University of Zagreb

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Danijel Mlinarić

EXTENSION OF DYNAMIC SOFTWARE UPDATE MODEL FOR CLASS HIERARCHY CHANGES AND RUN-TIME PHENOMENA DETECTION

DOCTORAL THESIS

Zagreb, 2020

University of Zagreb

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Danijel Mlinarić

EXTENSION OF DYNAMIC SOFTWARE UPDATE MODEL FOR CLASS HIERARCHY CHANGES AND RUN-TIME PHENOMENA DETECTION

DOCTORAL THESIS

Supervisor: Associate professor Boris Milašinović, PhD

Zagreb, 2020

Sveučilište u Zagrebu

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Danijel Mlinarić

PROŠIRENJE MODELA DINAMIČKOG AŽURIRANJA SOFTVERA NA PROMJENU HIJERARHIJE KLASA I DETEKCIJU FENOMENA IZVOĐENJA

DOKTORSKI RAD

Mentor: Izv. prof. dr. sc. Boris Milašinović

Zagreb, 2020.

Doctoral thesis has been made at the University of Zagreb, Faculty of Electrical Engineering and Computing, Department of Applied Computing.

Supervisor: Associate professor Boris Milašinović, PhD

Doctoral thesis has: 150 pages

Number of doctoral thesis.: ___________

About the Supervisor

Boris Milašinović was born in Metković in 1976. He graduated at the Department of Mathematics of the Faculty of Science, University of Zagreb in July 2001. He defended his Master's Thesis in 2006 and he defended his PhD thesis in 2010 at the Faculty of Electrical Engineering and Computing, University of Zagreb. Since October 2002, he has been employed at the Department of Applied Computing of the Faculty of Electrical Engineering and Computing, University of Zagreb as a research assistant. From September 2001 to October 2002, he was employed at the Department of Mathematics of the Faculty of Science, University of Zagreb as a research and teaching assistant. From June 2000 to September 2001, he worked as a software developer at Multimedia Lab. He has held the position of Associate Professor at the Faculty of Electrical Engineering and Computing since February 2019. He has been a supervisor of more than 20 Bachelor's and Master's theses. Within the Department's external cooperation, he worked on the design and development of databases and software and as a consultant in informatization projects for the economy and public administration. He is author or co-author of more than 20 scientific, professional and educational publications in the area of information systems and software engineering. Associate Professor Boris Milašinović is a member of IEEE and a member of several conference international programme committees.

O mentoru

Boris Milašinović rođen je u Metkoviću 1976. godine. Diplomirao je 2001. godine na Matematičkom odjelu Prirodoslovno matematičkog fakulteta u Zagrebu (dipl.ing. matematike – smjer računarstvo) te magistrirao 2006. i doktorirao 2010. godine na Sveučilištu u Zagrebu Fakultetu elektrotehnike i računarstva (FER). Od listopada 2002. godine zaposlen je na Zavodu za primijenjeno računarstvo FER-a. Prethodno je bio zaposlen kao znanstveni novak na Matematičkom odjelu PMF-a (2001.-2002.) te kao programer u tvrtki MultimediaLab d.o.o. (2000.-2001.). U veljači 2019. godine izabran je u zvanje izvanrednog profesora te je bio mentor više od 20 završnih i diplomskih radova. U okviru vanjske suradnje Zavoda radio je na projektiranju i izradi baza podataka i programske podrške te kao konzultant u projektima informatizacije za gospodarstvo i državnu upravu. Objavio je više od 20 radova u časopisima i zbornicima konferencija u području informacijskih sustava i programskog inženjerstva. Izv. Prof. Boris Milašinović član je stručne udruge IEEE i programskih odbora nekoliko međunarodnih konferencija.

To my family for their support and inspiration
Thank you mom:
“Keep moving forward”

Abstract

Software maintenance requires a software update that causes unavailability during the update. Software downtime introduces costs in business processes and can negatively affect critical processes, such as the software controlling bank transactions or traffic systems. One of the solutions to provide software updating without interruption is Dynamic Software Updating (DSU). However, there are several challenges in dynamic software updating. It is necessary to detect changes between versions of the program, apply them dynamically, and ensure the program's correct operation and stable performance. Aspect programming (AOP - Aspect Oriented Programming) by cross-cutting concerns allows modification of object-oriented programs, making it convenient to describe program changes. Furthermore, dynamic AOP (DAOP) allows program changes at run time, but currently available solutions contain limitations in the form of supported changes, such as changing the class hierarchy. Meanwhile, the state correctness of the dynamically updated program can be compromised, known as the occurrence of runtime phenomena. By detecting runtime phenomena, such program states can be prevented. Furthermore, a dynamically updated program may cause performance degradation due to an additional layer that allows dynamic updating. Therefore, dynamic updating implies the need to evaluate and compare the performance in order to use solutions that minimally affect execution performance.

In object-oriented programming, a class is the main construct and relationship between classes in the form of inheritance abstracts the software domain. Therefore, changes between program versions and runtime phenomena are considered from the class hierarchy perspective. Consequently, an extended DAOP model is proposed that extends the set of supported changes to class hierarchy and changes related to the class hierarchy. Furthermore, program analysis is used to detect and estimate runtime phenomena based on the changes between different program versions. In addition, the proposed methodology for performance evaluation of the various approaches is used to detect the impact on computer resources.

The prototype system is developed by implementing the proposed update model with runtime phenomena detection. Prototype evaluation is performed by several experiments, validating the proposed approach in terms of efficiency, applicability, and performance. Efficiency is evaluated by generated trees, whereas open-source programs are used for empirical analysis of supported changes to evaluate applicability. Moreover, a variant of the *Snake* game is developed, with two update versions used as a use case example to validate the model. As for performance evaluation, the prototype system and currently available approaches are compared using the proposed methodology implemented as a benchmark tool.

Keywords: dynamic software updating, dynamic AOP, class hierarchy, OOP, runtime phenomena, performance evaluation

Sažetak

Održavanje softvera zahtjeva ažuriranje softvera koje uzrokuje nedostupnost tijekom ažuriranja. Nedostupnost softvera uvodi troškove u poslovnim procesima i može negativno utjecati na kritične procese, poput softvera koji kontrolira bankovne transakcije ili sustav koji kontrolira promet. Dinamičko ažuriranje softvera (engl. *Dynamic Software Updating*) jedno je od rješenja za ažuriranje softvera bez prekida rada. Međutim, postoji nekoliko izazova u dinamičkom ažuriranju softvera. Potrebno je detektirati promjene između verzija programa i primijeniti ih dinamički. Tijekom i nakon dinamičkog ažuriranja potrebno je osigurati ispravan rad programa i stabilne performance. Slijedom toga, programiranje pomoću aspekata (AOP – Aspect Oriented Programming) omogućuje horizontalnu modifikaciju objektno-orijentiranih programa, što ga čini prikladnim za opisivanje promjena programa. Nadalje, dinamički AOP (DAOP – Dynamic AOP) omogućuje promjene programa tijekom izvođenja, ali trenutno dostupna rješenja sadrže ograničenja u obliku podržanih promjena, poput promjene hijerarhije klasa. S druge strane, ispravnost stanja dinamički ažuriranog programa može biti narušena, što je poznato kao nastanak fenomena izvođenja. Detektiranjem fenomena izvođenja takva stanja se mogu spriječiti. Nadalje, dinamički ažurirani program može uzrokovati pogoršanje performanci izvođenja zbog dodatnog sloja koji omogućuje dinamičko ažuriranje. Stoga dinamičko ažuriranje podrazumijeva potrebu za evaluacijom i usporedbom performanci, kako bi se koristila rješenja koja minimalno utječu na performance izvođenja.

Postoje različita rješenja za probleme povezane s dinamičkim ažuriranjem, međutim trenutno se niti jedno rješenje ne koristi u produkcijskom okruženju. Glavni motiv ove disertacije je napraviti korak prema korištenju DSU-a iz razvojnog u produkcijsko okruženje. Odnosno, u scenarijima u kojima se manje promjene između verzija, poput novih funkcionalnosti ili hitnih popravaka, mogu dinamički primijeniti s minimalnim rizikom i troškovima. Zatim se redovito ažuriranje programa može izvršiti u vremenskom periodu u kojem je upotreba softvera niska kako bi se izbjeglo ometanje rada krajnjih korisnika. Kako bi se program mogao dinamički ažurirati i podržao scenarij glavne motivacije, detektirano je nekoliko ključnih ograničenja u trenutačno dostupnim rješenjima za dinamičko ažuriranja. Detektirana ograničenja smatraju se područjima daljnjeg istraživanja: model dinamičkog ažuriranja koji podržava proizvoljne promjene programa, otkrivanje i sprječavanje mogućeg nepravilnog ponašanja programa i evaluacija performanci različitih pristupa dinamičkog ažuriranja.

U postojećim istraživanjima detektirani su problemi koje uvodi DSU u usporedbi s postupcima ažuriranja koji zahtijevaju zaustavljanje programa. Ključni problemi su: kada primijeniti ažuriranje na pokrenuti program, kako održati ispravno stanje programa nakon ažuriranja, kako podržati razne promjene te kako učinkovito detektirati i primijeniti promjene bez utjecaja na performance izvođenja. Predložena rješenja su u obliku različitih koncepata i tehnika.

U ovom doktorskom istraživanju fokus je postavljen na nekoliko karakteristika pristupa DSU-a: mehanizmi ažuriranja, specifikacija promjena između različitih verzija, podržane promjene, utjecaj na stanje ažuriranog programa i performance izvođenja.

U okruženjima viših programskih jezika gdje se koriste virtualni strojevi, postoji nekoliko koncepata s pristupima koji koriste različite mehanizme za postizanje dinamičkog ažuriranja. Java je trenutno dominantna u istraživanjima, ali predložena rješenja za određene probleme dinamičkog ažuriranja mogu se primijeniti na druga programska okruženja, kao što je Microsoft CLR. Izmijenjeni Java virtualni stroj odnosno JVM pristupi koriste modificirane interne strukture podataka JVM-a i modificirani sakupljač smeća (engl. *garbage collector*) za promjenu referenci na postojeće objekte. Nadalje, okruženje poput JVM sadrži API (engl. *Application Programming Interface*) sučelja, npr. JVMTI (engl. *Java Virtual Machine Tool Interface*), koje omogućuje kontrolu nad izvršavanjem pokrenutog programa i izmjene prijenosnog programskog kôda (bytecode). Java agenti se mogu koristiti kao sloj između programa i JVM-a, koji sa sučeljima poput JVMTI omogućavaju presretanje pristupa objektima i promjenu klasa. Aspektno orijentirana paradigma (AOP) omogućuje proširenje funkcionalnosti programa na horizontalnoj razini, odnosno uvodi razdvajanje funkcionalnosti. Programski kôd aspekta upliće se unutar programa tijekom kompilacije ili učitavanja, kao što je to slučaj za AspectJ. Međutim, dinamički AOP (DAOP) omogućuje dinamičko uplitanje aspekata što se može koristiti za dinamičko ažuriranje programa i može implementirati u obliku izmijenjenog JVM-a i JVM agenta.

Specifikacija promjena između različitih verzija programa nije univerzalno definirana, u postojećim pristupima definirana je u obliku modificiranih klasa programa, vlastito definiranog formata i vlastito definiranog programskog jezika. Istraživanja, gdje je koncept DAOP-a proširen kako bi omogućio DSU, pokazuju kako se dinamički aspekti sa svojom presječnom odnosno funkcionalnošću izolacije programskog kôda, mogu koristiti kao prikladan format za opisivanje promjena programa. Nadalje, dinamički aspekti omogućuju fleksibilno dodavanje i brisanje promjena programa i time mogu omogućiti inkrementalna dinamička ažuriranja.

Za podržane promjena, DAOP je ograničen na promjenu tijela metode. Kako bi se proširio podržani skup promjena, potrebno je generirati promjene u obliku drugih tipova aspekata (npr. aspeki za pristupanje polju klase) i dodatnih klasa. U pristupima proširenog DAOP-a, dinamičke promjene izvode se uz pomoć analize različitih verzija softvera i generiranja dinamičkih aspekata. Izmijenjeni JVM i Java agent omogućuju veći skup promjena u usporedbi s proširenim DAOP-om, koji je ograničen na promjene na razini klase. Cilj istraživanja DAOP-a je omogućiti što više aspektnih funkcionalnosti podržanih pristupima statičkih aspekata, kao što je AspectJ. Takvi DAOP pristupi omogućili bi jednostavnu specifikaciju i primjenu proizvoljnih promjena između inačica programa.

Aspektni jezici nisu izgrađeni sa svrhom definiranja specifikacije promjena programa i dinamičke evolucije softvera, stoga aspekti ne mogu definirati promjene u hijerarhiji klase. Npr. AspectJ podržava ograničeno dodavanje sučelja klase i ne podržava brisanje sučelja. Nadalje, Prose kao DAOP ne podržava pozive metoda iz uplitanog, tj. programskog kôda unutar savjeta (engl. *advice*) na metodu definiranu u istom aspektu, niti je moguće pristupiti polju definiranom u istom aspektu. Uz to, nije moguće pristupiti polju koje je zaštićeno modifikatorima pristupa (npr. `private`) i ne podržava zamjenu tijela konstruktora. Navedene funkcionalnosti moguće je postići deklariranjem promijenjenih članova klasa u dodatnim klasama i proširivanjem postojećih klasa pomoću međutipnih deklaracija, ali pristup koji rješava problem dinamičkih međutipnih deklaracija ne rješava problem dinamičke promjene hijerarhije klasa.

Nekoliko pristupa u području istraživanja napredovalo je za upotrebu u razvojnim okruženjima. Sljedeći korak je korištenje dinamičkog ažuriranja u produkcijskim okruženjima. Izvršavanje programa u razvojnim okruženjima pojednostavljuje uvjete koje bi pristup za dinamičko ažuriranje trebao ispunjavati. S obzirom na to da je izvršavanje programa kratko, pretvorba stanja je jednostavna i provodi se statička analiza kako bi se potvrdila ispravnost promjena u programu. Osim toga, pristupi iz razvojnog okruženja ne uzimaju u obzir inkrementalna ažuriranja. Kao rezultat, s dinamičkim ažuriranjima mogu se pojaviti fenomeni izvođenja. Fenomen izvođenja je stanje programa nakon dinamičkog ažuriranja koje nije isto kao što bi bilo nakon postupka ažuriranja u obliku koraka: zaustavi, ažuriraj i pokreni program. Pojedini radovi razmatraju potrebu alata s dinamičkom analizom promjena programa kako bi se utvrdilo mogu li dinamička ažuriranja dovesti do fenomena izvođenja. S druge strane, u proširenom DAOP-u, analiza programa se koristi za identifikaciju i pretvaranje promjena programa u aspekte. Nadalje, kod dinamičke evolucije aspektno-orijentiranih programa, analiza između različitih verzija programa koristi se za inkrementalna ažuriranja. Trenutno nedostaju istraživanja s analizom promjena programa povezana s fenomenima izvođenja. Detektiranje fenomena izvođenja omogućila bi DSU-u sprječavanje ažuriranja koja mogu uzrokovati fenomene izvođenja, odbijanjem ili dijeljenjem ažuriranja na manje dijelove kako bi se ažuriranje u konačnici izvršilo.

Performance dinamički ažuriranog softvera mogu biti narušene u odnosu na softver ažuriran procedurom s ponovnim pokretanjem. Kako bi se iskoristila prednost DSU-a, potrebno je minimizirati utjecaj na računalne resurse. Trenutni DSU pristupi evaluirani su prema brzini izvođenja implementiranih mehanizama koji omogućuju dinamičko ažuriranje. Npr. trajanje izvršavanja modificiranog sakupljača smeća (engl. *garbage collector*) ili utjecaj na izvršavanje programa kada se nisu dogodila dinamička ažuriranja, odnosno u stanju mirovanja (engl. *steady state*). Iako pojedini pristupi sadrže evaluaciju koristeći stvarne programe, takva evaluacija ne može prikazati ukupni utjecaj pojedinog pristupa na performance izvođenja jer su mjerenja ograničena domenom testiranog programa. Kako bi evaluirali pristupe sa stanovišta utjecaja na računalne resurse i usporedili rezultate, potrebno je razviti alat koji može obaviti različita mjerenja utjecaja dinamičkog ažuriranja na brzinu izvođenja i potrošnju memorije pomoću unaprijed pripremljenih testova.

Disertacija je strukturirana kako slijedi:

U prvom, uvodnom poglavlju (*1 Introduction*) opisani su motivacija, područje i ciljevi istraživanja te sadržaj rada. Objašnjen je znanstveni doprinos i njegova struktura.

Zatim slijedi poglavlje (*2 Dynamic Software Updating*) u kojem su kategorizirani očekivani zahtjevi koje treba ispuniti prilikom dinamičkog ažuriranja softvera, opisane općenite tehnike dinamičke promjene i klasični problemi vezani uz dinamičko ažuriranje po pitanju trenutka i opsega ažuriranja te konzistentnosti podataka. Poglavlje završava pregledom postojećih rješenja za dinamičko ažuriranje softvera te uočenim problemima i nedostacima postojećih rješenja.

Fokus disertacije je na dinamičkom ažuriranju softvera kao posljedice izmjene hijerarhije klasa u objektno-orijentiranoj paradigmi, stoga treće poglavlje (*3 Object-oriented environment*) opisuje moguće izmjene u hijerarhiji klasa te mogućnosti prikaza hijerarhije klasa u obliku stabla uz vizualizaciju promjena između stabala koji predstavljaju staru i novu verziju programa. U poglavlju je također detaljno opisani koncepti objektno-orijentirane paradigme i podrška za promjene u Javi kao tipičnom predstavniku objektno-orijentiranih programskih jezika, odnosno u Javinom virtualnom računalu i Javinoj okolini.

Temelj za detekciju izmjena hijerarhije klasa predstavljaju mjere za udaljenosti stabala definirane u četvrtom poglavlju (*4 Tree dissimilarity*). Mjere opisuju troškove izmjene među verzijama softvera, uzimajući u obzir direktne izmjene hijerarhije dodavanjem, brisanjem ili izmjenom bridova stabla (što odgovara dodavanju novih klasa, brisanju nekih od postojećih, odnosno promjenama u nasljeđivanju), ali i indirektne troškove kao posljedicu izmjene lanca nadjačavanja metoda u hijerarhiji klasa. Za programsku implementaciju dinamičke izmjene koristi se aspektno-orijentirana paradigma, odnosno varijanta s dinamičkim aspektima.

Nakon opisa postojećih istraživanja na području dinamičkih aspekata, u petom poglavlju (*5 Extended DAOP model*) definiran je algoritam koji omogućava detekciju izmjene hijerarhije klasa te model temeljen na dinamičkim aspektima za podršku dinamičkom ažuriranju.

Dinamičkom izmjenom softvera moguć je nastanak određenih artefakata korištenih u staroj, odnosno novoj verziji softvera, što se opisuje kao fenomeni izvođenja. Mogući fenomeni su kategorizirani u šestom poglavlju (*6 Runtime phenomena detection*) te su moguće promjene u hijerarhiji klasa uparene s odgovarajućim kategorijama fenomena. Nadogradnjom algoritama iz 5. poglavlja definiran je skup novih algoritama za detekciju mogućih fenomena izvođenja uslijed dinamičkog ažuriranja korištenjem dinamičkih aspekata definiranih u 6. poglavlju. Sedmo poglavlje (*7 Measurement methodology for performance benchmarking*) definira metode, mjere i arhitekturu za mjerenje performanci dinamičkog ažuriranja temeljem postojećih tehnika s ciljem evaluiranja vlastitog rješenja čiji je prototip opisan u osmom poglavlju (*8 Prototype system*).

Kako bi se evaluacija mogla provesti bilo je potrebno izraditi generator podataka, odnosno generator hijerarhije klasa, što je opisano u devetom poglavlju (*9 Evaluation*). Analizom dubine i širine hijerarhije klasa nekoliko postojećih programa otvorenog koda utvrđena je uobičajena distribucija čvorova po razinama te su analizirani razmjeri izmjena verzija tih programa. Temeljem dobivenih podataka oblikovani su parametri za generator stabala korištenih prilikom evaluacije.

Disertacija završava zaključkom (*10 Concluding remarks*) i pregledom citirane literature. Kroz zaključak je dan rezime disertacije i otvoreni problemi za planirana buduća istraživanja.

U ovoj disertaciji promjene između inačica programa razmatraju iz perspektive hijerarhije klasa, što je osnova za prošireni model ažuriranja temeljen na DAOP-u koji podržava promjene hijerarhije klasa. Razmotreni su fenomeni izvođenja koji nastaju nakon dinamičkog ažuriranja zbog promjena u nasljeđivanju klasa te predstavljeni algoritmi za detekciju i procjenu. Nadalje, metodologija za evaluaciju performanci omogućuje razvoj DSU pristupa s procjenom utjecaja na performance izvođenja. Stoga je implementiran prototipni sustav temeljen na proširenom modelu, algoritmima otkrivanja fenomena izvođenja i metodologiji evaluacije performanci. Korištenjem prototipnog sustava trenutno pokrenut program može dinamički ažurirati verzijom programa koja sadrži promjene članova klase i promjene u nasljeđivanju klasa s procjenom pojave fenomena izvođenja i malim utjecajem na performance izvođenja.

Znanstveni doprinos je sljedeći:

- **Prošireni DAOP model ažuriranja** Analiza odnosa između klasa omogućuje detektiranje promjena u hijerarhiji klasa između verzija programa i dinamičko ažuriranje tih promjena. Klase su povezane nasljednim odnosom, formirajući hijerarhiju klasa. DAOP sa svojstvom presijecanja kao razinom indirektnosti, za dinamičko ažuriranje omogućava promjene u nasljeđivanju klasa. Kako bi se omogućila promjena hijerarhija klase uvodi se klasa *dynamic*. Korištenjem klase *dynamic* i uzorka klijenta/dobavljač (engl. *client/supplier*) proširuje se postojeća hijerarhija klase, čime se omogućuju promjene tipa s obzirom na hijerarhiju. Nadalje, klase *dynamic aspect* i *diff* omogućuje promjene članova klase.
- **Algoritmi za detekciju fenomena izvođenja** Na temelju promjena između verzija programa predloženi su algoritmi za detektiranje i procjenu fenomena izvođenja. Fenomeni izvođenja analizirani su iz perspektive promjena u hijerarhiji klasa i ovisnosti o pozivima između klasa. U ovoj disertaciji fokus je na utjecaju promjena u nasljeđivanju na fenomene izvođenja. Algoritam za procjenu fenomena izvođenja procjenjuje rizik izvođenja dinamičkog ažuriranja. S druge strane, algoritam za detekciju fenomena izvođenja rezultira informacijama o promjenama programa koje mogu prouzročiti fenomeni izvođenja. Navedene informacije mogu se koristiti za prilagodbu ažurirane verzije programa za dinamičko ažuriranje bez detektiranih fenomena izvođenja. Nadalje, za ispravno izvođenje dinamičkog ažuriranja u modelu, prijenos stanja izvodi se postupkom kojim se inicijalizira novo stanje objekta pomoću trenutnog stanja.

- **Prototipni sustav** Prototipni sustav implementiran je na temelju proširenog DAOP modela i algoritama fenomena izvođenja. Sustav se sastoji od *offline* i *online* alata. *Offline* alat analizira izvorni kôd trenutno pokrenute i ažurirane verzije te izdvaja promjene u obliku Java klasa. *Online* alat učitava kreirane klase i koristeći DAOP primjenjuje promjene na pokrenutom programu. Kako Prose kao DAOP ne omogućava redefiniciju tijela konstruktora, implementiran je dinamički *weaver* temeljen na Prose-ovoj definiciji aspekata. Međutim, osim Prose-a, mogu se koristiti i drugi aspektni jezici. Prototipni sustav evaluiran je na testnom primjeru i empirijskoj studiji. Rezultati pokazuju da se promjene u hijerarhiji klasa i članovima klase kao posljedica evolucije softvera mogu primijeniti dinamičkim aspektima. Nadalje, procijenjeni fenomeni izvođenja, zajedno s evaluacijom performanci, odražavaju primjenjivost i učinkovitost predloženog pristupa.
- **Metodologija mjerenja za evaluaciju performanci** Jedan od zahtjeva DSU-a je minimalan utjecaj implementacije na resurse sustava i performance izvođenja programa. Predložena je metodologija za evaluaciju i usporedbu performanci na temelju koje je implementiran alat za mjerenje. Metodologija sadrži mjerenje utjecaja DSU-a na performance bez dinamičkih ažuriranja, s izvedenim dinamičkim ažuriranjem, trajanja dinamičkog ažuriranja i utjecaja na upotrebu memorije. S implementiranim testnim slučajevima koji predstavlju promjene u programima, evaluacija pokazuje prednosti i nedostatke evaluiranih pristupa, u skladu s implementacijom pojedinog pristupa.

Trenutna implementacija prototipa dinamičkog ažuriranja ograničena je na izvršavanje `.class` datoteka. Stoga bi budući rad uključivao definiranje promjena programa u `.jar` datotekama. Nadalje, u trenutnoj implementaciji nedostaje podrška za promjenu anonimnih (engl. *anonymous*) i unutarnjih (engl. *inner*) klasa. Posebni slučajevi za takve klase mogu se implementirati unutar algoritma za detektiranje promjena i biti upravljani *DSU manager*-om u *online* alatu. U slučaju promjena konstruktora u v_2, kada u verziji v_1 ne postoji podrazumijevani (engl. *default*) konstruktor u nadređenoj klasi, postupak ažuriranja stvorit će naredbe koje koriste nepostojeći konstruktor. Kao rezultat nastat će iznimka ili greška u izvođenju. Navedeno ograničenje može se riješiti zamjenom promijenjenih klasa s *dynamic* klasama. Implementacija *online* alata, općenito, koristi API refleksije (*reflection*) unutar JVM-a za manipulaciju objektima. U daljnjem razvoju prototipa, može se koristiti API instrumentacije (*engl. instrumentation*) za poboljšanje performanci, jer refleksija isključuje pojedine JVM optimizacije.

Algoritmi detekcije fenomena izvođenja detektiraju fenomene povezane s promjenama nasljeđivanja. Međutim, algoritmi se temelje na statičkoj analizi i procjenjuju moguće fenomene izvođenja. Budući rad bi trebao uključivati dinamičku analizu, gdje se *online* alat može proširiti analizom gomile kako bi se utvrdili trenutno aktivni objekti. Štoviše, analiza ovisnosti poziva između klasa može se provesti na temelju trenutnog stanja stoga. Stoga, dinamička analiza može poboljšati detekciju fenomena izvođenja, pri čemu se mogu izvesti dinamička ažuriranja

koja mogu uzrokovati fenomene izvođenja, ako analiza utvrdi kako objekti detektiranih klasa nisu aktivni. Nadalje, *DSU manager* može odgoditi ažuriranje kako bi analizirao trenutno stanje u unaprijed definiranom vremenskom intervalu i izvršiti ažuriranje kada je to moguće.

Alat za evaluaciju performanci mogao bi koristiti univerzalne testove za mikro mjerenja. U trenutnoj implementaciji testovi su ručno prilagođeni evaluiranom pristupu. Međutim, implementacija univerzalnih testova zahtijeva odgovarajuće sučelje za svaki pristup kako bi se test transformirao iz univerzalnog formata u format prikladan za određeni pristup. Osim toga, budući rad može uključivati nova mjerenja poput utjecaja dinamičkog ažuriranja na korištenje memorije u stanju mirovanja (engl. *steady state*).

Ključne riječi: dinamičko ažuriranje softvera, dinamički AOP, hijerarhija klasa, OOP, fenomeni izvođenja, evaluacija performanci

Contents

Chapter 1

Introduction

1.1 Motivation

Dynamic Software Updating (DSU) is required to update a running program without interruption. Although there are several solutions to the problems related to dynamic updating, none of these solutions are currently used in the production environment. The main motivation of this dissertation is to make a step towards the use of DSU from development to the production environment in scenarios where smaller changes between versions such as new features or bug fixes that are urgently needed can be applied dynamically with minimal risk and overhead. A regular update of the program can then be performed when there is a window of time where software usage is low to avoid disrupting the end-user's work. For the program to be dynamically updated and to support the main motivation scenario, several key constraints in the current available dynamic updating solutions are detected. Detected constraints are seen as areas for further research: dynamic update model to support as many program changes as possible, detection and prevention of possible incorrect program behavior, and performance evaluation of the dynamic updating implementation. The detected restrictions are described in the rest of this section.

Many studies have detected problems introduced by the DSU in comparison to the updating procedures that require the program to stop [1, 2, 3, 4, 5, 6]. The key problems are: when to apply an update to an already running program [5, 7, 8], how to preserve the state of the program after the update [9, 10], how to support arbitrary program changes, and how to effectively detect and apply changes without distortion on the execution performance [6, 11, 12]. The proposed solutions are in the form of various concepts and techniques [1], [5], [6], [8], [11], [13]. For this doctoral research, focus is set on several characteristics of the system with DSU, as follows: update mechanisms, change specification between different versions, supported changes, impact on the state of the modified program, and execution performance.

In environments with higher programming languages where virtual machines are used, there are several concepts that use different mechanisms to achieve dynamic updating. Java is currently dominant in research work, but proposed solutions for certain problems of dynamic updating can be applied to other programming environments, such as Microsoft CLR. Modified Java virtual machine – JVM approaches [14]–[15] use modified internal JVM data structures and a modified garbage collector to change existing object references. Environment such as JVM contains API interfaces, such as JVMTI (Java Virtual Machine Tool Interface), that allow control over the execution of the running program and modifications of portable programming code (i.e. bytecode). Java agents can be used as a layer between the program and the JVM, which with interfaces such as JVMTI enable access interception and changes of classes, as in [12], [16]. The Aspect-Oriented Paradigm (AOP) can extend a program's functionality on the horizontal level; that is, it introduces cross-cutting concern [17]. Aspect program code weaves within the program during the compilation or load, such as with AspectJ [18]. However, Dynamic AOP (DAOP) enables dynamic weaving of aspects that can be used for dynamic program update. DAOP can be implemented using a modified JVM [10, 19, 20] and JVM agent [21, 22].

The specification of the changes between different versions of programs is not universally defined because it is present in the form of modified program classes [12], [15], custom defined format [14], [9] or a custom defined programming language [19, 23]. In research work [24], [25], where the concept of DAOP is extended to enable DSU shows that dynamic aspects, with its cross-cutting functionality (i.e. program code isolation), can be used as a suitable format to describe changes. Furthermore, dynamic aspects provide flexible addition and deletion of the program changes, and thus can enable incremental dynamic updates [25].

For the supported changes, DAOP is limited to a method body change. To expand the supported change set, it is necessary to generate changes in the form of other types of aspects (e.g., accessing class field) and additional classes. In the extended DAOP approaches [24], [25], dynamic changes are performed by analyzing different software versions and generating dynamic aspects. The modified JVM and Java agent provide a greater set of changes in comparison to extended DAOP, which is limited to changes on the class level. The objective of research using DAOP is to enable as many aspect-functionalities as are supported by static aspect approaches, such as AspectJ [18]. These DAOP approaches would allow simple specification and application of arbitrary changes between program versions.

Aspect languages are not built to define program change specifications and support dynamic software evolution;; therefore, aspects cannot define changes in the class hierarchy [24]. For example, AspectJ supports a limited addition of class interfaces and it does not support interface deletion [26]. Furthermore, Prose [27] as DAOP used in [24], [25] does not support method calls from the weaved (i.e. advice) code to a method defined in the same aspect, nor is it possible to access the field defined in the same aspect. In addition, it is not possible to access a field protected by the access modifiers (e.g. `private`) and does not support constructor body replacement. Although these functionalities may be achieved by declaring changed members in additional classes and extending existing classes with inter-type declarations, the approach described in [10] does not address the problem of the class hierarchy change.

Several approaches in the field of research have matured to be used in development environments [3], [15], [16]. The next step is to use dynamic updating in production environments. Program execution in development environments simplifies the conditions that an approach for dynamic updating should meet. Considering that program execution is short, the state conversion is simple and static analysis is performed to validate the correctness of the program changes. In addition, approaches from the development environment do not take into account incremental updates. Consequently, runtime phenomena can occur with dynamic updates [3], [28]. Runtime phenomena is the state of the program after the dynamic update that is not the same as it would be after the standard update procedure, as follows: stop, update and start the program. Research work from [3], [28] discusses the need for the tool with dynamic analysis of program changes, to determine whether dynamic updates can lead to the runtime phenomena. Meanwhile, in the extended DAOP, analysis is used to identify and convert program changes to aspects. In the dynamic evolution of aspect-oriented programs [25], analysis between different versions of programs is used to support incremental updates. Currently, there is a lack of research on analysis of the program changes concerning the occurrence of the runtime phenomena. Detecting runtime phenomena would enable the DSU to prevent updates that could cause the runtime phenomena by refusing or by dividing the update into smaller parts to perform the update.

The performance of the dynamically updated software can be degraded compared to software updated with restart procedure [29]. To take advantage of DSU, it is necessary to minimize impact on computer resources. The current DSU approaches were evaluated according to the execution of implemented mechanisms that allows dynamic updates; for example, the execution duration of modified garbage collector in [15], [9] or the impact on the execution of the program in steady state [9, 14, 15], [19, 27]. Although some approaches contain evaluation by using real-world programs, such an evaluation cannot show an overall impact on the execution performance because measurements are limited by the domain of the tested program [14], [10], [9]. To evaluate the approaches from the point of impact on computer resources and compare the results, it is necessary to develop a benchmark tool that can perform various measurements of the impact of dynamic updating on execution duration and memory consumption by using pre-prepared tests.

1.2 Contribution

The result of this research is a DSU system based on dynamic aspects (DAOP) and runtime phenomena detection. The presented model of the DSU is based on dynamic aspects with support for the class hierarchy modification, where dynamic aspects and additional classes describe program changes to be applied in the process of DSU. Furthermore, as a consequence of dynamic updating, dynamically updated programs are exposed to an unwanted or unexpected program known as runtime phenomena. In this research, algorithms based on program analysis are presented to detect these program state occurrences. The proposed model, algorithms, and techniques are evaluated using a prototype system and by empirical evaluation of actual program modifications. Benchmark methods are created to evaluate the performance and to compare the prototype system to other approaches.

The contributions of this thesis are as follows:

- DAOP update model that enables class hierarchy modification;
- Algorithms to detect runtime phenomena;
- Prototype system based on DAOP update model and detection of the runtime phenomena;
- Benchmark methods for performance evaluation of DSU approaches.

1.3 Thesis outline

This thesis is structured as follows. Chapter 1 introduces the thesis, including the research motivation, scientific contribution, and thesis outline. Chapter 2 introduces the DSU by describing its requirements and challenges. Chapter 3 provides a background on the object-oriented environment, briefly describing current DSU approaches for Java and the class inheritance relationship necessary for the thesis. Chapter 4 introduces tree dissimilarity as the basis to detect changes between program versions. Chapter 5 presents the extended DAOP model to support class hierarchy changes and class member changes regarding inheritance. Chapter 6 discusses runtime phenomena and introduces algorithms to detect and estimate runtime phenomena based on the class hierarchy changes. Chapter 7 describes the impact of DSU implementation on the performance and system resources, introducing a methodology to evaluate and compare the DSU approaches. Chapter 8 presents the prototype system based on the extended DAOP model and runtime phenomena detection algorithms. Chapter 9 evaluates approach based on the prototype system and defined methodology for evaluation and comparison. Finally, Chapter 10 summarizes the thesis and draws a conclusion.

Chapter 2

Dynamic software updating

In this chapter, the DSU is described as the introduction to the dissertation research area. The presented categorization of existing DSU approaches based on the work from [30] is the result of DSU requirements and their relationships. The described characteristics of DSU approach depend on the mechanisms and techniques to achieve given DSU requirements. In addition, this chapter outlines existing challenges as motivation for further research, where some of the challenges are addressed the following chapters.

2.1 Introduction to dynamic software updating

Software changes over time because of changes or corrections in its functionalities. As a result of these changes, new software versions are produced. Software updating replaces the current software version with a new one. Early distributed software systems, such as the airline reservation system, required both software availability and functionality changes [2]. However, these two requirements are contrary to the updating procedure that has the following steps: stop, update, and software start. The main drawback of this cyclic process is that the program is unavailable during the update. Therefore, it is difficult to balance high availability with frequent changes.

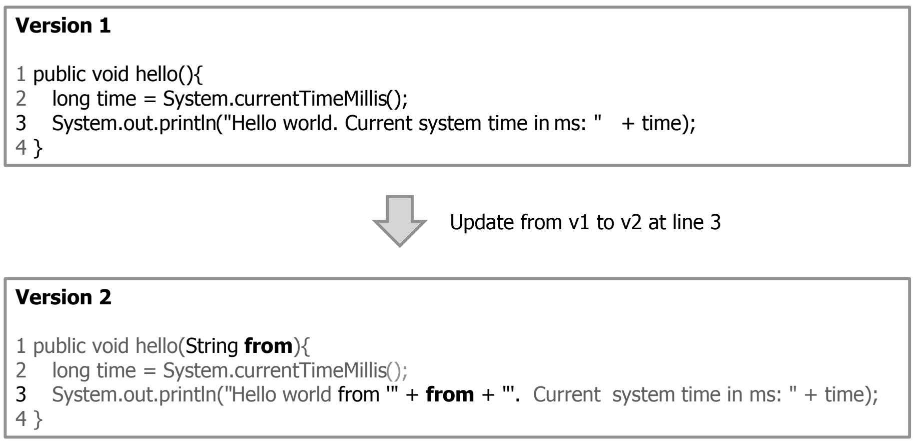
DSU has been the focus of many research studies. However, there is no agreed definition of DSU within the research community. Briefly, DSU replaces the software version in runtime, therefore DSU environments provide high availability and support for changes [11, 31]. Compared to the software updating cycle with restart, dynamic updating consists of two continuous steps: deploy and change. A wide range of applications and systems could benefit from the DSU. In embedded systems for mission critical applications availability is highly needed. For example, in power or traffic lights control, it is necessary to enable the changes with the new control strategies used for smart control in cities. Cloud systems are required to provide high availability in Platform as a Service (PaaS) and Software as a Service (Saas) services due to end-user agreements, such as Service Level Agreements (SLA) [32]. In the case of downtime, the service provider compensates the users for unavailability [33]. There are some solutions to support availability with changes, such as a rolling update for distributed and cloud systems that updates the nodes in a distributed system one by one, while redirecting clients to an active node. However, this can overload an active node with too many client requests, causing unavailability. Meanwhile, for operating systems (OSs), a restart of the system can negatively affect the working process of the end-user. Furthermore, client applications connected to other entities (e.g. web servers and remote databases) update due to changes in those entities, which can lead to lost data. In mobile applications, for example, an update of a communication application disrupts the end-user's possibility to communicate to other users during the update. In business web applications, where the application is a part of the business process, non-availability creates higher costs due to downtime. There is a constant struggle between availability and support for changes in software applications and systems, which is a key motivation for DSU.

2.2 Dynamic software updating requirements

To date, many different techniques and approaches have been developed using several systems and applications. However, DSU systems have requirements that are the same regardless of environment features and constraints. Together with the previously mentioned availability and change support, they are:

- *Availability* – performing DSU does not affect the availability of the software, or it is not noticeable by the end-user [12].
- *Correctness* – dynamic updateability does not violate correct execution of the software, before, during or after the update [12].
- *Changeability* – the set of possible changes between two versions of the software should be without constraints [12].
- *Performance* – the possible increase in demand for system resources (e.g. processing power, memory, and storage) to support DSU should be minimal [12, 29].
- *Usability* – DSU usage should be simple and transparent [12, 29].

