuutta. Nämä parannukset paikkaavat erityisesti PDG-pohjaisten tekniikoiden erästä suurimmista heikkouksista.

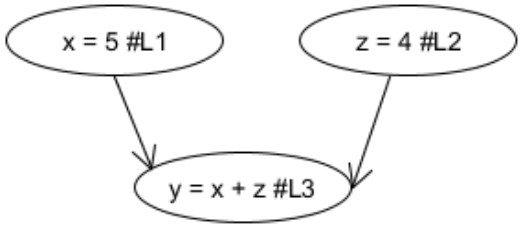
Scorpio on työkalu, joka tukee monisäieajoa ja näin ollen käyttää tehokkaasti hyväkseen moniydinprosesessorien täyttä laskentatehoa (Higo ja Kusumoto 2009). Tällä ollaan yritetty tehostaa menetelmän suoritusnopeutta. Työkalussa on monia mahdollisuuksia valita millaisia toisteisia koodinpätkiä pitäisi havaita koodiklooneina (Higo ja Kusumoto 2009) . Esimerkiksi havaittavien kloonien minimikoko voidaan asettaa kloonin PDG-esityksen sisältämien solmujen määrän avulla. Lisäksi parametrisointi voidaan erikseen konfiguroida monella eri tapaa.

PDG-pohjaisten perusalgoritmien eräänä suurimpana heikkoutena mainittiin jo peräkkäisten kloonien havaitseminen. Tämä on hankalaa normaalilla PDG:llä, sillä se vaatii tutkittavien koodirivien välille joko data- tai kontrolliriippuvuutta. Higo ja Kusumoto (2009) esittävät näiden ongelmien takia kaksi tekniikkaa, joita ovat suoritusriippuvuuksien ja edestakaisen viipaloinnin hyväksikäyttö.

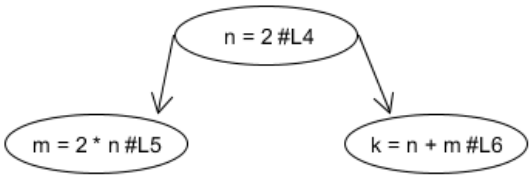
Listaus 4.2: Lauseita joiden välillä ei ole data- tai kontrolliriippuvuuksia.

```
x = 0              #L1
y= 5               #L2
function()         #L3
2nd_function()     #L4
```

Suoritusriippuvuus on tehty täydentämään PDG:n kuvaamia riippuvuuksia (Higo ja Kusumoto 2009) . Suoritusriippuvuus syntyy kahden solmun välille PDG:ssä, jos seuraava solmu tullaan suorittamaan heti sen jälkeen, kun toisessa lauseessa oleva solmu on suoritettu. Riippuvuutta käytetään, koska normaalisti PDG-pohjainen tekniikka ei havaitse peräkkäisiä koodiklooneja, koska kahden peräkkäisen lauseen välillä ei useinkaan ole data- tai kontrolliriippuvuutta. Tämän ansiosta listauksen 4.2 kaltaisina rakenteena esiintyviä toisteisia koodinpätkiä voidaan havaita klooneina.



(a) Esimerkki PDG.



(b) Esimerkki PDG.

Kuvio 4: Tilanteet, joiden läpikäynti erilaista eri viipalointimenetelmillä.

Perusalgoritmeissa koodikloonien vertailu suoritettiin pääasiassa käyttäen taaksepäin viipalointia ja vain joidenkin predikaattien tapauksessa käytettiin hieman eteenpäin viipalointia. Scorpiossa käytetään jatkuvasti kumpaakin eteen- sekä taaksepäin suuntautuvaa viipaloinitia (Higo ja Kusumoto 2009). Tämän avulla voidaan havaita entistä enemmän klooneja, sillä kaikki kloonit, joita havaitaan eteenpäin eivät välttämättä näy taaksepäin viipaloinnilla.

Kuviossa 4 on kaksi PDG:tä, jotka havainnollistavat viipalointien eroja. 4a kaaviossa saadaan käytyä kaikki solmut läpi taaksepäin viipaloinnilla, kun lähdetään solmusta L3 liikkeelle. Sen sijaan eteenpäin suuntautuneella viipaloinnilla voidaan havaita vain L1:n ja L3:n tai L2:n ja L3 :n luomat alikuvaajat. 4b kaaviossa sen sijaan taaksepäin suuntautuvalla viipaloinnilla saadaan käytyä läpi vain tilanteet L5 ja L4 tai L6 ja L4. Sen sijaan eteenpäin suuntautunut viipalointi onnistuu käymään kaikki solmut läpi.

