TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

TANELI PULKKINEN

LÄHDEKOODIN AUTOMAATTISEN KORJAUKSEN MENETELMÄT

Kandidaatintyö

Tarkastaja: professori Petri Ihantola
Tarkastaja ja aihe ei vielä hyväksytty

TIIVISTELMÄ

TANELI PULKKINEN: Lähdekoodin automaattisen korjauksen menetelmät
Tampereen teknillinen yliopisto
Kandidaatintyö, 18 sivua, 3 liitesivua
Toukokuu 2017
Tietotekniikan kandidaatin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Petri Ihantola

Avainsanat: Ohjelmointi, virheet, korjaus, testaus, automatiikka, synteesi, geneettinen ohjelmointi

Tässä työssä tutustutaan erilaisiin automaattiseen ohjelmakoodiin virheiden korjauksen työkaluihin, sekä yleistetään niiden taustalla olevia toimintamenetelmiä. Tavoitteena on selvittää, millaisia menetelmiä automaattisessa virheenkorjauksessa on kahden viime vuoden aikana käytetty. Lisäksi tutkitaan, kuinka suosittuja eri menetelmät ovat.

Työ on kirjallisuuskatsaus, jossa käytiin läpi 16 vuodesta 2015 alkaen julkaistua työkalua. Jokaisen työkalun käyttämään toimintatapaan tutustuttiin, ja ne ryhmiteltiin kuuteen toisistaan eroavaan korjausmenetelmään. Näiden menetelmien yleisyyttä tutkittujen työkalujen joukossa vertailtiin toisiinsa. Myös menetelmien yhdistelyä työkaluissa selvitettiin.

Kuudesta määritellystä korjausmenetelmästä yleisimmät ovat korjaussynteesi ja geneettinen ohjelmointi. Näistä ensimmäistä käytti kahdeksan työkalua ja toista viisi. Muut korjaustavat työssä esiteltävät menetelmät ovat symbolinen suoritus, kehityshistorian louhinta, koodinsiirto ja valmiit korjausmallit. Yksittäinen korjaustyökalu käyttää joko yhtä tai kahta menetelmää toiminnassaan. Yleisiksi menetelmien rajat ylittäviksi piirteiksi korjausohjelmissa havaittiin yleiskäyttöisyys ja testeihin perustuva toiminta.

Korjaukset jaetaan usein semanttisiin ja syntaktisiin. Jakoa käytettiin myös tässä työssä, minkä lisäksi kehitettiin toinen kahtiajako korjatun ohjelmakoodin lähteen perusteella. Uudessa jaossa korjausmenetelmä on joko koodia luova tai koodia hakeva. Määritellyt menetelmät asetettiin näistä kahdesta jaosta luotuun nelikenttään. Lopputuloksena on helposti omaksuttava jäsennys menetelmien suurista eroista.

ABSTRACT

Taneli Pulkkinen: Automatic source code repair methods
Tampere University of Technology
Bachelor of Science Thesis, 18 pages, 3 Appendix pages
May 2017
Bachelor's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Professor Petri Ihantola

Keywords: Programming, errors, repair, testing, automation, synthesis, genetic programming

This thesis showcases several tools for automatic program repair, and generalizes the methods behind them. The goal was to find out what methods have been used in automatic program repair in the past two years. The popularity of each method was also studied.

This paper is a literature review, going through 16 tools published since the year 2015. The way each tool works was studied, and their approaches were grouped into six general repair methods. The prevalence of each method was compared to the others. How the tools combine different methods was examined as well.

Of the six methods defined, the most popular ones are repair synthesis and genetic programming. The former was used by eight repair tools and the latter by five. The other four methods are symbolic execution, history mining, code transfer, and premade repair templates. Each tool uses one or two repair methods. General principles shared between methods are universality and test-based operation.

Repairs are often divided into semantic and syntactic. This division is used in this thesis, but another dichotomy was created as well, based on the source of the repaired code. In this new bipartition, repair methods are either code generating or code fetching. The defined methods were split into four categories using these two divisions. The result is an easy to digest presentation of the key differences between methods.

ALKUSANAT

Tämä kandidaatintyö on laadittu lukuvuonna 2016-2017 Tampereen Teknillisellä Yliopistolla. Haluan kiittää työn ohjaajaa Petri Ihantolaa, tietotekniikan laitosta, sekä perhettäni, tyttöystävääni ja ystäviäni. Tukenne ja neuvonne ovat auttaneet merkittävästi tämän työn valmistumisessa.

Tampereella, 18.5.2017

Taneli Pulkkinen

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

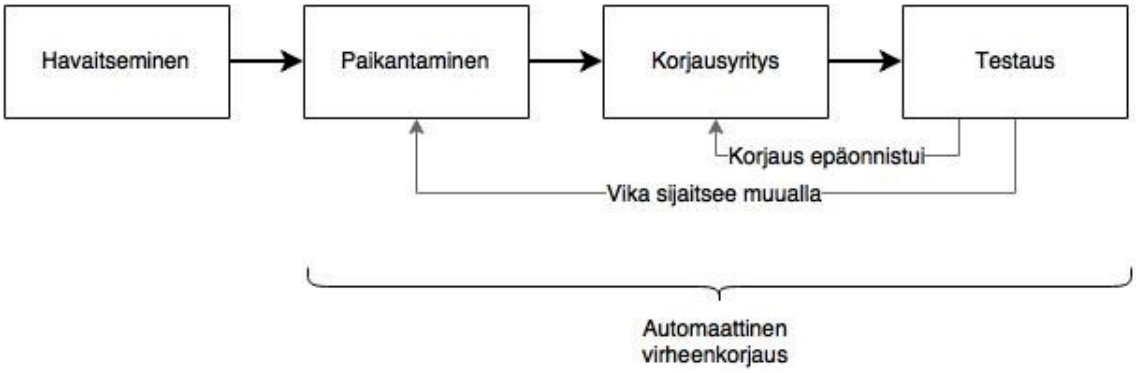
paikkaus	engl. patch, muutostiedosto. Muutokset, jotka korjaavat ohjelmavian.
korjausmenetelmä	engl. repair method. Sarja toimenpiteitä, jotka suorittamalla korjausohjelma luo paikkauksia.
generoida	engl. generate, luoda, tuottaa.
hakuavaruus	engl. search space. Alue, jolta jotakin etsitään. Esimerkiksi kokoelma ohjelmakoodia.
kandidaatti	engl. candidate, ehdokas. Kandidaatti pyrkii saavuttamaan jotain, mutta sen onnistumista ei ole varmistettu.
validointi	engl validation, varmistus. Asian oikeellisuuden tarkistus.
negatiivinen testitapaus	engl. negative test case. Ohjelmatesti, jossa syöte on virheellinen. Testataan ohjelman toimintaa normaalitilanteessa.
positiivinen testitapaus	engl. positive test case. Ohjelmatesti, jossa syöte on virheetön. Testataan ohjelman toimintaa ongelmatilanteessa.
mutaatio	engl. mutation. Lievä muokkaus ohjelmaan. Tavoitteena ohjelmavirheen korjaus.
mutantti	engl. mutant. Lievästi muokattu versio ohjelmasta. Yleensä näitä luodaan suuri määrä, joista valitaan parhaat.
iteraatio	engl. iteration. Jonkin asian toisto, sama asia tehdään useita kertoja peräkkäin.
synteesi	engl. synthesis. Korjauksen rakentaminen keinotekoisesti tai yhdistämällä osia toisiinsa.
suorituspolku	engl. program path. Reitti ohjelman läpi. Jos ohjelmassa on ehtolauseita, suoritus voi kulkea sen läpi useampaa eri reittiä.
TTY	Tampereen teknillinen yliopisto