Some of these goals are contrary. It is challenging to provide a very large set of possible changes and simultaneously provide minimal usage effort for the developer, while minimizing performance cost. Almost every DSU approach balances between these requirements. Meanwhile, availability and correctness should be viewed together; for example, by considering that the software is running without disruption, before, during and after the update but the program execution during or after the update is corrupted. This kind of software behavior is not desirable.



Version 1

```
1 public void hello(){
2    long time = System.currentTimeMillis();
3    System.out.println("Hello world. Current system time in ms: "   + time);
4 }
```

Version 2

```
1 public void hello(String from){
2    long time = System.currentTimeMillis();
3    System.out.println("Hello world from '" + from + "'.  Current  system time in ms: " + time);
4 }
```

Figure 2.1: An example of a failure: argument in new version do not exists in previous version

2.3 Related problems

To achieve the requirements from Section 2.2, DSU is confronted with several problems when compared to the software update procedure with restart. Updating an active program code at an unsafe moment can result in unpredicted software behavior. For example, in Figure 2.1 the dynamic updating could be performed in the middle of the function `hello` at line 3. A function in a newer version after the point of updating executes a statement comprising the newly introduced parameter `from` that is undefined in the previous version, which leads to a fault state. Another problem is to ensure a smooth end-user experience during the update. DSU is required to provide transparency in such a way that a component of the previous version can use a component in the new version. When DSU updates the entire program at once, this kind of problem is not an issue because the components are upgraded at once. However, the replacement of the whole program often takes long time, which degrades the availability requirement (Figure 2.2). Moreover, the whole program replacement often encounters issues of correctness because the current state of the program can be lost. In contrast, during the running application update, the DSU is required to preserve the current state to support correctness and consequently to provide transparency to the end-users. To preserve the program state, multiple program versions can exist simultaneously. If there are multiple versions of the same program object in the system (e.g. a class instance in memory), then the demand for resources increases and this can degrade system performance.

These problems differentiate the DSU update process in comparison to the classic update process. Other problems arising from the DSU implementation related to various software environments are described in Section 2.5.

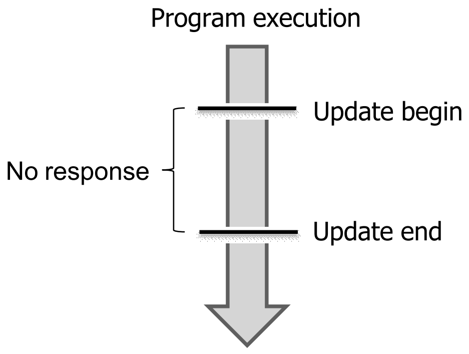


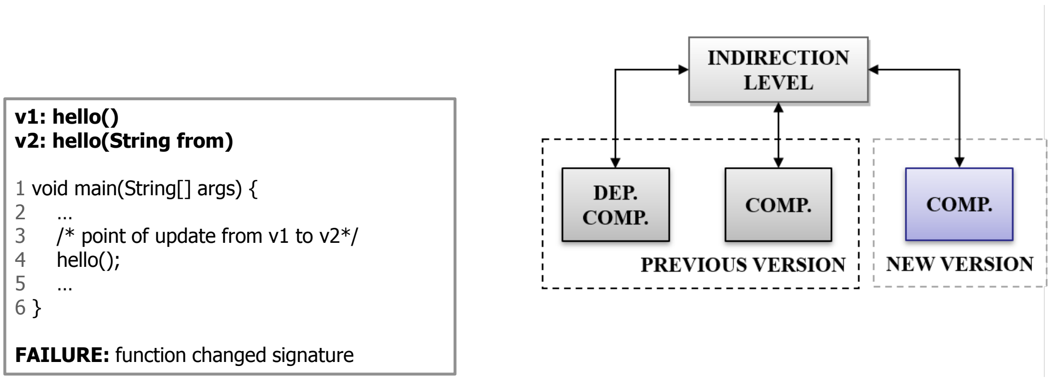
Figure 2.2: Dynamic update duration

2.4 Updating techniques and mechanisms

The DSU related problems described in Section 2.3 can be resolved with appropriate techniques divided into the following categories: level of update, update of dependent components, time of update, state transfer, and cleaning and rollback mechanisms. The level of update determines which parts of the software will be updated, such as a single component or the whole program. In dynamic updating, the dependency between components when components are of different versions is handled with an update of the dependent components. State transfer enables the state conversion from the previous to the new version in terms of system correctness and end-user level transparency. One of the questions addressed by the time of the update is when to update the active program code. Timing techniques determine the time of update in different parts of the software system. Meanwhile, the system resource requirements increase with dynamic updating. Therefore, proper cleaning of the unused memory fragments from the previous version maintains the system availability and performance. Errors may occur during the process of the dynamic update, and there is a need to provide the possibility to rollback changes to the previous version to maintain system availability and correctness.

2.4.1 Level of update

DSU can be designed to replace the whole program at once or to replace a single component (e.g. class [12, 15], class members [12], method [34, 35]) or instruction [36]. To reduce the time of update, smaller components of the update are chosen; for example, instruction-level as presented in DynSec [36]. Such a size of the unit is appropriate for security patches. Changing a single instruction can fix a bug (e.g. buffer overflow). It is unnecessary to update entire program because of single instruction change in security patch. To keep the update duration and performance overhead as minimal as possible compared to the whole program, components are used as the common level of update in various approaches.



v1: hello()
v2: hello(String from)

```
1 void main(String[] args) {
2    ...
3    /* point of update from v1 to v2*/
4    hello();
5    ...
6 }
```

FAILURE: function changed signature

(a) Example of a dynamic update (b) Indirection level between components

Figure 2.3: Dynamic update example requiring indirection logic

2.4.2 Update of dependent components

Fabry in [2] introduced the indirection level for dependent components to resolve problems that arise when dependent components have different program versions. Indirection level as a component connects the dependent component to the components in the different versions (Figure 2.3b). In the example shown in Figure 2.3a, the functions `hello` and `main` are in different components. A dynamic update is performed before the call to function `hello`, at line 2. Because the `hello` function signature changed in the new version (v_2), the caller method in dependent component from the previous version (v_1) cannot invoke the method without parameter in the calling component in v_2, and program execution ends with failure.

The connection between components is in both directions because the components have both an input and an output. The indirection level has built-in logic for handling the connection between the previous and the new component version. In the example in Figure 2.3a indirection level handle calls from `main` function dependent component to component in the new version. The simple approach is to use a jump or jump-like instruction as in [1, 31]. Frieder and Segal [31] use a special segment register as indirection level, which contains the address of the procedures lookup table. However, this approach depends on the CPU architecture. In the example in Figure 2.3a, register is pointing to method `hello` in v_1, after the dynamic update it points to method `hello(String from)` in $v2$. Other approaches to the indirection level can include proxies redirecting calls to new versions [12]; wrappers, where a new version "wraps" the previous [37]; and pointers, which updates pointers from the previous to the new version objects [11, 15]. The Dynamic Proxification Framework (DCF) [12] uses the byte-code rewriting technique to insert the code in the previous component version, which then calls the new version of the component. E.g. in an Object-Oriented environment, methods `hello` and `hello(String from)` from Figure 2.3a are methods belonging to updated class in v_1 and v_2, and method `main` to unchanged class. DCF is rewriting method `main` body with statements to invoke method `hello` defined in v_1 or v_2 depending on the currently running version. Wrapper approach also modifies statements of the `main` method in order to use wrapper classes. However, wrapper class contains only changed class members between versions. Meanwhile, pointers in [15] are changed from v_1 to the v_2 objects by analyzing heap. For the example in Figure 2.3a virtual method table in the updated class needs to be modified because parameter for method `hello` is added in v_2. Updated class is reloaded resulting in change of pointers for existing objects.

Dependent components can be dependent statically [7, 12]; determined in static state (e.g. in software code); or determined semantically (i.e. in runtime) [31]. Static dependency can be determined automatically, whereas semantic dependency is defined by the programmer or detected in a higher-level analysis. Both types of dependency between components are required to provide the correctness of the DSU. The most common example of both dependency types is a software system where communication is performed. The communication program consists of two functions: sending and receiving messages. Changing one function requires changing another; otherwise, the function for sending the message sends the message in the previous version but receives the message in the new version. Consequently, the receiving function cannot receive the message properly due to the change between the versions, which leads to incorrect behavior of the communication process. These cases are solved with the use of the programmer's defined semantic-dependencies list [31].

2.4.3 Time of update: safe point of update

The point in time when it is possible to make an update is important to maintain the correctness of the running software. In [38], the authors proposed three categories of update timing control regarding the update of active functions: activeness safety (AS), con-freeness safety (CFS), and manual identification. AS prevents the update of active functions (i.e. existing on the call stack). Con-freeness [7] allows updating of an active function when it is type-safe. An update is type-safe when the block of code after the point of the update is not affected by the update. To ensure the correctness, both AS and CFS requires code modification by the programmer [38]. Manual identification relies on the programmer-defined safe points of update [1, 38, 39]. In single-threaded applications with one loop, the point of the update is suggested to be at the end of the loop when there are no active transactions [15, 29]. Meanwhile, defining the points of the update is complex in multi-threaded programs. The programmers are required to follow programming patterns because of synchronisation problems, such as a suggestion for the loops.

Some approaches do not handle update timing [12, 40]. In contrast, other approaches such as: quiescence [5], tranquility [8], and relaxed synchronisation [41], manage updates with delay. A brief description for each follows. First, assume that in DSU, depending on the application and system type, entities of DSU environment affects the correct running and the behavior of the system; for example, entities are functions in procedural programming languages, nodes in distributed systems, processes in operating systems, transactions in databases and communication systems.

1. *Quiescence* [5]. To perform the operation of the dynamic update on an updatable entity, the entity is required to be inactive. Further on, entities reachable by an updatable entity are required to be inactive, and other entities that might reach the updatable entity will be inactive during the update. This concept has been used in [5, 31, 42].
2. *Tranquility* [8, 43]. As relaxed quiescence, it does not require that the entities that can reach updatable entity are required to strictly be inactive during the update. Instead, both the entities connected to the updateable entity and the updateable entity itself are inactive and will both remain inactive during the update process. Furthermore, the connected entities will not be active when the updateable entity is inactive.
3. *Relaxed synchronisation* [23]. There are points of update that are equivalent in the program code, which means that it does not matter where the update will occur in the program code. The equivalent update points create a block of code where the update does not affect any of the block statements. If the update request is received in the middle of the block, then program (thread) execution can continue until the end of the block and the update will then be executed.

The time of the update is not exactly determined in the First-Class Context [44] approach, which is called an incremental update. Active transactions and requests run in the previous version until the end of the transaction or request. After the update, every new transaction or request runs in the new version. This is a lazy update from the previous to the new version.

2.4.4 State transfer

State loss as a result of a dynamic update can lead to exceptions and end users can experience missing features or state, similar to a cold restart. A simple prerequisite that is used by some approaches is to assume that the new version uses the state of the previous version [40], which mostly induces reduced changeability. There are two basic categories: automatic conversion and programmer-defined conversion. Automatic conversion [12, 15] copies object data from the previous to the new version. This includes, in some cases, the conversion of complex types (e.g., parent class change), and for primitive types, the conversion of an associated value (e.g., a number to string). Moreover, if some of the structure is changed (e.g., when the field is added to class), then the default programmer-defined values are used. The manual approach [31, 33, 44] includes programmer defined transformation functions to transfer the state from the previous to the new version. This approach extends flexibility when a major change between versions occurs, but it also disrupts programming transparency. Defining a custom transformation code often requires the programmer to use new language constructs and follow conventions or design patterns, related to dynamic updating.

Another aspect of transformation is the time when it occurs, either immediately or after the incoming update request. On-demand is a popular technique for performing state transformation after receiving the update request [1, 2, 45], which is also called a lazy transformation. The value of the field is transformed on the first access, as seen in [12, 46]. Lazy transformation is recommended because it has a good impact on the performance and the duration of the update. Meanwhile, transformation functions can be bidirectional [1, 44, 45] to maintain synchronisation between the entities belonging to different versions, which is common in approaches with multiple versions. The change of entity state in one version affects the entity of another version.

2.4.5 Cleaning

In higher-level languages, DSU relies on the garbage collector mechanism. When the old components are no longer used, the garbage collector marks them for cleaning [9, 15]. In [44], when the old context is not in use, it is finalized. The finalisation process migrates old context objects to the new context and the old context is then garbage collected.

2.4.6 Rollback

In the case of errors during an update, to support the availability and correctness of DSU, the system is required to provide a rollback feature [1, 12]. However, rollback cannot be done in environments without reversible actions. In general, re-executing and rollback are mechanisms where the program execution rolls back to the first point where the previous and the new software versions are equal [47]. POLUS [1] supports reversible patches that convert the currently running to the previous version. Rollback may be designed as a part of multiple versions existence during the update when there is a need for the rollback in case of exceptions.

2.5 Dynamic software updating implementation

Given that DSU implementation largely depends on various environments and usage, there are different restrictions and features to consider. Programming language defines implementation platform, type of application and runtime environment determine architecture of DSU, whereas type safety, defining set of changes, concurrency, and coexisting of multiple versions are implementation features.

2.5.1 Programming language

Some DSU approaches define a custom programming language to enable a dynamic update capability, such as in [7, 45, 47]. Custom language DSU requires programmers to learn new programming languages. Such DSU supports a large set of changes and long-term usage simplicity, as the programming language is developed with dynamic updating features. Another approach is to use a programming language popular among developers, such as C, C++, Java [48]; as in [1, 12, 15]. Using a popular programming language without modifications can increase the complexity of the DSU implementation because of a lack of dynamic updating features in the programming language. Meanwhile, a less popular programming language, such as in [2, 44], is often used when the environment (e.g. legacy or system constraints) does not allow another solution. The programming platform is defined by the programming language, supported Application Programming Interfaces (APIs) and libraries connected to the running environment, such as the Operating System (OS).

2.5.2 Application type

Implementation depends on the application type because of the environment properties and the purpose of the system. DSU for an embedded environment [49] has a simpler implementation. Embedded software has a smaller memory footprint, which means that the state transfer is simpler for handling. Object-oriented programming languages are not often used for embedded systems as opposed to business applications, which increases the demand for system resources. Furthermore, current business applications often run on web [1, 7, 38] or cloud systems [33, 50]. These systems can be distributed due to load balancing. The technique for uniformly balancing load over each node serves a large set of users, which means that the DSU implementation performs synchronisation between multiple nodes [50]. Standardized business applications consist of three layers: database, server, and client. The dynamic update of a business application on a three-layer architecture requires a separate update of each layer, such as database [2, 51], with a synchronisation mechanism between these layers [50]. In Figure 2.4 shows a distributed system with multiple nodes and separate layers. Furthermore, for DSU handling operating systems, the time of the update can be simpler to detect because the function in the process can be blocked instead of inactive [29]. The Linux operating system provides a "hook" (*ptrace* function) for easy injection of code into a running application [1]. Built-in mechanisms such as *ptrace* simplify the implementation of DSU supporting the dynamic update of OS modules. Native (i.e. client) applications regarding DSU [12, 15, 40] perform updates during an inactive state similar to OS blocked processes.

2.5.3 Runtime environment

There is a difference between implementing long-term and short-term DSU. Current short-term DSU approaches are used for development environments, such as IDE (Integrated Development Environment) as in [12, 15]. Meanwhile, a long-term DSU would tend to be used in a

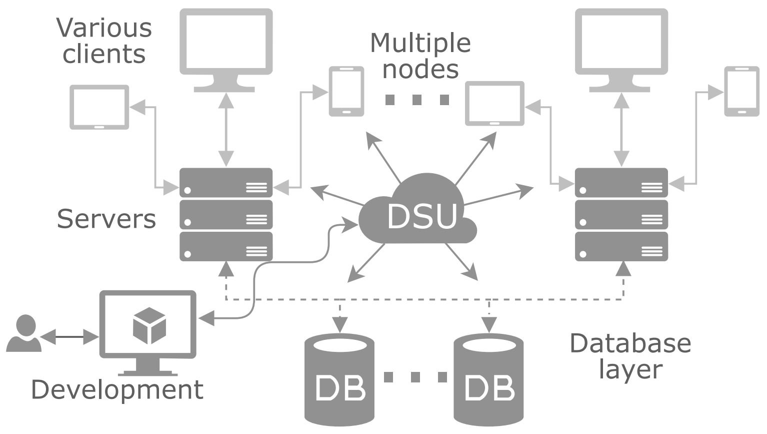


Figure 2.4: Distributed software system with dynamic updating

production environment. Although various approaches in the existing research can solve many DSU runtime challenges, there is no research on a long-term running DSU for the production environment.

In the development environment, DSU is a tool to aid software debugging activities (e.g. refactoring or bug correction). Current commercial level available solutions include HotSwap [34, 40] in Eclipse IDE, and *Edit and Continue* [35] in the Microsoft Visual Studio IDE. These IDEs support method body changes without changing the method signatures or type changes, with limitations for adding the class members in object-oriented languages. The programmer can change or add an active method statement while debugging using the same method, mainly to correct bugs. The change immediately affects the functionality of the program and further execution. In [12, 15] the authors extended the set of changes when compared to available IDE mechanisms, enabling the change of class hierarchy and adding or deleting members of classes. In these cases, it is necessary to ensure binary compatibility, described in more detail in Subsection 2.5.4.

The techniques and methods developed for the debugging environment could be applied in a production environment, although with modifications due to differences in these environments. The debugging environment provides simple detection of point of update and simplifies the state transfer process. The breakpoints in the debug mode are the points of the update, which are points in a program that can suspend execution to perform dynamic update. Meanwhile, production environments increase the complexity of the implementation because of continuous execution.

2.5.4 Type safety: binary compatibility

If the DSU level of the update is smaller than the replacement of the whole program, then the update of the dependent component handles type safety. This is particularly important in approaches relying on static programming languages because of the static type checking. Chang-

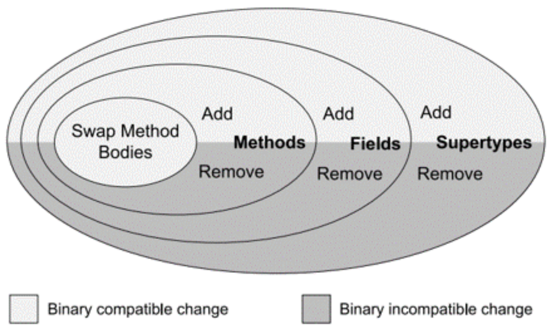


Figure 2.5: Binary compatible and incompatible changes [15]

ing the component interface is a modification of the component interface signature (i.e. type change). One of the first approaches using a static type checking was DYMOS [39] using StarMod distributed version of the Modula programming language. Approaches that do not support interface change do not have a requirement to handle type safety [40, 52]. Other approaches with support for interface change perform a static check [12, 15, 39] or use functions to provide type safety [1, 11, 45]. In a DSU with a static check, the updates that fail type safety checks are rejected to maintain the correctness.

DSUs with a convention that updates are required to be type-safe because they rely on the fact that in the update procedure with restart the programmers compile changes before deploying the program [12]. Compiling ensures the type safety (i.e. binary compatibility of the new version). As shown in Figure 2.5, adding new objects such as classes, methods, fields, and variables into the code is a binary compatible operation. Meanwhile, removing the objects is considered to be an incompatible binary operation (i.e. type unsafe). Changing the body of functions or methods is also binary compatible change, and is widely supported in the debugging environments that are described in Subsection 2.5.3. Binary incompatible changes can be handled with coexisting multiple versions and with the use of predefined behavior, such as in [15]. The binary incompatible functionalities that are introduced in [15] are:

- *Static check* – if a deleted object is accessed in further program execution, then the update is rejected,
- *dynamic check* – search for reference during runtime, returns an exception on fail.
- *Access deleted member* – supports the access of an old deleted method or static field, whereas for deleted instance fields returns an exception.
- *Access old members* – method in previous version access the old version of the method instead of the new version (i.e. version consistency).

Furthermore, in databases, type safety consists of a data scheme that compares the previous and the new version. In [33], scheme comparing is called a *safety check*, which ensures that accessing the modified object (in this case, the database table) is always performed on the new scheme version. There are two cases, presuming that Δ is a set of all differences in the scheme between versions. The differences are: added, deleted, or modified table attributes. A safety check before accessing the table T performs the intersection between Δ and the scheme of table S_T. In the case of the empty intersection $\Delta \cap S_T = \varnothing$, the table is accessed because there are no changes. Otherwise, $\Delta \cap S_T \neq \varnothing$ and access is delayed as long as DSU is performing the migration of the scheme and the existing data from the previous to the new version, corresponding to lazy transformation.

Dynamic languages such as PHP, JavaScript, and Ruby do not have explicit type declaration because types are handled dynamically. This feature can simplify the implementation of DSU because it does not have to handle type safety as in static programming languages.

2.5.5 Concurrency

Multithreading or concurrency support in DSU is closely connected with the time of update described in Subsection 2.4.3. Some approaches [40, 45] do not consider multithreading, even if their handling of update timing can provide multithreading support [40, 45]. Other approaches that support multithreading, such as [12], advise programmers to use programming patterns and practices in multithreaded programs. These are common when developing multithreaded programs because developing multithreaded applications is often complex due to synchronisation between threads. DSU that includes multithreading support should avoid the occurrence of deadlocks. A DSU can be implemented by using proper technique for the time of update and modifying system calls (e.g. with indirection) used for synchronising threads. In [12], the Java environment provides `wait` and `notify` synchronisation functions. The former is used to block the execution of the thread and the latter is used to release the waiting thread. A problem arises when in the previous version of the object, function `wait` is called, causing thread waiting, and after the update calling of the `notify` function is performed on the new version of the object. The thread in the previous version never ends because it will never receive unblock instruction. The changes between versions in program code handling synchronisation, such as unblocking the thread, should be done with caution. Changed functionality in the newer version of such objects can leave threads waiting, which causes deadlocks [12]. These changes are not compatible with dynamic updating.

2.5.6 Defining the set of changes: the set of differences

The difference between versions can be determined automatically by static analysis tools on a code level, defined by the programmer, or determined at the higher level of abstraction or design level [1, 47, 49]. The code level can be object code, such as byte-code level [52] or source code level [9, 11]. However, the source code of previous versions is not always available. In such cases, object code has an advantage over the source code. Furthermore, comparing on the lower level may introduce unnecessary updates of negligible changes. Meanwhile, comparing small changes in the case of optimisation or bug correction can on a higher level cause update rejection because no changes are detected [29]. For example, single changed instruction where the size of an array is defined can be neglected when comparing program versions on the higher level using, such as Control Flow Graphs (CFG) [49]. Lower level automatic comparing is simpler to implement than the automatic comparing on a higher level, such as class hierarchy. Automatic comparing relies on static analysis and is often implemented as a separate tool that produces differences in a format known in advance. The format can be custom [9] or in existing format such as XML (eXtensible Markup Language). In [36], the difference is produced by a special tool to compare the differences of the functions binary code. Furthermore, software versions can be stored in a repository, such as a database, where an update to a specific version is determined by repository analysis [24]. From the point of implementation, differences defined by the programmer as part of an update are the simplest case, but it degrades the simplicity goal.

2.5.7 Coexisting of multiple versions

It is desirable to have multiple versions at the same time, such as in cases of cyclic dependencies, when in one version the class depends on another, while in the next version the second class depends on the first [15]. Multiple versions of the same object in memory increases memory demands and affects the correctness. The correctness may be degraded because DSU provides version consistency [53], meaning that dependent objects are required to be compatible. If multiple versions of the same object exist, then it is challenging to ensure state synchronisation between them. In distributed systems, there can be more than two versions which increases the complexity of synchronisation between the system entities.

2.6 Challenges

Many software applications in a distributed environment consist of multiple layers and nodes. Figure 2.4 shows a software system that is based on three layers: database, server and client, with multiple nodes. Although layers are presented as separate entities, they can be on one physical place as a single node. In real-world situations, current business applications have physically separated layers. The database layer can be distributed on multiple servers or it can consist of a main database server with a secondary failover server. The main application logic layer often consists of multiple nodes (i.e. servers with load balancing). The clients include various devices, such as desktop computers, mobile or IoT devices. The client-side of the application usually resides on the Internet browser but can be the native application. DSU logic can be placed depending on the implementation: inside the IDE, distributed in each node, or on a single physically separated location.

Having an entire system in mind, several DSU challenges can be considered. Based on the requirements of the real production systems and related literature, the rest of this section discusses some of the general challenges. Furthermore, in the following subsections, challenges related to DSU addressed in this dissertation are discussed.

DSU concepts applied in the debugging environment could also be applied in the production environment. However, there is a lack of research on deploying dynamic updates in the production environment. Further DSU research and new approaches can provide developers in the development environment Existing DSU with further research and new DSU approaches can provide developers in the development environment the ability to change code on the fly when debugging and dynamically deploy a new version to the web, distributed or embedded distributed system, as shown in Figure 2.4. However, more research is needed to develop further DSU enhancements related to connecting development environments to the production environments.

In the literature, DSU is rarely found for cloud and web applications. Bhattacharya and Neamtiu [50] considered the problem of each layer separate dynamic update validation and synchronisation between multiple nodes. However, there is a lack of research beyond problem discussion related to dynamic updating managing and synchronisation between multiple layers. DSU is required to consider consistency between nodes and layers, including possibly different versions in nodes. A situation where the system contains different versions across nodes can appear temporarily during an update, or intentionally when different groups of nodes serve different clients. For example, the development system can contain a test instance. Therefore, before deploying to the production environment, a new version is deployed to the test instance. Distributed and multilayered systems require an appropriate centralized manager to analyze the current system state, and to perform dynamic updating across layers and nodes. However, it is challenging to determine and design parts of such a centralized updating manager handling different versions in nodes and layers.

2.6.1 Runtime phenomena: state artifacts

In [28], Gregersen et al. discussed runtime phenomena, which is a condition where the system state after the dynamic update is applied is not equal to a system state when the update procedure with software restart is applied. The following phenomena is detected [28]: phantom objects – removed class objects remain in application, absent state – object is missing state in newer version, lost state – state is lost with type change of the class member, oblivious update – features introduced in the newer version are missing, broken assumptions – after applying multiple new versions, state and logic dependency assumption may break objects, transient inconsistency – application state occurred after the update that will never be reachable by a restart.

More research is needed to detect and avoid such conditions. Chapter 6 explains in detail the problems of runtime phenomena and presents solutions as a result of dissertation research.

2.6.2 Evaluating dynamic software updating implementation

Performance benchmarks from existing research focus on overhead introduced by the DSU or compare program execution speed before and after performing a dynamic update. Memory demand comparison influenced by DSU capability is rarely provided. However, this comparison is needed because current applications are memory demanding. In the case of the DSU approach that supports the coexisting of multiple versions of the running program, the memory demand after multiple updates should be in the worst-case equal to a cumulative number of running versions. Meanwhile, a common situation in the computer systems when there is an advantage in the speed of software execution, there is higher memory usage, and vice versa. A stable DSU system is required to balance those two requirements in terms of dynamic updating. Further development of appropriate DSU benchmarks, similar to [54], considering the demands for speed and memory resource demands is needed.

Proper benchmarks are needed to evaluate availability across the system in different nodes and to check the correct behavior of the system. Besides correctness and availability evaluation, evaluation is also needed for comparing different DSU techniques and methods, and update failures analysis. The challenge is how to design and perform such an evaluation with different parts of the complex system in mind.

In order to compare the DSU approach presented in this dissertation with other approaches, Chapter 7 describes the methodology for evaluating different DSU approaches and benchmark tool.

Chapter 3

Object-oriented environment

Each programming environment contains specific constraints and mechanisms to support dynamic software updating. The focus of this dissertation is the hierarchy of classes in Object-Oriented (OO) programming languages. Consequently, this chapter will give an overview of DSU approaches in Java because an OO environment is given, and it will describe the current approaches regarding the Java environment. A brief description of the key requirements from the previous Chapter 2 is given. Although there are other OO programming environments, such as C#, the currently available research is based on Java. Therefore, OO concepts in this dissertation are explained in the Java environment, although these concepts should be applicable to similar OO environments. Furthermore, class hierarchy results in inheritance relationship between classes, introducing mechanisms such as method overriding. Changes between program versions are described regarding class hierarchy as preliminaries for the extended DAOP model and runtime phenomena. Class hierarchy changes between program versions (besides statistical and algorithm comparison) can be observed visually, as described in the last section of this chapter.

3.1 Java approaches

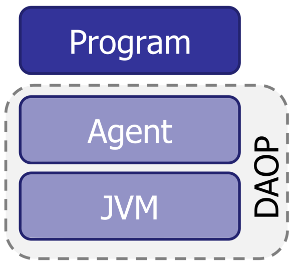


Figure 3.1: Java environment stack

The placement of the dynamic updating logic is a crucial factor of every DSU approach. In the Java programming language environment, dynamic updating can be executed at different environment hierarchy levels (Figure 3.1), and therefore using different techniques. The execution environment of a higher level programming language such as Java relies on a virtual machine (i.e. Java Virtual Machine – JVM). JVM is an intermediate level software execution environment that supports code portability, and is responsible for executing and translating intermediate portable code (Java bytecode) to machine code. Concepts involving the modification of the virtual machine perform the program change with the assistance of modified JVM internal data structures, class metadata, stack and modified garbage collection mechanisms; as seen in [9, 15].

Programming environments such as JVM support specific API (Application Programming Interface) mechanisms (e.g. JVMTI - Java Virtual Machine Tool Interface agent or Java agent). Those APIs can intercept access, and modify program constructs (e.g. classes) and current program state. JVM agents also provide bytecode manipulation, which all together can be utilized for dynamic software updating. Concepts utilizing JVM agent often involve intermediate objects between a previous and a new object version as "proxies" and wrappers in [12, 37]. Additionally, to cope with different class versions, different class loaders [12] or class renaming [37] can be used.

Aspect-oriented programming (AOP) enables the program functionality to be extended on horizontal hierarchy level (cross-cutting concern) [18], such as for logging or security access. AOP uses a weaving mechanism to load the program code within an aspect during compile or load time, as with AspectJ [11]. Dynamic aspect-oriented paradigm (DAOP) enables weaving during the runtime, which means it can be used for dynamic software updating. It is implemented as a JVM modification [10, 19, 20] or JVM agent [21, 22].

Regarding the architecture, mechanisms, and paradigms found in the currently available approaches, concepts are classified in the following three categories: modified virtual machine (modified JVM - mJVM), level between virtual machine and executed program (JVM agent - JVMa) and concepts based on DAOP. Figure 3.2 shows the Java environment vertical hierarchy and the proposed categorization.

Besides this categorization, Java concepts can also be compared with the following key characteristics, as mentioned in Chapter 2: update timing, necessary program adaptation and supported changes. These characteristics will be described in the following subsections.

3.1.1 Update timing

One of the crucial problems in dynamic software updating is to determine a proper moment in time to perform a dynamic update. This problem has been extensively discussed in the related literature [5, 7, 8]. However, Java concepts mainly rely on internal JVM mechanisms. A dynamic update is executed at JVM safe points [9, 14, 15], with delayed or refused update when updating active code [9, 14]. At safe points, threads are stopped; for example, to per-

Type	Level	Program adjustments
mJVM	3	DPR
JVMa	3	DPLR
DAOP (mJVM)	2	DPCLR
DAOP (JVMa)	2	DPCLR
Standard JVM	1	D

Figure 3.2: Java DSU approaches classification

form garbage collection which occurs on loop endings, method calls, endings. In addition to the approaches relying on JVM defined update points, there are approaches whose points are manually defined by the programmer, such as [13, 37]. Furthermore, in some cases a safe point may never be reached (e.g. in methods with a long running loop). These specific cases can be prevented with appropriate modification at the design time, such as by using the functional decomposition of code inside a long running loop [6].

Another point to consider is the moment of update completion. Modified JVM often uses garbage collection mechanism to replace references to the modified objects, which implicitly performs updates atomically during the garbage collection [9, 15]. In contrast, in lazy update approaches, objects are updated at the first access of a modified object [12, 14]. Atomic and lazy approaches can both be found in the modified JVM and JVM agent, while DAOP, due to join-point activation, atomically performs aspects update [22, 27].

3.1.2 Program adaptation

Several types of program adaptation techniques can be found in the approaches to DSU. Modified JVM and JVM agents introduce a mechanism to transfer the object state from previous to new program version. There are also approaches with programmer-provided or automatically generated state transfer functions that the programmer can modify [9, 14], while other approaches provide mechanisms to copy field values [3, 15]. DAOP concepts, except Dynamic ITD (Inter-Type Declarations) [10], generally do not provide these mechanisms. Moreover, in addition to state adaptation, manual program adaptation may be required to comply with conventions introduced by a specific approach, such as programming language extensions presented in [13].

Current approaches perform program adjustments at compile, load or run-time. In compile and load time, program adjustments are made with bytecode modification; for example, by inserting bytecode snippets, called “hooks”, to enable runtime join-point activation in DAOP [20, 23] or to enable proxy or wrapper objects activation in JVM agents [12, 37]. In addition, Jooflux [22] at load time replaces the static instruction for method invoke with dynamic instruction, which can be altered at runtime. In general, DAOP only provides dynamic method body modification. To perform incremental dynamic updates and to extend changes supported by DAOP, extended DAOP approaches such as [24, 55] perform program adaptation in the preparation steps and use program code analysis to convert changes between versions into aspects. Meanwhile, besides possible generated transformation functions [9, 14], modified JVMs do not require program adjustments in the preparation steps. Moreover, compared with JVM agents, they utilize smaller or negligible program adjustments, which are made during the runtime. DCEVM [15] is a modified JVM approach that adds dummy fields reduced-size classes to improve garbage collector execution performance. Jvolve [9] inserts a return barrier instruction on the stack to perform a delayed update, while Javelus [14] generates code validity checks to enable a lazy update. Furthermore, DAOP approaches Prose [27], and Hotwave [21] when implemented as JVM agents use runtime bytecode inlining, whereas Steamloom [19] and Dynamic ITD [10] use a bytecode modification that is built as a modified VM.

Therefore, program adjustments are categorized by the program life cycle where DSU is enabled, as follows: D - design, P - programming, C - compile, L - load and R - run time (Figure 3.2).

3.1.3 Supported changes

Method body change is a common modification with the HotSwap functionality, which has become part of Java Hotspot VM since version 1.4 [56]. Modified JVM works on the lower level of environment hierarchy, inside the JVM, and can handle more complex changes [57] by extending the existing HotSwap functionality [15]. JVM agent approaches can support unanticipated changes, from adding and deleting class members to class hierarchy modification; as seen in [12]. DAOP based approaches are limited to class member changes because they operate at the method level, but there are some DAOP approaches that could provide hierarchy change using inter-type declarations; as seen in [10].

Therefore, the changes supported by various approaches can be categorized into three basic levels, where each level supports all changes supported by lower levels, starting with 1, as follows:

1. Method body change;
2. Addition and deletion of methods, fields, constructors, and classes;
3. Addition and deletion of interfaces including changes in class hierarchy (i.e. change of the class or interface supertype).

Table 3.1 shows the proposed program change categorization. Changes of types and names of fields or classes, as well as changes of signatures of methods or constructors, are absent from the proposed categorization. These type of changes are made by multiple changes performed in sequence, such as deletion followed by addition, as in [9, 15]. Furthermore, real world changes frequently involve making multiple basic changes at once. Therefore, they can be categorized as compound changes. DSU approaches support various changes, while Prose [27] and Jooflux [22] as dynamic aspect approaches (DAOP) support program changes indirectly as in [24]. However, class hierarchy changes are not supported. DCEVM [15], as a modified VM (mJVM), supports arbitrary changes; whereas Jvolve [9] does not support supertype change. The adding and deleting class is a dependent modification (e.g. if the added class is not used, then it does not affect the running program).

Table 3.1: Categorized supported changes and DSU approaches support

Category			DSU			
Type	Lev.	Modification	Prose (Cech †)	Jooflux (Cech †)	DCEVM	Jvolve
Basic	1	Method body (MB)	•	•	•	•
	2	Method (M*)	**	**	•	•
		Constructor (Co*)	**	**	•	•
		Field (F*)	**	NA	•	•
		Class (C*)	****	****	****	****
	3	Supertype (S*)	NA	NA	•	***
		Interface (I*)	NA	NA	•	•
Compound	2	M* + MB	•	•	•	•
		F* + MB	•	NA	•	•
		Co* + MB	•	•	•	•
		C* + MB	•	•	•	•

† based on Cech [24]
* add and remove modification
** indirectly supported
*** partially supported
**** dependent modification

3.2 Hierarchy changes

Changes in the class hierarchy between two versions of the program can either be the type or member changes. Meanwhile, hierarchy changes in the form of changes in class members do not affect the class type, but through the inheritance changes in members do affect the classes that inherit the class with changes.

3.2.1 Type changes

When the relationship between classes in two versions of a program changes, a type change occurs; for example, if in the new version (v_2) of the program another class has been added to the class as a predecessor, then the class becomes a subtype of the added class. The same is true for deleting, which is the opposite case because the class is no longer a subtype of the deleted class. Furthermore, a class that has been added or deleted as a predecessor may be an existing class that has changed position in the tree; that is, it is not a new class added in the new version (v_2) or an existing one deleted from the current version (v_1).

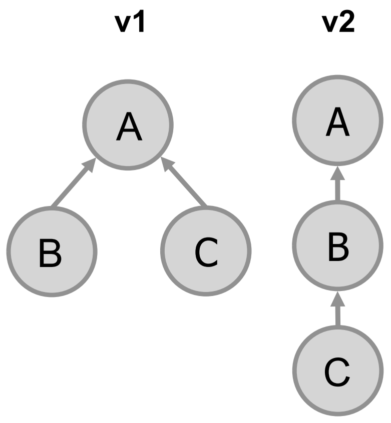


Figure 3.3: Hierarchy type change example

In the example in Figure 3.3, a change in inheritance is seen for class C, which in the updated version (v_2) inherits class B, and does not inherit in the current version (v_1) . In v_2, class C is a subtype of class B, which results in a change in statements that contains the type comparison that are shown on left-hand side in Example 3.1. In v_1 variable b is not an instance of the class C, however in v_2 statement evaluate as true. Casting the variable b to type of the class C is causing an exception in v_1, and executes without exception in v_2 .

Example 3.1: Java statements with type relationship when `m2()` and `f2` are defined in `B`

```
B b = new C();              C c = new C();
b instanceof C              c.m2();
(C)b                        c.i2 = 3;
```

3.2.2 Member changes

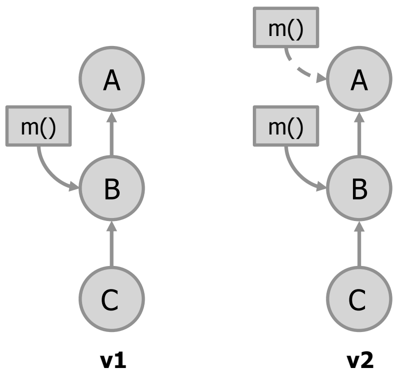
Classes in the hierarchy tree inherit class members. Therefore, any changes in the class member or when the member is added or deleted affect the classes that inherit the changed class. In Example 3.1, on the right-hand side are statements that contain access to added class members that class C inherits from class B. Changes of members, in the case of fields, is the field type change;

whereas in the case of methods is the method signature. Meanwhile, private fields and methods can be used within accessible methods or constructors,; therefore, class indirectly inherits changes in these members, which are manifested through a change in behavior. Constructors are changed similarly to the methods by signature; therefore, the class can use a modified constructor of the parent class. Meanwhile, changes in the body of inherited methods and parent class constructors affect the class that inherits them in the form of behavior changes. Because the fields contain the state of a class object, and methods and constructors define its behavior, any changes in class fields are state changes while changes in class methods and constructors are behavior changes. Moreover, if a class has a change in state or behavior between two program versions, then these changes are inherited by the child classes.