Vaikka edellä mainitut tekniikat parantavat PDG-pohjaisen tekniikan kloonin havaitsemiskykyä, niin ei se tule kuitenkaan ilmaiseksi, sillä se tekee havaitsemisesta raskaampaa ja täten enemmän aikaa vaativaa (Higo ja Kusumoto 2009). Parannus-

ten avulla saadaan kuitenkin havaittua entistä enemmän kiinnostavia ja mahdollisesti haitallisia koodiklooneja.

4.3 Havaitut kloonit

Komondoorin ja Horwitzin algoritmilla havaitaan ei-peräkkäisiä, uudelleenjärjesteltyjä ja toisiinsa kietoutuneita klooneja. Tämän tyyppiset kloonit ovat yleensä hyviä kanditaatteja poistettaviksi klooneiksi (Komondoor ja Horwitz 2001). Tällaiset kloonit ovat toiminnaltaan samankaltaisia ja ne voidaan korvata lähdekoodissa yleensä luomalla niitä vastaava proseduuri eli funktio, metodi tai aliohjelma ja kutsumalla tätä niiden tilalla. Tästä voi olla hyötyä myös miettiessä uusia kirjastokanditaatteja (Basit et al. 2015).

Tiivistämällä kloonit proseduurikutsuiksi edesautetaan samalla ohjelmiston ylläpitoa. Ohjelmakoodi lyhenee ja yhdenmukaistuu siitä, että alunperin on sama proseduuri tehty hieman toisistaan poikkeavilla tyyppi-3:n klooneilla. Tämä auttaa ylläpidossa, kun ohjelmakoodi on myös selkeämpää lukea ja se vie vähemmän kovalevytilaa (Patil et al. 2015).

Krinken algoritmilla havaitaan mahdollisimman suuria alikuvaajia ja täten mahdollisimman suuria klooneja. Göde ja Harder (2011) kertovat, että suuret kloonit ovat yleensä epävakaampia kuin pienet kloonit. Suuret kloonit vaativat todennäköisemmin jälkeenpäin muokkaamista kuin pienet. Näin ollen kaikki mahdollisesti suuremmat kloonit, joita Krinken algoritmilla havaitaan, ovat hyödyksi ohjelmistojen vakaudelle.

Göde ja Harder (2011) mainitsevat myös, että tyyppi-1:n kloonit ovat yleensä vähemmän vakaampia kuin tyyppi-2 ja tyyppi-3. Koska Scorpiossa on panostettu peräkkäisten eli näiden tyyppi-1:n kloonien havaitsemiseen, niin mahdollisesti poistamalla nämä kloonit saadaan ohjelmista vakaampia.

5 Yhteenveto

Kanditaatintutkielmassa tutkittiin miten PDG:n avulla havaitaan koodiklooneja ja miten näistä havaituista klooneista voidaan hyötyä. Aluksi luvussa 2 selvitettiin, mikä koodiklooni ylipäänsä on ja miten niihin on yleisesti suhtauduttu. Luvussa 3 tutustuttiin paremmin PDG:hen. Luvussa 4 katsastettiin PDG-pohjaisia havaitsemisalgoritmeja ja sitä millaisia klooneja näillä havaittiin. PDG-pohjaisia algoritmeja on olemassa useampia, mutta perusidea kaikissa oli sama: luotiin ohjelmasta PDG:t ja alettiin etsimään näiden pohjalta isomorfisia alikuvaajia käymällä solmuja läpi riippuvuuksien välityksellä.

Menetelmillä havaittiin yleensä ei-peräkkäisiä klooneja. Tämä onnistui siksi, koska riippuvuudet säilyivät yleensä muuttumattomina, vaikka rivejä lisättiin tai poistettiin ohjelmakoodin seasta. Lisäksi parannuksia omaavalla uudemmalla tekniikalla havaittiin myös peräkkäisiä klooneja, jolloin menetelmä alkoi havaita klooneja jo kattavasti.

Menetelmän yleinen heikkous kaikilla algoritmeilla oli hitaus. Ohjelman riippuvuuskaavioiden luominen ja vertailu jokaisen proseduurin osalta on työlästä. Tämän takia menetelmä ei sovellu suurien ohjelmien tutkimiseen.