1. JOHDANTO

Ohjelmakoodin virheet ovat perinteisesti olleet merkittävä ongelma ohjelmistoprojekteissa. Tietokoneohjelman virheellä tarkoitetaan ohjelman käyttäytymistä, joka poikkeaa suunnitellusta. Parhaista yrityksistä huolimatta virheitä syntyy paljon, ja niiden korjaaminen on kallista. Yhteensä virheenkorjaukseen käytetään alalla miljardeja dollareita vuodessa [1]. Mitä myöhemmin virheet löydetään, sitä enemmän kuluja niistä syntyy. Jo pelkkä virheiden olemassaolon tunnistaminen vaatii monipuolisia työkaluja sekä huolellista ja aikaa vievää työskentelyä. Tämän lisäksi virheiden havainnointia seuraava ongelman paikallistaminen ja korjaaminen voi kuluttaa ohjelmiston valmistuksessa lähes puolet ohjelmoijien koko työajasta [2]. Ohjelmistovirheet tuovat kuluja myös ohjelmien julkaisun jälkeen, sillä olemassa olevien järjestelmien viankorjaus on ylläpidon merkittävin kulunlähde. Näin virheet vaikuttavat kustannuksiin tuotteen koko elinkaaren ajan [3]. Tehostamalla virheenkorjausta voitaisiin siis saavuttaa mittavia säästöjä.

Automaattinen korjaaminen poistaa ohjelmakoodista löydettyjä virheitä ohjelmallisesti. Sen tavoitteena on tukea ja tehostaa ohjelmoijien työskentelyä sekä parantaa ohjelmien laatua. Lisäksi se on askel kohti automaattista ohjelmointia, eli ihmisohjelmoijista kokonaan luopumista [4]. Ohjelmallinen korjaus välttämättä vaatii, että korjaustyökalulle täytyy kertoa miten korjattavan ohjelman kuuluisi toimia. Tämä tieto voidaan välittää useilla eri tavoilla, mutta nykyiset korjaustyökalut suosivat ylivoimaisesti ohjelman omia yksikkötestejä. Ohjelman oletetaan siis toimivan halutulla tavalla, jos se läpäisee kaikki testinsä.

Ohjelmakoodin korjauksessa on neljä päävaihetta, jotka ovat virheen havaitseminen, virheen paikantaminen, korjauksen luonti, ja korjauksen testaaminen. Nämä vaiheet on esitetty alla kuvassa 1. Lähtökohtaisesti vaiheet käydään tässä järjestyksessä läpi. Hyvin usein ensimmäinen korjausyritys ei kuitenkaan ole täydellinen, joten testauksesta palaaminen uuden korjauksen luontiin on tyypillistä. Toisinaan myöskään virheen paikantaminen ei onnistu, jolloin useamman epäonnistuneen korjauksen jälkeen yritetään paikannusta uudelleen.



Kuva 1. *Ohjelmistovirheen korjauksen vaiheet*

Nykyiset automaatiotyökalut keskittyvät kolmeen viimeiseen vaiheeseen, tosin testaus suoritetaan tavallisilla ohjelmakoodin testausmenetelmillä, jolloin korjaustyökalun vastuulla on lähinnä testauksen käynnistys. Keskimmäiset kaksi vaihetta sen sijaan ovat automaattisen korjauksen ydintä. Niiden toteutukseen on monia eri tapoja, joita eri työkalut käyttävät. Korjauksen luonti on erityisen laaja vaihe, ja siinä esiintyy suurin vaihtelu. Täten on aiheellista kysyä, millaisia menetelmiä näiden vaiheiden suorituksessa käytetään. Korjausmenetelmällä tarkoitetaan sarjaa toimenpiteitä, jotka suorittamalla korjausohjelma luo paikkauksia. Tässä tutkimuksessa keskitytään menetelmiin, joita useat eri työkalut suorittavat samankaltaisina.

Virheiden automaattinen korjaaminen on viime vuosina kehittynyt erittäin aktiivisesti. Vanhemmissa korjausohjelmissa oli kapeasti rajatut käyttötilanteet, ja ne pystyivät korjaamaan vain muutamia erilaisia virheitä [4; 5]. Viime aikoina rajoitukset ovat kuitenkin keventyneet, ja työkalujen yleiskäyttöisyys sekä skaalautuvuus ovat yhä parempia [3]. Nykyään tarjolla on laaja valikoima työkaluja, joiden toiminta pohjautuu eri lähestymistapoihin. Ne eroavat niin tuettujen ohjelmointikielien kuin käytettyjen menetelmien osalta. Alan tulevaisuuden kannalta on tärkeää ymmärtää nykytila ja tähän mennessä kehitettyjen menetelmien vahvuudet ja heikkoudet.

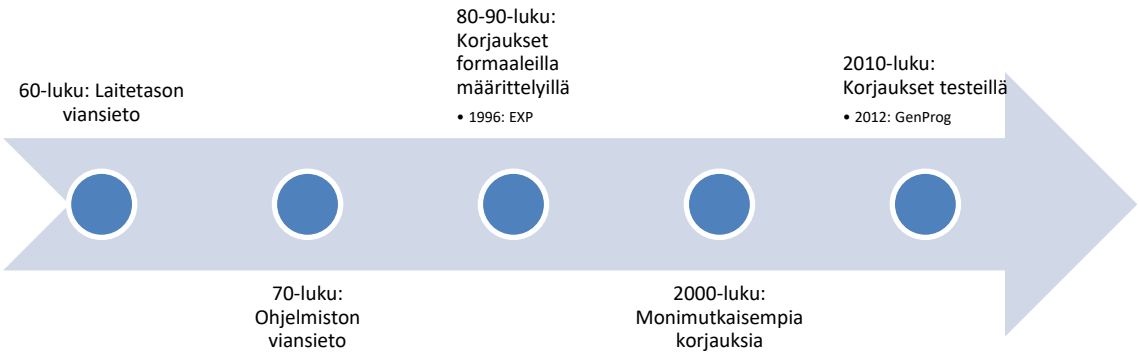
Tämän työn tarkoituksena on vastata kysymykseen: ”Millaisia eri menetelmiä ohjelmakoodin automaattiseen korjaukseen nykyään käytetään?” Lisäksi perehdytään eri menetelmien yleisyyteen.

Alan tutkimuksen suuresta määrästä ja nopeasta kehityksestä johtuen tässä työssä tutkitaan ainoastaan vuodesta 2015 alkaen englanninkielisissä julkaisuissa esiteltyjä korjaustyökaluja. Alan kehityksen kuvausta varten käytiin läpi myös noin 10 historiallisesti merkittävää vanhempaa artikkelia. Tuoreidenkin julkaisujen runsauden vuoksi tutkimus ei ole kattava. Tiedonhaku toteutettiin seuraavasti: tietokannoista IEEE Xplore ja Google Scholar haettiin sanoja ’automated program repair’, rajaten tulokset vuodesta 2015 alkaen julkaistuihin teksteihin. Tuloksia tarkasteltiin järjestyksessä noin 30 kummassakin tietokannassa, ja niistä kerättiin kaikki jotka esittelevät uuden työkalun tai vertailevat laajasti

aiempia työkaluja. Työkaluja esittelevät artikkelit luettiin kokonaan läpi, jotta saatiin selkeä käsitys työkalujen todellisista toimintatavoista. Vanhoja julkaisuja etsittiin samalla menetelmällä ilman aikarajausta, sekä käymällä läpi uusien julkaisujen lähteitä. Alan termistö ei ole täysin vakiintunut, joten samoille konsepteille käytettiin eri artikkeleissa toisinaan eri termejä.

Seuraavassa luvussa kuvataan ohjelmakoodin automaattisen korjauksen historiaa ja esitellään esimerkkejä työkaluista eri vuosikymmeniltä. Tämän jälkeen luvussa 3 esitellään tutkitut työkalut. Luvussa 4 kuvataan työkaluista löydetyt korjausmenetelmät. Lopuksi yhteenvedossa tiivistetään työn tulokset ja tarkastellaan menetelmiä suhteessa toisiinsa.