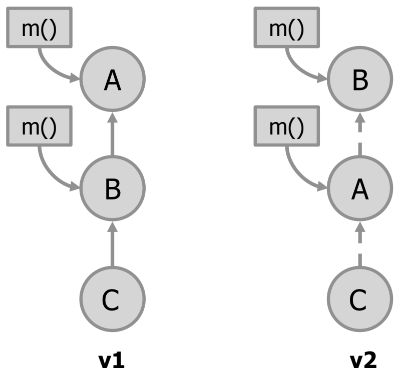
Overriding methods

Method overriding is related to the inheritance in the OO paradigm and runtime polymorphism. As already described, descendant classes inherit class members from predecessor classes. A key property of the OO paradigm is inheritance, where classes are an abstraction of the real-world and inheritance defines the relationship between classes that represent objects from the real world. For example, class animal is a parent class of classes cat and dog. Class animal defines behavior *sound* because each animal makes a sound. However, each animal sounds differently: a cat meows, purrs or hisses, whereas dogs bark, howl or growl. Therefore, subclasses of the class animal can have different implementations of the behavior. In the OO paradigm behavior is defined over methods, thus in the example method `sound` would be implemented in cat and dog classes, overriding the method defined in class animal. Both cats and dogs are animals. Therefore, if the animal object is initialized, for example as cat, then invoking method `sound` on the animal object would invoke the method implemented in class cat, corresponding to runtime polymorphism.

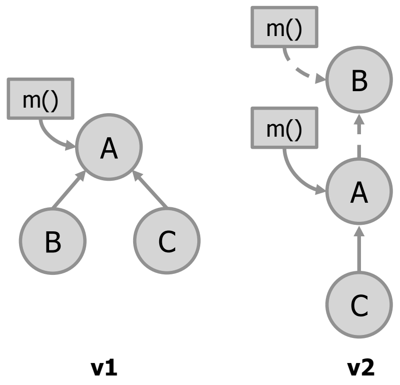
Changes in the class hierarchy or methods between program versions can result in changes in overriding methods (i.e. behavior relationship between classes). For example, in Figure 3.4c class hierarchy in v_1 is equal to the v_1 hierarchy shown in Figure 3.3; however, in v_2, class A changes parent to class B, instead of class C changing parent to class B. Regarding methods, class B in v_2 implements method `m()`, and class C inherits method implemented in class A, instead of method `B.m()` because the first predecessor class that implements method with the same signature is class A. Class A in v_2 overrides method `m()` added in class B. Therefore, a change of class B's position and implementation of method `m()` does not affect class C in the context of changed methods for class C. Similarly, in the opposite case, when the version v_2 is the current version, and version v_1 updated version, method `m()` is removed from class B, and class B is removed as predecessor of class C. Furthermore, similar to the Figure 3.4b, in Figure 3.4a A implements method `m()` in the v_2 class; however, class B overrides method in class A,



(a) Method `m()` added to class *A*



(b) Class *A* and *B* changed positions



(c) Class *B* with method `m()` added as parent to class *A*

Figure 3.4: Overriding method changes

resulting in no changes for methods of class *C*. The opposite is the case in Figure 3.4b, classes *A* and *B* change positions in v_2. Therefore, for class *C*, method `m()` in class *A* overrides method in class *B*.

Constructor changes

Constructors are used to initialize the state of objects. If there is no definition of the constructor, then a default constructor is implicitly added. Constructor body contains as the first statement an invocation to the parent constructor or another constructor defined within the same class. If the such statement is omitted by programmer, then an implicitly call to the parent constructor is inserted. For example, in Figure 3.5 when the default constructor on class *C* is invoked, a chain call of constructors is performed for each predecessor class until class *A*. Statements defined in constructor $A()$ are executed, then in $B()$ and finally statements in the invoked constructor $C()$. A chain invocation of the constructors initializes fields of predecessor classes. The initialization of predecessor class members first ensures that if the invoked constructor contains statements with inherited members, it will not cause runtime errors because of uninitialized fields.

Changes regarding the constructor can either be changes in the constructor body, or the constructor can be added or deleted similar to the methods. The return type for constructors corresponds to a class containing the constructor and cannot be changed. Changes in constructor parameters, type and number, results in deleted and added constructor. Regarding changes in the first statement, a invocation to the another constructor within the same class or to parent constructor can be replaced between two program versions. Furthermore, an added constructor or existing constructor can contain an invocation to the existing or added parent constructor, also as invocation to the existing or added constructor within the same class. These changes should be detected and resolved by dynamic software update model to support arbitrary hierarchy changes.

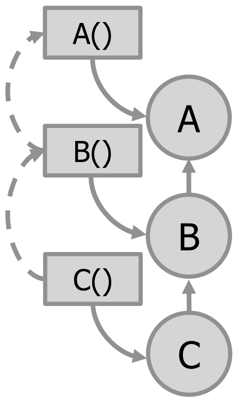
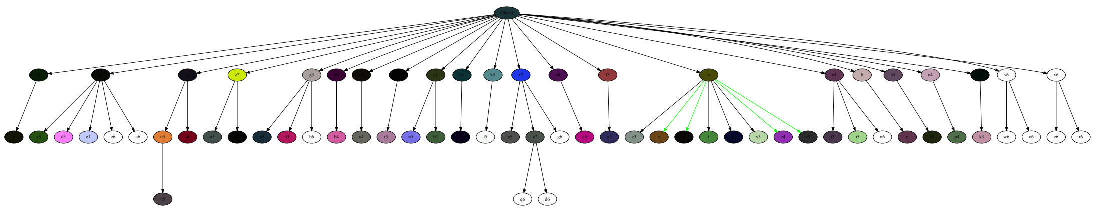


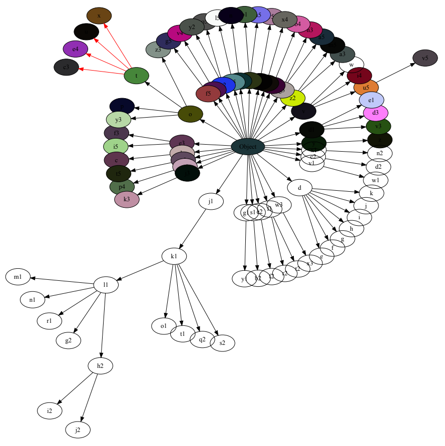
Figure 3.5: Constructor invocation in the class hierarchy

3.3 Class hierarchy visualization

To represent trees and graphs, as data structures, adjacency matrices and lists are usually used. It is intuitive to represent class trees by visual representation to demonstrate differences in the classes, and also the inheritance relationship between the classes as edges.There are various algorithms for drawing and visual description of trees and graphs [58, 59]. In this dissertation, dot language is used to describe the tree and dot visual representation is supported by the GraphViz tool [60]. Although there are other formats such as GML [61] and TGF [62], and various tools supporting these graph formats, GraphViz and dot are simple to use and are widely supported. The characteristics of dot representation are planar and symmetrical graphs, where nodes on the same level are horizontally aligned. This allows us to represent trees by levels, unlike the adjacency list and matrix. Some visual layouts of trees are better suited to reflect data from the problem domain than other layouts. To compare changes in the class hierarchy between two versions of object-oriented programs, represented by trees, it is convenient to use radial rendering of nodes per level (i.e. *twopi* layout). For example, Figure 3.6a shows the class hierarchy of program NewPipe [63] version 0.9.0, which is one of the open source programs analyzed in this dissertation. Nodes on the same level are visible in Figure 3.6a, but Figures 3.6b and 3.6c with radial layout are more suitable to compare two program versions than the default dot layout in

Figure 3.6a. In Figures 3.6b and 3.6c, the center is the root (i.e. `Object` class), which other classes implicitly inherit. Nodes on the same level are visible as “rings” around the root node. Nodes filled with a color exist in both trees, where they can be identified by the filled color. Nodes that are not filled with a color exist in only one of trees. From the transformation point of view, where the left tree is the source tree transformed to the right as the target tree, it can be concluded that the no-color nodes in the source tree are deleted and no-color node in the target tree are nodes that have been added.


(a) Dot layout: NewPipe v0.9.0


(b) Twopi layout: NewPipe v0.8.9

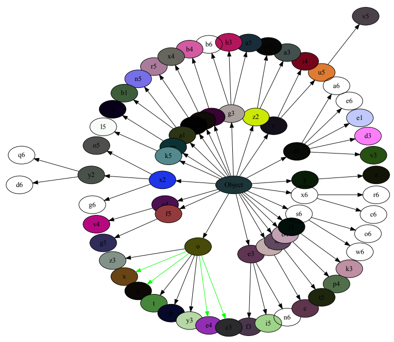

(c) Twopi layout: NewPipe v0.9.0

Figure 3.6: Class hierarchy comparison between 0.8.9 and 0.9.0 NewPipe program version

For nodes matched in both trees, red edges connected to the parent node denote deleted edges in the source tree and green edges denote added edges in the target tree. In Figure 3.6, because class t is deleted, classes x, u, $e4$, and $c3$ have changed parent from class t to class o.

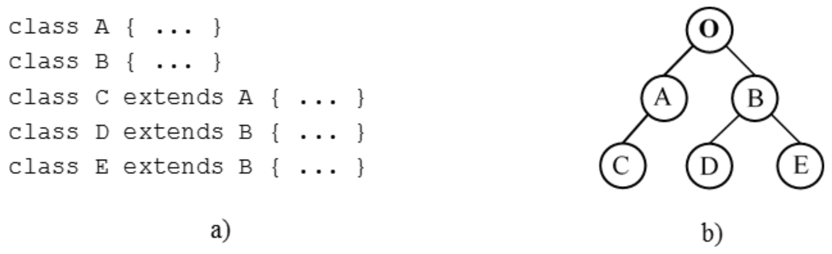
Chapter 4

Tree dissimilarity

The class hierarchy of a program version in OO paradigm can be represented as tree data structure. Class hierarchy (i.e. inheritance relationship) conforms to the relationship between nodes in the tree, where the root node is the root class (e.g. `Object` in Java). In the previous chapter, a tree is used to describe inheritance mechanisms and visualize class structure. Therefore, the premise is that tree comparisons can inherently be used to detect differences between two program versions in OO. However, currently available tree comparison algorithms focus on finding minimal structural differences between trees, which is (for example) used for computer vision applications. Therefore, in this chapter, as part of the work in [64], the currently available approach is analyzed and new algorithms are introduced. The work from [64] with emphasis on the inheritance relationship between nodes is included for completeness because the presented algorithms are used as a basis to detect program changes and runtime phenomena between two program versions in the latter chapters. In addition to detecting changes in the inheritance relationship between classes, it is possible to detect class member changes and assess the risk of the runtime phenomena.

4.1 Class hierarchy as tree data structure

The class hierarchy can suitably be represented as a tree if multiple inheritance is not supported. Multiple inheritance enable inheritance from multiple classes and is supported by some programming languages (e.g. C++). However, it introduces the inheritance diamond problem [65, 66]. Most of the current object-oriented languages support only single inheritance, where a class can only inherit a single class. Consequently, this dissertation will only consider the case of a single inheritance. Meanwhile, Java interfaces define a set of methods that class implements. Interfaces are organized hierarchically and allow multiple inheritance, while classes can implement more than one interface. Interface hierarchy is not considered because if the class in the modified version implements the methods specified in the interface, then the change reflects



```
class A { ... }
class B { ... }
class C extends A { ... }
class D extends B { ... }
class E extends B { ... }
```

Figure 4.1: a) Class declaration b) Class hierarchy represented by a tree

through the implemented methods.

4.2 Introduction to trees and dissimilarity measures

Because trees represent hierarchy organized data, they are widely used in areas such as computer vision [67], structured documents [68], natural language processing [69], phylogenetic studies [70], and molecular biology [71, 72]. The main goal in representing data patterns as trees or generally as graphs is to identify the changes in the data. In this dissertation, the motivation is to compare class hierarchy between program versions in object-oriented programming languages. In the related literature [70, 73, 74], trees such as Abstract Syntax Trees (ASTs) are used to determine the difference between program versions on the syntax level, which can be used for program analysis on the intraprocedural level. However, ASTs can be used with other tree representations to compare classes, such as in [75]. In this dissertation, the relationship between classes is considered in the hierarchy of object-oriented languages. The dissimilarity between hierarchies can be used for program analysis in various software engineering tasks, such as detection of code clones [76], regression testing [77], and dynamic software updating [74]. To compare the class hierarchy between two program versions, an unordered labeled tree is the most suitable hierarchy structure. The unordered tree corresponds to the class hierarchy in object-oriented languages, where the root node corresponds to the elementary class (e.g., Object class in the Java programming language), as illustrated in Figure 4.1.

To compare trees, or generally graphs, it is necessary to use a dissimilarity measure. In the related literature [78, 79], dissimilarity measures usually fall into two elementary categories: isomorphism [80] and edit distance [81, 82, 83]. Isomorphism represents an exact matching between two trees or subtrees. In contrast, edit distance provides inexact (i.e. error-tolerant) matching [79]. Isomorphism attempts to find a bijective function or mapping from one tree to another tree, or subtree, answering whether or not the tree is equal to another tree or subtree. Meanwhile, Tree Edit Distance (TED) [84] can compare entirely different trees, measuring the dissimilarity between them by the amount of modification of nodes and edges required to transform one tree into another.

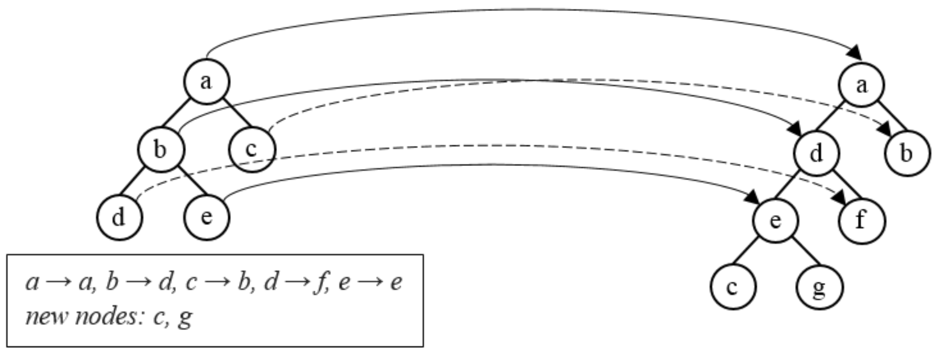

a
b
c
d
e
a
d
b
e
f
c
g
$a \rightarrow a$, $b \rightarrow d$, $c \rightarrow b$, $d \rightarrow f$, $e \rightarrow e$
new nodes: c, g

Figure 4.2: Tree edit distance operations (listed in the box) and nodes mapping (by arrows)

Tree edit distance is more suitable to detect differences in the trees representing the class hierarchy because various program versions can have an entirely different class hierarchy. However, existing tree edit distance measures are not appropriate to detect hierarchy changes in trees such as inheritance changes because their goal is to find similar parts of trees. In this dissertation, unordered tree transformation is discussed from the edge perspective to observe changes in relationships between nodes. The contribution of this dissertation is that the dissimilarities between unordered trees are considered based on changes in the edge between node and parent, introducing Edge Edit Distance (EED), and as a major contribution the dissimilarities are based on changes in inheritance relationships between nodes, introducing Tree Inheritance Distance (TID). For both distance measures, efficient algorithms are presented and evaluated by experiments.

4.3 Preliminaries

Tree edit distance evolved from the string comparison [84, 85, 86]. In general, it is based on finding the lowest transformation cost from one tree to another. Tree transformation is a sequence of elementary edit operations on nodes and edges such as add, delete and substitute. Cost is assigned to each edit operation. Consequently, the total cost of transformation from one tree to another is the sum of the costs of all edit operations in the sequence. Because there may be several ways to transform a tree into another, resulting in different total costs, edit distance is defined as the lowest transformation cost [67, 78, 79, 82, 83, 84, 86, 87]. Tree edit distance algorithms correspond to the process of mapping (i.e. fitting similar parts of two trees) according to the assigned cost. An example of mapping of the unordered trees is shown in Figure 4.2.

Edges define the relationship between nodes. Therefore, to detect changes in a tree, both edge edit operations and node edit operations could be used. Although edge operations in graph edit distance [79] are represented as the result of node operations, the presented idea to detect changes can be introduced more intuitively by using edge operations and will be used later to explain the indirect effect of edge edit operations on trees.

4.3.1 Edge edit operations

Three elementary edge operations are considered: *add*, *delete*, and *substitute* edge. Following the notation in related literature for edit distance [78, 79], let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be trees, where V_1 and V_2 are sets of nodes, and E_1 and E_2 are sets of edges. Let e_1 be an edge in E_1, e_2 in E_2, and λ represent empty edge not contained in E_1 and E_2. *Add* edge operation is denoted by $\lambda \to e_2$, *delete* by $e_1 \to \lambda$, and *substitution* by $e_1 \to e_2$.

Let u, $v \in V$, edge $e \in E$ is then defined by a pair of nodes u and v such that $e = (u, v)$. In the next three cases, edge operations and their relation to node operations can be recognised:

1. Edge is substituted: $e_1 \to e_2$
   $e_1 = (u_1, v_1) \in E_1$, $e_2 = (u_2, v_2) \in E_2$ implies $u_1 \to u_2$ and $v_1 \to v_2$
   meaning that node u_1 is substituted by u_2 and node v_1 is substituted by v_2
2. Edge is added: $\lambda \to e_2$
   $e_2 = (u_2, v_2) \in E_2$ implies $\nexists (u_1, v_1) \in E_1$ where $u_1 \to u_2$ and $v_1 \to v_2$
   meaning that nodes u_2 and/or v_2 are added
3. Edge is deleted: $e_1 \to \lambda$
   $e_1 = (u_1, v_1) \in E_1$ implies $\nexists (u_2, v_2) \in E_2$ where $u_1 \to u_2$ and $v_1 \to v_2$
   meaning that nodes u_1 and/or v_1 are deleted

These three cases are illustrated by Figure 4.3, in which the original tree could be transformed to one of three other trees depending if one edge is substituted (a), added (b), or deleted (c). Th edge involved in the operation is marked with the bold connecting line. It should be noted that because the operation in Figure 4.3 a) is performed on non-leaf nodes, and both nodes are substituted, it implies additional substitute operations $(a, b) \to (c, b)$, $(c, e) \to (a, e)$, and $(c, d) \to (a, d)$.

4.3.2 Relationship between nodes

The main motivation is to detect modification in the relationship between nodes from the aspect of inheritance change. As mentioned earlier, edge operations are the result of nodes operations. On Figure 4.2, node b is substituted by node d, and node c by node b, and consequently edge (a, b) is substituted by (a, d), and edge (a, c) by (a, b). However, the edge between nodes a and b exists in both trees. The relationship between nodes a and b is preserved, making node b and edge (a, b) substitution unnecessary. Because TED performs operations on preserved edges between the trees, it is not suitable for observing direct relationship modification and, therefore, inheritance changes. To detect edge modifications and, based on this, changes in inheritance, the edge operations and the new dissimilarity measure are presented in Section 4.4.

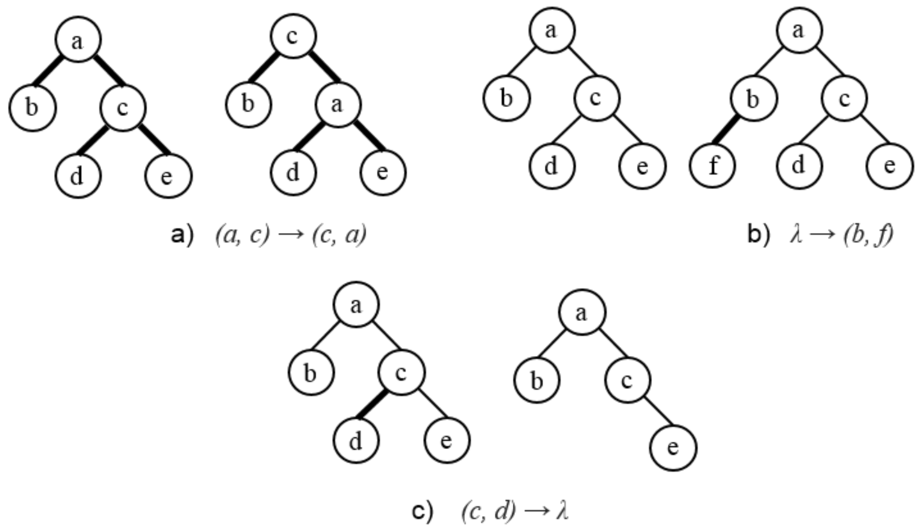


Figure 4.3: Edge edit operations: a) *substitute* b) *add* c) *delete*

4.4 Edge edit distance

Instead of finding similar parts of a tree, in the case of the inheritance tree [69, 88] the relevant information is the change in the relationship between nodes. Because every child node in a tree has only one parent, a child node and corresponding edge to the parent can be observed as a single operation unit. Therefore, the tree is defined with additional empty node and operations which reflects modification on relationship between nodes.

4.4.1 Edge extended tree

By considering the node and its edge to its parent as a single unit, edge edit operations could be divided into three cases. The *add* operation consists of adding a new node that by a new edge connects to an existing or in a previous operation added parent node. The *delete* operation removes the node and its edge that connects it to its parent. Instead of a *substitute* operation, the term *move* would be used and the operation is the change of only one incident node (i.e. the parent node).

In this way, if the parent function is defined, then every edge in a tree could be denoted as pair $(parent(u), u)$, where $parent(u)$ and u are nodes in the tree. However, because the root node does not contain any edge to the parent, to define the parent function for the complete node set a dummy or empty node $\varepsilon \notin V$ is introduced in such way that edge $e = (\varepsilon, r)$ exists, where r is the root node.

Definition 1 *The Edge Extended Tree (EET) is an unordered tree $X = (V \cup \{\varepsilon\}, E)$ where $\varepsilon \notin V$ is an empty node connected only to the root node r with an edge $e = (\varepsilon, r) \in E$.*

Remark *Let n be a number of nodes in V, then the tree X contains n edges. This claim is a direct consequence of the fact that unordered labeled tree $T = (V,E)$ with n nodes contains $n - 1$ edges, and adding an empty node ε and corresponding edge (ε, r), leads to the tree $X = (V \cup \{\varepsilon\}, E)$ containing n edges.*

Formally, the parent function for the EET $X = (V \cup \{\varepsilon\}, E)$ could be described as:

$$p : V \to V \cup \{\varepsilon\} \text{ such that } p(v) = u \text{ iff } \exists u \in V \cup \{\varepsilon\} \mid (u,v) \in E$$

By using the parent function, edge edit operations on the EET can be defined as follows:

Definition 2 *Let $X_1 = (V_1 \cup \{\varepsilon\}, E_1)$ and $X_2 = (V_2 \cup \{\varepsilon\}, E_2)$ are EET's with parent functions p_1 in X_1 and p_2 in X_2, edit edge operations are:*

1. *Edge is moved: $e_1 \to e_2$*
   Only one of the nodes incident to edges $e_1 = (p_1(v), v)$ and $e_2 = (p_2(v), v)$ is changed (i.e. the parent node).
   This operation can be reduced to the following situation: $v \in V_1 \cap V_2$ and $p_1(v) \neq p_2(v)$
2. *Edge is added: $\lambda \to e_2$*
   As e_2 could be defined as $(p_2(v), v) \in E_2$, this implies $\nexists (p_1(v), v) \in E_1$
   Therefore this operation is reduced to $v \notin V_1 \wedge v \in V_2$; that is, $v \in V_2 \setminus V_1$
3. *Edge is deleted: $e_1 \to \lambda$*
   Similar to the previous operation, it is reduced to $v \notin V_2 \wedge v \in V_1$; that is, $v \in V_1 \setminus V_2$

Edge operations are shown in Figure 4.4, where edges involved in operations are given in bold. Furthermore, child nodes included in the edge operation are marked with the dashed surrounding ellipse, together with the edge to the parent node - $(parent(u), u)$. Edge *move* operation from the edge (a,c) to edge (b,c) is shown in Figure 4.4 a). *Move* operation performs the move of the entire subtree rooted at the node c, from the position where the previous parent node is a to the position where node b is the new parent. *Add* edge (b, f) operation is shown in Figure 4.4 b). Adding edge is the result of adding node f to the tree, such that the parent of the new node is node a. *Delete* edge (c,d) operation is shown in Figure 4.4 c). *Delete* edge is the result of the node delete operation from the tree. *Add* and *delete* edge operations are shown only on leaf nodes because non-leaf *add* and *delete* edge operations are followed by at least one more edge edit operation.

When a non-leaf node is added, at least one child of a node, which becomes a parent to a new node, becomes a child of a newly added node. In Figure 4.5 a), node f is added as a non-leaf child node to parent node a, with edge operation $\lambda \to (a, f)$. Node b as the previous child of the node a is consequently moved to be the child of the added node f, with edge operation $(a,b) \to (f,b)$. However, edge (a,c) to another child c of node a is preserved; therefore, only one child node is moved. Meanwhile, the deletion of the non-leaf node requires that all children

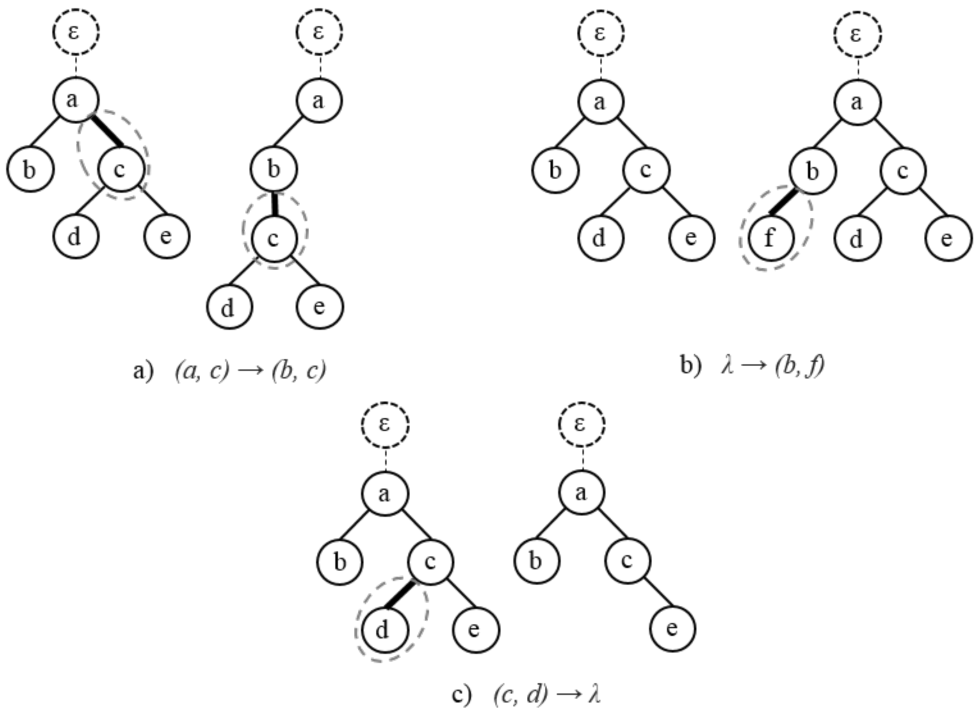


Figure 4.4: Edge edit operations on EET

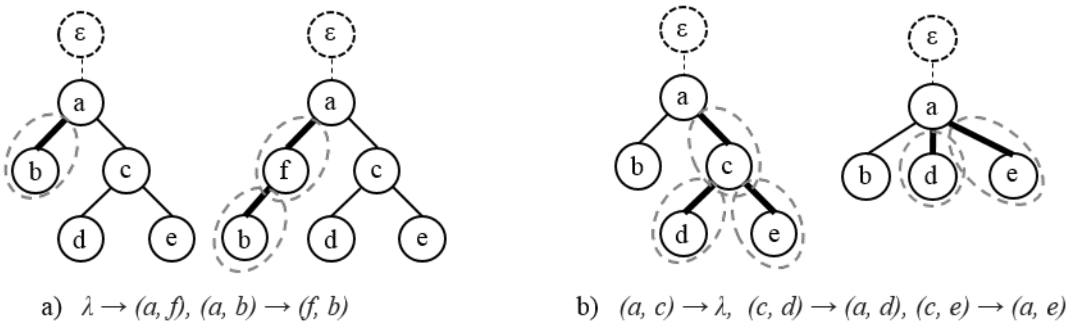

ε
a
b
c
d
e
f
a) λ → (a, f), (a, b) → (f, b)
b) (a, c) → λ, (c, d) → (a, d), (c, e) → (a, e)

Figure 4.5: Non-leaf node *add* (a) and *delete* (b) edge operations

of the deleted node change their parent to the parent of the deleted node. Figure 4.5 b) shows the deletion of non-leaf node c, with edge operation $(a,c) \rightarrow \lambda$, followed by changing the parent of node d to node a, with edge operation $(c,d) \rightarrow (a,d)$. Identical operation is performed on the node e, with edge operation $(c,e) \rightarrow (a,e)$. There is a specific case when the root node is deleted. In this case, by an arbitrary procedure, one of the child nodes of the previous root node is promoted to be the new root node, and other siblings of this node are moved to be the children of the new root. It is mandatory because only one edge can exist from the empty node ε to the root node. Otherwise, there are a smaller number of edges than nodes, resulting in the fact that a tree is not EET. Furthermore, the root resolving procedure is arbitrary depending on the application (e.g. it could be based on specific node properties).

4.4.2 Set of edit operations

Generally, the transformation of tree T_1 to another tree T_2 can be observed by a sequence $s_1, s_2, \ldots, s_n$ of edit operations on nodes or edges [67, 78, 82, 83, 86]. For example, sequence of edge edit operations to transform a source tree in Figure 4.2 on the left-hand side to the target tree on the right side is: $(a,b) \rightarrow (a,d)$, $(a,c) \rightarrow (a,b)$, $(b,d) \rightarrow (d,f)$, $(b,e) \rightarrow (d,e)$, $\lambda \rightarrow (e,c)$, $\lambda \rightarrow (e,g)$. Similar is for EETs X_1 and X_2, where edge edit operations defined by Definition 2 are *add*, *delete* and *move*. Sequence of EET edge edit operations to transform tree in Figure 4.2 is: $(b,d) \rightarrow (a,d)$, $(b,e) \rightarrow (d,e)$, $\lambda \rightarrow (d,f)$, $(a,c) \rightarrow (e,c)$, $\lambda \rightarrow (e,g)$. There is a smaller number of EETs than TED operations because EET consider edge modifications based on unchanged node labels, while TED maps similar tree parts by nodes label transformation, i.e. nodes substitution

The sequence of edit operations influences the intermediate results. If edge edit operations are applied so that *add* edge operation is followed by a *move* and then by the *delete* operation, while consecutive *add* and *delete* operations are executed from leaf nodes upwards, then each step in the sequence produces a tree. Otherwise, the intermediate result depending on the involved edge operations can be tree forest. TED fulfils these conditions in [73, 78, 86] because edit operations are performed only on leaf nodes. Meanwhile, such conditions do not produce valid EET in each possible step, such as adding the new root node before moving or deleting the old root node results in two edges from the empty node. Similar to the [78] the sequence of edge edit operations that transforms a tree can be written as an ordered relation $R \subseteq (E_1 \cup \{\lambda\}) \times (E_2 \cup \{\lambda\})$, where an edge edit operation is represented as a pair of edges (e_1, e_2), such that $e_1 \in E_1 \cup \{\lambda\}$, $e_2 \in E_2 \cup \{\lambda\}$. However, only the final result is considered (i.e. operations between two trees: source and target tree), without intermediate results. Therefore, instead of edit sequence, where operations order is essential, a set of edge edit operations S_e are used.

Considering that there may be several possible edit sets that perform the same transformation, let $W(X_1, X_2)$ denote the set $\{S_{\mathrm{e}1}, \ldots, S_{\mathrm{en}}\}$ of all sets of edit operations to transform X_1 to X_2, possibly containing superfluous operations, such as additional subsequent *delete* and *add* edge operations or *move* edge operation. Furthermore, *move* edge operation $e_1 \rightarrow e_2$ can be described as a *delete* edge operation $e_1 \rightarrow \lambda$ followed by an *add* edge operation $\lambda \rightarrow e_2$. These additional operations for TED [78] and Graph Edit Distance (GED) [79] generally increase the additional cost to the tree transformation. However, a question arises as to how the relationships (i.e. edges between nodes in the source and target tree) are changed. Accordingly, the cost for the minimum number of required edge edit operations to transform a tree is calculated, which implicitly excludes such operations.

4.4.3 Edit set cost

To determine the cost for the set of edit operations S_e, a cost function $\gamma : E_1 \cup E_2 \cup \{\lambda\} \times E_1 \cup E_2 \cup \{\lambda\} \rightarrow \mathbb{R}$ is assigned for each edge edit operation. The total cost to transform from tree X_1 to X_2 by applying edge edit operations from the set of edit operations S_e is equal to:

$$c(S_e) = \sum_{s_i \in S_e} \gamma(s_i)$$

Edge dissimilarity measure between two trees is EED, formally defined as:

Definition 3 *EED between trees X_1 and X_2 is the total cost of the set of edit operations S_e that contains the minimum number of edge edit operations to transform X_1 to X_2:*

$$d_e(X_1,X_2) = c(S_e) \,|\, S_e \in W(X_1,X_2) \textit{ and}$$
$$|S_e| \leq |S_e'| \; \forall S_e' \in W(X_1,X_2)$$

If the cost for all edge edit operations is 1, then the above formulation is equal to finding the edit set with the minimum cost; although generally the cost of the minimum number of operations could be greater than the minimum cost. Note that even with the same cost of operations, this formulation is not equal to the definition of TED [78] or GED [79] because of the different formulation of edit operations, primarily because the *move* edge operation is different from substitution. Furthermore, if the cost of a *move* operation is greater than the sum of the cost of *add* and *delete* operations, then the set of the minimum number of edit operations is not changed compared to equal costs.

Because for the *move* operation a node changes the parent, the cost is equal to the cost of the parent change. For *add* and *delete* operations, the cost is equal to the cost of node adding to a tree or deleting from a tree. Furthermore, due to Definition 2 of edit operations, because only the source and the target tree are considered, finding minimal transformation edit set could be determined by nodes and the cost function could be defined as $\gamma : V_1 \cup V_2 \rightarrow \mathbb{R}$.

$$\gamma(v) = \begin{cases} 0, & v \in V_1 \cap V_2 \wedge p_1(v) = p_2(v) \\ m, & v \in V_1 \cap V_2 \wedge p_1(v) \neq p_2(v) \\ a, & v \in V_2 \setminus V_1 \\ d & v \in V_1 \setminus V_2 \end{cases}$$

Here m, a and d are the cost of the *move*, *add* and *delete* edge operations. The costs of these operations could be a constant number (e.g. equal to 1) but it can be generalised to situations in which m, a, and d could be cost functions ($m(v)$, $a(v)$, $d(v)$) of a node v (e.g. based on the number of node properties or the distance from the root node).

By using cost function defined over a node, the EED between trees X_1 and X_2 is equal to:

$$d_e(X_1,X_2) = \sum_{v \in V_1 \cup V_2} \gamma(v)$$

Proof *Let the set of edge operations S_e consists of the following sets of operations: S_a, S_d, and S_m, where $(\lambda,e_2) \in S_a$, $(e_1,\lambda) \in S_d$, and $(e_1,e_2) \in S_m$. Without loss of generality, let the costs for add, delete, and move edge operations are equal to the constants a, d, and m. According to Definition 2 edge operations are reduced to nodes, and according to the Definition 3, the following applies:*

$$\begin{aligned} d_e(X_1,X_2) = c(S_e) &= \sum_{s_i \in S_e} \gamma(s_i) \\ &= \sum_{s \in S_a} a + \sum_{s \in S_d} d + \sum_{s \in S_m} m \\ &= \sum_{v \in V_1 \cap V_2 \wedge p_1(v) \neq p_2(v)} m + \sum_{v \in V_2 \setminus V_1} a + \sum_{v \in V_1 \setminus V_2} d \\ &= \sum_{v \in V_1 \cup V_2} \gamma(v) \end{aligned}$$

The number of operations is minimal because the number of edge operations is limited by the number of nodes and the following applies $|V_1 \cup V_2| = |V_1 \setminus V_2| + |V_2 \setminus V_1| + |V_1 \cap V_2|$.

Furthermore, EED by using constants for m, a, and d can be formulated as:

$$d_e(X_1,X_2) = a\,|V_2 \setminus V_1| + d\,|V_1 \setminus V_2| + \sum_{v \in V_1 \cap V_2 \wedge p_1(v) \neq p_2(v)} m$$

Considering Definition 2 and Definition 3 Algorithm 1 is proposed to calculate EED. Note that constants for cost of the edge edit operations could be replaced with functions $a(v)$, $d(v)$, and $m(v)$ or $\gamma(v)$.

To determine time complexity of the algorithm, let the n_1 be the number of edges in X_1, and n_2 be the number of edges in X_2. Algorithm time complexity is $O(n_1 + n_2) * O(\texttt{contains})$ because the algorithm iterates through a set of nodes V_1 and V_2, and the size of node sets corresponds to the size of E_1 and E_2, equal to n_1 and n_2. The cost of `contains` function could be $O(1)$ if an appropriate hashing is used. Meanwhile, function `contains` can be implemented by a simple searching loop with complexity $O(n)$, where n corresponds to n_1 in X_1 and n_2 in X_2. In this case, the time complexity is $O(n_1 * n_2)$. It should also be noted that storing processed nodes set from the set $V_1 \cap V_2$ could be used to reduce time complexity. In the first loop processed nodes from V_1 could be saved to skip comparison of the already processed nodes in the second

Algorithm 1: Edge edit distance algorithm

```
input : trees $X_1(V_1 \cup \{\varepsilon\}, E_1)$ and $X_2(V_2 \cup \{\varepsilon\}, E_2)$, parent functions
        $p_1 : V_1 \to V_1 \cup \{\varepsilon\}$, and $p_2 : V_2 \to V_2 \cup \{\varepsilon\}$, and cost of the edit operations $a$, $d$,
        and $m$
output: an edit set $S_e$ with the minimum number of operations and an EED $d_i$

$S_e \leftarrow \varnothing$, $d_i \leftarrow 0$;
foreach node $v$ in $V_1$ do
    if contains($V_2$, $v$) then
        if $p_1(v) \neq p_2(v)$ then
            $S_e \leftarrow S_e \cup \{(p_1(v), v) \to (p_2(v), v)\}$;
            $d_e \leftarrow d_e + m$;
        end
    else
        $S_e \leftarrow S_e \cup \{(p_1(v), v) \to \lambda\}$;
        $d_e \leftarrow d_e + d$;
    end
end
foreach node $v$ in $V_2$ do
    if not contains($V_1$, $v$) then
        $S_e \leftarrow S_e \cup \{\lambda \to (p_2(v), v)\}$;
        $d_e \leftarrow d_e + a$;
    end
end
```

loop, but space complexity would increase and `contains` functions should be additionally used on processed set for each node from V_2.