Havaituilla klooneilla voidaan parantaa ylläpidettävyyttä ja vakautta. Perusalgoritmeilla havaitut kloonit ovat yleensä tyyppi-3:n klooneja ja näitä poistamalla saadaan ohjelmakoodia yksinkertaistettua ja lyhennettyä. Suurehkot kloonit, joita Krinken algoritmilla havaittiin ja peräkkäiset kloonit, joita havaittiin Scorpiolla, voivat yleisesti havaittuina ja käsiteltyinä parantaa ohjelman vakautta.

Tulevaisuudessa voisi tutkia paremmin, miten PDG-pohjaisia menetelmiä voisi tehdä tehokkaammiksi eri kloonien havaitsemisessa tai millaisia parannuksia algoritmit voisivat tarvita. Varsinkin suorittamisajat tulisi saada järkevämmäksi, että menetelmiä voisi käyttää suuremmissakin ohjelmistoissa.

Kirjallisuusluettelo

Avetisyan, A., S. Kurmangaleev, S. Sargsyan, M. Arutunian ja A. Belevantsev. 2015. "LLVM-based code clone detection framework". Teoksessa *Computer Science and Information Technologies (CSIT), 2015*, 100–104. Syyskuu. doi:10.1109/CSITechnol.2015.7358259.

Baker, B. S. 1995. "On finding duplication and near-duplication in large software systems". Teoksessa *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*, 86–95. ID: 1.

Basit, Hamid Abdul, Muhammad Hammad, Stan Jarzabek ja Rainer Koschke. 2015. "What do we need to know about clones? deriving information needs from user goals". Teoksessa *Software Clones (IWSC), 2015 IEEE 9th International Workshop on*, 51–57. IEEE.

Bellon, S., R. Koschke, G. Antoniol, J. Krinke ja E. Merlo. 2007. "Comparison and Evaluation of Clone Detection Tools". ID: 1, *Software Engineering, IEEE Transactions on* 33 (9): 577–591.

Ferrante, Jeanne, Karl J Ottenstein ja Joe D Warren. 1987. "The program dependence graph and its use in optimization". *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9 (3): 319–349.

Gabel, Mark, Lingxiao Jiang ja Zhendong Su. 2008. "Scalable detection of semantic clones". Teoksessa *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, 321–330. IEEE.

Göde, Nils, ja Jan Harder. 2011. "Clone stability". Teoksessa *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, 65–74. IEEE.

Göde, Nils, ja Rainer Koschke. 2011. "Frequency and risks of changes to clones". Teoksessa *Proceedings of the 33rd International Conference on Software Engineering*, 311–320. ACM.

Higo, Yoshiki, ja Shinji Kusumoto. 2009. “Enhancing quality of code clone detection with program dependency graph”. Teoksessa *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, 315–316. IEEE.

Hotta, Keisuke, Yoshiki Higo ja Shinji Kusumoto. 2012. “Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph”. Teoksessa *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, 53–62. IEEE.

Juergens, E., F. Deissenboeck, B. Hummel ja S. Wagner. 2009. “Do code clones matter?” Teoksessa *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, 485–495. Toukokuu. doi:10.1109/ICSE.2009.5070547.

Kapser, C., ja M. W. Godfrey. 2006. “"Cloning Considered Harmful"Considered Harmful”. Teoksessa *Reverse Engineering, 2006. WCRE '06. 13th Working Conference on*, 19–28. Lokakuu. doi:10.1109/WCRE.2006.1.

Komondoor, Raghavan, ja Susan Horwitz. 2001. “Using slicing to identify duplication in source code”. Teoksessa *Static Analysis*, 40–56. Springer.

Koschke, Rainer. 2007. “Survey of research on software clones”. Teoksessa *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Krinke, Jens. 2001. “Identifying similar code with program dependence graphs”. Teoksessa *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, 301–309. IEEE.

Patil, R.V., S.D. Joshi, S.V. Shinde, D.A. Ajagekar ja S.D. Bankar. 2015. “Code clone detection using decentralized architecture and code reduction”. Teoksessa *Pervasive Computing (ICPC), 2015 International Conference on*, 1–6. Tammikuu. doi:10.1109/PERVASIVE.2015.7087126.

Roy, Chanchal Kumar, ja James R Cordy. 2007. *A survey on software clone detection research*. Tekninen raportti. Technical Report 541, Queen's University at Kingston.