2. AUTOMAATTISEN KORJAUKSEN KEHITYS



Kuva 2. *Automaattisen korjauksen historia*

Automaattisen virheenkorjauksen vanhimmat juuret löytyvät 60-luvulta, jolloin tutkittiin tietokoneen oikean toiminnan jatkamista laitetason vaurioista huolimatta [6; 7]. Tilanne tällöin poikkesi merkittävästi nykyisestä, sillä keskeisenä ongelmanlähteenä nähtiin laitteisto ohjelmiston sijaan. Lisäksi tavoitteena oli saada tietokone toimimaan viasta huolimatta, eikä vian korjaus. Tämä on ymmärrettävää, sillä fyysisten komponenttien luotettavuus oli nykyistä huonompi, niiden korjaaminen täysin automaattisesti ei nykyäänkään ole mahdollista, ja ohjelmistot olivat pieniä ja yksinkertaisia. Ohjelmistovirheiden käsittelyyn ei vielä ollut automaattisia työkaluja.

Ohjelmiston virheensietoa eli oikean toiminnan jatkamista virheistä huolimatta alettiin tutkia 70-luvulla [7]. Komponenttien luotettavuuden paraneminen sekä ohjelmien koon ja monimutkaisuuden kasvu tekivät ohjelmistopuolen ongelmista yhä tärkeämpiä suhteessa laitteistoon. Tavoitteena ei tässä vaiheessa ollut virheiden automaattinen poistaminen ohjelmakoodista, vaan ohjelman toimivuuden parantaminen virheistä huolimatta ja niiden vaikutusten paikkaaminen suorituksen aikana. Tähän pyrittiin lisäämällä koodiin osia, jotka tunnistivat suorituksen saapuneen virheelliseen tilaan ja palauttivat ohjelman tilaan, jonka tiedettiin olevan virheetön. Tämän jälkeen viallinen toimenpide voitaisiin mahdollisesti suorittaa vaihtoehtoisella tavalla, jolloin virhe kierretään. Ohjelmistovirheitä ei kuitenkaan voitu havaita tai paikantaa automaattisesti.

80- ja 90-luvuilla korjaustyökalut tarkistivat ohjelman toiminnan oikeellisuuden lähes yksinomaan invarianteilla ja muilla formaaleilla määrittelyillä. Määrittelyjen vahvuutena on täsmällisyys, mutta merkittävänä heikkoutena niiden kirjoitus aiheuttaa runsaasti lisätyötä. Ohjelmatestejä sen sijaan käytetään vähintään osittaisena ohjelmiston määrittelynä käytännössä jokaisessa ohjelmistoprojektissa. Korjaustyökalu, joka vaatii toimiakseen vain ohjelman omat testit, voidaan siis ottaa käyttöön helpommin.

Testitapauksiin pohjautuva automaattinen ohjelman korjaus kehitettiin vuonna 1996, jolloin konseptia tutkittiin tarkoitusta varten kehitetyssä yksinkertaisessa funktionaalisessa kielessä EXP [8]. Korjaustyökalu osasi jo paikantaa virheen, luoda siihen korjauksen ja testata korjatun ohjelman toimivuuden suorittamalla alkuperäiset yksikkötestit. Korjaus suoritettiin etsimällä yksi viallinen komento ja tekemällä siihen pieni muutos. Työkalu hidastui voimakkaasti korjattavan ohjelman koon kasvaessa eikä osannut käsitellä esimerkiksi päättymättömiä silmukoita, mutta vastasi perustoiminnaltaan selvästi nykyisiä työkaluja. Menetelmän yleisenä haittapuolena puutteet testien kattavuudessa johtavat vajaaseen määrittelyyn, jolloin korjausohjelma ei voi tietää, miten ohjelman kuuluu todella toimia. Tämän vuoksi automaattisia korjauksia käytettäessä on tärkeää luoda testit, jotka kattavat mahdollisimman monta eri suorituspolkua eli reittiä ohjelman läpi.

Alan tutkimuksen määrä kasvoi merkittävästi 2000-luvun ensimmäisellä vuosikymmenellä. Vuonna 2004 julkaistiin menetelmä, jolla voidaan esi- ja jälkiehtojen pohjalta paikallistaa ja korjata virheellinen ohjelma [9]. Aiemmin kertaalleen korjatun ongelman paikkaaminen automaattisesti tehtiin mahdolliseksi vuonna 2005 [10]. Vuonna 2008 esiteltiin työkalu sekä ohjelman että sen yksikkötestien korjaamiseen rinnakkain [11]. Käyttäjän täytyi kuitenkin yhä tarjota korjattavan ohjelman formaali spesifikaatio.

Modernien korjaustyökalujen esikuva on vuonna 2012 julkaistu GenProg [12]. Se on ensimmäinen työkalu, joka ei vaadi muita syötteitä kuin viallisen ohjelman ja sen yksikkötestit, ja jonka geneerinen lähestymistapa voi korjata laajasti eri tyyppien virheitä. Myöhemmin kuitenkin valitettavasti havaittiin, että GenProg tuottaa huomattavasti vähemmän onnistuneita korjauksia kuin sen kehittäjät väittivät [13]. Tästä huolimatta GenProg toimi lähtöpisteenä koko nykyiselle automaattisen ohjelmistokorjauksen alalle. Tällä hetkellä sen esittelyyn on viitattu Google Scholarissa ainakin 264 kertaa, ja sen julkaisusta alkaen uusien työkalujen määrä on kasvanut räjähdysmäisesti.

3. TYÖKALUT

Tutkimusta varten analysoitiin yhteensä 16 työkalua. Tarkasteltavana ovat kaikki vuodesta 2015 alkaen julkaistut työkalut, jotka esiteltiin kirjallisuuskatsauksen kohteena olleissa artikkeleissa. Työkalut on lueteltu alla, taulukossa 1. Seuraavassa osiossa esitellään jokainen työkalu erikseen.

Työkalun nimi	**Julkaisuvuosi**
Angelix [14]	2016
Bach ja Xuan* [15]	2016
CDRep [16]	2016
DynaMoth [17]	2016
Genesis [18]	2016
HDRepair [19]	2016
Prophet [20]	2016
Code Phage [21]	2015
DirectFix [22]	2015
Kali [13]	2015
Leon [23]	2015
PBRepair [24]	2015
PCR [25]	2015
relifix [26]	2015
SearchRepair [27]	2015
SPR [28]	2015

Taulukko 1. *Tarkastellut työkalut.*

*Työkalun kehittäjät Bach ja Xuan eivät antaneet sille nimeä.

Kunkin työkalun toiminnan vaiheet kuvataan kappaleen mittaisena tiivistelmänä. Ellei asiasta erikseen mainita, työkalu pitää korjattavan ohjelman testejä sen määrittelynä. Mikäli paikattu ohjelma läpäisee kaikki testit, sen katsotaan olevan korjattu.

3.1 Työkalujen kuvaukset

Angelix [14] tutkii korjattavan ohjelman toimintaa korvaamalla todennäköisesti viallisia komentoja symboleilla. Muuttamalla näiden symbolien arvoja työkalu paikallistaa virheet ja niihin tarvittavat muutokset. Käytetty menetelmä on hyvin skaalautuva, joten sitä voidaan käyttää suuriinkin ohjelmiin.

Bach ja Xuan yhdistivät työkalussaan [15] useita toimintatapoja. Työkalu vaatii testien sijaan korjattavan ohjelman funktionaalisen määrittelyn, esimerkiksi esi- ja jälkiehdot. Korjaus yritetään luoda samanaikaisesti sekä analysoimalla poikkeamia ohjelman määrittelystä, että satunnaisesti muuttamalla viallisia komentoja. Mikäli kumpikaan korjaus ei onnistu välittömästi, parhaiksi arvioituja paikkauksia muokataan ja yhdistellään toisiinsa. Tätä toistetaan, kunnes virhe on korjattu tai korjaaminen katkaistaan.