Regarding the number of operations, the upper bound for move edge operations is $|V_1 \cap V_2|$. In that case, all edges from X_1 are moved in X_2, therefore all nodes from both trees are matched, and nodes from X_1 changed their parents in X_2. In the case when all edges from E_1 are deleted, and all edges from E_2 are added because there is an equal number of edges and nodes, the maximum number of edge operations; that is, the upper bound for edge edit operations n_{max} can be stated as:

$$n_{max} = |E_1| + |E_2| = |V_1| + |V_2| > |V_1 \cap V_2|$$

4.5 Tree inheritance distance

Edge edit operations are suitable to describe changes in the direct relationship between nodes. However, a tree is observed as an inheritance tree [88], where a node is in inheritance relationship to predecessor nodes. Therefore, the node is affected by any changes in its predecessor nodes. EED defined by appropriate cost function can describe changes in predecessor nodes only for nodes directly involved in edge edit operations, such as added, deleted and moved nodes. Meanwhile, indirect effects of edge edit operations on descendant nodes cannot be described by EED. Those changes are the consequence of edge edit operations. The following subsections describe the direct and indirect effects of editing operations on the inheritance relationship between nodes, and they then define the indirect edit operation. Afterwards, the inheritance operations, the cost function for inheritance operations and tree inheritance distance are introduced.

4.5.1 Tree editing impact on the inheritance

A suitable example to explain the effects of edge edit operations on the inheritance relationship between nodes should include *add*, *delete* and *move* edge edit operations. In Figure 4.6, for example, *delete* (Figure 4.6 a) and *add* (Figure 4.6 b) edge edit operations, performed on non-leaf nodes, are tagged with number 1 and induce *move* edge operation, tagged with number 2. From the inheritance aspect for the added node, all nodes on the path to the empty node are added to the inheritance relationship; that is, the newly added node can inherit ancestors' nodes properties. On the other side, when a node is deleted, inheritance relationship to nodes on the path to the empty node is deleted; that is, deleted node does not any longer inherit prospective ancestors' properties. Furthermore, the inheritance relationship to ancestor nodes is also changed for the moved node, involving at least the parent node changed by an edge edit operation. In Figure 4.6 a), node c is deleted, and as a consequence the edge (a,c) is also deleted, inducing move edge operation $(c,e) \rightarrow (a,e)$. Path (ε,a,c) is removed from the tree, meaning that node c loses inheritance relationship to the node a. Moreover, path (ε,a,c,e) is deleted, and path (ε,a,e) is added to the tree, meaning that node e loses inheritance relationship to the node c. Figure 4.6 b) shows a similar case where the edge (c,i) and the node i are added, inducing move edge operation $(c,e) \rightarrow (i,e)$. Node i obtains an inheritance relationship to the nodes a and c, and node e obtains an inheritance relationship to the added node i. For example, obtaining inheritance relationship in this dissertation means that class as node obtained predecessor classes members. Meanwhile, losing inheritance relationship means that class losses class members implemented in the predecessor classes that were in the inheritance relationship.

However, if the inheritance relationship is changed for nodes directly involved in edit operations, then it is indirectly changed for their descendants because the paths from descendants to the empty node are changed. In Figure 4.6, descendant nodes of a moved node are affected by move operations. These indirect effects of the edge edit operations are tagged with number 3. As already observed in Figure 4.6 a), node c is removed from the path to the empty node; that is, removed from the inheritance relationship for node e. Moreover, node c is also removed from the inheritance relationship for node e descendants f, g and h. Similarly in Figure 4.6 b), where node i is added to the inheritance relationship for moved node e but also for its descendants f, g, and h.

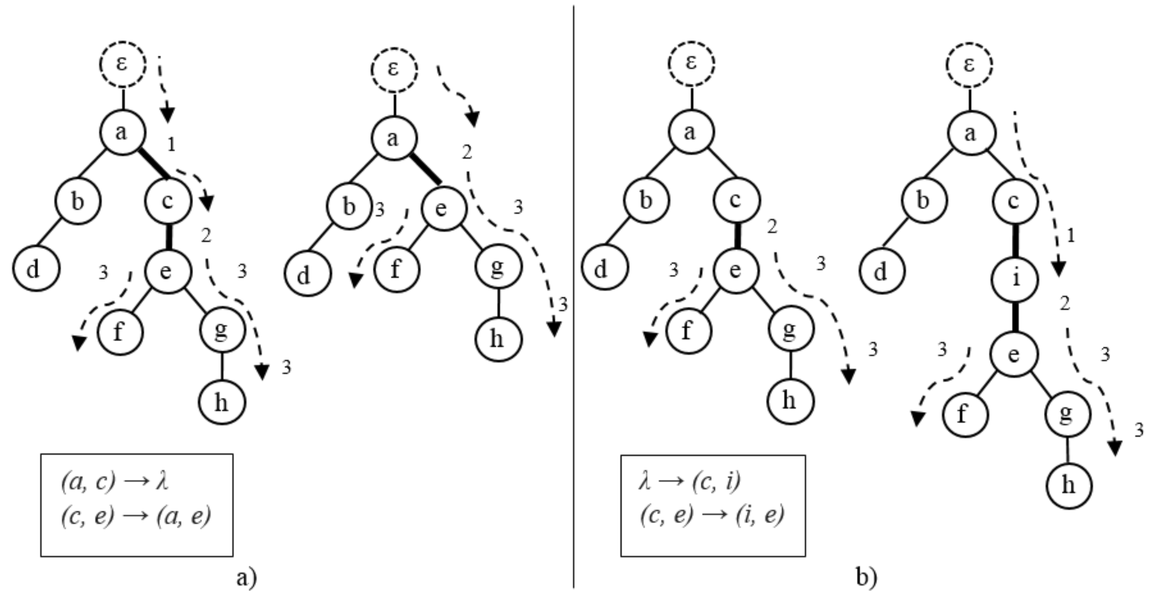


Figure 4.6: Impact of the move edge edit operation on the inheritance relationship induced by: a) deleted edge and b) added edge

Similar would happen if the move edge operation is not induced by other edge edit operations. If the tree in Figure 4.6 a) would be modified only by the move edge operation $(c,e) \to (d,e)$, then node e would lose inheritance to node c and would obtain inheritance to nodes b and d. Furthermore, the same change in the inheritance relationship would be applied to the descendants of node e.

4.5.2 Detecting inheritance changes

The previous section describes the change of the inheritance relationship as the consequence of edge edit operations. To determine changes in the inheritance relationship, it is necessary to compare nodes on the paths to the empty node from node involved in edge edit operations and its descendants.

Definition 4 *Let $path(u,v)$ be the path from the node u to the node v in the tree $X(V \cup \{\varepsilon\}, E)$, where $u \in V \cup \{\varepsilon\}$ and $v \in V$. Let $P(u,v)$, shortly $P_u(v)$ represents the set of nodes on the $path(u,v)$ without node v. $P_\varepsilon(v)$ then contains at least empty node ε for all nodes in V. Furthermore, if $v \notin V$ then $P_\varepsilon(v)$ is an empty set ($\varnothing$). Formally:*

$$P_\varepsilon(v) = \begin{cases} \{u \in V \cup \{\varepsilon\} \mid u \in path(\varepsilon, v) \wedge u \neq v\}, \ v \in V \\ \varnothing, \ v \notin V \end{cases}$$

Remark *Expressing the $path(u,v)$ expressed as path from the node u to the node v is synonym ti path from the node v to the node u.*

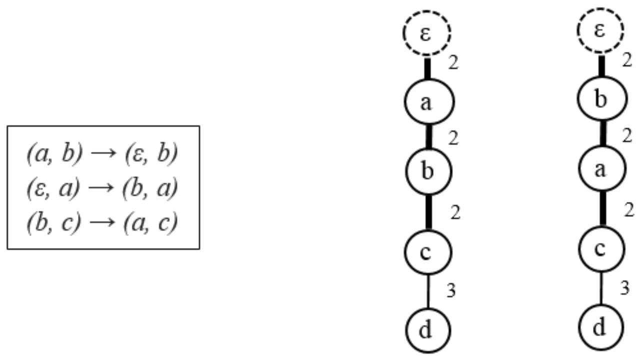


Figure 4.7: Move edge operations with equal $P_\varepsilon(c)$ and $P_\varepsilon(d)$ node sets

Node set $P_\varepsilon(v)$ contains a set of predecessor nodes for node v; therefore, condition $u \neq v$ is required because node v cannot precede itself. Note that for the node set to the empty node $P_\varepsilon(v)$, the term predecessor node set will be used. If node sets $P_{\varepsilon 1}(v)$ and $P_{\varepsilon 2}(v)$ in EETs X_1 and X_2 are different for node v from $V_1 \cup V_2$, then there is a change in inheritance relationship for node v. However, if the node sets $P_{\varepsilon 1}(v)$ and $P_{\varepsilon 2}(v)$ are equal, then it does not imply that the inheritance relationship is not changed for node v. In Figure 4.6, move edge operation implies a change of the parent node, and consequently different sets $P_{\varepsilon 1}$ and $P_{\varepsilon 2}$. Meanwhile, in Figure 4.7, as the result of move edge edit operations, nodes a and b switched positions, which for node c and consequently node d caused different paths in two trees, although the node sets $P_{\varepsilon 1}(v)$ and $P_{\varepsilon 2}(v)$ are equal. There is a change in the inheritance relationship for node c and d in a form of change in the position between nodes on the path to the empty node. Change in position between nodes on the path to the empty node can result in a change of inherited properties (e.g. a node can lose inherited properties).

The change of position or order of predecessor nodes can be observed as the change in the distance between the nodes; that is, the number of edges on the path between nodes. In Figure 4.7, the distance between nodes a and d in the source tree is 3, and in the target tree is 2. Furthermore, the distance between nodes b and d is 2 and 3, in the source and the target tree, respectively. Similar can be observed for node c, where predecessor node sets remain equal but the distance to nodes a and b changes.

The distance between nodes on the path to the empty nodes is equal to the difference of node depths. However, to simplify distance comparison between nodes, the node distance is defined as the cardinality of the set of nodes on the path from the node u to the node v. Formally, a distance function is defined as:

$$d : V \times V \to \mathbb{R}, \text{such that } d(u,v) = |P_u(v)|$$

In the example illustrated by Figure 4.7, $d_1(a,c) \neq d_2(a,c)$ because $d_1(a,c) = 2$ and $d_2(a,c) =$

1, and $d_1(b,c) \neq d_2(b,c)$. Note that the distance function defined in this way reflects a structural change in the tree but the distance function can also reflect modified properties of nodes or edges. It can be concluded that the inheritance relationship for the node has changed if the predecessor node sets are different or if there is a node on the path that has changed the distance relative to the node.

Definition 5 *Let $X_1(V_1 \cup \{\varepsilon\}, E_1)$ and $X_2(V_2 \cup \{\varepsilon\}, E_2)$ be EETs. Let $P_{\varepsilon 1}(v)$ in X_1 and $P_{\varepsilon 2}(v)$ in X_2 for $v \in V_1 \cup V_2$ be predecessor node sets. Let $d_1(u,v)$ be nodes distance function in X_1 and $d_2(u,v)$ in X_2, where $u \in P_{\varepsilon 1}(v) \cup P_{\varepsilon 2}(v)$. The inheritance relationship is changed; that is, node v is edited, if the following applies:*

$$P_{\varepsilon 1}(v) \neq P_{\varepsilon 2}(v) \text{ or } \exists u \in P_{\varepsilon 1}(v) \cap P_{\varepsilon 2}(v) \text{ such that}$$
$$d_1(u,v) \neq d_2(u,v)$$

4.5.3 Direct and indirect edit operations

The edge edit operations described by Definition 2 have direct and indirect effects on inheritance editing. To define indirect effects of edge edit operations on inheritance editing, first, the direct effects of edge edit operations on the changes in node sets $P_\varepsilon(v)$ will be determined, and distance between nodes on the path to the empty node.

Let X_1 and X_2 be EETs, $P_{\varepsilon 1}(v)$ in X_1 and $P_{\varepsilon 2}(v)$ in X_2 be set of predecessor nodes for v in $V_1 \cup V_2$. Edge *add* operation imply empty set $P_{\varepsilon 1}(v)$ and non-empty set $P_{\varepsilon 2}(v)$ because the path for the added node v in X_1 is not defined because the added node is not part of X_1. Similarly, for *delete* edge operation, set $P_{\varepsilon 1}(v)$ is not empty, and $P_{\varepsilon 2}(v)$ is empty because the path to the deleted node v is not defined in X_2. Furthermore, *move* edge edit operation involves a change of the node v parent, resulting in either different sets $P_{\varepsilon 1}(v)$ and $P_{\varepsilon 2}(v)$ or a change in the distance between nodes in these sets.

By Definition 2 and Definition 5, for edge edit operations the following applies:

1. *add*: $v \in V_2 \setminus V_1$, implies $P_{\varepsilon 1}(v) = \varnothing \wedge P_{\varepsilon 2}(v) \neq \varnothing$
2. *delete*: $v \in V_1 \setminus V_2$, implies $P_{\varepsilon 1}(v) \neq \varnothing \wedge P_{\varepsilon 2}(v) = \varnothing$
3. *move*: $v \in V_1 \cap V_2 \wedge p_1(v) \neq p_2(v)$, implies $P_{\varepsilon 1}(v) \neq P_{\varepsilon 2}(v)$ or $\exists u \in P_{\varepsilon 1}(v) \cap P_{\varepsilon 2}(v)$ such that $d_1(u,v) \neq d_2(u,v)$

Meanwhile, as an indirect result of edge edit operations, by Definition 5, the path to the empty node is changed for descendant nodes. Consequently, descendant nodes are involved in indirect edit operations.

Definition 6 *Node $v \in V_1 \cap V_2$ is indirectly edited if $p_1(v) = p_2(v)$ and $P_{\varepsilon 1}(v) \neq P_{\varepsilon 2}(v)$ or $\exists u \in P_{\varepsilon 1}(v) \cap P_{\varepsilon 2}(v)$ such that $d_1(u,v) \neq d_2(u,v)$. Edge $e = (p(v), v)$ is indirectly edited if node v is indirectly edited, as a consequence of a change in inheritance relationship for the node v.*

Because *add*, *delete*, and *move* edge edit operations induce changed paths to the empty node, it can be concluded that indirect operations are an indirect result of the *move* edge edit operations, and possibly an indirect result of the *add* and *delete* edge edit operations, in order 1-2-3 or 2-3, where the numbers represent operation type, as shown in Figure 4.6 and Figure 4.7 (1: *add* or *delete*, 2: *move*, 3: *indirect*).

4.5.4 Inheritance edit operations

In the previous subsections, inheritance change is detected by changes in paths to the empty node. These changes were only observed by detecting whether the inheritance relationship for a node is changed or not; that is, whether the node is edited directly or indirectly. However, to compare inheritance changes between two trees, it is necessary to determine how inheritance has changed for each node. For a single node, one or more nodes can be added, deleted or can change their position or other properties on the path to the empty node. Each such modification on the path to the empty node is a single inheritance operation. Similar to the edge edit operations, *add*, *delete*, and *move* inheritance operations are defined:

Definition 7 *Let $X_1(V_1 \cup \{\varepsilon\}, E_1)$ and $X_2(V_2 \cup \{\varepsilon\}, E_2)$ be EETs, $P_{\varepsilon 1}(v)$ in X_1 and $P_{\varepsilon 2}(v)$ in X_2, for $v \in V_1 \cup V_2$ be sets of predecessor nodes, $d_1(u,v)$ be node distance function in X_1 and $d_2(u,v)$ be node distance function in X_2, where $u \in P_{\varepsilon 1}(v) \cup P_{\varepsilon 2}(v)$.*

Inheritance edit operations on node $v \in V_1 \cup V_2$ are:

1. *add, $u \in P_{\varepsilon 2}(v) \setminus P_{\varepsilon 1}(v)$,*
   node u is added to the inheritance relationship for node v
2. *delete, $u \in P_{\varepsilon 1}(v) \setminus P_{\varepsilon 2}(v)$,*
   node u is deleted from the inheritance relationship for node v
3. *move, $u \in P_{\varepsilon 1}(v) \cap P_{\varepsilon 2}(v) \wedge d_1(u,v) \neq d_2(u,v)$,*
   nodes u and v are moved relative to one another

Remark *As already shown, edge edit operations implicitly result in inheritance changes for the involved node. Consequently, inheritance operations occurred by such changes are direct inheritance operations. Meanwhile, inheritance operations occurred on nodes indirectly are indirect inheritance operations.*

A single edge edit operation could result in multiple inheritance operations. In Figure 4.6 *move* edge operation $(c,e) \rightarrow (a,e)$ results in four *delete* inheritance operations because node c is removed from the path for node e and its descendants f, g, and h. Meanwhile, *move* edge edit operations in Figure 4.7 resulted in overall six inheritance operations. Node a is added to the inheritance relationship for node b. Analogously, node a is removed from the inheritance relationship for node b. Consequently, nodes a and b are relatively moved to nodes c and d.

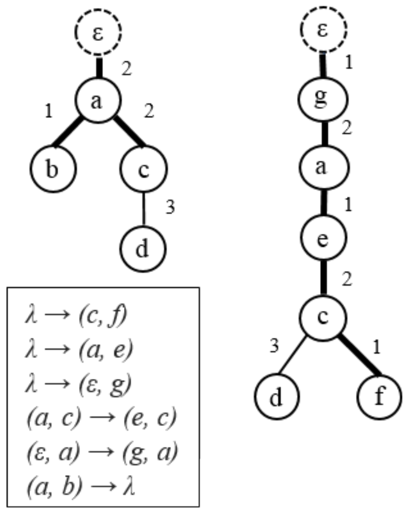


Figure 4.8: Inheritance edit operations example (edge operations: 1 - *add*/*delete*, 2 - *move*, 3 - indirect)

Table 4.1: Inheritance edit operations in tabular form

$v \in V_1 \cup V_2$	$V_a(v)$	$V_d(v)$	$V_m(v)$
a	$\{g\}$	$\varnothing$	$\varnothing$
b	$\varnothing$	$\{\varepsilon, a\}$	$\varnothing$
c	$\{g, e\}$	$\varnothing$	$\{a\}$
d	$\{g, e\}$	$\varnothing$	$\{a\}$
e	$\{\varepsilon, g, a\}$	$\varnothing$	$\varnothing$
f	$\{\varepsilon, g, a, e, c\}$	$\varnothing$	$\varnothing$
g	$\{\varepsilon\}$	$\varnothing$	$\varnothing$

In the previous figures, inheritance operations are intuitively observed as sets of added, deleted, and moved node sets for the edited node, which corresponds to *add*, *delete* and *move* inheritance operations. By using Definition 7, these sets are defined as:

Definition 8 *Let $P_{\varepsilon 1}(v)$ be set of predecessor nodes for v in X_1, $P_{\varepsilon 2}(v)$ in X_2, and $d_1(u,v)$ and $d_2(u,v)$ be distance functions in X_1 and X_2, where $u \in P_{\varepsilon 1}(v) \cap P_{\varepsilon 2}(v)$. Then, set of added nodes $V_a(v)$, deleted nodes $V_d(v)$, and moved nodes $V_m(v)$ on the path from empty node ε to the node v, for $v \in V_1 \cup V_2$ are defined as:*

$$V_a(v) = P_{\varepsilon 2}(v) \setminus P_{\varepsilon 1}(v)$$
$$V_d(v) = P_{\varepsilon 1}(v) \setminus P_{\varepsilon 2}(v)$$
$$V_m(v) = P_{\varepsilon 1}(v) \cap P_{\varepsilon 2}(v) \mid d_1(u,v) \neq d_2(u,v)$$

Inheritance operations can be conveniently shown in tabular form, where columns represent node sets $V_a(v)$, $V_d(v)$, and $V_m(v)$ for every node v in the table rows, whether or not the node is edited. In Figure 4.8, for example, inheritance edit operations occur on all nodes. Consequently, for each node v in X_1 and X_2, at least one of the node sets $V_a(v)$, $V_d(v)$, and $V_m(v)$ is not empty. From the Table 4.1, it can be detected that there are indirect inheritance operations. Node c and d have equal node sets, because direct inheritance changes on node c are propagated to the node d.

From node sets $V_a(v)$, $V_d(v)$, and $V_m(v)$, the connection can be observed between edge edit operations and inheritance operations. For some nodes v, sets $V_a(v)$ and $V_d(v)$ contain empty node ε, which means that these nodes are added or deleted. In Table 4.1, set $V_a(g)$ contains only node ε because node g is added as the new root node. In contrast, nodes v without empty node ε in node sets $V_a(v)$ and $V_d(v)$, are edited by *move* edge operations. By Definition 2 and Definition 7, three cases can be observed for how inheritance edit operations affect $V_a(v)$, $V_d(v)$, and $V_m(v)$ node sets:

1. add, $P_{\varepsilon 1}(v) = \varnothing \wedge P_{\varepsilon 2}(v) \neq \varnothing$ implies $V_a(v) \neq \varnothing \wedge V_d(v) = \varnothing \wedge V_m(v) = \varnothing$
2. delete, $P_{\varepsilon 1}(v) \neq \varnothing \wedge P_{\varepsilon 2}(v) = \varnothing$ implies $V_a(v) = \varnothing \wedge V_d(v) \neq \varnothing \wedge V_m(v) = \varnothing$
3. move, $P_{\varepsilon 1}(v) \neq P_{\varepsilon 2}(v)$ or $d_1(u,v) \neq d_2(u,v)$ implies $V_a(v) \neq \varnothing$ or $V_d(v) \neq \varnothing$ or $V_m(v) \neq \varnothing$

These cases are similar to the observation of edge edit operations impact on the predecessor node set P_ε in the previous section. The main difference is in the first case, where both direct and indirect edit operations are included; that is, cases where $p_1(v) \neq p_2(v)$ or $p_1(v) = p_2(v)$.

4.5.5 Inheritance cost

Inheritance operations describe inheritance changes between trees. To define inheritance dissimilarity measure, it is necessary to determine the cost of inheritance changes between trees, which is equal to the total cost of inheritance operations. In the previous subsection, it was determined how inheritance edit operation corresponds to nodes contained in sets $V_a(v)$, $V_d(v)$, and $V_m(v)$. Let $V_i(v)$ contain all nodes involved in the inheritance operation for node v such that $V_i(v) = V_a(v) \cup V_d(v) \cup V_m(v)$. Accordingly, to determine inheritance cost of nodes contained in $V_i(v)$ on node v, cost function φ is defined for the inheritance edit operation.

Definition 9 *Let $\varphi : (V_1 \cup V_2) \times (V_1 \cup V_2) \rightarrow \mathbb{R}$ be the cost function of the inheritance operation of node $u \in V_i(v)$ on node $v \in V_1 \cup V_2$.*

In the case of *add* operation, both node u from $V_a(v)$ and node v are contained in V_2. Similar to the *delete* operation, where both node u from $V_d(v)$ and node v are in V_1. For *move* operation, node u from $V_m(v)$, and v are in $V_1 \cap V_2$. Furthermore, such arguments are associated with the symmetry property of the function φ, which ensures that adding and deleting the same node on the path to the empty node is complementary. Similar applies to complementary move inheritance operations, involving the same nodes u and v, where nodes repeatedly exchange their relative positions, resulting in the preserved distance.

Function φ can be as simple as $\varphi(v_a, v_b) = c$ for all v_a, $v_b \in V_1 \cup V_2$, where c is a constant number or function φ can be dependent on the distance between involved nodes: $\varphi(v_a, v_b) = c * \frac{d(v_a,v_b)}{k}$, or alternatively, $\varphi(v_a, v_b) = c * e^{-\frac{d(v_a,v_b)}{k}}$, where $d(v_a, v_b)$ is the distance function between nodes v_a and v_b, and k is the constant to control inheritance range and strength of the edit operation. Namely, in the context of edge operation nodes closer to the edge operation are more influenced by the inheritance change than leaf nodes.

Inheritance editing of a single node has previously been shown by the set of nodes $V_i(v)$, which corresponds to the set of inheritance edit operations performed on the node v. Consequently, the inheritance cost for a single node v is defined by the sum of inheritance operations costs on the node v; that is, by the sum calculated by using a function φ over nodes from set $V_i(v)$.

Definition 10 *Let $V_i(v) = V_a(v) \cup V_d(v) \cup V_m(v)$ are node sets containing nodes involved in inheritance operations on node v. Inheritance cost function for node v is $\delta : V_1 \cup V_2 \to \mathbb{R}$, defined as:*

$$\begin{aligned}\delta(v) &= \sum_{u_a \in V_a(v)} \varphi(u_a, v) + \sum_{u_d \in V_d(v)} \varphi(u_d, v) + \sum_{u_m \in V_m(v)} \varphi(u_m, v) \\ &= \sum_{u \in V_i(v)} \varphi(u, v)\end{aligned}$$

where φ is an inheritance operation cost function $(V_1 \cup V_2) \times (V_1 \cup V_2) \to \mathbb{R}$.

If function φ is constant number 1 for all nodes, then the inheritance cost for node f in Figure 4.8 is 5, because set $V_a(f)$ contains five nodes ε, g, a, e, and c. If the source and target trees exchange places, then set $V_d(f)$ contains nodes ε, g, a, e, and c, while set $V_a(f)$ is empty, which results in inheritance cost that is again equal to 5. It can be observed how function δ inherits symmetry property from function φ, because adding and deleting node are complementary operations regarding inheritance operations. The same applies to moved and, indirectly edited nodes.

The inheritance cost between trees is determined by the total cost of inheritance editing of all nodes. Let S_i be the set of edited nodes between trees X_1 and X_2, where $S_i = \{v_1, \ldots, v_i, \ldots, v_n\}$, such that according to Definition 5, $\forall v_i \in V_1 \cup V_2$ implies that $P_{\varepsilon 1}(v_i) \neq P_{\varepsilon 2}(v_i)$ or $d_1(u, v_i) \neq d_2(u, v_i)$. Now, inheritance tree distance can be defined as follows:

Definition 11 *Tree Inheritance Distance (TID) between EETs X_1 and X_2 is the total cost of the edited nodes $S_i = \{v_1, \ldots, v_i, \ldots, v_n\}$ between X_1 and X_2:*

$$d_i(X_1, X_2) = \sum_{v_i \in S_i} \delta(v_i)$$

It can be observed that set of inheritance edited nodes S_i is an extension of the set of edge edit operations S_e with added indirect operations. Because S_i is an extension of S_e, the main difference of inheritance distance algorithm shown in Algorithm 2, from EED Algorithm 1 is in detecting indirect operations. Therefore, the condition at line 4 in Algorithm 1 where parent nodes are compared is replaced by detection of inheritance operations in Algorithm 2 by using Algorithm 3 (procedure `detectInheritanceChanges`). Algorithm 3 detects inheritance operations between source and target trees by detecting changes in predecessor nodes. In Algorithm 2, procedure `detectInheritanceChanges` is used to detect inheritance operations occurred on nodes which are equal in both trees. However, Algorithm 3 could be used to detect direct

Algorithm 2: Tree Inheritance Distance algorithm

```
input  : trees $X_1(V_1 \cup \{\varepsilon\}, E_1)$ and $X_2(V_2 \cup \{\varepsilon\}, E_2)$, distance functions
         $d_1 : V_1 \times V_1 \to \mathbb{R}$, and $d_2 : V_2 \times V_2 \to \mathbb{R}$, and cost function
         $\delta' : V_1 \cup V_2 \times V_1 \cup V_2 \to \mathbb{R}$
output: an inheritance edit distance $d_i$ and set of edited nodes $S_i$

$S_i \leftarrow \varnothing$, $d_i \leftarrow 0$;
foreach node $v$ in $V_1$ do
    $V_i \leftarrow \varnothing$;
    $P_{\varepsilon 1} \leftarrow$ getPreds($X_1$, $v$);
    if contains($V_2$, $v$) then
        $P_{\varepsilon 2} \leftarrow$ getPreds($X_2$, $v$);
        $V_i \leftarrow$ detectInheritanceChanges($v$, $P_{\varepsilon 2}$, $P_{\varepsilon 1}$, $d_1$, $d_2$)
    else
        $V_i \leftarrow P_{\varepsilon 1}$;
    end
    if $V_i \neq \varnothing$ then
        $S_i \leftarrow S_i \cup \{v\}$;
        $d_i \leftarrow d_i + \delta(v, V_i)$;
    end
end
foreach node $v$ in $V_2$ do
    $V_i \leftarrow \varnothing$;
    if not contains($V_1$, $v$) then
        $V_i \leftarrow$ getPreds($X_2$, $v$);
        $S_i \leftarrow S_i \cup \{v\}$;
        $d_i \leftarrow d_i + \delta(v, V_i)$;
    end
end
```

inheritance operations caused by add and delete edge edit operations, but such operations are straightforwardly detected over predecessor nodes. To detect predecessor nodes for a given node, by traversing the path to the empty node, function `getPreds` is used. Note that the result of function `getPreds` is used as input for Algorithm 3, where function `getPreds` is used to obtain predecessor node sets for nodes equal in both trees (line 4 and 6 in Algorithm 2). In Algorithm 2, cost function δ' is used to calculate inheritance operations cost. Input parameters to cost function δ' are node and its inheritance operations, as a set of changed predecessor nodes V_i.

Algorithm 3 is similar to Algorithm 1 because *add* and *delete* inheritance operations are determined by the difference between node sets, in this case by predecessor node sets. However, instead of parent functions, move inheritance operation is determined by given distance functions d_1 and d_2.

To determine the time complexity of the algorithm to calculate tree inheritance distance,

Algorithm 3: Procedure `detectInheritanceChanges`(v, $P_{\varepsilon 1}$, $P_{\varepsilon 2}$, d_1, d_2)

```
procedure detectInheritanceChanges($v$, $P_{\varepsilon 1}$, $P_{\varepsilon 2}$, $d_1$, $d_2$)
    $V_i \leftarrow \varnothing$;
    foreach node $u$ in $P_{\varepsilon 1}$ do
        if contains($P_{\varepsilon 2}$, $u$) then
            if $d_1(u,v) \neq d_2(u,v)$ then
                $V_i \leftarrow V_i \cup \{u\}$;
            end
        else
            $V_i \leftarrow V_i \cup \{u\}$;
        end
    end
    foreach node $u$ in $P_{\varepsilon 2}$ do
        if not contains($P_{\varepsilon 1}$, $u$) then
            $V_i \leftarrow V_i \cup \{u\}$;
        end
    end
    return $V_i$;
```

let the n_1 be the number of edges in X_1, and n_2 be the number of nodes in X_2. Let k_1 and k_2 denote the average distance from a node to the empty node; that is, average node depth in X_1 and X_2, respectively. The time complexity of the tree inheritance distance algorithm (Algorithm 2) is $O(n_1 * k_1 + n_2 * k_2)$. It is assumed that an appropriate `contains` function is used with the complexity $O(1)$, e.g., hash function. Similar to Algorithm 1, this algorithm iterates through a set of nodes V_1 and V_2, searching for matching nodes by using the lookup function `contains`. This function determines predecessor nodes, with path traversal complexity $O(k)$.

First, let us consider the case discussed in Section 4.4 with a maximum number of edge operations; that is, when all nodes are deleted in the source tree, and all nodes are added to the target tree. For each of n_1 deleted nodes in X_1, the algorithm determines its predecessor's nodes by using function `getPreds`, where traversal complexity to the empty node is k_1. Analogous is for added nodes, where traversal to the empty node is performed n_2 times, where average node depth is k_2. Therefore, the complexity is $O(n_1 * k_1 + n_2 * k_2)$.

Furthermore, let us consider the case when all nodes are moved. In this case $n = n_1 = n_2$. The traversing of paths in both trees is performed only for matched nodes, thus n times, with the complexity $O(k_1)$ in the first tree and $O(k_2)$ in the second tree. Therefore, the complexity is $O(n * (k_1 + k_2))$.

Procedure `detectInheritanceChanges` (Algorithm 3) additionally compares paths to the empty node for each matched node in both trees with the complexity $O(k_1 + k_2)$, if an appropriate $(O(1))$ function contains is used. Furthermore, the complexity of Algorithm 3, $O(k_1 + k_2)$, is valid if the complexity of distance functions d_1 and d_2 is $O(1)$. If the complexity of distance functions is greater than $O(1)$, then it would increase the algorithm complexity. To calculate the tree inheritance distance, for each node the traversing of predecessor nodes is performed twice: once in Algorithm 2, and once in Algorithm 3. However, the time complexity is not increased; for example, in the case when all nodes are moved ($n = n_1 = n_2$), the complexity is $O(n*(k_1+k_2))+n*O(k_1+k_2) = O(2*n*(k_1+k_2)) \approx O(n*(k_1+k_2))$.

Chapter 5

Extended DAOP model

In this chapter, the identification of dynamic update model properties to support class hierarchy changes is performed, with an emphasis on efficiency and functionality. According to the DSU requirements and the existing specifications of program changes, it is determined that differences between versions can be described with aspects. Although existing dynamic aspect weaving approaches are limited with dynamic aspect functionality, by providing additional classes they can be used for dynamic updating. Changing the hierarchy of classes is analyzed as part of the main topic of the dissertation. Therefore, based on the identified properties and algorithms defined in Chapter 4, a dynamic software update model that supports the class hierarchy change by using dynamic aspects is presented and described.

5.1 Model properties

Chapter 2 describes the main requirements for a DSU, as follows: *availability*, *changeability*, *performance*, and *usability*. However, to achieve a functional dynamic update based on these requirements, the dynamic update model should consist of several components. The first component requires two versions of the comparison program as source code. The conversion of program versions into a suitable data structure is performed for program analysis to detect differences between versions. The differences are produced in a suitable descriptive format of changes for dynamic updating. The executable environment then accepts and applies any differences to the running version. This conceptual update model is given in Figure 5.1.

The following steps define the required model properties:

1. receiving different versions of the program
    - receives two different program versions as program code: the currently running program version - v_1, and the program version to which the running program needs to be updated v_2
2. program version preparation

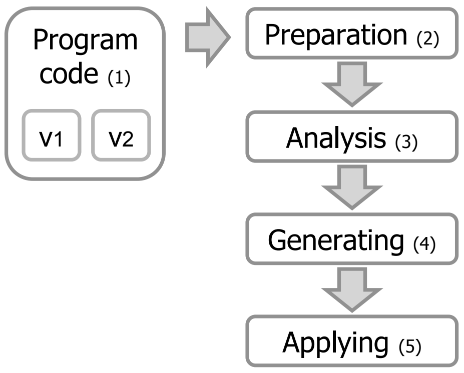


Figure 5.1: Conceptual dynamic update model

- program version code is converted to a suitable data structure for comparison

3. detecting the differences between versions
   - data structures from two program versions are compared, and any differences between them are detected
4. generating differences in a suitable format
   - detected differences between versions are stored in a suitable format
5. applying the changes
   - detected differences (introduced by the updated program version) are applied to the running program

The dynamic updating process in [9, 24] is similar to the such model. The dynamic aspects are interesting as a specification of changes because there is no specified format to describe the changes. However, an aspect paradigm as cross cutting concern is suitable for this use.

5.2 Dynamic aspects (DAOP)

The Aspect-Oriented Paradigm (AOP) introduces separation of cross-cutting concerns as aspects to retain the modularity and code reuse [17]. Aspects enable extending the program functionality on different hierarchy levels. Program code in the form of aspects is weaved into the program during compile or load time, as with AspectJ [18]. It is used, for example, to introduce logging or access control to objects belonging to different classes. For example, in Figure 5.2, to enable logging for methods in classes in the Object-Oriented Paradigm (OOP), it is necessary to change every method in all classes. An aspect can be used to easily define a set of methods; for example, to enable logging, called pointcut, and program code to execute at those pointcuts called advice.

To show how weaving works, in Figure 5.3 an example aspect is defined in AspectJ [18]. The advice in the example contains the printing *hello* statement and is weaved after the call defined by pointcut to method `print()`. When method `run()` is executed, the console will

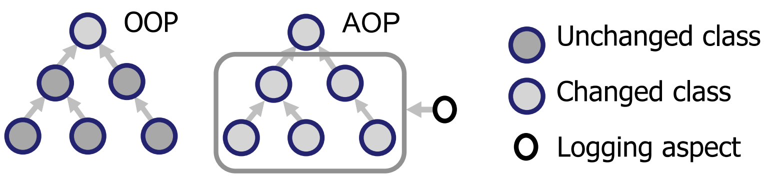
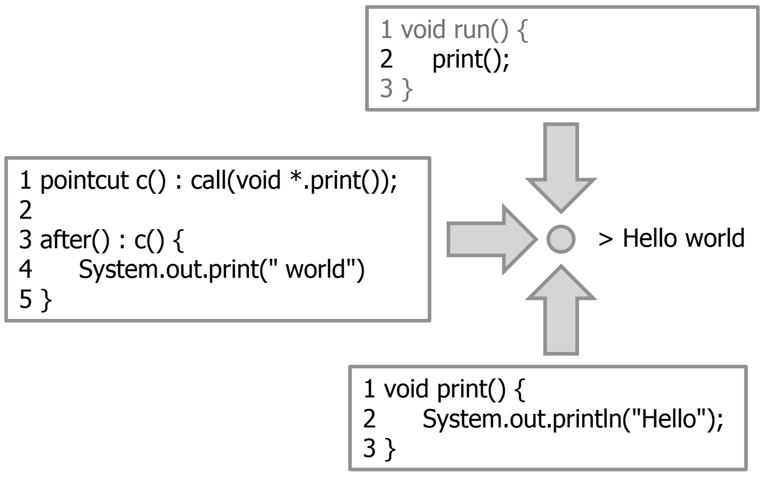


Figure 5.2: Program changes to enable logging OOP vs AOP

then print “Hello world”.



```
void run() {
    print();
}
```

```
pointcut c() : call(void *.print());

after() : c() {
    System.out.print(" world")
}
```

```
void print() {
    System.out.println("Hello");
}
```

```
> Hello world
```

Figure 5.3: "Hello world" aspect example in *AspectJ*

Meanwhile, Dynamic AOP (DAOP) enables aspect code weaving during the runtime, which can be used for dynamic updating [10, 19, 20, 21, 22]. The general objective of research using DAOP is to enable as many aspect-functionalities as static approaches support, such as AspectJ [18]. With their separation characteristics, the aspects provide a suitable update specification unit. Dynamic aspects also provide a simple way to add or remove aspects, consequently dynamic changes can be applied and reversed. It has been shown that DAOP [54] introduces less overhead in steady-state, thus having less impact on performance.

The downside is that dynamic aspects only support method body modification. To expand the supported set of changes, it is necessary to generate changes in the form of other types of aspects and possible additional classes by using offline code analysis performed on different software versions, as described in DSU approaches based on DAOP [24, 25]. These approaches analyze code to extract program changes in the form of additional classes, static and dynamic aspects to extend the supported changes. They correspond to the conceptual model shown in Figure 5.1, where the program code is prepared for analysis and produces an update specification in the form of aspects. These aspects are then used as input to dynamic aspect system to perform an update.

Example 5.1: Prose method redefinition dynamic aspect

```java
public class MethodDynamicAspect extends DefaultAspect {
    public Crosscut c1 = new MethodRedefineCut(){

        /* advice */
        public String METHOD_ARGS(A target, int arg1) {
            /* new body statements */
        }

        /* pointcut */
        protected PointCutter pointCutter() {
            return Within.method("m").AND(Within.type("A"));
        }
    };

    public Crosscut c2 = new MethodRedefineCut(){
        public String METHOD_ARGS(ANY target, REST params) {
    ...
}
```

5.2.1 Prose

Several systems have been developed to support dynamic aspects. For example, *Hotwave* [21] is one of the latest systems and it supports the AspectJ language for defining aspects; however, it does not directly support the *around* advice used to replace method body and cross-cutting for inter-type declarations that affect class hierarchy. Additionally, as for most of the existing dynamic aspect systems, the source code is not available. Meanwhile, Prose [27] in its third version is based on JVM hot-swap feature. Prose enables method redefinition, which can be used for dynamic updates, and its source code is available. However, the constructor redefinition is not supported because the constructor aspects behave as *before* advice. Furthermore, Prose defines the custom format to define aspects in the form of Java classes and methods. In this dissertation, the Prose definition of aspects is used to redefine the method bodies with *around* advice and constructor advice to redefine the constructor body.

An aspect in Prose is defined as a Java class that inherits `DefaultAspect` class defined by the Prose library. The aspect class contains one or more crosscuts, where each crosscut defines advice and pointcut, shown in Example 5.1 as `c1` and `c2`. Crosscuts are defined as classes depending on the crosscut type. In addition to `MethodRedefineCut` for redefining the method body there is crosscut `MethodCut` used in methods for *before* and *after* advice. Furthermore, `GetCut` and `SetCut` are used to intercept reading and writing to a class field. The advice in a crosscut is identified by method `METHOD_ARGS`. The first parameter is the object on which method has been invoked, where the joinpoint is activated. Additional arguments correspond to the matched method arguments that the statements contained in the advice can access. Furthermore, as parameters, the type `ANY` can be used to denote any class for advice

Example 5.2: Prose constructor dynamic aspect

```java
public class ConstructorDynamicAspect extends DefaultAspect {
    public Crosscut c = new ConstructorCut(){

        public void METHOD_ARGS(A target, int arg1) {
            /* new body statements */
        }

        protected PointCutter pointCutter() {
            return Within.type("A");
        }
    };
}
```

activation, and `REST` for any additional method parameters. Joinpoint is defined by pointcut, where method `Within.method` determines the name of the method to activate joinpoint, and method `Within.type` determines the class for joinpoint activation. Furthermore, methods `AND` and `OR` can be used for various joinpoint combinations. Prose enables joinpoint definition as pointcut in combination with advice method arguments. In Example 5.1, the advice method contains an argument with integer type. Therefore joinpoint is activated for method `m`, with one parameter of type integer, which is defined in class `A`. There are other possible pointcut and advice combinations, but they are out of the scope for this dissertation.

The dynamic aspect for the constructor is similarly defined as the dynamic aspect for method redefinition, as shown in Example 5.2. However, crosscut is defined as `ConstructorCut`, and pointcut is only defined over the type. To determine the joinpoint for the target constructor, a combination of advice method arguments and pointcut is used. In the example, joinpoint is activated for constructor of class *A* with one parameter of integer type.

5.3 Extended model

The specification of changes between different versions of programs is often given in the form of modified program classes [12, 15], custom defined format [9, 14], or a custom defined programming language [19, 23]. The program version can be represented as a data structure, such as a tree. Meanwhile, aspects with its cross-cutting concern functionality can be used as a suitable format for program changes description. Dynamic aspects provide flexible addition and deletion of the program changes. Existing research [24, 25] on extending the DAOP shows the feasibility of such a model.

Dynamic aspects approaches currently support class member changes only because the class hierarchy changes are not supported. Static aspects, such as in AspectJ, support some limited type changes with intertype declarations mechanism, but currently none of the dynamic aspect approaches include class hierarchy changes. Cech [24] conclude that there is a need for dynamic aspect system that can perform class hierarchy changes. Consequently, in this dissertation the DAOP model for dynamic updates is extended to support class hierarchy changes.

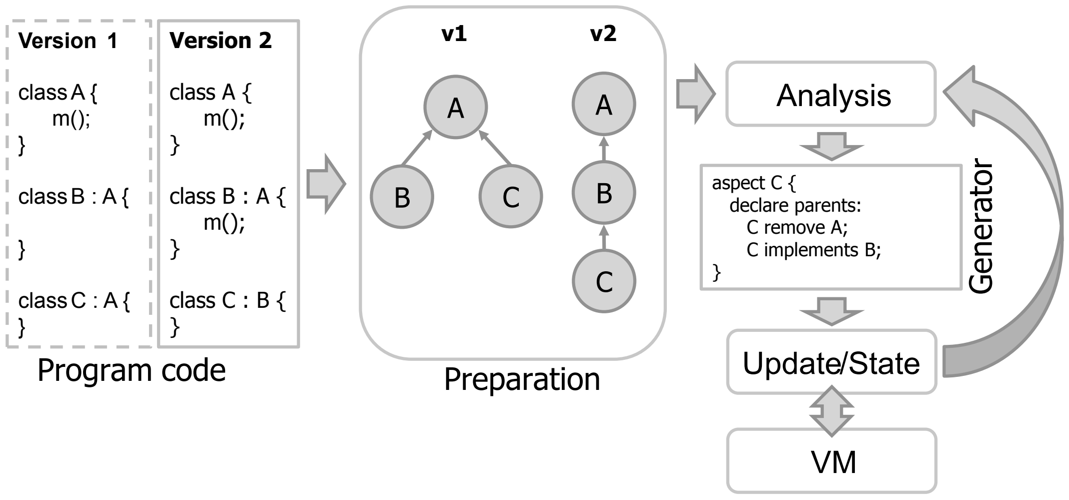


Figure 5.4: Extended DAOP update model

The extended DAOP update model that supports class hierarchy changes and runtime phenomena detection is shown in Figure 5.4. Object-oriented program code is converted to an abstract structure, such as abstract tree, to detect and extract program differences. The program differences are then used for analysis, for two different purposes: update specification and program state. Program state is related with runtime phenomena described in Chapter 6. Update specification is generated in the form of dynamic aspects.

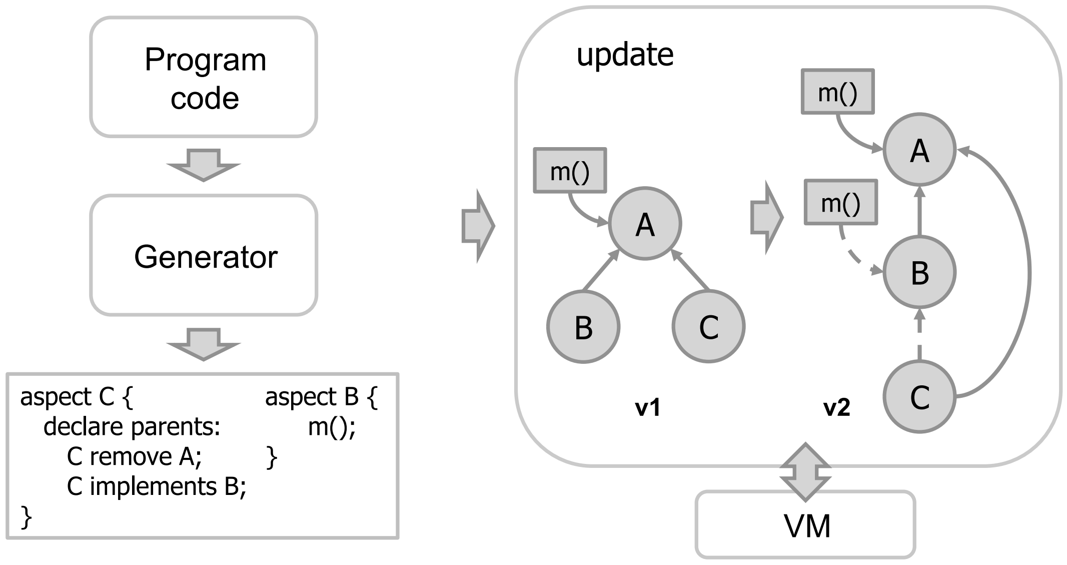
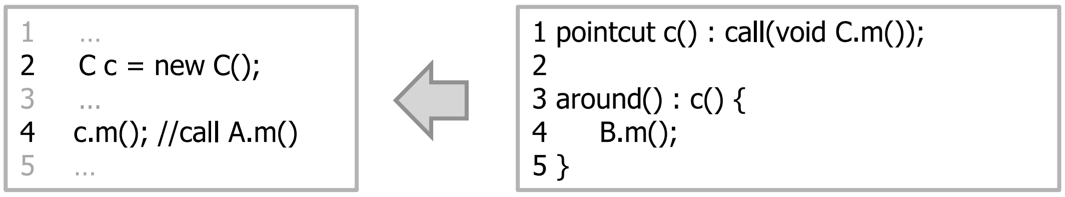


Figure 5.5: Extended DAOP model with hierarchy change support

The Extended DAOP update model shown in Figure 5.5 accepts two versions of Object-Oriented program. Program code is converted to a tree, where each class represents one node in the tree. Analysis performed on two trees determine changes in class hierarchy between versions. By using the generator, changes between versions are produced in the form of aspects. For example, in Figure 5.5 classes *B* and *C* in version v_1 inherit class *A*, whereas class *C* in version v_2 inherit class *B*. In version v_2, class *B* implements method `m()` with the same signature as the method implemented in class *A*. Consequently, class *C* in v_2 inherits method `m()` from class *B* instead from class *A* as in the v_1. In the update model, such changes are detected by analysis, and the generator creates the aspects with additional classes to support dynamic updating. Figure 5.6 shows a program code snippet executed in v_1. The call to method `m()` of class *C* is a call to method `m()` implemented in class *A* because class *C* does not override the method. In v_2, it is necessary to redirect the call from `A.m()` to `B.m()`. With DAOP, this is achieved with the aspect defined on the right-hand side in Figure 5.6. The aspect with the *around* advice replaces the call to `A.m()` with statements defined in the advice body, containing the call to method `B.m()`. To achieve this behavior, the advice body does not contain a call to method `proceed`, which is usually called in the *around* advice. The following subsection describes algorithms for detecting changes in the class hierarchy between program versions.



```
  ...
  C c = new C();
  ...
 c.m(); //call A.m()
 ...
```

```
pointcut c() : call(void C.m());

around() : c() {
    B.m();
}
```

Figure 5.6: Replacing the call to method A.m() in v_1 to B.m() in v_2

5.3.1 Program changes detection algorithm

In the context of the behavior, class in Object-Oriented paradigm is identified with its interface to other classes in the form of accessible methods. Hierarchy is determined in the runtime environment, such as Java Virtual Machine (JVM) that executes intermediate program code (i.e. bytecode). Existing dynamic updating solutions regarding changes in the class hierarchy rely on the modifications in the executing environment, such as modifying data structures containing information about parent classes (JVM), intervening compiled code or calls. By using dynamic aspects (i.e. DAOP) to redirect method calls, the program's behavior can be altered during the runtime, as shown by Cech [24]. To modify a running program to match the program running in the updated version, it is necessary to determine changed relationship between classes and changed class members in the updated version. In the previous section, in Figure 5.5, class *C* changed the parent, instead of class *A* in v_1, class *B* is the parent class in v_2. Consequently, there is a change in type, which is reflected as change in the class hierarchy. To support type changes, Algorithm 2 from Chapter 4 is used. Type changes for each class are detected as changes on the path to the root class for classes matched in both versions. Besides the type changes, inheritance member changes (as described in the previous section) also affect the descendant classes. Therefore, it is necessary to determine changes in class members. Accordingly, the Tree Inheritance Distance (TID) algorithm is modified by detection of changes in class members for matched classes, as shown by procedure `detectMatchedChanges` in Algorithm 4. Meanwhile, in the previous section, in Figure 5.5 class C contains method `B.m()` instead of method `A.m()`. In the v_2 method overriding occurs as described in Chapter 3. To apply dynamic update regarding method overriding, it is necessary to detect such methods by procedure `getOverridingMethods`. Moreover, instead of detecting added and deleted predecessor nodes, similar to the Extended Edge Distance (EED) algorithm shown in Algorithm 1, added and deleted classes are detected. Furthermore, in comparison to the Algorithm 2, distance calculation is omitted, and the input trees T_1 and T_2 represents class hierarchy from versions v_1 and v_2. The algorithm to detect changes between two program versions is shown in Algorithm 4.

Algorithm 4: Program changes detection algorithm

```
input  : class inheritance trees $T_1$ in $v_1$ and $T_2$ in $v_2$
output: set of type changed classes $C_t$, changed members $C_m$ for matched classes and
        overriding methods $C_o$

$A, D, \leftarrow \varnothing$;
$C_t, C_m, C_o \leftarrow \varnothing$;
foreach class c in $T_1$ do
    $V_i \leftarrow \varnothing$;
    $P_{\varepsilon 1} \leftarrow$ getPreds($T_1$, $c$);
    if contains($T_2$, $c$) then
        $P_{\varepsilon 2} \leftarrow$ getPreds($T_2$, $c$);
        $V_i \leftarrow$ detectInheritanceChanges($c$, $P_{\varepsilon 1}$, $P_{\varepsilon 2}$);
        $C_m(c) \leftarrow$ detectClassChanges($c$, $T_1$, $T_2$);
        $C_o \leftarrow C_o \cup$ getOverridingMethods($P_{\varepsilon 1}$, $P_{\varepsilon 2}$, $C_m(c)$);
        if $V_i \neq \varnothing$ then
            $C_t \leftarrow C_t \cup \{c\}$;
        end
    else
        $D \leftarrow D \cup \{c\}$;
    end
end
foreach class c in $T_2$ do
    if not contains($T_1$, $c$) then
        $A \leftarrow A \cup \{c\}$;
    end
end
```

The algorithm to detect program changes is based on the Algorithm 2 to detect inheritance changes, where procedure `detectMatchedChanges` is introduced to detect class member changes for matched classes, and procedure `getOverridinMethods` is used to detect overriding methods. The algorithm to detect member changes for matched class is shown in Algorithm 5. Member changes are detected based on the class in versions v_1 and v_2 obtained by method `getClass` for the given class trees T_1 and T_2. A comparison is performed on the set of methods and fields in c_1 and c_2, where constructor changes are detected as special kind of methods. Changes are detected as difference between set of members of the same kind (e.g. fields) in both versions. Members that exist in v_1 and that do not exist in v_2 are deleted (m_d and f_d). In contrast, members that exist in v_2 and do not exist in v_1 are added (m_a and f_a). Member type changes are detected as member deletion and addition. Furthermore, for methods and constructors with the same signature, body comparison is performed. The difference between statements of matched methods or constructors is detected as member body change (m_b).

Method overriding is detected separately based on the hierarchy tree. Methods detected as changed class member are candidates for changed overriding methods. Detection of changed overriding methods is a prerequisite for creating classes and statements to support dynamic changes related to method overriding mechanism. Changes in overriding method are detected as changes in overriding method implementation, or if the overriding method is added or deleted regarding inheritance relationship. For example, in Figure 5.5 method $m()$ is added to class B. Because method $m()$ with the same signature is implemented in class A, method in B overrides method in A. Furthermore if one of the overriding method implementations is changed, deleted or added, then every method in the inheritance relationship is collected for the further steps. The algorithm to collect methods in overriding relationship where changes occur between two program versions is given in Algorithm 6. For each class with changed method members, predecessor classes are traversed in both v_1 and v_2. First, methods with body changes m_b are compared by signature (`contains` method) with methods in predecessor class from $P_{\varepsilon 2}$ in v_2. If a method with an equal signature exists, then an overriding mechanism is detected, and because there is a change in the method implementation in one of the classes, both methods override and overridden are added as changed overriding methods (O). A similar process is used with added methods, where the added method is compared by signature with methods in predecessor classes. Method overriding in v_1 might not exist, but method added in v_2 of the same signature as method defined in the predecessor classes results in method overriding in v_2. Meanwhile, if overriding exists, then the added method overrides one of the predecessor method implementation. Therefore, both method of the same signature in predecessor classes and added method are added as changed overriding methods. Contrary to the added methods, deleted methods are compared to the methods of predecessor classes $P_{\varepsilon 1}$ in v_1. By deleting the overriding method, the method in the predecessor class is used for the class where the method is deleted, or the overriding mechanism is removed if all methods of the same signature are removed. Therefore, if a method with the same signature as the deleted method is found in the predecessor classes, then

Algorithm 5: Procedure detectClassChanges(c, T_1, T_2)

```
procedure detectClassChanges($c$, $T_1$, $T_2$)
    $c_1 \leftarrow$ getClass($c$, $T_1$);
    $c_2 \leftarrow$ getClass($c$, $T_2$);
    $M_c, F_c \leftarrow \varnothing$;
    $m_a, m_d, m_b \leftarrow \varnothing$;
    foreach method $m$ in $c_1$ do
        if contains($c_2$, $m$) then
            if compareBody($m$, $c_1$, $c_2$) then
                $m_b \leftarrow m_b \cup \{m\}$;
            end
        else
            $m_d \leftarrow m_d \cup \{m\}$;
        end
    end
    foreach method $m$ in $c_2$ do
        if not contains($c_1$, $m$) then
            $m_a \leftarrow m_a \cup \{m\}$;
        end
    end
    $M_c \leftarrow (m_a, m_d, m_b)$;
    $f_a, f_d \leftarrow \varnothing$;
    foreach field $f$ in $c_1$ do
        if not contains($c_2$, $f$) then
            $f_d \leftarrow f_d \cup \{f\}$;
        end
    end
    foreach field $f$ in $c_2$ do
        if not contains($c_1$, $f$) then
            $f_a \leftarrow f_a \cup \{f\}$;
        end
    end
    $F_c \leftarrow (f_a, f_d)$;
    return $(M_c, F_c)$;
```

both methods, deleted and predecessor method, are added as the changed overriding methods.

5.4 Classes for dynamic update

To perform a dynamic update, classes are created that reflect changes between program versions, as follows: *dynamic*, *difference* (*diff*) and *dynamic aspect* classes. *Dynamic* classes are classes with relationship change in the class hierarchy tree, created to support type changes regarding hierarchy. *Difference* classes are classes with members changes such as added, deleted or type

Algorithm 6: Procedure getOverridingMethods(c, $P_{\varepsilon 1}$, $P_{\varepsilon 2}$)

```
procedure getOverridingMethods($P_{\varepsilon 1}$, $P_{\varepsilon 2}$, $m_c$)
    $O \leftarrow \varnothing$;
    $(m_a, m_d, m_b) \leftarrow m_c$;
    foreach class $p$ in $P_{\varepsilon 2}$ do
        foreach method $m$ in $m_b$ do
            if contains($p$, $m$) then
                $O \leftarrow O \cup$ getMethod($p$, $m$);
                $O \leftarrow O \cup \{m\}$;
            end
        end
        foreach method $m$ in $m_a$ do
            if contains($p$, $m$) then
                $O \leftarrow O \cup$ getMethod($p$, $m$);
                $O \leftarrow O \cup \{m\}$;
            end
        end
    end
    foreach class $p$ in $P_{\varepsilon 1}$ do
        foreach method $m$ in $m_d$ do
            if contains($p$, $m$) then
                $O \leftarrow O \cup$ getMethod($p$, $m$);
                $O \leftarrow O \cup \{m\}$;
            end
        end
    end
    return $O$;
```

changes. *Dynamic aspect* classes are created to replace method and constructor bodies; that is, statements to use created *dynamic* and *difference* classes. Furthermore, dynamic aspects are used to enable method and constructor body changes.

5.4.1 Type changes

An example of a class inheritance tree change shown in Figure 5.7 induces type change; however, this change cannot be performed dynamically as current VM does not support superclass change during runtime. To provide a dynamic update with type change, inheritance tree in the currently running version (v_1) is modified by differences in the relationships between classes. Difference is reflected with *dynamic classes*, identical to classes defined in v_2 with changed name, such that suffix "Dynamic" is added. Suffix "Dynamic" is abbreviated with "dyn" in further text. By changing the name, the classloader in the VM can load the class with a changed parent that exists in the currently running version. For example, in Figure 5.7 the relationship is changed for class *C* in v_1, where the parent is class *B* instead of class *A* in v_1. However, class *A* is a parent of class *B*, therefore class *C* still inherits from class *A* in v_2 through class *B*. The parent change for class *C* is reflected by creating *dynamic* class C_{dyn}. Because class *C* in v_2 defines class *B* as parent, class C_{dyn} also defines class *B* as parent instead of class *A*.

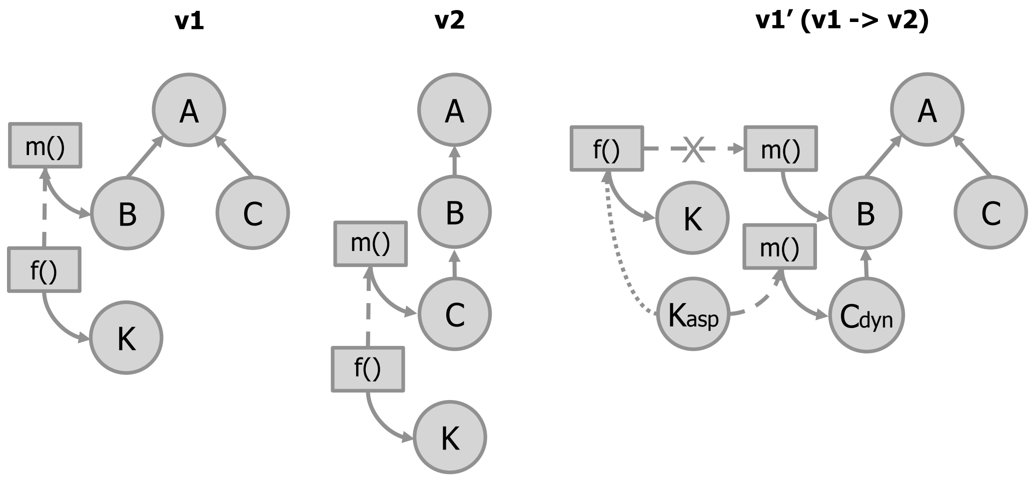


Figure 5.7: Dynamic class C_{dyn} example

In Figure 5.7 differences between class inheritance trees in v_1 and v_2 are shown in the form of changed parent classes for class *C*. To reflect type change, *dynamic* class C_{dyn} is loaded in version v_1. However, to load *dynamic* class in the currently running version, it is necessary to redefine usage of the class *C* in v_1 by changing the statements that declare class *C* with the statements that declare class C_{dyn}. Class usage redefinition is supported by creating *dynamic aspect* classes. Consequently, type change is loaded into the running version conforming to the client-supplier pattern introduced in [24], where *dynamic aspect* and *dynamic* classes represent supplier and class using *dynamic* class represent client. In Example 5.3, class *K* contains method $f()$, which defines variable *b* of type *B* and returns value returned by method `m()` defined in class *B*. Class *K* is the client because method `f()` contains definition of variable with changed type in v_2 (line 4 Example 5.3). Executing method `f()` from *K* in version v_2 with classes from v_1 would raise exception of incompatible types. Therefore *dynamic aspect* class *KDynamicAspect* is created to replace method `m()` with method containing *dynamic* class *CDynamic* instead of class *C*. The aspect method contains statements in the method `m()` with changes in line 4, where type *CDynamic* is instantiated; that is, `B b = new CDynamic()`. *KDynamicApect* class as Prose [27] dynamic aspect is shown in Example 5.4. Similar would be found with the other type related statements shown in Example 3.1, where the class *C* identifier would be replaced with *dynamic* class *CDynamic*.

If the field or the constructor contains *dynamic* class invocation, additionally to *dynamic aspect* and *dynamic* classes, then *difference* classes are created. *Difference* classes contains difference between classes in two program version, such as added or deleted class members (e.g. methods). *Difference* classes are named with added suffix "diff" to the name of the class

Example 5.3: Type changed class usage example

```
/*      v1      */              /*      v2      */
public class K {                public class K {
    public String f() {             public String f() {
        B b = new B();                  B b = new C();
        return b.m();                   return b.m();
    }                               }
}                               }
```

Example 5.4: Prose dynamic aspect containing type change example in Figure 5.7

```
public class KDynamicAspect extends DefaultAspect {
    public Crosscut c = new MethodRedefineCut(){

        public String METHOD_ARGS(K target) {
            B b = new CDynamic();
            return b.m();
        }

        protected PointCutter pointCutter() {
            return Within.method("f").AND(Within.type("K"));
        }
    };
}
```

with differences between versions. *Difference* class is abbreviated with "diff" in further text. Cases when *diff* classes are created are described in the following subsection.

5.4.2 Member changes

Beside type changes as the result of changes in the class inheritance tree, classes in the inheritance tree can also be affected by changes in predecessor classes members. In the previous subsection, method is replaced with dynamic aspect to support type change; however, the same procedure is applied when class method body is changed. *Dynamic aspect* class is created for changed class containing method with statements from version v_2, when *dynamic* class is not necessary because there is no type change. Furthermore, if the class members are changed, added or deleted, then the *diff* class is created, containing changed class members. When the class is candidate for *dynamic* and *diff* class, a *dynamic* class is created. In the rest of this subsection, each of the class member changes are described by creating classes for a dynamic update.

Methods

The introduction mentions that the changed method body results in *dynamic aspect* class for container class. However, signature change, added and deleted methods, require a *diff* class to

Example 5.5: Added method usage example

```
/*      v1      */              /*      v2      */
public class K {                public class K {
    public String f() {             public String f() {
        B b = new B();                  B b = new B();
        return b.m();                   return b.m2();
    }                               }
}                               }
```

Example 5.6: *Diff* class example with added method

```
public class BDiff {
    B target = null;

    public BDiff(B target) {
        this.target = target;
    }

    public String m2() {
        return "B.m2()";
    }
}
```

be created for the changed class to support a dynamic update. For example, if the method is added, then the method definition is extracted as member of *diff* class. Each *diff* class contains "target" reference to the modified object similar to the Cech [24] "that" to access modified object members. In the Example 5.5, in v_2 method `m()` from class B returns value of method `m2()` added in v_2.

To support a dynamic update, *diff* class B_{diff} is created (Example 5.6), containing definition for method $m_2()$ defined in v_2. Furthermore, *target* object is not used to access class members because the new method does not access any of the changed class members. The constructor of *diff* class contains class B object to which *diff* class belongs. Pairing and access to *diff* class objects is handled by *DSU manager* described in Chapter 8.

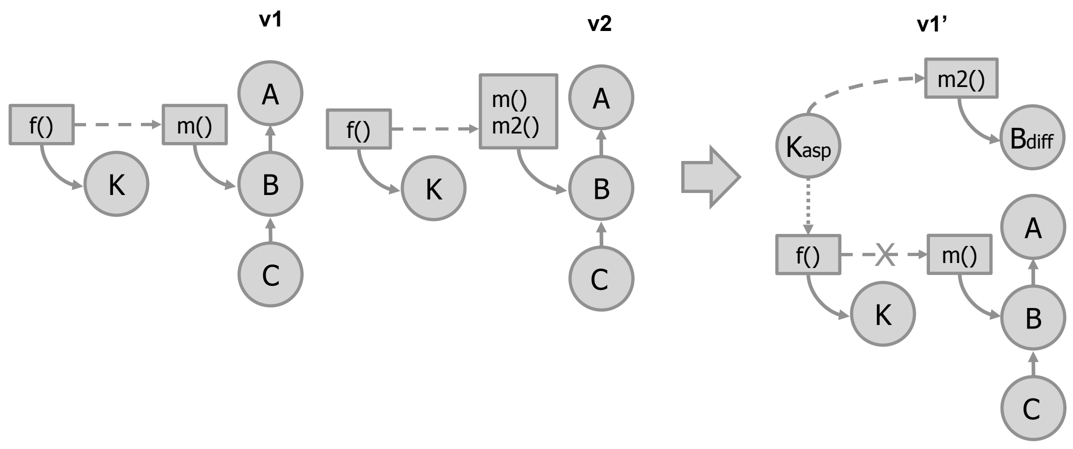


Figure 5.8: Added method for class B and corresponding diff class B_{diff} and K_{asp} classes relationship

Example 5.7: *Dynamic aspect* class for class *K* in Figure 5.8

```
public class KDynamicAspect extends DefaultAspect {
    public Crosscut c0 = new MethodRedefineCut(){

        public String METHOD_ARGS(K target) {
            B b = new B();
            return ((BDiff)DSUManager.Diff((Object)b)).m2();
        }

        protected PointCutter pointCutter() {
            return Within.method("f").AND(Within.type("K"));
        }
    };
}
```

Example 5.8: Added method with overriding example

```
1 /*      v1      */              /*      v2      */
2 public class K {                public class K {
3     public String f() {             public String f() {
4         B b = new B();                  B b = new C();
5         return b.m();                   return b.m2();
6     }                               }
7 }                               }
```

Figure 5.8 shows the class hierarchy in v_1 and v_2, together with changes for class B in the form of an added method `m2()`. Furthermore, classes K_{asp} (Example 5.7) and B_{diff} (Example 5.6) are *dynamic aspect* and *diff* classes for class K and B respectively. The slashed lines indicate calls to methods, whereas the dotted lines indicate aspect pointcut. K_{asp} class replace class K method `f()` with statement invoking method `m_2()` defined in class B_{diff}. *Dynamic aspect* class K_{asp} applies new method introduced in class B in v_2 to class K. If there were more matched classes that use the new method in the v_2, then the number of *dynamic aspect* classes would increase according to the number of such classes.

The method deleted in v_2 results in a *dynamic aspect* class that replaces the method body with a statement throwing an exception. It is expected that methods deleted in v_2 are not used, therefore statements containing invocation to deleted methods are changed in v_2. Statements are changed by *dynamic aspect*, *diff* and *dynamic* classes depending on the context of change.

Inherited and overriding methods

Considering the overriding method mechanism described in Chapter 3, in Example 5.5, a call to method $m2()$ can be performed directly on *diff* class. However, the method $m()$ in class K is changed as shown in Example 5.8 on line 4, where variable b is instantiated as object of class C. Since class C does not implement $m2()$ method, method is inherited from class B.

In this case, *dynamic aspect* class K_{asp} can invoke method $m2()$ defined in class B_{diff} similar to the process described for added method in the this subsection. However, in this case object of class B_{diff} is created for object of class C. Class B_{diff} contains differences for class B because in this case class C is only changed through added inherited method $m2$, while class C_{diff} is not created. Constructor of B_{diff} accepts object of class C as target because class C is descendant (i.e. subtype of class B). Therefore, *diff* classes are created only for classes with changed members, and these changes are reflected to descendant classes by creating and pairing the predecessor *diff* class object to the descendant class objects.

Meanwhile, if class C in v_2 implements method $m2$ in Example 5.5, overriding mechanism is used. Although variable b is declared as object of class B, method defined in class C is used because class C overrides method $m2()$ defined in class B. As in the case of inheritance, the *dynamic aspect* class cannot directly invoke method $m()$ defined in class B_{diff}. Because of subtype relationship, VM performs dynamic method dispatching to invoke correct method based on the object type in runtime. Therefore, to support dynamic update in the case of inherited and overriding method, altering of dynamic method dispatch is required. Altered dispatch is handled by *DSU manager* with algorithm given in Algorithm 7. Altered dispatch is performed in *DSU manager*, where methods detected as overriding methods in Chapter 3, are invoked by *DSU manager*. The algorithm first searches for *diff* class and the corresponding method according to method name, number and type of parameters. If the corresponding *diff* class does not exist or the method is not defined in the *diff* class, then the method is searched in the object class. At the end of each step, for the next search step, the parent of the current class is selected. This process is iterative until the method is found.

For Example 5.8, current classes and *diff* class are shown in Figure 5.9. In the example, the algorithm for altered dynamic dispatch retrieves the C_{diff} class and then method $m_2()$ defined in the *diff* class. Meanwhile, if the method $m_2()$ is only defined in class A, then the algorithm iteratively searches in the *diff* and classes of C and B in the running version, until method $m_2()$ is finally found in class A_{diff}.

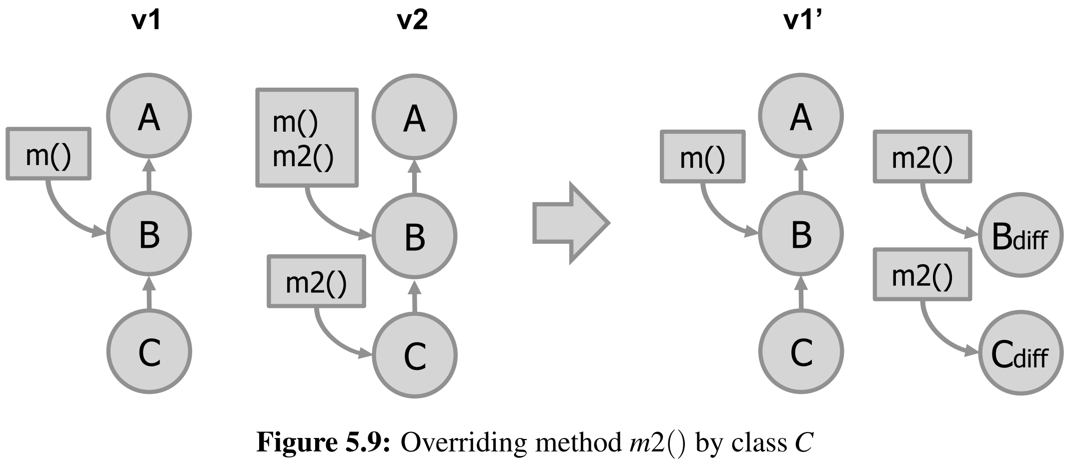


Figure 5.9: Overriding method $m2()$ by class C

Algorithm 7: Altered dynamic dispatch procedure

```
input : Current object ($obj$), method $name$ and arguments ($args$)
$m \leftarrow \varnothing$;
$invObj \leftarrow obj$;
$par \leftarrow$ getPars($args$);
for $c \leftarrow$ getClass($obj$); $c \neq \varnothing \land m \neq \varnothing$; $c \leftarrow$ getParentClass($c$) do
    $d \leftarrow$ getDiff($c$);
    if $d \neq \varnothing$ then
        $m \leftarrow$ getMethod($d$, $name$, $par$);
    end
    if $m \neq \varnothing$ then
        if isDeleted($m$) then
            continue;
        end
        $invObj \leftarrow$ diff($obj$, $c$, $d$);
    else
        $m \leftarrow$ getMethod($c$, $name$, $par$);
    end
end
if $m \neq \varnothing$ then
    invoke($invObj$, $m$, $args$)
end
```

Fields

Field changes regarding dynamic update are reflected through *diff* and *dynamic aspect* classes similar to the changes in methods. Added and type changed fields are added to the corresponding *diff* class as members. In Example 5.9, class B method $m1()$ in v_2 use field f_2 which does not exist in v_1. To support dynamic update, *diff* class B_{diff} is created, containing field f_2 as a member, and *dynamic aspect* class B_{asp} replaces the method $m1()$ to use the field $f2$ defined in B_{diff}.

Meanwhile, fields are inherited in the class inheritance tree. As described in Chapter 3, if the accessible field is changed in predecessor class it is changed for class inheriting from predecessor class. Similar to the methods, objects of descendant classes are paired with *diff* objects of predecessor classes containing changed field. For example, if the class B from Example 5.9 is inherited by class C, added field f_2 is added to class C by class B_{diff}. Figure 5.10 shows the case where method $m2()$ defined in class C uses the field f_2 defined in class B. Therefore, class C_{asp} (Example 5.10) is created to replace $m2()$ body in order to use $f2$ defined in B_{diff}. *DSU manager* resolves pairing of C and B_{diff} objects. Moreover, if the class B in v_2 contains added methods, then class C inherits both methods and fields over the same B_{diff} class.

Deleted fields or fields with changed type are set to *null* value for reference types. Dynamic

Example 5.9: Added field inheritance example

```
/*      v1      */              /*      v2      */
public class B {                public class B {
    int f1;                         int f1, f2;

    public String m1() {             public String m1() {
        return "B.m1()" + f1;            return "B.m1()" + f2;
    }                               }
}

public class C extends B {      public class C extends B {
    public String m2() {             public String m2() {
        return "C.m2()" + f1;            return "C.m2()" + f2;
    }                               }
}                               }
```

Example 5.10: *Dynamic aspect* class for class *C* in Figure 5.10

```
public class CDynamicAspect extends DefaultAspect {
    public Crosscut c0 = new MethodRedefineCut(){

        public String METHOD_ARGS(C target) {
            return "C.m2() " + ((BDiff)DSUManager.Diff((Object)target).f2;
        }

        protected PointCutter pointCutter() {
            return Within.method("m2").AND(Within.type("C"));
        }
    };
}
```

aspects to intercept read or write to such fields can be created to raise an exception. However, it is expected that statements using such fields in v_2 are changed with corresponding *dynamic aspect*, *diff* and *dynamic* classes, depending on the context.

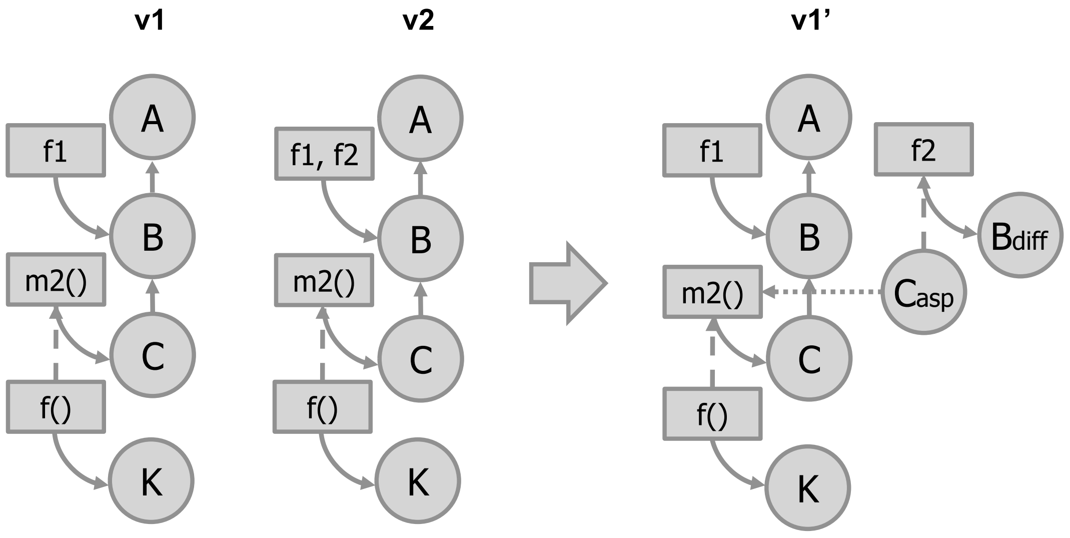


Figure 5.10: Added field inheritance example

State transfer and initialization

In Section 3.2, the method and constructor changes are classified as behavior changes and fields changes as state changes. If the class in v_2 changes type relationship, as described in Subsection 5.4.1, then the corresponding *dynamic* class is created. Therefore, it is necessary to transfer the state from the existing object to the object as an instance of *dynamic* class. If the class determined as *dynamic* class does not contain any changes between fields in two program versions, then the state transfer copies the field values from the existing object to *dynamic* object. However, if the class contains field changes in v_2, then a value copy is not possible. Furthermore, the subsection *Fields* (20) shows how field changes result in the creation of a *diff* class to support the dynamic field changes. The state of changed fields in *diff* and fields in *dynamic* classes should be initialized, otherwise (as aforementioned) runtime phenomena could occur. For example, for integer type, default value of 0, if used in the calculation could introduce divide by zero error, or for reference type uninitialized object, null reference error. Instead of a straightforward value copy, the most common solution is to use default values or provide programmer defined transfer function. However, the former may introduce runtime errors and the latter reduces programmer transparency requirement described in Chapter 2. To reduce errors and programmer involvement, the algorithm to initialize changed fields is presented in Algorithm 8.