CDRep [16] tarjoaa valmiit korjausmallit seitsemälle kryptograafiselle heikkoudelle Android-sovelluksissa. Mallit on luotu korjaamalla virheitä manuaalisesti useissa eri sovelluksissa ja tutkimalla tehtyjä muutoksia. Korjausten yhteisten piirteiden pohjalta yleistettiin kunkin virheen vaatimat muokkaukset. Työkalu soveltuu rajattuihin tilanteisiin, mutta toimii niissä tehokkaasti.

DynaMoth [17] on synteesimoottori Nopol-työkaluun [29]. Nopol selvittää, missä ehtolauseissa ohjelman virheet ilmenevät. DynaMoth puolestaan kerää testien suorituksen aikana kontekstitietoa, kuten muuttujien arvot ja eri metodien paluuarvot. Tietojen pohjalta luodaan korjaus, jolla konteksti ja suorituksen kulku ohjelman läpi vastaavat toivottua.

Genesis [18] kerää avoimen lähdekoodin projekteista ihmisten tekemiä korjattavaa tilannetta vastaavien virheiden paikkauksia. Näistä abstrahoidaan yleiskäyttöisiä paikkausmalleja, jotka sovelletaan alkuperäiseen lähdekoodiin. Täten generoidaan joukko mahdollisia korjauksia, joiden seasta lopullinen ratkaisu etsitään. Tämä tehdään keräämällä paikkauskandidaatit, jotka läpäisevät kaikki korjattavan ohjelman yksikkötestit.

HDRepair [19] käy läpi monia ihmisten tekemiä korjauksia ja käyttää niitä perustana paikkauksien luonnissa sekä niiden laadun arvioinnissa. Se mutatoi korjattavasta ohjelmasta muokattuja versioita. Parhaista paikkauskandidaateista luodaan iteratiivisesti uusia versioita. Niiden toimivuutta ei määritetä pelkästään annettujen ohjelmatestien perusteella, vaan lisäksi tarkastetaan kuinka hyvin paikkaus vastaa ihmisten luomia korjauksia. Työkalu rakentaa joukon paikkauksia jotka läpäisevät kaikki yksikkötestit, ja antaa ihmisen valita niistä parhaan.

Prophet [20] kehittää ihmisten kirjoittamien korjausten pohjalta mallin oikeaoppisesta ohjelmakoodista ja vertaa luomiaan korjauskandidaatteja tähän malliin. Kandidaateista validoidaan laadukkaimmat korjaukset ensin, mikä lyhentää suoritusaikaa kokonaisuutena. Korjauskandidaatit luodaan SPR-työkalun (alla) menetelmillä.

Code Phage [21] etsii tietokannastaan ehjiä ohjelmia, jotka läpäisevät kaikki viallisen ohjelman mukana annetut testit. Tutkimalla eri testien suorituspolkuja se etsii jostakin ehjästä ohjelmasta ehtolauseen, joka käsittelee onnistuneesti vian aiheuttavan syötteen. Tämän ehtolauseen sijoittamista testataan viallisen ohjelman eri kohdissa. Tarvittaessa iteroidaan eri vaiheita.

DirectFix [22] eroaa muista työkaluista yhdistämällä vian etsinnän ja korjauksen luonnin samaan suoritusvaiheeseen. DirectFix luo viallisesta ohjelmasta polkukaavion, etsii siitä korjauksen loogiset ehdot, poistaa tarvittaessa rakenteen rajoitteita ja palauttaa korjausta kuvaavan mallin. Tavoitteena on löytää yksinkertaisin, eli pienin, mahdollinen korjaus. Suorituksen päättää korjausmallin yhdistäminen alkuperäiseen ohjelmaan, mikä korjaa vian.

Kali [13] etsii lähdekoodista komennot, joita suoritetaan eniten negatiivisissa testitapauksissa, ja luo vian hakuavaruuden. Avaruus käydään läpi yrittäen seuraavia muutoksia komennoille: käskyn poistaminen, return-komennon asetus sen eteen, tai suorituksen ohjaaminen toiselle polulle (mikäli kyseessä on haarakohta ohjelmassa). Rajoitettujen muutostyyppien lisäksi Kalin luomat korjaukset tekevät ohjelmaan ainoastaan yhden muutoksen. Korjaus onnistuu siis ainoastaan, jos ohjelmassa on yksi tarpeeton komento tai osio.

Leon [23] luo ohjelman esi- ja jälkiehdoista testitapauksia, jotka suorittamalla se löytää vikoja ohjelman toiminnassa. Vian oletetaan olevan alueella, jonka kaikki epäonnistuvat testit suorittavat. Tämä alue korvataan ”ohjelmareiällä”, jonka tilalle syntetisoidaan uusi ohjelman osa. Aiempaa toteutusta käytetään synteesiä ohjaavana vihjeenä, siten että korjaus luodaan testaamalla erilaisia alkuperäisen toteutuksen muokkauksia.

PBRepair [24] tarvitsee lähtötiedoksi virheen sijainnin. Suorituksessa työkalu luo yksi kerrallaan symbolisia vastaesimerkkejä, joissa ohjelman suoritus epäonnistuu. Vastaesimerkeistä muodostetaan synteesiongelma, joka ratkaisemalla ohjelma päivitetään niin, ettei yksikään tähän mennessä löydetyistä vastaesimerkeistä voi toteutua. Tätä iteroidaan, kunnes yhtäkään uutta vastaesimerkkiä ei löydetä.

PCR [25] etsii potentiaaliset vialliset komennot ja käyttää valmista muokkausoperaatioiden listaa luodakseen korjauskandidaatteja. Kandidaatteja ei luoda satunnaisesti, vaan synteesillä. Ensin tutkitaan, saadaanko suorituspolkuja muuttamalla negatiiviset testitapaukset tuottamaan oikeita tuloksia. Jos tämä onnistuu, luodaan uusi symbolinen ehto, jolla nämä onnistuneet suorituspolut toteutuvat mahdollisimman monessa tapauksessa. Tästä ehdosta tehdään korjauskandidaatti. Luotu kandidaatti tutkitaan suorittamalla ohjelman testit, ennen seuraavan mahdollista generointia.

relifix [26] korjaa päivitysten tuomia regressiovikoja, hyödyntäen ohjelman edellisen version testituloksia sekä tietoa tehdyistä muutoksista. Työkalu etsii potentiaaliset vialliset komennot ja käyttää valmista muokkausoperaatioiden listaa luodakseen korjauskandidaatteja. Käytettävä operaatio valitaan satunnaisesti, toteutetaan, ja syntyneelle mutantille suoritetaan ohjelman testit. Tätä iteroidaan, kunnes testit läpäistään, yhdenkään muokkausoperaation esiehdot eivät täyty, tai suoritus aikakatkaistaan.

SearchRepair [27] määrittää mitä tuloksia sen tietokannan koodinpätkät synnyttävät eri syötteillä. Viallisessa ohjelmassa virheen todennäköisimmistä sijaintipaikoista määritetään samalla tavoin toivottua käytöstä kuvaavat syöte-tulos-parit. Tietokannasta etsitään koodinpätkiä, jotka tuottavat kaikilla syötteillä haluttuja tuloksia. Korjaus luodaan siirtämällä niitä viallisten koodin osien tilalle ja testaamalla näin muokatun ohjelman toimintaa.