To set the field value, an algorithm search for field initialization in class from v_2. If the initialization exists in both v_1 and v_2, then it is used to initialize the field, otherwise the default value for specific type is used (e.g. for integer value of 0). Meanwhile, for the reference type

Algorithm 8: Field initialization procedure

```
input  : Field $f_1$ defined in $v_1$, and $f_2$ defined in $v_2$, initializations inits in class for field
         $f_2$ defined in $v_2$
output: Field initialization init
init $\leftarrow \varnothing$;
if isValueType($f_2$) then
    if getInit(inits, $f_2$) $\neq \varnothing$ then
        init $\leftarrow$ getInit(inits, $f_2$);
    else
        init $\leftarrow$ getDefaultValue($f_2$);
    end
else
    const $\leftarrow$ getDefConstr($f_2$);
    if const $\neq \varnothing$ then
        init $\leftarrow$ getDefConstr($f_2$);
    else
        sortInitsByMatchingPars($f_1$, $f_2$, inits);
        foreach i in inits do
            if matched(i, $f_1$, $f_2$) then
                init $\leftarrow$ i;
                break;
            end
        end
        foreach i in inits do
            if canAddDefVal(i, $f_1$, $f_2$) then
                init $\leftarrow$ addDefValues(i, $f_1$, $f_2$);
                break;
            end
        end
    end
    if init $= \varnothing$ then
        init $\leftarrow$ null;
    end
    return init;
end
```

field, if any, the default constructor is used to initialize the object. In the case when there is no default constructor, the algorithm searches between constructors used in v_2. If there is a constructor invocation with fields that exist in v_1 and v_2, or constant values as arguments, then the field is initialized with such a constructor. Otherwise, the field is initialized with the constructor invoked with the largest number of arguments available, in the form of fields that exist in v_1 and constant or default values. If no constructor with defined criteria is found, then the object value is set to *null* value, where programmer should ensure proper field initialization in the constructor of the *diff* class.

Example 5.11: Added constructor example

```
 1  /*      v1      */               /*      v2      */
 2  public class C {                 public class C {
 3      int f1 = 1;                      int f1 = 10;
 4      public C() {}                    public C() {}
 5                                       public C(int i1) {
 6                                           f1 = i1;
 7                                       }
 8      public String m() {              public String m() {
 9          return "C.m()" + f1;             return "C.m()" + f1;
10      }                                }
11  }                                }
```

Constructors

Constructor body changes are similar to the method body changes. The constructor body change results in *dynamic aspect* class that replaces the body during execution. Statements of constructor defined in v_1 are replaced with statements defined in v_2. Meanwhile, the added constructor is similar to the added method, it is added to the corresponding *diff* class. Because the constructor cannot be defined in another class, the constructors in *diff* classes are defined as *diff* class constructors. Statements from the added constructor are copied to the *diff* constructor. In Chapter 3, it is described that in the class hierarchy a default constructor is implicitly invoked as the first statement. If the class does not contain a constructor definition, then a default constructor is implicitly created. Furthermore, the first statement in the constructor can be an invocation of another constructor defined in the class, or an invocation to super class constructor. Therefore, the first statement in the constructor defined in *diff* class initializes the target object. Consequently, there are several cases to consider: when the first statement is an invocation to the parent default constructor, another constructor defined within the same class, and invocation to the parent constructor.

In Example 5.11, a constructor with argument of integer type is added in v_2 for class C. C_{diff} class is created containing $C_{diff}(int)$ constructor. As the first statement, instead of the implicit invocation to the parent (class B) default constructor, *target* is initialized with the class C default constructor. The rest of the statements are a copy from new constructor body. In the case of class C from Example 5.11, single statement (line 6) is copied.

Figure 5.11 shows class hierarchy for Example 5.11, where constructor is added in class C. Class K is the client with method $m()$ that invokes the default constructor $C()$ in v_1, whereas the added constructor $C(int)$ is invoked in v_2. *Dynamic aspect* class K_{asp} is created to replace the method $K.m()$ body with a body containing the initialization of class C object with added constructor $C_{diff}(int)$.

Meanwhile, if the first statement in the added constructor is an invocation of another constructor within the same class, then there are two possible cases: an invoked constructor exists

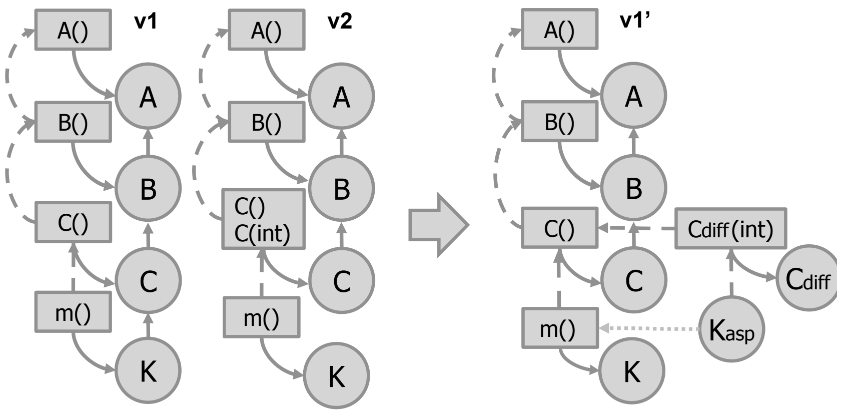


Figure 5.11: Added constructor example

in v_1 or it is added in v_2. For the former case, instead of an invocation of default constructor to initialize *target* object as in the previous example, *target* is initialized with the invoked constructor. For the latter case, *target* is initialized with the corresponding added constructor defined in *diff* class (C_{diff}). Therefore, the statement ($this()$) to invoke constructor within the same class remains unchanged.

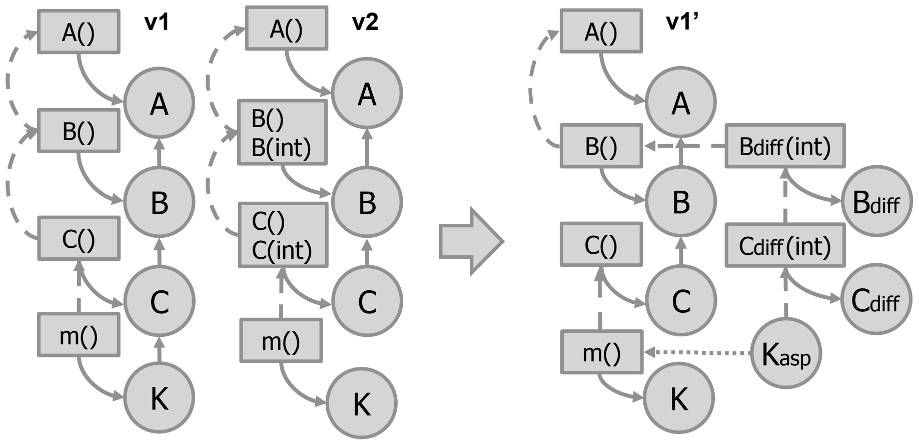


Figure 5.12: Added constructor inheritance example

Regarding inheritance, if the first statement in the added constructor is an invocation to the parent class constructor, then there are two cases identical to the case of another constructor invocation within the same class. The parent constructor can be defined in v_1 or added in v_2. If the parent constructor does not exist in v_1, then it is added to the parent *diff* class. The first statement is replaced by a statement initializing the *target* with the default constructor, and then the constructor defined in the parent *diff* class is invoked. Figure 5.12 shows an added constructor to classes *B* and *C* in v_2. As a result, classes B_{diff} and C_{diff} are created, where the constructor defined in C_{diff} invokes the constructor defined in B_{diff}.

Meanwhile, if the invoked parent class constructor exist in v_1, then it is not possible to invoke parent constructor, such as $B(int)$ from the $B_{diff}(int)$ constructor. Therefore *dynamic* class $B_{Dynamic}$ is created that contains the added constructor with invocation to the parent constructor.

Furthermore, any necessary *dynamic aspect* classes are created for *client* classes that use *B* class.

For deleted constructors, similar to the deleted methods, a *dynamic aspect* class is created to replace the constructor body with a statement to raise an exception on constructor execution. It is expected that statements from $v1$ that use deleted constructors are changed in v_2 with the corresponding dynamic classes (*dynamic aspect*, *diff*, and *dynamic*), depending on the context of change.

Chapter 6

Runtime phenomena detection

State artifacts in program execution that occur after the dynamic update are referred to as runtime phenomena. In this dissertation, the focus is placed on Object-Oriented languages; therefore, runtime phenomena caused by constructs of Object-Oriented languages are observed. In this chapter, runtime phenomena are described through identified changes between program versions. Furthermore, the characteristics of the update model presented in Chapter 5 related to runtime phenomena are analyzed. According to the update model, the focus in detecting runtime phenomena is placed on changes in the class hierarchy. Therefore, the TID algorithm presented in Chapter 4 is modified to detect and estimate the risk of these state artifacts.

6.1 Runtime phenomena

Runtime phenomena are an invalid state of the program after dynamic update that would not be in the update procedure with the restart [28, 89]. These program states are a side effect of the particular dynamic update procedure and environment features. For example, depending on how state transfer is handled or the used programming language. Because the focus of this dissertation is on Object-Oriented languages, the effect of the dynamic update procedure on the program state is related to the supported changes of the dynamic update model described in Chapter 5.

Figure 6.1 shows an example of a breakout game in [28] with dynamic change to the previous version. The runtime phenomenon that occur is the lost state because in the updated version on the right-hand side, the class describing the block in the game no longer contains the color and gift information that can be found in the initial version on the left-hand side.

In the related literature [3, 28, 89, 90], runtime phenomena are categorized by changes in program versions, such as adding or removing fields from class. Table 6.1 contains currently known runtime phenomena caused by program code changes.

After a class is removed in a running program, the existing objects of class remain as *phan-*

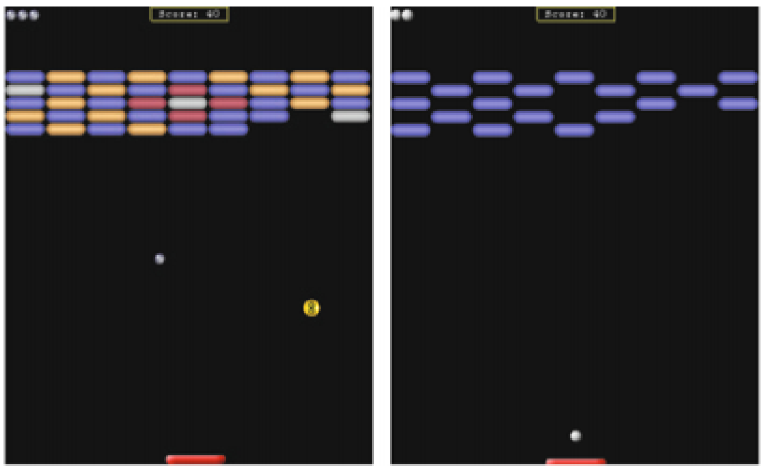


Figure 6.1: Runtime phenomena example (lost state and phantom objects) [28]

tom objects because they are not used in the updated version. *Phantom objects* can also occur when the class is modified to an abstract class or replaced with the interface. In general, *phantom objects* are objects that would not exist in the updated version. The *absent state* is related to the state that is missing in the updated version, when classes introduce new functionality. Occurs when added fields are not properly initialized during a dynamic update. Adding a class in the class hierarchy introduces an *absent state* when objects are compared by type. This comparison could fail because existing objects do not inherit the added class. Furthermore, for an added predecessor class, the introduced fields for existing objects can be uninitialized. Related to the Java programming language, when the static modifier is removed from the inner class, the inner class object belongs to the outer class object. In an updated version, field in inner class pointing to outer class can be uninitialized, introducing an *absent state*. The *absent state* is related to missing the state in the updated version, whereas the *lost state* is related to the missing state from the current version. The state is lost by changing the field type or when the field is removed from the class. When a field type is changed, the field's value in existing objects is lost due to the initialization of the new type's value to the default value. However, the *lost state* is related to the state transfer of the dynamic update system. By using the state transformation functions, *lost state* can be avoided, at least to some extent. Meanwhile, the *oblivious update* is related to the missing functionality introduced in the updated version, when the added fields are initialized in the changed constructor, or there is a change in static initialization of the class. It is a result of changed implementation of constructor or when static initialization cannot be executed for existing objects after the dynamic update because the objects are already initialized. *Broken assumptions* occur in the updated version as invalid dependency between state and method maintaining state of objects. For example, in [28] the field as a class member is depending on another constant field, where the constant value or method that maintains fields dependency changes. Furthermore, if the *broken assumptions* do not cause runtime exceptions, then an updated program is temporary in the runtime state that would not be reachable if the

Table 6.1: Runtime phenomena categorized by program changes

Runtime phenomena	Program code change
Phantom objects	Class removed Class renamed* Modifier abstract added to class Class replaced by interface
Absent state	Class added Predecessor of class changed Instance/static field added to class Removed static modifier from inner class
Lost state	Instance/static field type changed in class Class renamed* Instance/static field removed from class*
Oblivious update	Static initialization implementation changed in class Constructor body changed in class
Broken assumption	Static field value changed Instance/static method body changed*
Transient inconsistency	Instance/static method implementation changed*

* involve multiple runtime phenomena

program was started in the updated version. This is observed as a *transient inconsistency* runtime phenomena. *Transient inconsistency* is related to the program semantics changed by the modifications of methods. If such program state is not temporary, then the program is in an invalid state. Variants of the *transient inconsistency* are *transient state inconsistency* regarding the invalid state of fields and *delayed effect* when the update changes are not visible immediately.

Both *transient inconsistency* and *broken assumptions* are related to the program semantics, which needs to be determined by a detailed analysis of the programs behavior in both current and the updated version during the prolonged execution of the program. This makes such analysis extensive and challenging.

6.2 Runtime phenomena analysis in update model

Runtime phenomena can frequently occur with the long running applications after continuous updates, therefore detection and handling of such a state is required to achieve correctness DSU requirement from Chapter 2. Figure 6.2 shows part of DAOP model from Chapter 5. Runtime phenomena analysis is performed based on the detected changes between the program versions v_1 and v_2. To detect changes, an abstract data structure is created from the program code in each version. In addition to generating dynamic aspects, changes are used as input for runtime phenomena analysis. Runtime phenomena analysis is performed in the *Detect* component connected to the dynamic updating component (*update/state*). Based on the analysis, a decision is made to execute a dynamic update. If the analysis detects program changes that introduce runtime phenomena, update component can defer or reject the dynamic update or require the user's decision to proceed. Furthermore, analysis can be connected to the runtime environment to observe current program state; for example, to detect if the objects of the changed class exists that can cause the runtime phenomena. Therefore, in Figure 6.2 the arrow is shown from *update/state* to the *detect* analysis component.

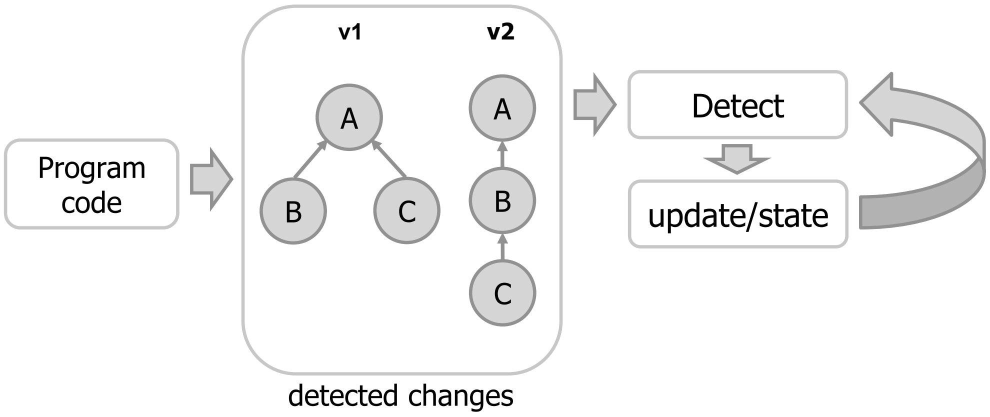


Figure 6.2: Runtime phenomena detection in dynamic update model

In Chapter 3, program changes are categorized as *basic* and *compound*. Runtime phenomena in the related literature are categorized over basic changes; however, the dynamic update model described in Chapter 5 uses dynamic aspects to apply *compound* changes. *Compound* changes are more suitable for runtime phenomena analysis because if the method is added but it is not called from any other method in the updated version, then it cannot create potential runtime phenomena states. Furthermore, in [28] the authors identify runtime phenomena according to code refactoring and proposes solutions for code changes to avoid runtime phenomena. In [89, 90], runtime phenomena are categorized by the results of queries executed during runtime, where the current program state is analyzed to detect runtime phenomena. To detect runtime phenomena by program changes between versions detected in extended DAOP model from Chapter 5, runtime phenomena are categorized by *compound* program changes and analyzed by changes dependency. Program changes dependency can be call and inheritance dependency. *Compound* changes are simplified variants of call dependency changes.

Table 6.2 shows *compound* changes from Chapter 3, and possible runtime phenomena which may result from these changes. *Method body change* depending on the changed program se-

Table 6.2: Possible runtime phenomena categorized by *compound* program changes

Compound change	Possible runtime phenomena
Added/Removed method + Method body change	*
Added field + Method body change	Absent state Lost state** Oblivious update *
Removed field + Method body change	Lost state** *
Added/Removed constructor + Method body change	Oblivious update *
Added class + Method body change	Oblivious update Absent state *
Removed class + Method body change	Phantom objects Lost state *

* Broken assumption and Transient inconsistency
** Type change

mantic and program state at the time of dynamic update, in general can result with *broken assumption* or *transient inconsistency*. *Added* or *deleted method* affects caller's method body, resulting in possible permanent (*broken assumption*) or temporary (*transient inconsistency*) invalid state of the objects. When a field is added in the updated version, in general this results in a missing state because the existing objects are already initialized and added field contains default values. An introduced field initialized with the dynamic update procedure can result in an *oblivious update*. Furthermore, combination of added and deleted fields, when field type is changed results in *lost state*. *Removed field* introduce lost state of the existing objects. Similar to the added method, an added constructor in classes of existing objects can result in improper object initialization, which would not occur in the updated version. An added class in the class hierarchy for existing objects results in *absent state* or *oblivious update*. Meanwhile, *removed class* implies *phantom objects* and *lost state*.

Because the *broken assumption* and *transient inconsistency* can occur as the result of changed program semantic, it requires analysis of changes in call dependency and the current state of the running program. Furthermore, changes in class hierarchy require analysis of changes in the inheritance relationship. The following two subsections describe changes in call dependency and inheritance.

6.2.1 Call changes dependency

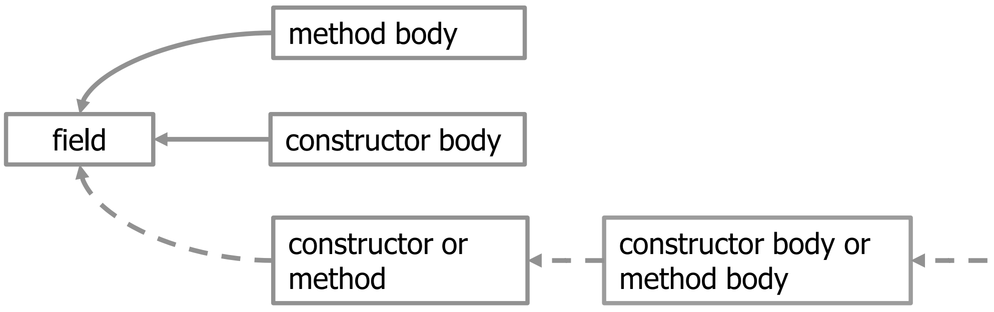


Figure 6.3: Program changes call dependency (slashed arrow – multiple instances)

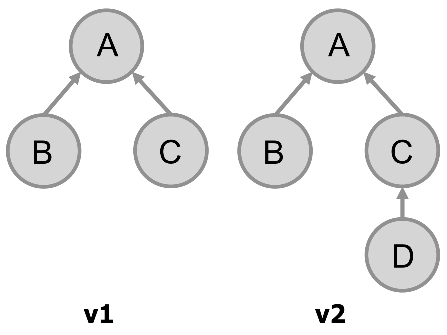
The impact of program changes on call dependency is shown in Figure 6.3. An added and deleted field affect the existing method or constructor body that uses the field. If the field is deleted, then the method or constructor body in the updated version does not contain statements with the deleted field. The opposite is found with an added field, where the method or constructor body is changed because the updated version contains statements with the added field. Field type and modifiers can be changed, which affect the body statements. Depending on the change, modifiers can produce similar effect as the deleting and adding field, whereas type changes may involve changes in statements using the field. Furthermore, the program can be changed by adding or deleting method, which affects method or constructor body that calls method. Similar to the field, statements that contains calls to deleted method are deleted in the updated version, whereas for added method the call statements are added. Changes in modifiers of method and method return type are similar to field modifiers and type change. Adding and removing constructor corresponds to the adding and removing methods. Meanwhile, calls can be nested forming a chain of calls, slashed arrows in Figure 6.3 denote nested calls. As a result of the chain of calls and changes in the program, a chain of changes is formed. For example, when a field is added, a method that use added field is added, and then the added method can be called from another added or changed method or constructor. Constructor, method and field changes can form an atomic unit as as a changed class. However, program changes can occur in different class, where each changed member belongs to another class, forming dependency of program changes between classes.

Changes in call dependency can be analysed to detect chain of runtime phenomena and consequently changes in semantics required to detect phenomena, such as *broken assumptions* and *transient inconsistency*. In Figure 6.3 direct dependency changes are shown based on static analysis. However, an entire call stack can be analyzed to the root method in the running program.

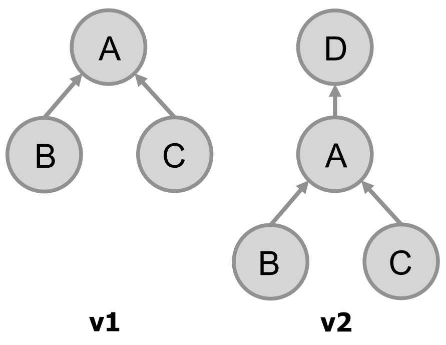
Consequently, it can be determined during program execution whether runtime phenomena will occur based on the current program state. For example, if the changed constructor is not executed in the running program, then a dynamic update can be performed, otherwise *oblivious update* can occur.

6.2.2 Inheritance change dependencies

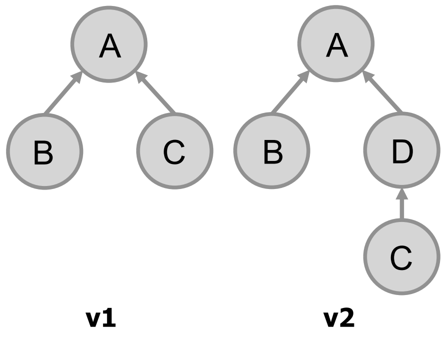
Inheritance change dependencies are a direct result of the changes in the class hierarchy. An added or removed class can impact other classes in the hierarchy through the inheritance relationship between classes. In addition to the added or removed class, changes in fields, methods and constructors of the matched classes can affect the descendant classes by inheritance. The inheritance relationship between classes and changes in inheritance results in runtime phenomena, which is analysed for each case.



(a) Class *D* added as child to class *C*



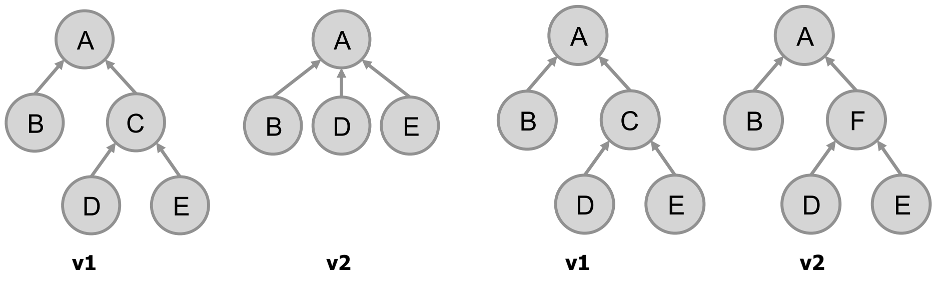
(b) Class *D* added as parent to class *A*



(c) Class *D* added as parent to class *C* and as child of class *A*

Figure 6.4: Examples of added classes in class hierarchy that cause runtime phenomena

Depending on where the class is added, removed or moved in the hierarchy, various runtime phenomena occur. Class can be added in the class hierarchy as descendant class and predecessor class. Figure 6.4 shows the addition of class *D* in the class hierarchy. Class *D* is added as only child class (Figure 6.4a). Existing objects of classes *A* and *C* created in the version v_1 are not of the same type as the objects created in the version v_2. Objects of class *A* and *C* created in the version v_2 are type comparable with objects of class *D*. The relationship between types is missing for the objects of classes *A* and *C* in v_1 to the class *D* added in v_2, therefore such objects are missing the proper state which corresponds to the *absent state*. The case where class D is added as a root class is shown in Figure 6.4b. Class D is the parent of the previous root class A. Objects of classes A, B and C created in v_1, are missing state and behavior introduced by class D in v_2. In general, existing objects of descendant classes for the newly added root class are missing introduced state in the form of class fields. Therefore, as a consequence of adding the predecessor class, an *absent state* occurs. Meanwhile, missing functionality is the result of the class C constructors that have not been executed, introducing the *oblivious update*. In Figure 6.4c, class D in v_2 is added as predecessor of class C and descendant of class A, replacing the class C as the child of class A. This example corresponds to the combination of basic examples shown in Figure 6.4a and Figure 6.4b. *Oblivious update* and *absent state* occur on objects of class C created in v_1 because class D constructor has not been executed and class D fields are uninitialized. Furthermore, *absent state* occurs on objects of class A created in v_1 because class D in v_1 is not descendant of class A. Consequently, existing objects of class A are missing the information for the type comparison with class D.



(a) Class C is deleted (b) Class F replaced class C

Figure 6.5: Examples of deleted classes in class hierarchy that cause runtime phenomena

When a class is deleted, the existing objects of the deleted class are *phantom objects*. However, if the class is deleted from the class hierarchy, objects of the deleted and descendant classes are *phantom objects*. Objects of classes that are descendants of the deleted predecessor class do not exist in v_2, therefore such objects are *phantom objects*. Figure 6.5a shows that class C has been deleted from the class hierarchy in v_2. Objects of class C in version v_2 are *phantom objects* together with objects of class D and E created in v_1. Meanwhile, in contrast to the *absent state*, when the predecessor class is added, deleted predecessor class introduce *lost state*. When converting the state of class D and E objects to the v_2, the current information in class C fields may be lost. However, if the existing objects of descendant classes are not converted, *broken assumption* may occur. For example, assuming that class C in v_1 contains fields and methods used by class D and E to maintain objects state in v_1, objects of class D and E in v_2 do not contain these members, which may break the expected state and logic dependency in the existing objects.

In Figure 6.5b class F in v_2 replaced class C in v_1, as a result of deleting class C and adding class F. As previously described, several runtime phenomena occurs on objects of classes with changed predecessor classes. Objects of class C, D, and E created in v_1 are *phantom objects*, because class C is deleted. Furthermore, as a result of adding class F, objects of class D and E created in v_1 are missing proper state of class F fields as constructor of class F is not executed, introducing *absent state* and *oblivious update*. Moreover, existing objects of class D and E depending on dynamic update procedure can introduce *lost state* for fields of class C or *broken assumption* because of changed initialization in v_2.

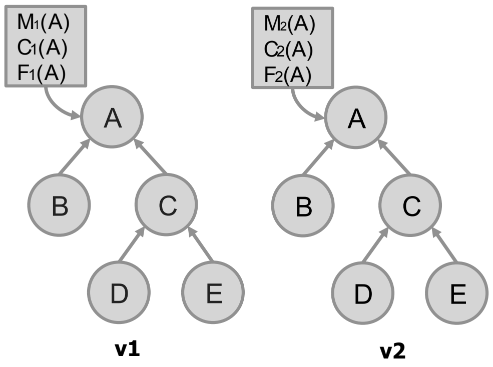


Figure 6.6: Runtime phenomena as the result of changed class members

Changes of class members affects descendant classes in the class hierarchy, which may result in runtime phenomena for the existing objects created in v_1. In Figure 6.6, a set of class A methods, constructors and fields is represented as $M_1(A)$, $C_1(A)$, and $F_1(A)$ in v_1 and $M_2(A)$, $C_2(A)$, and $F_2(A)$ in v_2, respectively. Changes between versions is denoted as difference between sets in v_1 and v_2, e.g. for methods $M_1(A) \neq M_2(A)$.

If method $m \in M_1(A) \cap M_2(A)$ body is changed, then the existing objects of descendant classes are also affected. As a result of the dynamic update, *broken assumption* and *transient inconsistency* can occur on existing objects of class and descendant classes, whether or not the method due to modifier is accessible by descendant classes. If the method is not accessible by the descendant class, then it can affect the fields of the changed class. Furthermore, if the method is used to initialize the fields, then it can introduce *oblivious update* for existing objects of changed class and descendant classes. In Figure 6.6 if method m changed body in v_2, classes (B, C, D, and E) that are descendant classes of class A are affected by the change. However, call dependency analysis is required to determine semantic change to detect such runtime phenomena. Similarly, adding method to predecessor class, depending on the semantics, can introduce *broken assumption* and *transient inconsistency* for existing objects of the changed class and descendant classes. For example, method m_{e2} of class E ($m_{e2} \in M_2(E)$) calls added method m_{a2} of class A, and m_{a2} changes the value of fields from A. Furthermore, deleting method can require changing caller methods body, resulting in runtime phenomena, which can be detected by call dependency analysis. For example, method $m_e \in M_1(E) \cap M_2(E)$ changes body because in v_1 it contained call to the method m_a of class A, deleted in v_2.

Changing the constructor body, such as for constructor $c \in C_1(A) \cap C_2(A)$ in Figure 6.6, introduces an *oblivious update*. It is similar to the changed body of method used for object initialization. Objects of class A and its descendant classes created in v_1 are missing initialization from v_2. Similar is found for adding a constructor $C_2(A)$ in v_2, existing objects are possibly missing initialization introduced in v_2. Deleting a constructor $C_1(A)$ can introduce *broken assumption* and *transient inconsistency* for existing objects of class A and its descendant classes. Similarly to the methods, adding the constructor and changing the constructor body can also introduce such runtime phenomena, which depends on the program semantics and can be detected by call dependency analysis.

Field changes introduce an *absent state* and *lost state* for descendant classes. In the case of adding and deleting a class in the class hierarchy, these runtime phenomena are related to type comparison. However, in the case of fields as described in Section 6.1 are related to the state of objects. For example, in Figure 6.6 field $f_{a2} \in F_2(A)$ added in v_2 ($f_{a2} \notin F_1(A)$) is missing state for objects of class A created in v_1, and for objects of class A descendant classes B, C, D, and E. Meanwhile, by deleting the field $f_{a1} \in F_1(A)$ in v_2 introduces a lost state for objects of class A and objects of descendant classes created in v_1. Existing objects contain information that can be lost in the v_2 because deleted fields may be replaced with added fields initialized with default values, rather than by copying values from the deleted fields. Furthermore, field type change can be observed as delete and add field change, which introduces *absent state* and *lost state* in the descendant classes.

6.3 Runtime phenomena detection algorithm

To detect possible runtime phenomena, based on the static analysis of changes in the class hierarchy, Algorithm 2 is adapted. In the runtime phenomena usage scenario, instead of calculating the cost of transforming from one class tree to another, it is necessary to estimate the risk of a runtime phenomena. Using runtime phenomena estimation, a decision can be made on whether or not to proceed with the dynamic update. This can be used as a guide for dynamic update operator.

The algorithm to estimate runtime phenomena occurrence is given in Algorithm 9. This algorithm is based on Algorithm 2 and Algorithm 4. The algorithm detects changes in predecessor classes C_p related to the runtime phenomena, according to the cases described in the previous section. Furthermore, identical to Algorithm 4, it detects changes in class members C_m for the matched class by using procedure `detectClassChanges`.

Cost function δ_{RP} is used to estimate runtime phenomena based on the detected changes. The input data of the cost function are: class with runtime phenomena c, predecessor classes with changes associated with runtime phenomena C_p for class c and changes of members C_m. The function implementation is arbitrary. The function can be implemented for different sce-

Algorithm 9: Runtime phenomena estimation algorithm

```
input  : class inheritance trees $T_1$ in $v_1$ and $T_2$ in $v_2$, cost function $\delta_{RP}$
output: runtime phenomena estimation $d_{RP}$ based on the cost function $\delta_{RP}$

$d_{RP} \leftarrow 0$;
foreach class c in $T_1$ do
    $C_p \leftarrow (\varnothing,\varnothing,\varnothing)$;
    $C_m \leftarrow \varnothing$;
    $P_{\varepsilon 1} \leftarrow$ getPreds($T_1$, $c$);
    if contains($T_2$, $c$) then
        $P_{\varepsilon 2} \leftarrow$ getPreds($T_2$, $c$);
        $C_p \leftarrow$ detectRPInheritanceChanges($c$, $P_{\varepsilon 1}$, $P_{\varepsilon 2}$, $T_1$, $T_2$);
        $C_m \leftarrow$ detectClassChanges($c$, $T_1$, $T_2$);
    else
        $C_p \leftarrow (\varnothing,P_{\varepsilon 1},\varnothing)$;
    end
    if $C_m \neq \varnothing \vee C_p \neq (\varnothing,\varnothing,\varnothing)$ then
        $d_{RP} \leftarrow d_{RP} + \delta_{RP}(c,C_p,C_m)$;
    end
end

foreach class c in $T_2$ do
    $P_{\varepsilon 2} \leftarrow$ getPreds($T_2$, $c$);
    if not contains($T_1$, $c$) then
        $C_p \leftarrow (P_{\varepsilon 2},\varnothing,\varnothing)$;
        $d_{RP} \leftarrow d_{RP} + \delta_{RP}(c,C_p,\varnothing)$;
    end
end
```

narios, where the emphasis of cost can be on specific runtime phenomena or changes in the relationship between classes, therefore it is set as the algorithm input. For example, runtime phenomena *absent* and *lost* state that occur due to field changes can be calculated based on the number of changed fields. Furthermore, the cost function can differentiate runtime phenomena related to class local and inherited changes. Inherited changes may be considered less significant if the changed member is less accessible by a restrictive modifier (e.g. `private`).

The algorithm uses several procedures, which are described as follows. Procedure `detect-ClassChanges` returns class member changes as a tuple (C_m). The first value in the tuple is set of changed methods and constructors, whereas the second, set of changed fields. While the `detectRPInheritanceChanges` procedure returns the inherited changes C_p as a triplet (3-tuple), which contains added, deleted, and moved predecessor classes. Added and deleted predecessor classes, detected at lines 11 and 20 in Algorithm 9, are used to estimate possible *absent state* due to changed type relationship between classes. Meanwhile, at line 8 the detected added, deleted and moved predecessor classes are related to estimation of *absent state* related to fields, *oblivious update*, *lost state* and *phantom objects*. In this dissertation call dependency

Algorithm 10: Procedure detectRPInheritanceChanges(c, $P_{\varepsilon1}$, $P_{\varepsilon2}$, T_1, T_2)

```
procedure detectRPInheritanceChanges($c$, $P_{\varepsilon1}$, $P_{\varepsilon2}$, $T_1$, $T_2$)
    $(C_a, C_d, C_m) \leftarrow (\varnothing, \varnothing, \varnothing)$;
    foreach class c in $P_{\varepsilon1}$ do
        if contains($P_{\varepsilon2}$, $c$) then
            if detectClassChanges($c$, $T_1$, $T_2$) then
                $C_m \leftarrow C_m \cup \{c\}$;
            end
        else
            $C_a \leftarrow C_a \cup \{c\}$;
        end
    end
    foreach class c in $P_{\varepsilon2}$ do
        if not contains($P_{\varepsilon1}$, $c$) then
            $C_d \leftarrow C_d \cup \{c\}$;
        end
    end
    return $(C_a, C_d, C_m)$;
```

analysis is not performed, and therefore *broken assumptions* and *transient inconsistency* are not considered. Furthermore, procedure `getPreds` returns predecessors and corresponds to the procedure used in the TID algorithm (Algorithm 2), with a difference that classes are used instead of nodes.

Procedure `detectRPInheritanceChanges` is used to detect changes in predecessor classes related to the runtime phenomena. The difference to the procedure `detectInheritanceChanges` in TID algorithm (Algorithm 3) is in the detection of moved classes. Regarding runtime phenomena in the inheritance relationship, as mentioned in Chapter 5 and the previous section, the distance between classes in the class hierarchy is not important as the changes in class members. Therefore, the *move* operation is detected as change in class members for predecessor class. Procedure `detectRPInheritanceChanges` is shown in Algorithm 10. The changes in added and deleted predecessors correspond to the procedure `detectInheritanceChanges`. However, to detect moved classes by changes in the class members, procedure `detectClassChanges` is used. The return value is triplet where the values are: set of added (C_a), deleted (C_d), and moved (C_m) predecessor classes.

As already mentioned, runtime phenomena are detected over changed predecessor classes and changed members. According to the runtime phenomena in the class hierarchy described in Section 6.2.2, added predecessor class can result in *absent state* and *oblivious update*. Meanwhile, the deleted predecessor class is found in *phantom objects* and the *lost state*. Changes in class members or members of predecessor class can cause *absent state* if the fields are added, *lost state* if the fields are deleted, and *oblivious update* in the case of constructors changes. This process to determine runtime phenomena can be used in the δ_{RP} function, and it can also be used to determine classes with risk of runtime phenomena.

The algorithm to detect classes in the class hierarchy classified by the possible runtime phenomena is shown in Algorithm 11. Changes in predecessors are obtained by the procedure `detectRPInheritanceChanges` as triplet (P_a, P_d, P_m). If there is deleted predecessor for the class c, then it is added as the *phantom object* class to P_h at line 11. In contrary, if a class is added as predecessor to the class c, it is added as the *oblivious update* class to O_b. Similarly, if class c or its predecessor class from P_m contains constructors changes, then class c is added to O_b. Furthermore, class c is added as the *absent state* class to A_{sf}, if the predecessor class is added ($P_a \neq \emptyset$). The same is true for for cases when the field is added to class c (set C_m) or to the predecessor class from P_m (line 17). Classes are added as *lost state* classes L_s if predecessor is deleted, or a field is removed from classes c or P_m. Classes related to the *absent state* for type comparison are straightforwardly detected as added and deleted predecessor classes of class c at lines 29 and 23. As a result, Algorithm 11 provides a classification of classes involved in possible runtime phenomena after the dynamic update. To detect a particular runtime phenomenon, the algorithm can be enhanced by additional constraints to classify phenomena. For example, detection of *absent state* can be constrained by the existence of a class field, and *lost state* by detecting a change in field type.

6.4 Discussion of runtime phenomena in extended DAOP update model

In the introduction to runtime phenomena, Section 6.2.2 describes that runtime phenomena depends on the dynamic update procedure. For example, in the presented DAOP model in Subsection 5.4.2 in Chapter 5, state transfer procedure copies field values from the object in the current program version to the updated version. The described procedure avoids the occurrence of *lost state* phenomena in cases when *dynamic* class is used. Furthermore, the state transfer procedure provides the initialization of changed field with the current field value or constant value. Initialization of the added field with constant value can avoid an *absent state* for fields. Similarly, initialization of field with a changed type (e.g. integer to string) using the current value can prevent a *lost state*. Moreover, if the constructors are changed to accommodate initialization of the added field and the field value is correctly initialized by the state transfer procedure, then an *oblivious update* is prevented. By introducing *dynamic* classes to handle type relationship changes in the class hierarchy, an *absent state* regarding type changes is avoided. Furthermore, *dynamic aspect* classes are created to replace statements that contain type comparison, therefore in these cases an *absent state* regarding type relationship cannot occur. *Phantom objects* are handled by DAOP model with the *DSU manager* and virtual machine (VM). DSU manager

Algorithm 11: Runtime phenomena classes detection algorithm

```
input  : class inheritance trees $T_1$ in $v_1$ and $T_2$ in $v_2$
output: classes with changes that results in possible runtime phenomena, $P_h$ - phantom,
        $O_b$ - oblivious, $A_{st}$ - absent state type, $A_{sf}$ - absent state fields, $L_s$ - lost state
$P_h, O_b, A_{st}, A_{sf}, L_s \leftarrow \varnothing$;
foreach class $c$ in $T_1$ do
    $P_a, P_d, P_m \leftarrow \varnothing$;
    $C_m \leftarrow \varnothing$;
    $P_{\varepsilon 1} \leftarrow$ getPreds($T_1$, $c$);
    if contains($T_2$, $c$) then
        $P_{\varepsilon 2} \leftarrow$ getPreds($T_2$, $c$);
        $(P_a, P_d, P_m) \leftarrow$ detectRPInheritanceChanges($c$, $P_{\varepsilon 1}$, $P_{\varepsilon 2}$);
        $C_m \leftarrow$ detectClassChanges($c$, $T_1$, $T_2$);
        if $P_d \neq \varnothing$ then
            $P_h \leftarrow P_h \cup \{c\}$;
        end
        if $P_a \neq \varnothing \vee$ constrChange($P_m$) $\vee$ constrChange($C_m$) then
            $O_b \leftarrow O_b \cup \{c\}$;
        end
        if $P_a \neq \varnothing \vee$ addFieldChange($P_m$) $\vee$ addFieldChange($C_m$) then
            $A_{sf} \leftarrow A_{sf} \cup \{c\}$;
        end
        if $P_d \neq \varnothing \vee$ removeFieldChange($P_m$) $\vee$ removeFieldChange($C_m$) then
            $L_s \leftarrow L_s \cup \{c\}$;
        end
    else
        $A_{st} \leftarrow A_{st} \cup P_{\varepsilon 1}$;
    end
end
foreach class $c$ in $T_2$ do
    $P_{\varepsilon 2} \leftarrow$ getPreds($T_2$, $c$);
    if not contains($T_1$, $c$) then
        $A_{st} \leftarrow A_{st} \cup P_{\varepsilon 2}$;
    end
end
```

sets unused fields of reference type to *null* value. In the current prototype implementation, when the *dynamic* class objects are created. Any object from previous version that is not referenced, VM marks for garbage collection.

The presented model cannot handle cases that require the intervention of a programmer to transfer the state. For example, if the changed field is an array or collection of reference type. Because of runtime polymorphism, the automatic state transfer procedure cannot predict correct initialization in updated version. Objects contained in the array can be of different types that require runtime analysis to resolve objects with dynamic changes. Such cases requires higher risk in the estimation by appropriate cost function. Moreover, Algorithm 11 detects changes in classes with runtime phenomena related to field changes, whether one or multiple fields change. However, multiple field changes pose higher risk of runtime phenomena. Chapter 9 further discusses runtime phenomena detection regarding the presented update model and cost function δ_{PR} implementation for the model.

Chapter 7

Measurement methodology for performance benchmarking

Although there are many studies and solutions to various problems in dynamic software updating, there is a lack of research that compares various approaches concerning supported changes and demands on resources. This chapter describes the benchmark methodology regarding the computer resources to compare DSU approaches in Java. Based on the given methodology, the design of the benchmark tool is described.

7.1 Discussion

There are currently several approaches for Java programming language that use techniques which differ in dynamic updating logic [12, 15, 21, 22, 27]. Some of these approaches can be found in development environments [3, 11], which grows the acceptance of dynamic updating. However, there is a lack of comparison of different approaches. Existing approaches are categorized in this chapter and the impact of DSU on computer resources is measured in the Chapter 9.

Seifzadeh et al. in [29] introduce various environment independent evaluation metrics of the DSU, which are a generalization of certain metrics used in this dissertation, such as update timing and supported changes. However, in this dissertation the focus is on Java-based approaches because of the OO paradigm, which is additionally categorized by used mechanisms and necessary program adaptations.

In the related literature [3, 9, 14, 15, 27] the steady state is measured on a specific DSU approach, while it is necessary to compare various DSU approaches. In [9, 14, 15], modified JVM approaches are evaluated by comparing the duration of modified garbage collection, measured by microbenchmark tests that perform class field changes. In contrast, to compare and evaluate different approaches various tests are necessary. DAOP approaches, [21, 22, 27] evaluate dynamic aspect weaving and woven code performance, because they are dynamic aspect systems, where DSU requires the performance comparison of program before and after the dynamic changes.

In [91], the authors introduce a quantitative cost-benefit model to estimate gain when using the dynamic updating approaches compared to other updating techniques. This comparison is expressed by a single revenue value calculated based on the estimated model parameters. This estimation differs because the parameters significantly differ across various application domains.

7.2 Measurement methodology

Software performance in DSU environments is inherently lower when compared to environments without DSU functionality [29]. The time required to perform dynamic updates should be as minimal as possible as well as demands on resources before and after the update [1, 11, 12, 20]. To compare DSU approaches, focus is on several measurements: steady state overhead, time duration necessary to perform the dynamic update, modified state overhead, and impact on memory usage.

7.2.1 Steady state overhead

Dynamic updating logic should not affect program execution when there are no dynamic updates, in steady state. To measure the impact of dynamic updating logic on program execution in steady state, it is necessary to compare program execution time in the unmodified environment to execution time in an environment that enables dynamic updating; as found in [9, 14, 15, 15, 19, 27]. These environments differ only in dynamic updating logic. Execution overhead is calculated as the difference in percentage between execution time in an environment with and without dynamic updating.

7.2.2 Update duration

Dynamic update duration is one of the performance characteristics that is used in DSU evaluation [1, 11]. It consists of preparation time and time to perform an update. Preparation time, for example, can include the time waiting for a safe point to occur and the time to load class files into memory. Update duration may also include the time to restore the program to normal execution. In more detail, DSU approach to apply a dynamic update may discard JVM optimizations, such as previously optimized compiled code. Consequently, the program executes more slowly because JVM requires some time to adapt to a current update, as shown by Würthinger et al. in [15]. The duration of the performance-related transient state is affected not only by dynamic update implementation but also by internal JVM mechanisms (e.g. JIT, OSR). Therefore, it is complex to measure, and it is also out of the scope of this dissertation. Update duration is measured as time elapsed between the moment when the dynamic update is invoked and the moment when dynamic update logic resumes normal program execution, as shown in Figure 2.2.

7.2.3 Modified state overhead

DSU can introduce performance overhead in method executions after the dynamic update [3, 20]. Depending on the approach, overhead can be persistent or temporary. Overhead can be persistent in approaches with inserted code snippets, such as join-point activation in DAOP [20, 23] and proxy calls in JVM agents [3, 37]. Temporary execution performance degradation is a characteristic of some modified virtual machine approaches [9, 14, 15]. Optimizations are invalidated after the update and gradually recovered. As stated in the previous subsection, the time between invalidated optimization and recovery is denoted as transient time. To compare approaches by execution overhead, it is necessary to measure the execution time for the dynamically updated method and method updated with the restart procedure. Furthermore, two separate measurements are performed: short and long term. Short term execution refers to the first method call after the update. Long term execution involves multiple calls of the same method in a sequence. These two cases differ because VM perform optimizations in the case of methods with multiple calls.

7.2.4 Memory usage

Using a DSU can increase the risk of excessive memory usage, especially in approaches that allow different program version coexistence, such as [1, 9, 15]. Memory usage can be monitored before the dynamic update, during the dynamic update, immediately after the dynamic update and when modified code resumes execution. However, there is a lack of research which includes this type of measurements for Java approaches. To determine dynamic updating impact on the memory usage, the difference in memory usage is measured before and after the update. Furthermore, difference in memory usage can be measured running a method on updated and unmodified class.

7.3 Benchmark architecture

Although many dynamic updating approaches rely on the HotSwap mechanism interface [16, 21, 23, 27], a standard dynamic updating management interface across different approaches currently does not exist. Each approach has a specific implementation and interface. Figure

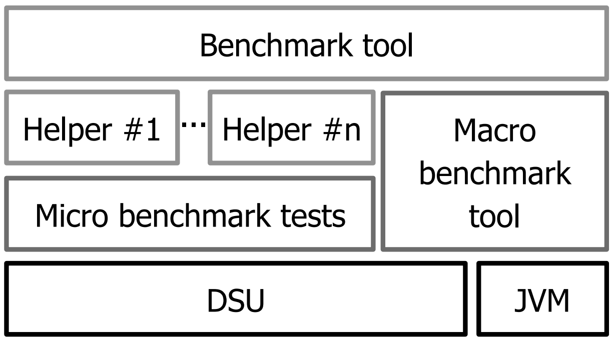


Figure 7.1: Benchmark architecture

7.1 shows a benchmark architecture that consists of several tools to perform performance evaluation. A benchmark tool is developed to perform micro and macro benchmark tests. Macro benchmark tests are part of existing benchmark tools, such as [92], which are usually used to evaluate system performance by typical software usage scenarios as they simulate real-world applications (e.g. scientific computation, text processing). In the context of DSU [3, 15, 22, 27], macrobenchmark tests are used to evaluate steady state overhead of DSU on program execution. Macrobenchmark tests are performed on the environment without DSU and the results are compared to the results of the environment with DSU support. Meanwhile, microbenchmark tests evaluate DSU implementation by various program change test cases. Evaluation is performed by measuring dynamic update duration, program execution speed before and after the dynamic update, and memory overhead. Furthermore, to perform measurements, additional helper tools for each approach are required. Helper tools are used as interfaces to invoke dynamic update on a specific approach, which contain microbenchmark tests that are supported by the approach and contain shared logic to perform measures. Microbenchmark tests are manually adapted for each approach to achieve the same program change across approaches. For example, DAOP approaches require program adjustments similar to those described in [24].

7.3.1 Benchmark tool

The benchmark tool that is developed to evaluate performance consists of components: *macrobenchmark*, *runner* and *result*. Each component implements interfaces to support various benchmarks and DSU approaches. The steady state measurement described in the previous section can be measured with existing macrobenchmark tools as standard performance evaluation tools. Therefore, a benchmark tool uses an interface `Macrobenchmark`, which each used macrobenchmark component should implement. The interface consists of `run` method that receives options to select tests that the macrobenchmark tool supports and options to adjust the testing environment (e.g. heap size). Meanwhile, microbenchmark tests, in contrast to the existing macrobenchmark tests, are used to evaluate DSU performance. The *runner* component shown in Figure 7.2 implements an interface with methods for executing selected microbenchmark

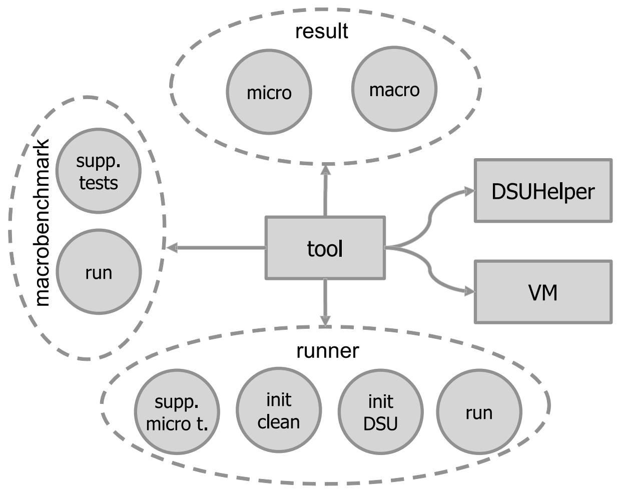


Figure 7.2: Benchmark tool components

tests on a particular approach. Component defines supported microbenchmark tests and initialize environment for the approach (e.g. Java version). Furthermore, method `run` in *runner* component, executes tests with the helper tool (*DSUHelper*). Helper tool is developed for the specific DSU approach, and is used as interface to approach. DSU approach performance is measured with helper tool, while VM performance is measured as a reference for comparison. Furthermore, the *result* component manages measured values and perform statistical calculation on the values, such as mean, standard deviation, error and confidence intervals.

7.3.2 DSU interfaces (helper tools)

To achieve “write once” tests for every tested approach, a universal format is required for the description of changes. However, the universal format does not currently exist. Therefore, for each tested DSU approach helper tool contains microbenchmark tests and interface to run selected tests. Helper tool is used by the *runner* component of benchmark tool as described in the previous subsection. Each test extends *RunTest* component (Figure 7.3), with functionality to perform measurement for DSU approach. *RunTest* component based on the arguments from the benchmark tool measures dynamic update duration, memory usage, and execution time of the test method. Furthermore, *RunTest* component provides a warmup phase by running the provided warmup method and by cleaning the memory before proceeding the measurement. Warmup methods are used to achieve state of the VM after the VM start up phase, when the execution of the program code is at an optimal level. Memory cleaning is performed to avoid the influence of garbage collection phase to measure only dynamic changes. To prevent the garbage collection phase being measured, the memory is cleaned by performing garbage collection before the change duration test. Therefore, *RunTest* component is connected to the running environment (i.e. VM), as shown by benchmark architecture in Figure 7.1. The *Runtest* component can run tests in multiple times on a single VM instance or by running a separate VM

Example 7.1: `Fact` test class example

```java
public class Fact implements TestClass {
    /** Recursive fact function implementation
    * @param n
    * @return
    */
    public int calculate(int n) {
        if(n <= 1) {
            return 1;
        } else {
            return calculate(n - 1) * n;
        }
    }

    @Override
    public Object warmupMethod(Object... args) {
        return this.calculate(args[0]);
    }


    @Override
    public Object measureMethod(Object... args) {
        return this.calculate(args[0]);
    }
}
```

instance for each iteration. To perform change from the initial to the modified version of the program code within the test, each test extends *RunTest* component by implementing `change` method. Furthermore, the tests contain test classes with changes in program code. The test class implements `TestClass` interface, which defines warmup method and method used for execution time measurement, before and after the update. An example of test class `Fact` is shown in Example 7.1.

7.3.3 Microbenchmark test cases

Microbenchmark tests are developed in accordance to the program changes classifications given in Chapter 3 and shown in Table 7.1. Manual adaptation of tests is performed for evaluated DSU approach, and tests are part of the corresponding helper tool. In extended DAOP approaches, as a consequence of client/supplier architecture, changes in class members that are not used in program code are neglected. Therefore, *basic* tests are omitted for DAOP. However, *basic* changes are supported by the presented benchmark tool because modified VM and Java agent detect such changes. The emphasis is on the *compound* changes, as the DSU approach presented in this dissertation is based on DAOP. For *compound* changes, microbenchmark test is created as a member change and method body change, where the method with changed body uses changed members. The created tests are based on changes in `Fact` class members. For example, *multiple* test (Example 7.2) consists of several changes: removed field (RF), removed constructor (RCo),

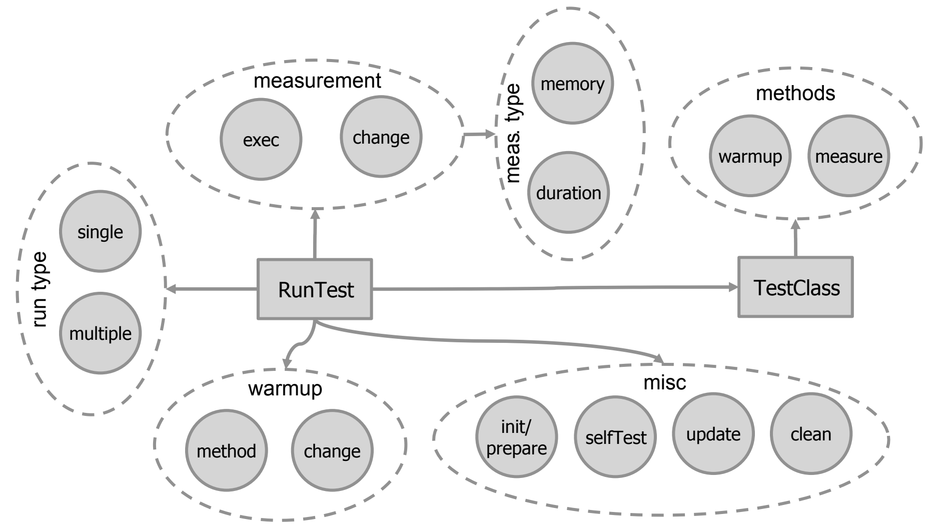


Figure 7.3: RunTest and ClassTest component

Table 7.1: Microbenchmark tests by classification

Type	Level	Test/Modification
Basic	1	Method body (MB)
	2	Add/Remove Method (AM/RM)
		Add/Remove Constructor (ACo/RCo)
		Add/Remove Field (AF/RF)
		Add/Remove Class (AC/RC)
	3	Add/Remove supertype (AS/RS)
		Add/Remove Interface (AI/RI)
Compound	2	AM/RM + MB
		AF/RF + MB
		ACo/RCo + MB
		AC/RC + MB
	3	Multiple

method body change (MB), added field (AF), added constructor (ACo), added method (AM), added class (AC), and hierarchy change as added supertype (AS). Such a change in one class is not likely in actual scenarios, but as a microbenchmark test, it can provide insight into how a particular DSU approach performs multiple changes.

In the *multiple* test case, class `Fact` changes the implementation of method `calculate` from recursive to iterative algorithm. Furthermore, fields `count` and `counter` are deleted together with method `getCallsCount`, which returns the value of the removed field `counter`. The abstract class `Algorithm` is added as a parent to class `Fact`containing the field `name`. Fur-

Example 7.2: Program changes for *Multiple* microbenchmark test

```
/*      original        */                    /*      modified        */
public class Fact {                           public class Fact extends Algorithm { /* (AS) */
    private int counter = 0;       /* (RF) */     private int result;               /* (AF) */
    private boolean count = false;

    public Fact(boolean count){    /* (RCo) */    public Fact(){                    /* (ACo) */
        this.count = count;                           this.name = "fact";
    }                                             }

    public int calculate(int n) {                 public int calculate(int n) {
        if(count) counter++;       /* (MB) */         int res = 1;                  /* (MB) */
                                                      if(n > 1) {
        if(n <= 1) {                                      for (int i = 1 ; i <= n; i++)
            return 1;                                         res *= i;
        } else {                                      }
            return calculate(n - 1) * n;              return result = res;
        }                                         }
    }
    public int getCallsCount(){    /* (RM) */     public void getLastResult(){      /* (AM) */
        return counter;                               return result;
    }                                             }
}                                             }

                                              public abstract class Algorithm{      /* (AC) */
                                                  protected String name;
                                              }
```

thermore, the constructor that uses the deleted fields is removed and a new constructor is added that initializes the inherited field `name`. The field (`result`) containing the last calculated result is added along with the method that returns the value of the field (`getLastResult`).

Chapter 8

Prototype system

The prototype system consists of several components implemented in two tools: offline tool and online. The offline tool as standalone application contains analysis of program versions, and a generator that produces classes to perform dynamic updates. Whereas, the online tool starts with a program that is dynamically updated and is responsible for loading changes and managing objects to perform dynamic updating. This chapter describes each component as a part of prototype system in relation to the process of the dynamic updating.

8.1 Prototype structure

The offline tool analyzes program versions source code and produce change specifications in the form of aspect and Java classes. Analysis is in the form of previously described algorithms for detecting changes between versions and for detecting runtime phenomena. The online tool is implemented as a Java agent. The online tool loads dynamic update classes, manages the update process and performs updates by dynamically inserting (i.e. weaving) aspects.

The steps of dynamic update process are described by Algorithm 12. The offline tool loads source code of currently running version v_1, and dynamically updated version v_2. Furthermore, the offline tool uses the Java compiler to compile source code and to access information about versions to create trees T_1 and T_2 representing class hierarchy in versions v_1 and v_2. Algorithms to detect program changes presented in Chapter 5 and runtime phenomena in Chapter 6 are executed on the created trees. The user performing the update is notified about the results of these algorithms. If the user approves the update, then the classes to perform the dynamic update are generated. After generating update classes, the online tool loads aspects and wove advice code at the required join point. Finally, the online tool manages the update process by performing object conversion to updated version and allowing access to restricted class members in the currently running version.

Figure 8.1 shows both the offline and online tool, and also the main components. The offline

Algorithm 12: Prototype system dynamic update process steps

```
1   load source code $v_1$ and $v_2$;
2   compile source code to get object code;
3   create class hierarchy trees $T_1$ and $T_2$;
4   detect hierarchy and class members differences;
5   detect runtime phenomena;
6   notify user and ask for proceeding;
7   if user approves update then
8       generate dynamic update classes;
9       perform update by DAOP;
10  end
```

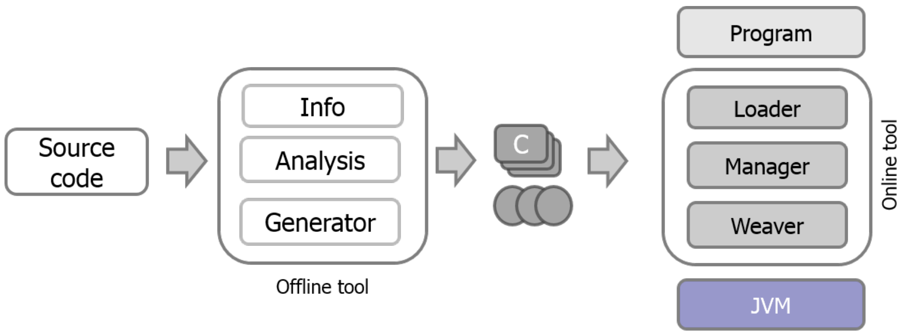


Figure 8.1: Prototype system tools

tool consists of: version *info* manager, program changes and runtime phenomena *analysis*, and dynamic update class *generator*. The online tool contains dynamic changes *loader*, dynamic update *manager* and dynamic *weaver*. These components are described in more detail in the following sections.

8.2 Creating version and source info

The *version manager* provides analysis of local and remote repositories. Remote repositories are obtained from online source code repositories, such as *GitHub*. The *version manager* determines the version details based on the release tag (e.g. whether version is major, minor or pre-release). The *version manager* enables evolution analysis by obtaining version information based on the version tag. In Chapter 9, it is used to compare versions in a sequence of releases or to compare major version and the corresponding minor versions and revisions.

The *version manager* contains information about the source for each version obtained by source code analysis. The process of creating source information in the *info* component is presented in Figure 8.2. The *version manager* obtains versions from local or remote repositories. It analyzes version tags and locates source files for each version. Retrieved source code is compiled using the Java compiler to create source code information based on the information about classes. Source code information consists of metadata objects about packages, classes, and class members (i.e. fields, methods, and constructors).

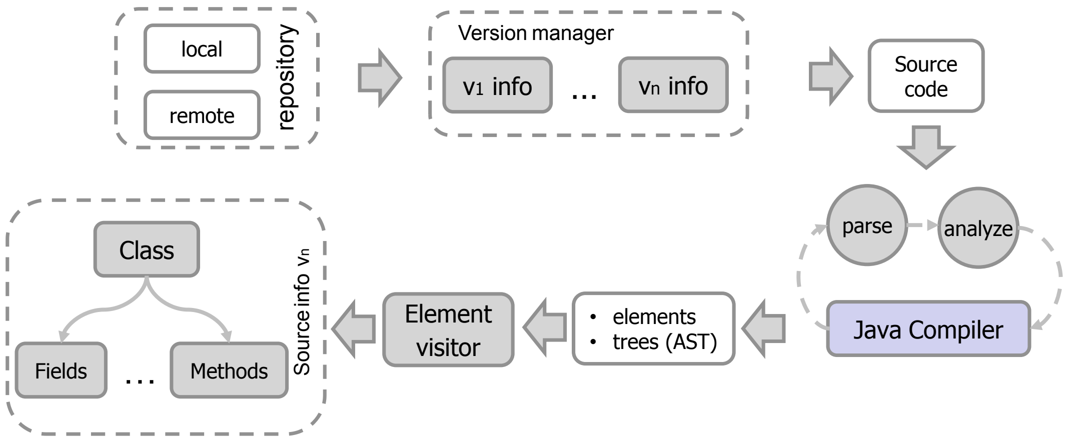


Figure 8.2: Version info manager and source info creation process (*info* component)

8.3 Source code analysis

Source code analysis is performed in different steps as part of the offline tool. Because the Java compiler API is used to obtain the version information and to generate classes, in this section the visitor pattern is described.

The Java compiler internally uses the visitor design pattern to generate the class files from the given source files. Two types of the visitor pattern can be used to analyze program from source through javac API. First, `ElementVisitor` (Figure 8.2) enables access to program elements, such as methods and constructors (also referred to as executables), packages, and variables. Second, `TreeVisitor` enables access to various program Abstract Syntax Trees (AST), such as blocks, loops, assignments. The first can be used for structural and the second can be used for intraprocedural and interprocedural analysis of the program. The *Version manager* in Figure 8.2 uses the `visitType` method in the element visitor to create information about classes for the program version. Element visitor is used to compare classes and class members as structural analysis, but it cannot be used to detect constructor or method body changes. Therefore, AST `MethodTree` is used to obtain executable statements, conforming to the intraprocedural analysis. Meanwhile, tree visitor is also used to generate classes for the dynamic update. In prototype implementation, tree visitor is used to extract detected changes as ASTs and to perform statement adjustments for dynamic update.

Tree visitor is used to generate classes instead of using the bytecode manipulation framework, such as ASM [93] or Javassist [94]. Because the Java compiler provide both element and visitor pattern, in a single invocation to the compile process, elements and trees (ASTs) can be obtained to perform both program analysis and adjustments. Consequently, when using the standard Java compiler there is no need for additional frameworks and program analysis. Adjustments are not performed on the bytecode but are instead performed on the AST as higher level of abstraction, which simplifies implementation. To create statements as AST, the `TreeMaker` class provided by Javac compiler is used and extended, similar to the *NetBeans* IDE [95].

8.4 Changes and runtime phenomena detection

The algorithms presented in Chapters 5 and 6 are implemented in the *analysis* component of the offline tool. As described, both algorithms as input receive class trees T_1 and T_2 representing two program versions. The class trees are created based on the source code information obtained in the *info* component. Information about the structure of the source code in the form of parent for each class is used to create a class hierarchy.

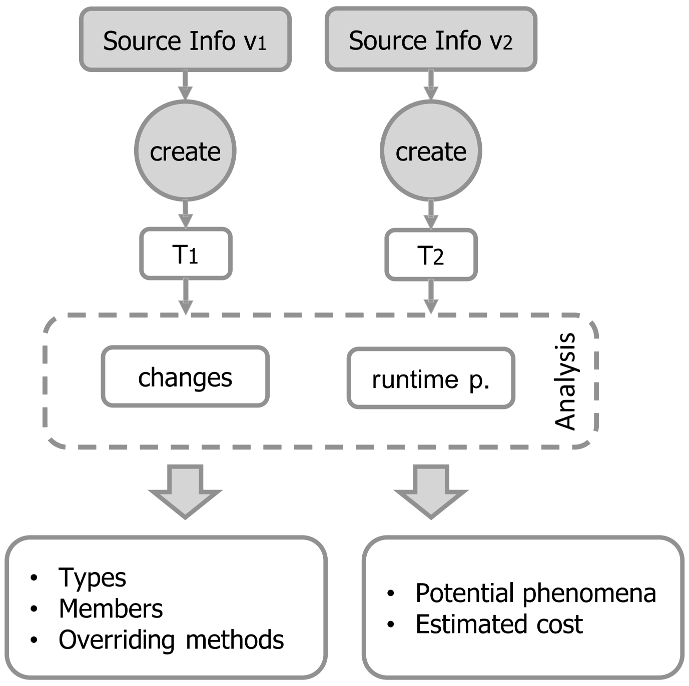


Figure 8.3: Analysis component process

An analysis process is shown in Figure 8.3. Based on the source information in v_1 and v_2, trees T_1 and T_2 are created as input for algorithms. Algorithm 4 detects changes between two program versions, and Algorithms 9 and 11 detect runtime phenomena. The changes detection algorithm results in a list of changes in: hierarchy of classes (i.e. type change), members and overriding methods. The produced lists are used as an input for the *class generator* component. The runtime phenomena algorithms estimate the runtime phenomena between two program versions and provide list of the potential runtime phenomena. The results of the runtime phenomena analysis are available to the dynamic update operator. Based on the given information, the operator can decide whether to proceed with dynamic update or to make additional adjustments. For example, the operator can make source code adjustments to the dynamic update classes and restart the analysis, or add appropriate state transfer logic based on the program semantics and analysis results.

8.5 Class generator

Chapter 5 describes how several classes are generated to enable dynamic updating, as follows: *dynamic*, *diff* and *dynamic aspect* classes. Generating classes relies on the standard Java compiler (*javac*) API. AST trees are used to modify statements and generate classes for update (Figure 8.4). The Java compiler consists of several phases, where in the parse phase, AST trees are generated from the input source code (Figure 8.2). However, to obtain statements in methods and constructors, it is necessary to obtain ASTs after the *analyze* phase.

Classes detected as type changed classes, as a result of Algorithm 4, are used to create ASTs of *dynamic* classes. AST of *dynamic* class is copy of AST of the changed class in v_2, where the class name is modified by adding the suffix "Dynamic". Created AST of *dynamic* class is added to the list for the *dynamic* class. Furthermore, classes detected as type changed classes, are used to detect statements in v_2 that use these classes as types. Every statement containing a changed class as type is changed to use *dynamic* class. Change is performed by the tree visitor, where on each visit the AST containing changed class is replaced by the AST containing introduced *dynamic* class. Furthermore, each class member containing changed class as type is added to the *diff* class. Therefore, methods and constructors with parameters of changed type result in adding their AST to the list for the *diff* class. Similarly, AST of fields containing changed classes as type are also added to the list for the *diff* class. Meanwhile, *dynamic aspect* classes are created to replace usage of these class members in the methods and constructors, conforming to the client-supplier pattern described in Chapter 5. Also statements with changed class as type in the constructor and method body are replaced by dynamic aspect. Therefore, ASTs of affected methods and constructors are added to the list for *dynamic aspect* classes.

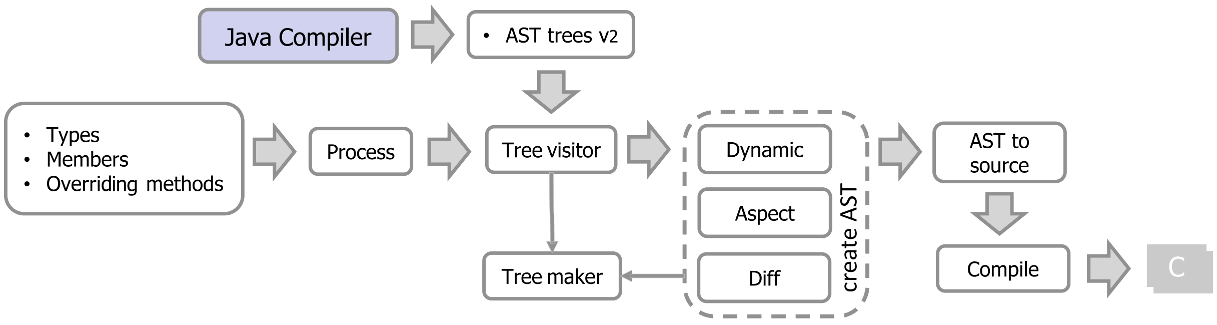


Figure 8.4: Generator component steps

Based on the detected changed class members as a result of Algorithm 4, methods with changed body are added to the aspect method list, whereas added and deleted methods are added to the list for the *diff* class list. Because constructors are detected as a special kind of method, the same applies to the constructors. Furthermore, added and deleted fields are added to the list for the *diff* classes.

Before creating AST for dynamic update classes based on the lists defined in the previous steps, it is necessary to process the statements in methods and constructors added for diff and *dynamic aspect* classes. Such statements are detected by *Tree visitor* (Figure 8.4) and modified by *Tree maker*. Statement processing copies and modifies statements that access class members to allow access within the created *diff* and *dynamic aspect* classes. There are two particular cases: when accessing local members and when the access modifier is restrictive. In the first case, statement is modified by replacing the keyword `this` with `target`, or by adding the keyword `target` if `this` is not used. In second case, the statement is replaced by the invocation to method provided by DSU manager to enable access to the restricted class member. For example, statement in the *diff* class accesses private members of the paired class. The method of DSU manager for accessing a restricted member is described in Section 8.6.2. Furthermore, overriding methods detected by Algorithm 4 are used to modify the invocation of such methods. Similar to the restricted member access, the invocation statement is replaced by the invocation to a method defined by the DSU manager, which handles the dynamic dispatch for changed overriding methods. The altered dynamic dispatch Algorithm 7 is described in Chapter 5 and the implementation is described in Section 8.6.2.

After the modification of statements it is necessary to generate dynamic update classes. The classes for dynamic update are created based on the ASTs lists and by using *Tree maker*. Then, ASTs of *dynamic*, *diff*, and *dynamic aspect* classes as compilation unit trees are converted to the source code. In the last step, the v_2 source code with the source code of generated classes is compiled to bytecode, creating the `.class` files that are loaded by the online tool described in Section 8.6.

8.6 Java agent

To instrument JVM and execute together with the program, the online tool is implemented as a Java agent. The online tool loads changes in the form of classes generated for dynamic update, activate dynamic aspects by using DAOP and manages updated program execution. These functionalities are implemented as components described in this section. Furthermore, in order to perform a dynamic update using the DAOP model, it is necessary to use a DAOP implementation. In this dissertation two implementations of the DAOP are used for prototype: Prose and dynamic weaver built in the online tool. The implemented dynamic weaver is intended for dynamic updating, while Prose is intended to support dynamic aspects and does not support constructor redefinition. Figure 8.5 shows the Java environment stack where online tool and DAOP as Java agents are on the top of the virtual machine. Therefore, presented architecture of online tool is flexible regarding used DAOP implementation.

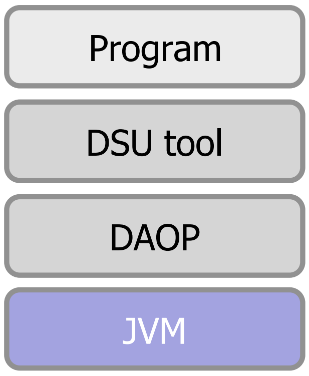


Figure 8.5: Online DSU tool and DAOP in Java environment stack

8.6.1 Loading changes

The online tool loads changes in the form of generated bytecode files (`.class`) by instantiating dynamic aspects. The online tool component for loading changes consists of a *file watcher*, *aspect manager* and *weaver*. The file watcher monitors the folder containing the running program class files, whereas the *aspect manager* manages changes by inserting or withdrawing dynamic aspects. To dynamically insert aspects, the *aspect manager* uses dynamic *weaver*.

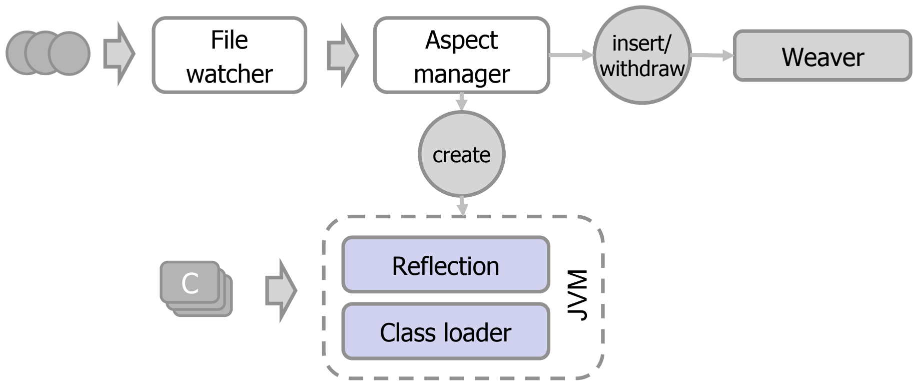


Figure 8.6: Dynamic aspects loading and weaving process

Figure 8.6 shows the process of loading changes. When dynamic aspect file is added to the monitored folder, the *file watcher* detects a change and invokes the *aspect manager*. The *aspect manager* instantiates a dynamic aspect class using the JVM reflection API, stores the aspect object in the register of created aspects, and then inserts the aspect into the running program using the *weaver*. In the case of a deleted dynamic aspect file, *aspect manager* retrieves the dynamic aspect object from the register and uses the *weaver* to withdraw the aspect from the running program. After inserting dynamic aspects, the statements in the aspect advices use dynamic update classes. These classes include generated *diff*, *dynamic* and standard Java classes that are defined in the updated program version. Therefore, when JVM executes statements that use these classes, the class loader in JVM loads the classes by searching the folder containing the running program.

8.6.2 DSU manager

Statements in the generated classes, as described in Section 8.5, contain invocation of methods defined in *DSU Manager*. The *DSU Manager* is implemented as a class containing static methods to provide dynamic changes of objects. Several methods can be used for executable members invocation and field access, a list is given in Example 8.3. A detailed functional description and algorithms implemented in these methods are described in Chapter 5.