SPR [28] valitsee valmiista listastaan muunnoskaavan ja käyttää sitä vialliseen komentoon. Operaation seurauksena syntyy korjausmalli, jonka parametrina voi olla abstrakti lauseke. Tämän jälkeen SPR tutkii, voiko malli luoda onnistuneen korjauksen millään

parametreilla. Jos korjauksen luonti valitulla muunnoskaavalla on mahdollinen, syntetisoidaan mahdollisia parametrien arvoja, kunnes saadaan luotua todellinen korjaus. Mikäli korjaus ei ole mahdollista, yritetään seuraavaa muunnoskaavaa.

4. MENETELMÄT

Korjaustyökalujen välillä havaittiin selviä yhtäläisyyksiä. Työkalujen käyttämiä tapoja korjata virheitä vertailtiin toisiinsa, jolloin todettiin monien niistä muistuttavan läheisesti toisiaan. Abstrahoimalla pois toteutusten teknisiä yksityiskohtia voitiin useiden eri työkalujen katsoa toteuttavan pohjimmiltaan samaa virheenkorjauksen menetelmää. Näitä korjauksentapoja oli tutkitussa 16 työkalun ryhmässä yhteensä kuusi kappaletta.

Tässä luvussa esitetään tutkimuksessa löydetyt korjausmenetelmät. Jokaisesta selostetaan toimintatapa, luetellaan sitä käyttävät työkalut ja esitetään käytännön esimerkki menetelmän toiminnasta. Esimerkit koskevat alla esitellyn ohjelman 1 korjausta.

Aloitteleva ohjelmoija on luonut funktion *onkoSuurempi,* joka tarkistaa onko ensimmäinen parametri suurempi kuin toinen. Virheellisesti funktio palauttaa arvon *true* parametrien ollessa samanarvoiset. Ongelman lähde on rivillä 2. Tulevissa korjausesimerkeissä ohjelmaan lisätyt rivit ilmaistaan merkillä + ja poistetut rivit merkillä -.

```
1     bool onkoSuurempi (int tarkistettava, int vertauskohde) {
2           if (tarkistettava >= vertauskohde) {
3                 return true;
4           }
5           else {
6                 return false;
7           }
8     }
```

Ohjelma 1. *Esimerkeissä korjattava ohjelma*

Testatessaan funktiota ohjelmoija loi joukon yksikkötestejä, jotka on esitetty alla tuloksiensa kanssa.

Testitapaus	**Toivottu tulos**	**Todellinen tulos**
tarkistettava = 2 *vertauskohde* = 1	true	true
tarkistettava = 3 *vertauskohde* = 2	true	true
tarkistettava = 1 *vertauskohde* = 1	false	true
tarkistettava = 3 *vertauskohde* = 3	false	true
tarkistettava = 1 *vertauskohde* = 3	false	false

Taulukko 2. *Funktion onkoSuurempi yksikkötestit.*

Funktio läpäisee testitapaukset 1, 2 ja 5, mutta epäonnistuu tapauksissa 2 ja 3. Automaattisen korjauksen tulisi siis muokata ohjelmaa siten, että se tuottaa testeissä 2 ja 3 eri tuloksen. Kuitenkaan tulos muissa testeissä ei saa vaihtua.

4.1 Korjaussynteesi

Korjaussynteesissä työkalu määrittää ensin, miten virheellisen ohjelman kuuluisi toimia. Tämän jälkeen tutkitaan, millaisia muokkauksia ohjelman käyttäytymiseen tarvitaan, jotta se saadaan toimimaan tarkoitetulla tavalla. Näin saadaan loogiset korjausehdot eli tieto sekä vaadituista (virheen korjaus) että kielletyistä (ehjän osan rikkominen) muutoksista ohjelman käytökseen. Kerätty tieto todellisesta ja halutusta toiminnasta auttaa paikallistamaan virheen sijainnin. Lopuksi muodostetaan korjaus, joka poistaa erot nykyisen ja toivotun toiminnan väliltä, ja tarkistetaan sen toimivuus. Korjauksen luontia varten työkalulla on tyypillisesti käytössään lista muunnosoperaatioita, joilla virheellisiä komentoja säädetään. Korjaussynteesi on työkalujen Prophet [20], Angelix [14], DynaMoth [17], DirectFix [22], Leon [23], PBRepair [24], SPR [28] ja PCR [25] käyttämä menetelmä.

Esimerkki synteesin käytöstä funktion *onkoSuurempi* korjauksessa: korjaustyökalu tutkii funktion rakennetta, ja toteaa että sen käytöstä hallitsee yksi ehtolause. Paikkaukset kohdistetaan siis ohjelman riviin 2. Ehtolauseeseen lisätään toinen, toistaiseksi määrittämätön ehto *abstract_cond()*. Tämä ehto kuvastaa ehtoon tarvittavaa muokkausta. Työkalu aloittaa eri muokkausvaihtoehtojen testauksen, selvittäen mitkä sen käytössä olevista operaatioista ovat hyödyllisiä. Muutaman epäonnistuneen kokeilun jälkeen tarkastellaan ehtolauseen tiukennus -operaatio, jonka todetaan olevan mahdollinen tapa luoda korjaus. Epäonnistuvissa testeissä ehtolause läpäistään, joten tekemällä siitä rajoittavamman ongelma voitaisiin korjata.

```
1     bool onkoSuurempi (int tarkistettava, int vertauskohde) {
2           if (tarkistettava >= vertauskohde
3 +         and abstract_cond()) {
4                 return true;
5           }
6           else {
7                 return false;
8           }
9     }
```

Ohjelma 2. *Korjaussynteesin alku*

Tämän jälkeen määritetään täsmällinen ehto, jolla kaikki testit läpäistään. Tämä voidaan luoda työkalusta riippuen usealla eri menetelmällä, esimerkiksi ohjelman toiminnallisen määritelmän perusteella tai jopa satunnaisesti kokeilemalla. Tässä tapauksessa löydetään

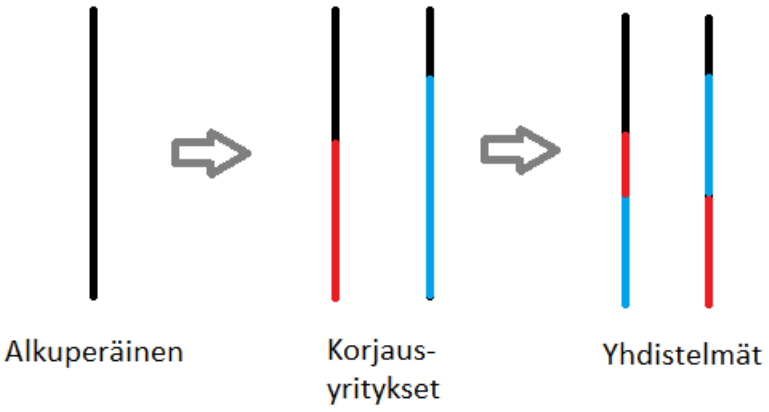
ehto, joka tarkistaa ovatko parametrit yhtä suuria. Sen lisäys läpäisee kaikki testit, joten korjaus on valmis.

```
1     bool onkoSuurempi (int tarkistettava, int vertauskohde) {
2           if (tarkistettava >= vertauskohde
3 -         and abstract_cond()) {
4 +         and tarkistettava != vertauskohde) {
5                 return true;
6           }
7           else {
8                 return false;
9           }
10    }
```

Ohjelma 3. *Korjaussynteesin lopputulos*

4.2 Geneettinen ohjelmointi

Geneettisessä ohjelmoinnissa [12; 30] löydetystä virheellisestä ohjelman osasta luodaan suuri määrä mutaatioita eli lievästi muokattuja versioita, joita sitten yhdistellään toisiinsa. Prosessi on esitetty alla kuvassa 3. Näiden mutaatioiden toimivuus arvioidaan suorittamalla ohjelman testit, ja parhaiten toimivat säilytetään pohjaksi seuraavalle evoluution kierrokselle. Toisella kierroksella säilytettyjä versioita muokataan ja yhdistellään jälleen. Tätä toistetaan, kunnes saadaan luotua kaikki testit läpäisevä versio, tai jokin ennalta määritetty rajoite (esimerkiksi aikaraja) keskeyttää suorituksen. Tutkituista työkaluista geneettisestä ohjelmointia käyttävät Kali [13], Genesis [18], HDRepair [19], Bachin ja Xuanin kehittämä työkalu [15] ja relifix [26].



Kuva 3. *Ohjelmakoodin mutaatiot ja niiden yhdistely*

Geneettinen korjaustyökalu luo funktiosta *onkoSuurempi* mutantteja tekemällä satunnaisia muutoksia funktion eri kohtiin, keskittyen niihin joissa virhe todennäköisimmin sijaitsee. Tästä seuraa, että epäonnistuneita yrityksiä syntyy tyypillisesti valtava määrä. Monet niistä ovat ihmiselle ilmiselvästi virheellisiä. Yksi tällainen epäonnistunut paikkaus on esitetty alla.

```
bool onkoSuurempi (int tarkistettava, int vertauskohde) {
      return true;
}
```

Yritettyään riittävän monta kertaa geneettinen menetelmä saa kuitenkin luotua mutaation, joka läpäisee enemmän testejä kuin alkuperäinen funktio, esimerkiksi tällä tavoin:

```
1     bool onkoSuurempi (int tarkistettava, int vertauskohde) {
2           if (tarkistettava >= vertauskohde) {
3 +               if (tarkistettava == 1) {
4 +                     return false;
5 +               }
6                 return true;
7           }
8           else {
9                 return false;
10          }
11    }
```

Ohjelma 4. *Alkuperäistä parempi mutaatio*

Tällaiset muita parempia tuloksia tuottavat mutantit otetaan seuraavan iteraation pohjaksi, jolloin syntyy jälleen monia viallisia mutantteja, mutta myös entistä toimivampi paikkaus. Edellistä mutanttia voidaan jalostaa kehittämällä äsken lisättyä ehtolausetta, jolloin saadaan lopullinen korjaus.

```
1     bool onkoSuurempi (int tarkistettava, int vertauskohde) {
2           if (tarkistettava >= vertauskohde) {
3 -               if (tarkistettava == 1) {
4 +               if (tarkistettava == vertauskohde) {
5                       return false;
6                 }
7                 return true;
8           }
9           else {
10                return false;
11          }
12    }
```

Ohjelma 5. *Geneettisen ohjelmoinnin lopputulos*

4.3 Symbolinen suoritus

Ohjelman symbolinen suoritus [31] on keino löytää virheitä. Menetelmässä ohjelman syötteille asetetaan symbolisia arvoja oikeiden lukujen sijaan. Ohjelman eri suorituspolut kuljetaan läpi, ja jokaisella kuljetulla polulla kerätään muistiin syötteille asetetut rajoitukset. Lopuksi näiden rajoitusten pohjalta ratkaistaan, millä todellisilla syötteillä ohjelmassa saadaan aikaan kyseinen suorituspolku. Tällöin voidaan myös tarkistaa, tuottaako jokainen polku odotetun tuloksen vai onko jokin niistä virheellinen. Näin saadaan selville virheen sijainti, sekä tieto millä syötteillä kyseinen ohjelman osa suoritetaan. Lopuksi korjataan virhe joko muokkaamalla virheellistä ohjelman osaa tai ohjaamalla vian aiheutta-

vien syötteiden käsittely toiselle suorituspolulle. Menetelmä tuottaa runsaasti tietoa virheiden sijainnista ja ominaisuuksista, muttei itsessään korjaa niitä. Niinpä se esiintyy toisten menetelmien rinnalla, tukien niiden toimintaa. Yleisimmin lopussa suoritettava virheenkorjaus tehdään synteesillä. Menetelmää käyttävät työkalut Angelix [14], DirectFix [22], DynaMoth [17] sekä Bachin ja Xuanin kehittämä nimetön työkalu [15].

Korjattaessa funktio *onkoSuurempi* symbolisesti se suoritetaan ensin antamalla parametreille abstraktit arvot, esimerkiksi *a* ja *b*. Ehtolauseen kohdalla suoritus jaetaan kahdelle polulle, joista toisessa ehto täytetään ja toisessa ei. Molemmilla poluilla suoritus jatkuu funktion loppuun, jolloin polkuihin merkitään niiden tuottamat lopputulokset. Samalla kummallekin suorituspolulle kirjataan muistiin myös sen asettamat vaatimukset muuttujille. Näin luodaan funktion polkukaavio.

Polku 1	**Polku 2**
a >= b & return true	*a < b & return false*

Kuva 4. *Funktion onkoSuurempi polkukaavio.*

Funktiolla on yksikkötestejä, jotka osoittavat toteutuksesta syntyvän polkukaavion olevan virheellinen. Vika sijaitsee siis ehtolauseessa. Tämän tiedon perusteella voidaan kohdistaa korjaustyökalun tekemät muokkaukset oikeaan kohtaan ohjelmassa.

4.4 Kehityshistorian louhinta

Kehityshistorian louhinnassa virhe korjataan keräämällä toisista projekteista suuria määriä ihmisten tekemiä paikkauksia vastaaviin ongelmiin. Näistä abstrahoidaan yleiskäyttöisiä paikkausmalleja, jotka sovelletaan alkuperäiseen lähdekoodiin. Täten luodaan hakuavaruus mahdollisia korjauksia, joiden seasta lopullinen ratkaisu etsitään. Tämä tehdään vertaamalla korjauksia ohjelman testeihin ja keräämällä paikkauskandidaatit, jotka läpäisevät ne kaikki. Menetelmää käyttävät työkalut Genesis [18], Prophet [20] ja HDRepair [19]. Niiden väliset erot ovat varsin pieniä ja teknisiä, perusmenetelmä on kaikissa sama.

Funktion *onkoSuurempi* korjauksessa hyödyllisiä ovat aiemmat numerojen vertailun paikkaukset toisissa ohjelmissa. Korjaustyökalu käy läpi toisia projekteja, etsien ihmisten tekemiä samankaltaisten virheiden paikkauksia. Esimerkiksi alla esitetyt paikkaukset toimivat hyvin paikkausmallin pohjana.

```
1 -   if (a > b) {
2 +   if (a >= b) {
3           tarkista(a);
4     }

1 -   if (creature.pos <= map.width) {
2 +   if (creature.pos < map.width) {
3           throw OutOfBoundsException();
4     }
```

Tämän kaltaisista paikkauksista saadaan luotua malli, jossa ehtolauseen vertailuoperaattoria muokataan. Mallia käytetään apuna sekä paikkauskandidaattien luonnissa, että niiden oikeellisuuden arvioinnissa. Mallin mukaisesti muutokset pyritään kohdistamaan ainoastaan ehtolauseeseen, joten esimerkiksi osiossa 4.2 esitetty korjaus nähdään huonolaatuisena. Korjaustyökalu kokeilee erilaisia operaattorin muutoksia, kunnes havaitsee, että > on tässä tapauksessa oikea valinta.

```
1     bool onkoSuurempi (int tarkistettava, int vertauskohde) {
2 -         if (tarkistettava >= vertauskohde) {
3 +         if (tarkistettava > vertauskohde) {
4                 return true;
5           }
6           else {
7                 return false;
8           }
9     }
```

Ohjelma 6. *Kehityshistorian louhinnan lopputulos*

4.5 Koodinsiirto

Koodinsiirto on menetelmä, jossa siirretään korjattavan ohjelman ulkopuolisesta lähteestä viallisen osion tilalle samanlaisen tehtävän suorittavaa toimivaa ohjelmakoodia. Siirrettävä koodi voidaan noutaa esimerkiksi korjattavan ohjelman aiemmasta versiosta, ohjelman toisesta osasta tai ulkopuolisesta tietokannasta. Taustalla on oletus, että ohjelmissa usein toistuu samankaltaisia rakenteita [32], joten sama ongelma on todennäköisesti ratkaistu muualla onnistuneesti. Siirto pyritään tekemään mahdollisimman suoraan, mutta pieniä muutoksia siirrettävään koodiin joudutaan tavallisissa tapauksissa tekemään, jotta se toimii uudessa ympäristössään. Koodinsiirtoa soveltavat Code Phage [21] ja SearchRepair [27; 32].

Esimerkkitapauksessa funktion *onkoSuurempi* suorituksesta muodostetaan lähtö- ja tuloarvojen pareja, kuten ”syöte (2, 1) tuottaa tuloksen *true*, mikä on haluttu tulos”. Tämän jälkeen korjaustyökalu etsii tietokannastaan ohjelmakoodin pätkiä, joka vastaanottavat kaksi kokonaislukua ja palauttavat totuusarvon. Tällainen pätkä voi olla esimerkiksi toisesta projektista löytynyt samaan tarkoitukseen kirjoitettu ohjelma 7.

```
1     bool ekaIsompi (int eka, int toka) {
2           if (eka < toka) {
3                 return false;
4           }
5           return true;
6     }
```

Ohjelma 7. *Siirrettävä ulkopuolinen ohjelmakoodi*

Valittu koodi siirretään korjattavan ohjelmakoodin tilalle, muuntaen tarvittaessa siirrettävän koodin muuttujien ja funktioiden nimiä vastaamaan korjattavan ohjelman ympäristöä. Tässä tapauksessa näin joudutaan tekemään, koska funktion nimen vaihtaminen voisi aiheuttaa ongelmia muualla ohjelmassa. Lopputuloksena saadaan korjattu ohjelma.

```
1     bool onkoSuurempi (int tarkistettava, int vertauskohde) {
2 -         if (tarkistettava >= vertauskohde) {
3 -               return true;
4 +         if (tarkistettava < vertauskohde) {
5 +               return false;
6           }
7 -         else {
8 -               return false;
9 -         }
10 +        return true;
11    }
```

Ohjelma 8. *Koodinsiirron lopputulos*

4.6 Valmiit korjausmallit

Valmiita korjausmalleja käyttävillä työkaluilla on tiedossaan sarja malleja, joista jokainen korjaa tietynlaiset virheet. Mallit ovat etukäteen luotuja, esimerkiksi työkalun kehittäjän manuaalisesti rakentamia. Yksittäinen malli on kuin käyttöohje, joka sisältää tarvittavat askeleet yhdenlaiselle korjaukselle. Se koostuu sarjasta koodimuunnoksia eli komentojen lisäyksiä ja poistoja. Lisäksi siinä on kuvaus kontekstista, eli mitkä ohjelmakoodin piirteet osoittavat mallin sopivan kyseiseen tilanteeseen. Korjaustyökalu vertaa korjattavaa ohjelmakoodia eri malleihin, ja jos jonkin mallin konteksti sekä poistettavat komennot vastaavat viallisesta ohjelmasta löytyviä komentoja, käytetään kyseistä mallia. Näin työkalu tunnistaa millaisesta ongelmasta on kyse ja käyttää kyseiseen tilanteeseen sopivinta mallia korjauksen suoritukseen. Tästä toimintatavasta seuraa, että nämä työkalut ovat käyttökelpoisia rajatuissa tilanteissa: jos tiettyyn tilanteeseen sopivaa mallia ei ole tarjolla, korjaus ei ole mahdollinen. Mikäli virhettä vastaava laadukas malli taas on käytettävissä, korjaus on erittäin nopea ja luotettava. Tutkituista työkaluista ainoastaan CDRep [16] toimii korjausmalleilla.

Korjausmalleja käyttävä työkalu toimii esimerkkitilanteessa vain, mikäli sillä on funktiolle *onkoSuurempi* sopiva malli. Oletetaan että näin on. Mallissa muuttujat merkitään geneerisillä nimillä *M1* ja *M2*, vakiot merkillä *, ja kontekstin kannalta tärkeät komennot merkillä *§*.

```
1 -         if (M1 >= M2) {
2 +         if (M1 > M2) {
3 §               return *;
4           }
5 §         else {
6 §               return *;
7           }
```

Ohjelma 9. *Ohjelman 1 korjaukseen sopiva valmis korjausmalli*

Yllä esitetyn mallin käyttö alkaa tarkistamalla, että poistettaviksi ja kontekstille tärkeiksi merkityt komennot löytyvät korjattavasta funktiosta. Tämän jälkeen ohjelmasta poimitaan muuttujien nimet ja vakioiden arvot geneeristen arvojen tilalle. Lopuksi suoritetaan komentojen lisäykset ja poistot. Tuloksena saadaan korjattu ohjelma.

```
1     bool onkoSuurempi (int tarkistettava, int vertauskohde) {
2 -         if (tarkistettava >= vertauskohde) {
3 +         if (tarkistettava > vertauskohde) {
4                 return true;
5           }
6           else {
7                 return false;
8           }
9     }
```

Ohjelma 10. *Valmiin korjausmallin lopputulos*

5. PÄÄTELMÄT

Tutkimalla 16 eri työkalua löydettiin ohjelmakoodin virheiden korjaamiseksi kuusi erilaista menetelmää. Yleisimmät korjausmenetelmät ovat korjaussynteesi ja geneettinen ohjelmointi, joista ensimmäistä käyttää kahdeksan työkalua ja toista viisi. Muut menetelmät ovat yleisyysjärjestyksessä symbolinen suoritus, kehityshistorian louhinta, koodinsiirto ja valmiit korjausmallit. Yksittäinen korjaustyökalu käyttää näistä joko yhtä tai kahta toiminnassaan.

Yhteistä tutkituille työkaluille on niiden yleiskäyttöisyys ja testeihin perustuva toiminta. Vain CDRep on tiukasti rajattu tietynlaisten virheiden käsittelyyn, muut ovat geneerisiä. Lisäksi ainoastaan CDRep sekä Bach ja Xuan eivät oleta, että viallisen ohjelman mukana tulleet testit määrittelevät sen oikeanlaisen toiminnan. Erityisesti testeihin keskittyminen on tuore, viimeisen viiden vuoden aikana ilmaantunut ilmiö. Katsauksen tulosten mukaan nämä kaksi piirrettä ovat nykyaikaisen automaattisen ohjelmakoodin korjauksen tunnusmerkkejä.

Automaattisen korjauksen yhteydessä puhutaan laajalti syntaktisista ja semanttisista korjauksista [15; 33; 34]. Syntaksiin perustuvat menetelmät luovat laajan hakuavaruuden eli joukon mahdollisia paikkauksia, ja etsivät sieltä kandidaatteja korjaukselle. Tämä hakuavaruus voidaan luoda keräämällä olemassaolevia koodinpätkiä valitusta lähteestä, tai mutatoimalla olemassaolevaa koodia satunnaisesti. Semanttiset menetelmät puolestaan keräävät ensin tietoa korjaukselta vaadittavista ominaisuuksista, ja tekevät sitten ohjelmaan tarvittavat muutokset. Tutkimuksessa löydetyt kuusi menetelmää asettuvat näihin kahteen yläluokkaan selkeästi, kuten nähdään alla taulukossa 3. Kumpikin yläluokka on yhtä suosittu.

	Semanttinen	**Syntaktinen**
Koodin luonti	Korjaussynteesi, Symbolinen suoritus*	Geneettinen ohjelmointi
Koodin haku	Valmiit korjausmallit	Kehityshistorian louhinta, Koodinsiirto

Taulukko 3. *Menetelmien luokittelu*

**Symbolinen suoritus voi työkalusta riippuen luoda tai hakea koodia, mutta luominen on yleisempää.*

Löydetyt menetelmät jakautuvat kahtia myös korjatun koodin lähteen suhteen. Osa kehittää korjauksia itse, rakentaen täysin uutta ohjelmakoodia. Toiset taas noutavat sitä muualta, siirtäen olemassaolevaa koodia uuteen paikkaan. Kuten taulukosta 3 näkyy, sekä semanttiset että syntaktiset menetelmät käyttävät kumpaakin lähdettä. Koodin luominen on yleisempää: kolme suosituinta menetelmää käyttävät tätä korjaustapaa.

Taulukko 3 auttaa ymmärtämään eri menetelmien eroja ja yhtäläisyyksiä. Yleisimmin yhdessä käytetty pari on korjaussynteesi ja symbolinen suoritus, ja molemmat asettuvat samaan kategoriaan eli semanttiseen koodin luontiin. Kumpikin selvittää ensin korjaukselta vaadittavia ominaisuuksia, ja luo sitten uutta ohjelmakoodia joka täyttää nämä vaatimukset. Myös keskenään samankaltaisia ovat kehityshistorian louhinta ja koodinsiirto. Kummassakin kerätään korjattavan ohjelman ulkopuolista koodia, jota liitetään vialliseen ohjelmaan ja näin saadaan aikaan monia paikkausyrityksiä. Niitä testaamalla etsitään lopullinen korjaus.

Osa tutkituista työkaluista yhdistelee toisistaan voimakkaasti eroavia menetelmiä. Esimerkiksi Prophet yhdistää korjaussynteesiä ja kehityshistorian louhintaa, Genesis puolestaan geneettistä ohjelmointia ja kehityshistorian louhintaa. Kahden erilaisen lähestymistavan käyttö rinnakkain tarjoaa laajemmat mahdollisuudet onnistua korjauksessa. Mikäli yksi menetelmä on tehoton tiettyjen virheiden korjaamisessa, voi toinen saada parempia tuloksia aikaan.

Aiheen ympärillä on yhä runsaasti tilaa jatkotutkimukselle. Eri menetelmien ominaisuuksia ei tässä tutkimuksessa selvitetty kvantitatiivisesti, eikä kullekin parhaiten soveltuvia käyttötilanteita ole määritelty. Myös menetelmien yhdistelyyn voitaisiin perehtyä kokeellisesti.

LÄHTEET

[1] V. Mittal, S. Aditya, Recent Developments in the Field of Bug Fixing, Procedia Computer Science, Vol. 48, No. C, 2015, pp. 288-297.

[2] B. Jobstmann, S. Staber, A. Griesmayer, R. Bloem, Finding and fixing faults, Journal of Computer and System Sciences, Vol. 78, No. 2, 2012, pp. 441-460.

[3] C. Le Goues, S. Forrest, W. Weimer, Current challenges in automatic software repair, Software Quality Journal, Vol. 21, No. 3, 2013, pp. 421-443.

[4] A. Arcuri, Evolutionary repair of faulty software, Applied Soft Computing Journal, Vol. 11, No. 4, 2011, pp. 3494-3514.

[5] F. Long, M. Rinard, Staged program repair with condition synthesis, Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, pp. 166-178.

[6] W.H. Pierce, Failure-tolerant computer design, Academic Press, 1965, .

[7] B. Randell, System structure for software fault tolerance, ACM SIGPLAN Notices, ACM, pp. 437-449.

[8] M. Stumptner, F. Wotawa, A model-based approach to software debugging, Proceedings on the Seventh International Workshop on Principles of Diagnosis, Citeseer, .

[9] H. He, N. Gupta, Automated debugging using path-based weakest preconditions, International Conference on Fundamental Approaches to Software Engineering, Springer, pp. 267-280.

[10] M. Brodie, S. Ma, G. Lohman, L. Mignet, N. Modani, M. Wilding, J. Champlin, P. Sohn, Quickly finding known software problems via automated symptom matching, Second International Conference on Autonomic Computing (ICAC'05), IEEE, pp. 101-110.

[11] A. Arcuri, X. Yao, A novel co-evolutionary approach to automatic software bug fixing, 2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence), IEEE, pp. 162-168.

[12] C. Le Goues, T. Nguyen, S. Forrest, W. Weimer, Genprog: A generic method for automatic software repair, IEEE Transactions on Software Engineering, Vol. 38, No. 1, 2012, pp. 54-72.

[13] Z. Qi, F. Long, S. Achour, M. Rinard, An analysis of patch plausibility and correctness for generate-and-validate patch generation systems, Proceedings of the 2015 International Symposium on Software Testing and Analysis, ACM, pp. 24-36.

[14] S. Mechtaev, J. Yi, A. Roychoudhury, Angelix: Scalable multiline program patch synthesis via symbolic analysis, pp. 691-701.

[15] L.D.X. Bach, Q.L. Le, D. Lo, C.L. Goues, Enhancing Automated Program Repair with Deductive Verification, .

[16] S. Ma, D. Lo, T. Li, R.H. Deng, CDRep: Automatic Repair of Cryptographic Misuses in Android Applications, Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, ACM, New York, NY, USA, pp. 711-722.

[17] T. Durieux, M. Monperrus, DynaMoth: Dynamic code synthesis for automatic program repair, pp. 85-91.

[18] F. Long, P. Amidon, M. Rinard, Automatic Inference of Code Transforms and Search Spaces for Automatic Patch Generation Systems, 2016, .

[19] X. Le, History Driven Program Repair, The Institute of Electrical and Electronics Engineers, Inc.(IEEE) Conference Proceedings, Vol.1, pp.213-224, Vol. 1, 2016, pp. 213-224.

[20] F. Long, M. Rinard, Automatic patch generation by learning correct code, ACM SIGPLAN Notices, Vol. 51, No. 1, 2016, pp. 298-312.

[21] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, M. Rinard, Automatic Error Elimination by Horizontal Code Transfer Across Multiple Applications, SIGPLAN Not., Vol. 50, No. 6, 2015, pp. 43-54.

[22] S. Mechtaev, J. Yi, A. Roychoudhury, DirectFix: looking for simple program repairs, IEEE Press, pp. 448-458.

[23] E. Kneuss, M. Koukoutos, V. Kuncak, Deductive program repair, International Conference on Computer Aided Verification, Springer, pp. 217-233.

[24] H. Riener, R. Ehlers, G. Fey, Path-Based Program Repair, arXiv preprint arXiv:1503.04914, 2015, .

[25] F. Long, Z. Qi, S. Achour, M. Rinard, Automatic Program Repair with Condition Synthesis and Compound Mutations, 2015, .

[26] S. Tan, A. Roychoudhury, relifix: automated repair of software regressions, IEEE Press, pp. 471-482.

[27] Y. Ke, An automated approach to program repair with semantic code search, 2015, .

[28] F. Long, M. Rinard, Staged program repair in SPR, 2015, .

[29] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. Lamelas Marcote, T. Durieux, D. Le Berre, M. Monperrus, Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs, IEEE Transactions on Software Engineering, 2016, pp. 1-1.

[30] S. Forrest, Genetic Algorithms: Principles of Natural Selection Applied to Computation, Science, Vol. 261, 1993, pp. 13.

[31] J.C. King, Symbolic Execution and Program Testing, Commun.ACM, Vol. 19, No. 7, 1976, pp. 385-394.

[32] Y. Ke, K.T. Stolee, C. Le Goues, Y. Brun, Repairing programs with semantic code search (T), Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on, IEEE, pp. 295-306.

[33] X.B.D. Le, D. Lo, C. Goues, Empirical study on synthesis engines for semantics-based program repair, 2016, .

[34] X.D. Le, Towards Efficient and Effective Automatic Program Repair, Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ACM, New York, NY, USA, pp. 876-879.