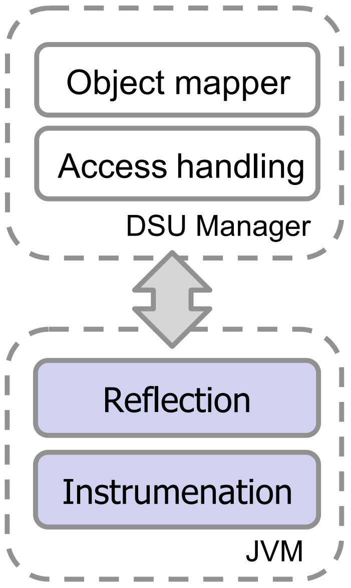


Figure 8.7: Dynamic object managing based on the Java instrumentation and reflection API

Figure 8.7 shows the two main components of the *DSU Manager*: *Object mapper* and *Access handling*. Both components in the implementation use methods provided by reflection and instrumentation API of JVM. *Object mapper* is used to pair objects of initially running version to the corresponding *diff* objects. When statements access class members defined in the *diff* class, then the *DSU Manager* on the first access instantiates a *diff* class object and stores the *diff* and initial version (v_1) object as a pair using the *Object mapper*. On subsequent access, the *DSU Manager* retrieves the corresponding *diff* object from *Object mapper*, based on the given initial version object. *Object mapper* is implemented in method `Diff`. In Example 8.1, `Diff` method is used to invoke the method `m()` from the updated version implemented in the *diff* class. Therefore, to invoke method `m()` on the existing object *obj*, `Diff` method is used to obtain the *diff* object. Furthermore, as part of the *Object mapper*, `copyState` is implemented as state transfer method; as described in Subsection 20. In Example 8.2, it is assumed that the `Board` class contains a field of type `Food`, which is a *dynamic* class. Therefore, according to the Section 8.5, the *diff* class `BoardDiff` is created. Method `copyState` is used to transfer the state when instantiating the *diff* object, by copying fields values from an existing object of the class `Food` to the object of *dynamic* class `FoodDynamic`.

Example 8.1: *DSU Manager* `Diff` method invocation usage example

```
/*   original   */    /*           dynamic           */
obj.m();              DSUManager.Diff((Object)obj).m();
```

Example 8.2: Use of method `copyState` in *diff* class to convert *dynamic* class field `food`

```
public BoardDiff(Board target) {
    this.target = target;
    this.food = new FoodDynamic(target, target.elementSize);
    Food oldfood = target.food;
    DSUManager.copyState(oldfood, food);
}
```

Access handling is implemented as methods providing access to fields and methods with restricted access modifiers and enables changes in the constructors and overriding methods. Regarding executable members, there are methods to invoke constructor, parent constructor, method, and overriding method. Handling constructor invocation is required for constructor changes when constructor is defined in the *diff* classes. Meanwhile, method invocation is used by statements in the *diff* and *dynamic aspect* classes that invoke methods with restricted access. Furthermore, changes regarding overriding methods are handled by the dynamic dispatch algorithm defined in Algorithm 7, implemented as a method in the *DSU Manager*. As aforementioned, invocation statements to the changed overriding methods are replaced as invocation to the dynamic dispatch method. Meanwhile, regarding access to the fields, two methods are provided for reading from (`getField`) and writing to (`setField`) a field. Therefore, statements in *diff* and *dynamic aspect* classes that read or write to a field with restricted access modifier, are replaced with the statement invoking methods `getField` and `setField`. Both methods for accessing restricted fields and methods (`invMethod`) are implemented by using method `setAccessible` from the reflection API.

Example 8.3: *DSU Manager* methods prototype

```
Object DiffConstr(Class objClass, Object ... args);
Object DiffConstrSuper(Object obj, Object ... args);
Object invMethod(Object target, String method, Object ... args);
Object invMethodOver(Object target, String method, Object ... args);
Object Diff(Object key, Class objClass, Class diffClass);
Diff(Object key);
Object getField(Object target, String field);
setField(Object target, String field, Object value);
void copyState(Object src, Object dst);
```

8.7 Dynamic aspect (DAOP) weaver

In the prototype system Prose is used as one of the implementations of DAOP. However, Prose does not support redefinition of constructor body. To provide dynamic updates of constructor body by Prose, in [24] constructor body is copied to special `init` method inserted in the initial program version (v_1). A built in dynamic weaver is implemented to support constructor changes without modification on the initial program version. Since a built-in dynamic weaver is intended for dynamic updating, both implementations are used in evaluation for comparison. However, Prose is considered for dynamic update cases without constructor modification.

To implement DSU based on dynamic aspects, *around* advice is used only; therefore, it is not necessary to support other advice types. Furthermore, because only the method and constructor bodies are replaced, DAOP for DSU can be considered as a single join point single advice DAOP. For example, in Figure 8.8, the CFG graph show how executing a statement with invocation of method `m()` is followed by executing the statements contained in the method body. In the case where multiple aspects define join-point for the same method, a dynamic aspect system is required to execute advice from the aspects in the correct order. In the case of DSU, only single aspect is processed to redefine body, which simplifies the DAOP implementation.

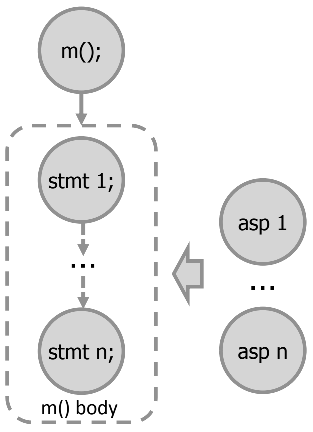


Figure 8.8: Multiple aspects applied on single join point example

For the builtin weaver, similar to the Prose [27], the *HotSwap* feature of the JVM is used. *HotSwap* enables redefinition of the running program method. Figure 8.9 shows the implemented weaver structure. Component *Interpreter* analyzes the dynamic aspect structure. Prose

API is used to determine join point from the crosscut defined in the Prose dynamic aspect. The first step is to determine the type of aspect: constructor or method. Then the Prose API is used to determine the join point and to extract the advice body as bytecode statements. Furthermore, to insert dynamic aspect, component *Transformer* is used. The *Transformer* component based on the determined join point replaces the body with the bytecode extracted from the advice. To replace the body, Javassist [94] bytecode manipulation framework is used. Finally, to perform weaving, the *Transformer* component redefines the changed classes by using the JVM instrumentation API.

The described weaver structure allows the use of another aspect language (e.g. AspectJ instead of Prose) if the appropriate library and aspect analysis is implemented. Furthermore, bytecode manipulation is not restricted to Javassist, solutions such as BCEL [96] or ASM [97] can also be used.

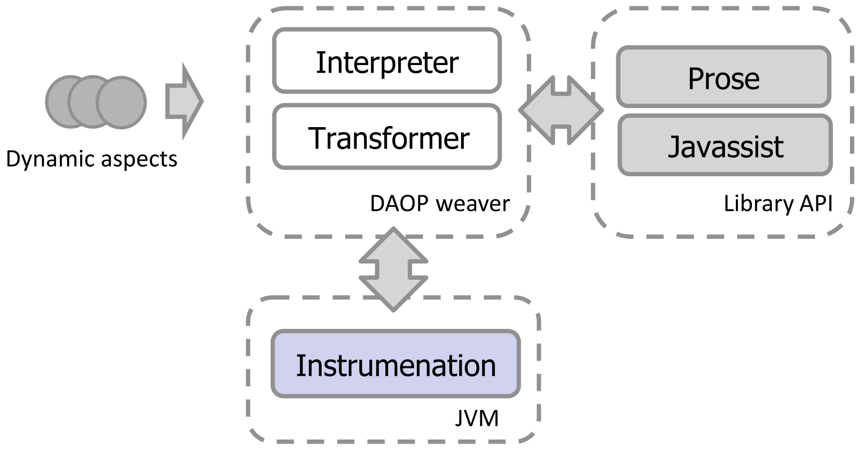


Figure 8.9: Weaver structure

Chapter 9

Evaluation

In this chapter the algorithms to detect program changes and runtime phenomena presented in Chapter 5 and Chapter 6 are evaluated according to efficiency and applicability. Efficiency is evaluated as a comparison of the algorithm's execution time on the generated trees, where the generated trees corresponds to the class trees with the distribution of nodes observed in several open-source programs. Applicability is demonstrated by an empirical study conducted with the algorithms on two open-source programs. Furthermore, the motivation to perform changes in production environment is demonstrated by the implementation of a simple game, and by examples of update versions. Meanwhile, measurement methodology is used on the various DSU approaches and prototype system for comparison. The results are discussed to determine whether the presented approach conforms to the main requirements of the DSU system described in Chapter 2.

9.1 Applicability

The ability of the algorithms to detect program changes and runtime phenomena is evaluated by an empirical study on the source code in versions of two open-source programs. Program changes are analyzed by Algorithm 4 and possible runtime phenomena are analyzed by Algorithm 11. Furthermore, prototype applicability is evaluated by the implemented *Snake* game with dynamic updates to two different versions.

9.1.1 Program changes analysis

Algorithms to detect program changes (Algorithm 4) and runtime phenomena (Algorithm 11) are executed on program versions of two publicly available Java programs: *Pinpoint*[*] and *NewPipe*[†]. *Pinpoint* is *Application Performance Management* (APM) tool with about 4000 classes

[*]https://github.com/pinpoint-apm/pinpoint
[†]https://github.com/TeamNewPipe/NewPipe

in the latest version [93]. *NewPipe* is a streaming Android application with about 350 classes in the latest version [63].

The class hierarchy of each version of the program is used as input for algorithms. Furthermore, for each program, there are two setups. In the first setup, algorithms are executed on trees created from subsequent program versions; for example, the first pair is the first and second version, the second pair is the second and third version, and so on until the last version pair. In the second experiment setup, each program version (i.e. revision) is compared to its corresponding minor version. For example, in *PinPoint* revision 1.7.3 is compared to minor version 1.7.0. Given that some minor versions are unavailable, corresponding prerelease versions are used as minor versions instead: release candidate (RC) in *Pinpoint*, and beta version in *NewPipe*. A comparison to the major version could not be made because the major versions are unavailable for the evaluated programs. Changes between program versions are detected in the form of class, member, and inherited member changes. The results are shown in the following tables. The first column in the tables denotes the program version. Changes analysis is performed on all currently available versions, but the results shown in the tables are from the last 14 versions for *Pinpoint* and the last 15 versions of *NewPipe*.

Table 9.1: Pinpoint class changes

Version	NOC	NOCH	added	deleted	matched	changed type	changed members
			p/m	p/m	p/m	p/m	p/m
1.7.0-RC1	3848	928	*/*	*/*	*/*	*/*	*/*
1.7.0-RC2	3854	931	7/*	1/*	3848/*	2/*	74/*
1.7.0	3854	931	0/*	0/*	3855/*	0/*	74/*
1.7.1	3854	931	0/0	0/0	3855/3855	0/0	66/66
1.7.2	3899	911	181/181	136/136	3719/3719	6/6	326/326
1.7.3-RC1	3901	909	4/183	2/136	3898/3719	2/7	68/328
1.7.3	3901	909	0/183	0/136	3902/3719	0/7	62/328
1.8.0-RC1	4339	959	522/701	84/216	3818/3639	10/16	453/630
1.8.0	4347	961	8/709	0/216	4340/3639	0/16	80/632
1.8.1-RC1	4563	970	386/386	170/170	4178/4179	17/17	406/406
1.8.1	4563	970	0/386	0/170	4564/4178	0/17	80/421
1.8.2-RC1	4567	968	6/390	2/170	4562/4178	2/17	90/428
1.8.2	4568	969	3/392	2/171	4566/4177	0/17	70/430
1.8.3	4569	969	2/394	1/172	4568/4176	0/17	68/432

* results from previous versions are excluded

Table 9.2: NewPipe class changes

Version	NOC	NOCH	added	deleted	matched	changed type	changed members
			p/m	p/m	p/m	p/m	p/m
0.13.0-b	280	170	*/*	*/*	*/*	*/*	*/*
0.13.1	280	170	0/0	0/0	281/281	0/0	32/32
0.13.2	287	174	9/9	2/2	279/279	0/0	48/51
0.13.3	274	162	49/58	62/64	226/217	0/0	45/53
0.13.4	272	163	1/59	3/67	272/214	0/0	39/60
0.13.5	271	163	0/59	1/68	272/213	0/0	34/65
0.13.6	276	166	5/64	0/68	272/213	2/2	58/70
0.13.7	276	166	0/64	0/68	277/213	0/2	24/70
0.14.0	296	180	23/87	3/71	274/210	0/2	56/77
0.14.1	296	18	0/0	0/0	297/297	0/0	25/25
0.14.2	303	187	7/7	0/0	297/297	0/0	91/91
0.15.0	350	207	59/66	12/12	292/285	4/4	44/91
0.15.1	349	206	4/4	5/5	346/346	0/0	38/38
0.16.0	359	215	10/14	0/5	350/346	0/0	45/57
0.16.1	359	215	0/0	0/0	360/360	0/0	26/58

* Results from previous versions are excluded

In Table 9.1 and Table 9.2, the number of classes is denoted as the total number of classes (NOC) and the number of classes in hierarchy (NOCH) with a depth greater than 1. NOC includes classes that explicitly do not inherit any other class, and neither they are inherited by any class. These classes introduce a new behavior into the program, but the behavior is not further propagated through the hierarchy. As these are single classes in terms of inheritance, they are in a hierarchy with a depth equal to 1. Meanwhile, classes that form a hierarchy with a depth greater than 1 introduce program behavior that propagates through the class hierarchy. Other columns in the tables, show the results of the comparison with the previous version; that is, revision and previous minor version (p/m). Minor versions that are used for comparison are highlighted in bold in the first column. The values as a result of the comparison are various changes in classes. Column *added* corresponds to the number of added classes in comparison to the previous versions. Similarly, columns *deleted*, *changed type*, *changed members* corresponds to the number of deleted, changed type, and classes with changed members. *Matched* is the number of matched classes that is classes that exist in both compared versions of the program.

Although *Pinpoint* is larger than *NewPipe* in the number of classes (NOC), the results for class changes shown in Table 9.1 and Table 9.2 are similar. There is a smaller number of changes between subsequent versions in the form of *added* and *deleted* classes. As expected, these changes are greater when the revision is compared with the minor version. It is noticeable that there is a smaller number of classes with a type change. Classes with type change exist in both compared versions (*matched*); however, there is a change in predecessor classes, which results in type changes between versions. Most changes in *matched* classes are related to member changes. Therefore, the approach presented in this dissertation in real-world scenarios would mostly generate *dynamic aspect* and *diff* classes with several *dynamic* classes to perform the dynamic update.

The class members changes are analyzed separately by the results shown in Table 9.3 and Table 9.4. The results are shown for subsequent program version changes because it is expected that a comparison with minor versions would result in a greater number of changes with a similar tendency as shown for the type change. Fields are analyzed by the number of added (A), deleted (D), and matched fields (M). In comparison to matched (i.e. unchanged) fields for both evaluated programs, the number of added fields is not large, besides in some cases of comparison with minor versions. Meanwhile, deleted fields occur rarely. In the case of *NewPipe*, there are no deleted fields between the shown subsequent versions. As described in Chapter 2, this scenario conforms to the binary compatibility resulting in a feasible dynamic update. Column *executables* includes the results of constructors and methods. Executables are analyzed by added (A), deleted (D), matched (M), and executables with body changes (B). Similar to the fields, the number of added executables is not large compared to the number of unchanged executables (M), besides in minor versions with an overall large number of changes. At the same time, there is a similar number of deleted executables. However, class member changes are mostly in executable bodies, which simplifies dynamic updating. Dynamic updates for body changes are performed with *dynamic aspect* classes, without other dynamic update classes.

Table 9.3: Pinpoint class members changes

Version	fields			executables			
	A	D	M	A	D	B	M
1.7.0-RC1	584	33	3263	847	627	1852	6704
1.7.0-RC2	16	0	193	4	3	122	560
1.7.0	9	0	205	0	0	124	611
1.7.1	9	0	170	0	0	116	527
1.7.2	174	1	1344	181	194	510	2432
1.7.3-RC1	10	0	174	1	0	121	535
1.7.3	9	0	156	0	0	113	499
1.8.0-RC1	352	22	1949	472	596	700	3341
1.8.0	20	0	203	11	15	140	584
1.8.1-RC1	215	16	1427	353	355	526	2746
1.8.1	11	0	237	7	3	130	643
1.8.2-RC1	35	0	234	12	5	140	611
1.8.2	13	0	280	13	2	116	655
1.8.3	15	0	181	38	0	110	545

* Results from previous versions are excluded

Table 9.4: NewPipe class members changes

Version	fields			executables			
	A	D	M	A	D	B	M
0.13.1	0	0	222	1	0	95	560
0.13.2	22	0	417	46	33	160	904
0.13.2	22	0	417	46	33	160	904
0.13.3	21	0	344	72	68	90	801
0.13.4	4	0	273	10	4	97	595
0.13.5	3	0	209	24	12	90	590
0.13.6	59	0	550	71	47	224	1103
0.13.7	0	0	165	0	0	77	424
0.14.0	42	0	472	44	19	153	1075
0.14.1	0	0	166	0	0	81	438
0.14.2	82	0	747	54	21	155	1363
0.15.0	62	0	328	53	32	132	806
0.15.1	7	0	301	10	5	129	610
0.16.0	12	0	337	21	13	139	833
0.16.1	0	0	163	0	0	97	457
0.16.1	13	0	504	34	23	144	1107

Table 9.5: Pinpoint inherited class members changes

Version	inh. fields		inh. executables		
	A	D	A	D	B
1.7.0-RC1	114	2	335	155	472
1.7.0-RC2	2	0	9	0	393
1.7.0	0	0	0	0	397
1.7.1	0	0	0	0	393
1.7.2	1	0	9	36	364
1.7.3-RC1	0	0	0	0	343
1.7.3	0	0	0	0	343
1.8.0-RC1	12	0	24	25	353
1.8.0	0	0	0	0	362
1.8.1-RC1	96	0	80	67	372
1.8.1	0	0	0	0	397
1.8.2-RC1	0	0	8	0	413
1.8.2	0	0	0	0	395
1.8.3	0	0	0	0	393

* Results from previous versions are excluded

Table 9.6: NewPipe inherited class members changes

Version	inh. fields		inh. executables		
	A	D	A	D	B
0.13.1	0	0	0	0	92
0.13.2	16	0	26	22	150
0.13.3	16	0	47	38	56
0.13.4	0	0	0	0	77
0.13.5	0	0	0	0	80
0.13.6	47	0	52	40	239
0.13.7	0	0	0	0	70
0.14.0	44	0	59	17	130
0.14.1	0	0	0	0	98
0.14.2	65	0	50	2	136
0.15.0	8	0	4	0	122
0.15.1	3	0	6	4	157
0.16.0	0	0	0	0	159
0.16.1	0	0	0	0	163

Since the focus in this dissertation is on inheritance, class changes are also analyzed through changes in the inherited member. The results are shown in Table 9.5 and Table 9.6. There are mostly no inherited member changes for *Pinpoint*, besides in some cases related to the comparison with minor versions with a large set of changes. The only significant changes are executable (i.e. method) body changes. However, the results for *NewPipe* as the program with fewer classes compared to the *Pinpoint*, show changes in inherited class members. In addition to body changes, the results also show the addition and deletion of fields and methods. These results show that changes in inherited member are related to the program domain and should be considered for dynamic updating.

The results of the two open-source programs show that the algorithms presented in Chapter 5 are applicable and useful to the analysis of program changes. Change analysis includes both class member changes and type (i.e. class hierarchy) changes. Detection of both type of changes is necessary to allow dynamic updates with arbitrary changes between program versions. Therefore, Algorithm 4 can be used to detect differences between program versions in a format that divides changes by type for applying dynamic updates.

9.1.2 Runtime phenomena detection

Runtime phenomena detection is evaluated by Algorithm 9 and Algorithm 11. These algorithms are executed on versions of *Pinpoint* and *NewPipe* open-source programs, as described in the previous subsection. The algorithm to detect runtime phenomena (Algorithm 11) is used to obtain total number of classes with possible runtime phenomena: phantom objects, oblivious update, absent state, and lost state. Possible runtime phenomena as the result of class changes are shown in Tables 9.7 and 9.8. The results include possible runtime phenomena on objects of class as a result of class member changes and changes in predecessor classes; as described in Chapter 6. For comparison, the results for runtime phenomena without the impact of inheritance are shown separately as an oblivious update, and absent and lost state on fields, which are denoted with suffix *(m)*. Meanwhile, the results of the algorithm to estimate runtime phenomena (Algorithm 9) are presented as a single value denoting the estimation as a cost calculated by the function `costRP`, implemented as shown in Algorithm 13. Function `costRP` corresponds to function δ_{RP} described in Chapter 6. As described in Chapter 6, the input for the function is the class with possible runtime phenomena c, changed predecessor classes C_p, and member changes C_m for class c. The estimation algorithm (Algorithm 9) is generic in the way that it can use any cost function with defined parameters. The cost function implemented as Algorithm 13 is appropriate for the prototype system in this dissertation. Runtime phenomena that can occur in the prototype are calculated according to changes in class members, with each change in class member equally contributing to the cost. However, various implementations can be used covering specific cases. For example, the case of changing a field that contains a collection of

Algorithm 13: Procedure `costRT`(c, C_p, C_m) as implementation of δ_{RP} cost function

```
procedure costRT($c$, $C_p$, $C_m$)
    $(C_a, C_d, C_{mp}) \leftarrow C_p$;
    $r \leftarrow 0$;
    foreach class $p_a$ in $C_a$ do
        $r \leftarrow r +$ getNoOfFields($p_a$);
        $r \leftarrow r +$ getNoOfConstructors($p_a$);
    end
    foreach class $p_d$ in $C_d$ do
        $r \leftarrow r +$ getNoOfFields($p_d$);
    end
    foreach class $p_m$ in $C_{mp}$ do
        $r \leftarrow r +$ getNoOfAddedFields($p_m$);
        $r \leftarrow r +$ getNoOfDeletedFields($p_m$);
        $r \leftarrow r +$ getNoOfBodyChangedConstructors($p_m$) +
          getNoOfAddedConstructors($p_m$) + getNoOfDeletedConstructors($p_m$);
    end
    foreach class $c_m$ in $C_m$ do
        $r \leftarrow r +$ getNoOfAddedFields($c_m$);
        $r \leftarrow r +$ getNoOfDeletedFields($c_m$);
        $r \leftarrow r +$ getNoOfBodyChangedConstructors($c_m$) +
          getNoOfAddedConstructors($c_m$) + getNoOfDeletedConstructors($c_m$);
    end
    return $r$;
```

objects, the state transformation mechanism described in this dissertation cannot automatically convert state of such a field. Therefore, these cases could contribute to a higher cost of the runtime phenomena. In the previous example, the cost of runtime phenomena on a collection field would be higher than on a field containing a single object.

Procedure `costRT` implemented as shown in Algorithm 13 calculates the estimate of runtime phenomena as cost based on the number of class members. Each changed class member is valued as constant cost 1. Added predecessor class p_a can introduce an oblivious update and an absent state. Therefore, the cost of the *oblivious update* is calculated by the number of constructors (function `getNoOfConstructors`) and the number of fields (function `numberOfFields`) in predecessor class p_a. Meanwhile, deleted predecessor class p_d can introduce *phantom objects* and *lost state*. *Phantom objects* are not calculated, because in the approach presented in this dissertation it is expected that unused objects will be garbage collected. However, since the lost state depends on fields, the cost is calculated by the number of fields in deleted predecessor class p_d. Furthermore, for predecessor class p_m with changed members, the cost of the *oblivious update* is determined by added, deleted, and constructors with a changed body. Furthermore, the cost of *absent state* is determined by the number of added fields (function `getNoAddedFields`), and the *lost state* is calculated by the number of deleted fields (function `getNoDeletedFields`). The runtime phenomena cost for class with changed members c_m is calculated in a similar way as for p_m class. The *absent state* regarding type comparison is not included in this estimation because the approach presented in this dissertation enables class hierarchy changes and thus the comparison of objects with the changed type.

The results of runtime phenomena detection and estimation algorithms are shown in Table 9.7 for *Pinpoint* and Table 9.8 for *NewPipe*. The results show that the estimation of the runtime phenomena for *NewPipe* is lower in compared to *Pinpoint*, which is a consequence of the number of changed fields as shown in Table 9.3 and Table 9.4. As expected, the runtime phenomena estimation is higher for dynamic updates from minor version to revision. Meanwhile, based on the type of runtime phenomena, the smallest number of objects with *lost state* will occur, which corresponds to the small number of deleted fields. The number of other runtime phenomena depends on the number of changes in the subsequent program versions and the program domain. For example, the *oblivious update* in *Pinpoint* is estimated for a large number of classes between several revisions, such as *1.8.0-RC1* and *1.7.3*. Meanwhile, runtime phenomena are not expected to occur in *NewPipe* for a dynamic update from the corresponding previous version to the *v.0.16.1* and *v0.14.2*, because changes are only in the executables bodies. Furthermore, the results show that the total number of possible runtime phenomena is greater than the number of those that occur as a result of changes only in class members. Therefore, it can be concluded that it is necessary to include the inheritance relationship for the runtime phenomena estimation. Furthermore, the results for *absent state* regarding type, show that many objects can be affected when the hierarchy with a large number of classes changes, depending on the position of the change and the height of the class tree. However, the DSU approach that provides changes in the class hierarchy implicitly prevents *absent state* related to type comparison.

Table 9.7: Pinpoint runtime phenomena estimation

version	cost	phantom	oblivious	absent state type	absent state fields	lost state	oblivious (m)	absent state fields (m)	lost state fields (m)
1.7.0-RC2	32/*	0/*	4/*	0/*	10/*	1/*	2/*	8/*	1/*
1.7.0	9/*	0/*	0/*	0/*	5/*	0/*	0/*	5/*	0/*
1.7.1	9/9	0/0	0/0	0/0	5/5	0/0	0/0	5/5	0/0
1.7.2	530/530	50/50	84/84	70/70	132/132	42/42	64/64	113/113	2/2
1.7.3-RC1	29/535	2/51	2/84	0/69	8/132	2/43	2/65	8/114	0/2
1.7.3	9/535	0/51	0/84	0/69	5/132	0/43	0/65	5/114	0/2
1.8.0-RC1	910/1364	21/72	192/253	81/150	228/333	27/69	149/194	183/271	10/1
1.8.0	25/1370	0/72	2/254	1/151	12/335	0/69	1/194	11/272	0/11
1.8.1-RC1	951/951	55/55	137/137	101/101	175/175	30/30	88/88	121/121	10/10
1.8.1	11/951	0/55	0/137	0/101	6/175	0/30	0/88	6/121	0/10
1.8.2-RC1	51/949	2/55	5/138	0/99	12/177	2/30	5/91	12/125	0/10
1.8.2	31/969	0/55	2/140	2/101	10/179	0/30	0/91	8/125	0/10
1.8.3	18/974	0/55	2/142	0/101	9/181	0/30	2/93	9/127	0/10

* Results from previous versions are excluded

Table 9.8: NewPipe runtime phenomena estimation

version	cost	phantom	oblivious	absent state type	absent state fields	lost state	oblivious (m)	absent state fields (m)	lost state fields (m)
v0.13.0-beta	*/*	*/*	*/*	*/*	*/*	*/*	*/*	*/*	*/*
v0.13.1	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
v0.13.2	48/48	1/1	5/5	7/7	12/12	0/0	3/3	8/8	0/0
v0.13.3	527/560	87/88	39/39	148/15	51/54	49/49	5/3	13/16	0/0
v0.13.4	10/570	0/88	3/42	1/156	4/58	0/49	3/6	4/20	0/0
v0.13.5	4/574	0/88	1/43	0/156	1/59	0/49	1/7	1/21	0/0
v0.13.6	117/676	0/88	7/46	0/156	22/70	0/49	7/10	15/29	0/0
v0.13.7	0/676	0/88	0/46	0/156	0/70	0/49	0/10	0/29	0/0
v0.14.0	111/775	3/91	10/55	19/175	48/94	0/49	3/12	14/36	0/0
v0.14.1	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
v0.14.2	214/214	0/0	20/20	23/23	96/96	0/0	4/4	42/42	0/0
v0.15.0	213/404	7/7	18/38	22/45	18/102	1/1	9/13	12/43	0/0
v0.15.1	54/54	4/4	8/8	9/9	12/12	2/2	3/3	6/6	0/0
v0.16.0	83/137	0/4	12/20	18/27	14/26	0/2	1/4	3/9	0/0
v0.16.1	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0

* Results from previous versions are excluded

9.1.3 Use case example

The object-oriented variant of *Snake* is developed as a use case example of software evolution. Two versions are developed to validate the applicability of the proposed update model. In the first version, class hierarchy is changed such that class is added as the parent of an existing class. Besides the change of parent, there are no other changes. A type change occurs that creates several dynamic update classes. In the second version, there are several changes. Class changes the parent with the existing class, and the class members change as well. Both cases are used to confirm the applicability of the proposed prototype regarding changes in inheritance and class members.

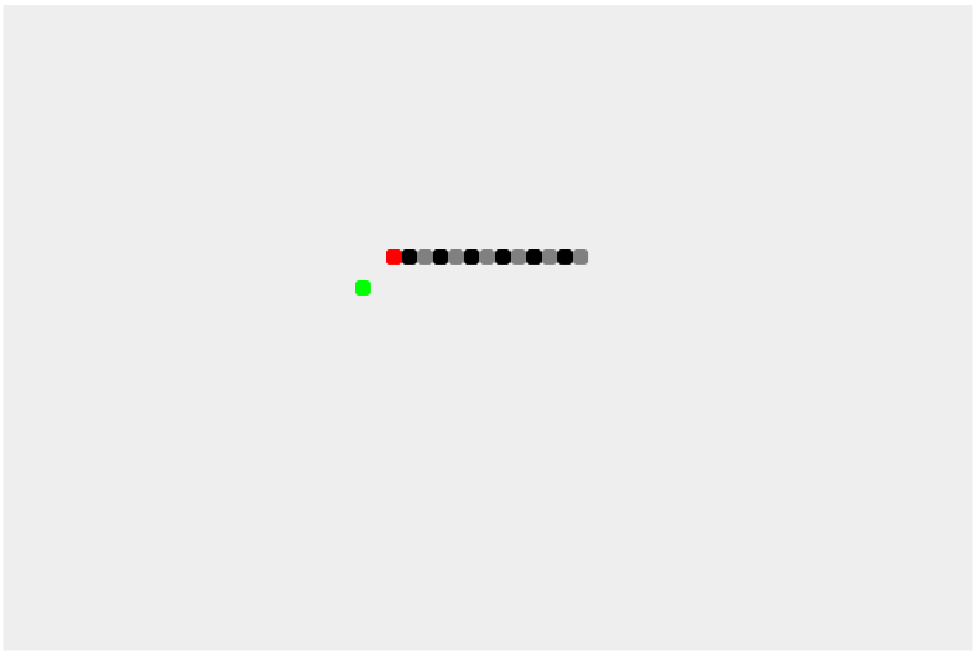


Figure 9.1: Implemented Snake game screen (*green dot – food, red head with white/black body dots – snake*)

The game consists of a player controlling the snake using four arrow keys on the keyboard guiding it to the food. After the snake consumes food, it grows by one element; that is, dot on the game screen (Figure 9.1). The game ends when the snake collides with its own body.

This game is implemented in Java using several classes; as shown in Figure 9.2. Class `Game` initializes the game and controls the gameplay. To control snake by keyboard, class `KeyLis-tener` (short as `KeyList`) is implemented. Class `Board` defines board responsible for drawing elements on the board, as objects of class `Element`. Elements contain information about position on the board and are drawn on the screen. `Food` extends `Element` by random positioning on the screen. `Movable` extends `Element` by enabling the repositioning of element. `Snake` is a movable element that can grow and eat food.

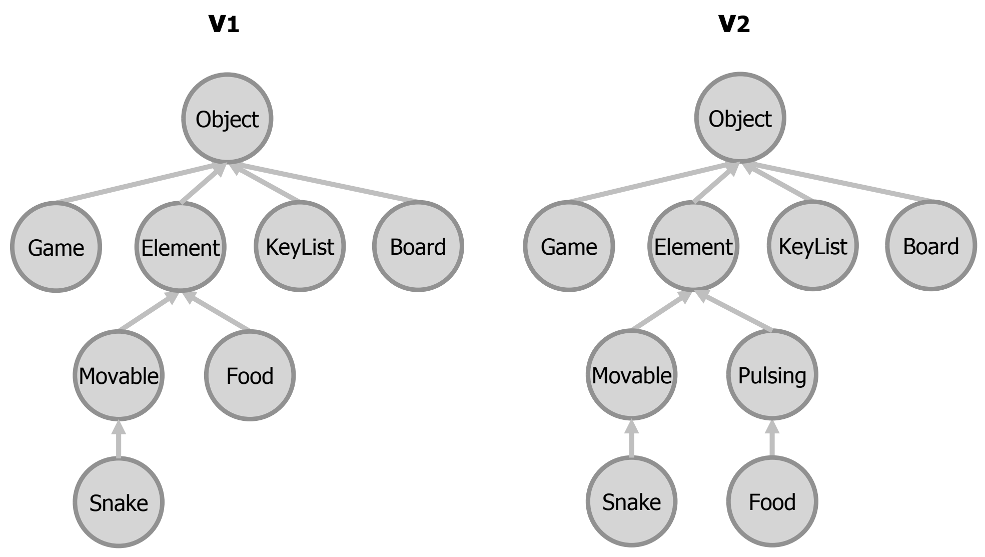


Figure 9.2: Added `Pulsing` element class as parent to `Food` class

Example 1: Added class as a new parent class

In the first update example (Figure 9.2), the `Pulsing` class is added, which extends the `Element` class. `Pulsing` introduces a new element behavior, temporary invisibility. New behavior is applied to the `Food` class by changing the parent. Instead of the `Element` in v_1, parent for `Food` in v_2 is `Pulsing`. Furthermore, a change in functionality happens when food is invisible: the snake cannot eat food. Furthermore, the body of method `eatsFood` in class `Snake` is modified.

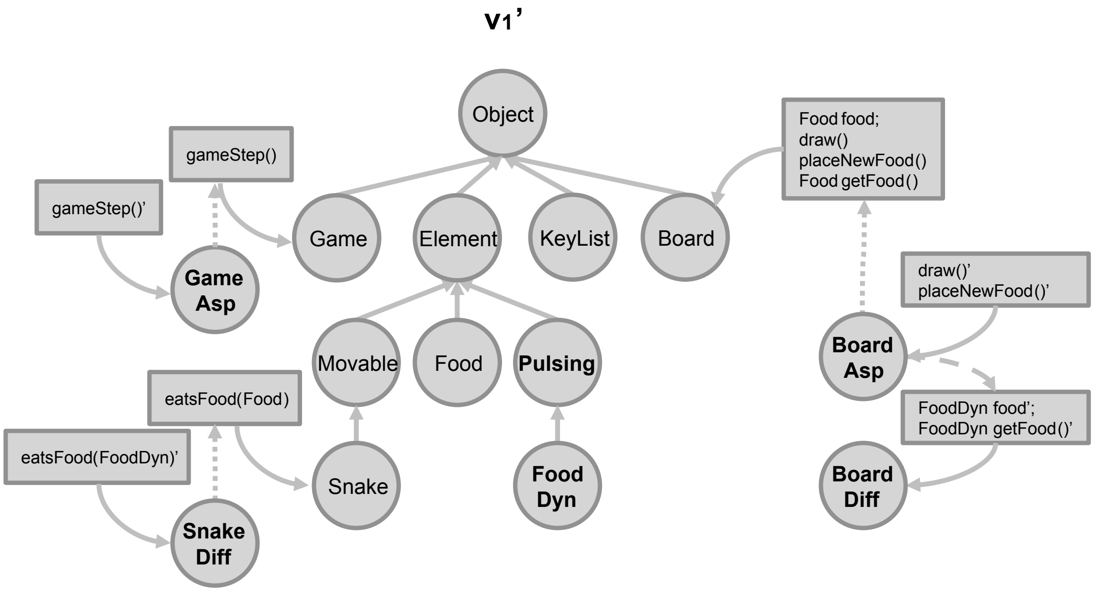


Figure 9.3: *Snake* updated to version with *dynamic* class `Food` that inherits from new class `Pulsing`

Figure 9.3 shows the dynamically updated version v_1 for the first example. To change the parent of the `Food` class, *dynamic* class `FoodDyn` is created. The added class `Pulsing` inherits `Element`, whereas `FoodDyn` inherits `Pulsing`. For each class that contains a method that uses the `Food` class as a type, *dynamic aspect* classes are created. In class `Game`, method `gameStep()`

uses *Food* object to detect whether the snake ate the food. Therefore, `GameAsp` class is created to replace `gameStep()` method body with the `gameStep()'` advice method body, such that statement that uses `Food` as type is replaced with the statement that uses `FoodDyn`. Furthermore, `BoardAsp` is created to replace the two methods in `Board`, `draw()` and `placeNewFood()`. The `draw()` method uses a *Food* field to draw the food on the board, whereas the `placeNewFood()` method creates new food on the board and stores the food in the field. Method `placeNewFood()` is invoked from the `gameStep()` method, whereas `draw()` is used by internal Java drawing classes. Furthermore, class `Board` contains a field of type `Food` and method `Food getFood()` with the return type `Food`. Furthermore, method `getFood()` is invoked by method `gameStep()`. Therefore, *diff* class `BoardDiff` is created. The `BoardDiff` class contains field definition of type `FoodDyn`, used by redefined methods `draw()'`, `placeNewFood()'`, and `getFood()'`. Meanwhile, method `eatsFood(Food)` in the `Snake` receives an object of type `Food` as a parameter. Consequently, `SnakeDiff` class is created that contains the `eatsFood(FoodDyn)` method with the `FoodDyn` parameter. The method defined in the `SnakeDiff` class is invoked from the `gameStep'()` method defined in the `GameAsp` dynamic aspect class.

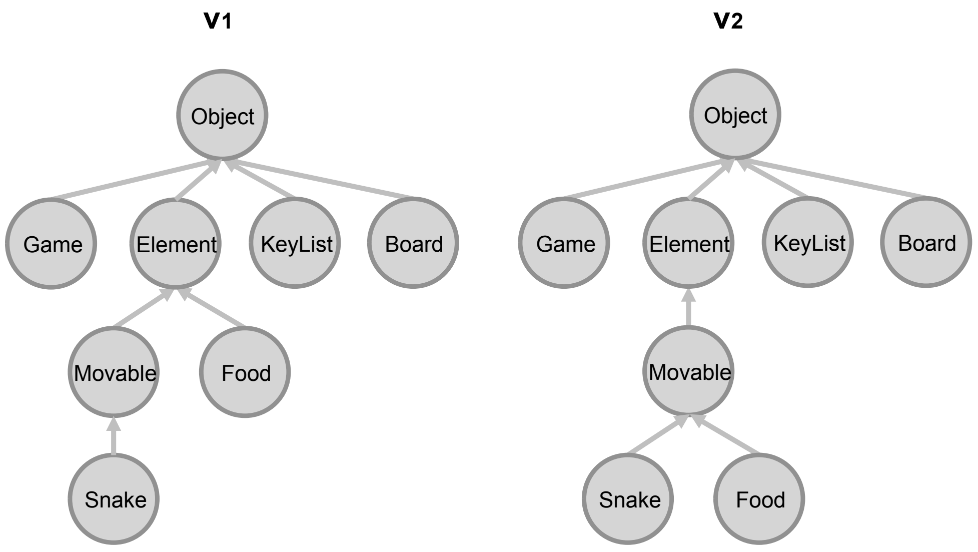


Figure 9.4: `Food` inherits `Movable` class to introduce second player that controls food

Example 2: Existing class as a new parent class

In the second update example (Figure 9.4), to introduce a second player who controls the movement of food, the `Food` class changes the parent from `Element` in v_1 to `Movable` in v_2. Furthermore, another direction field is added to the `KeyList` class to receive keyboard entries for the second player.

Figure 9.5 shows v_1' with applied dynamic updates from the second example. Since the `Food` class changed parent in v_2, *dynamic* class `FoodDyn` is created. Therefore, similar to the first example, statements that use `Food` objects are replaced by *dynamic aspect* and *diff*

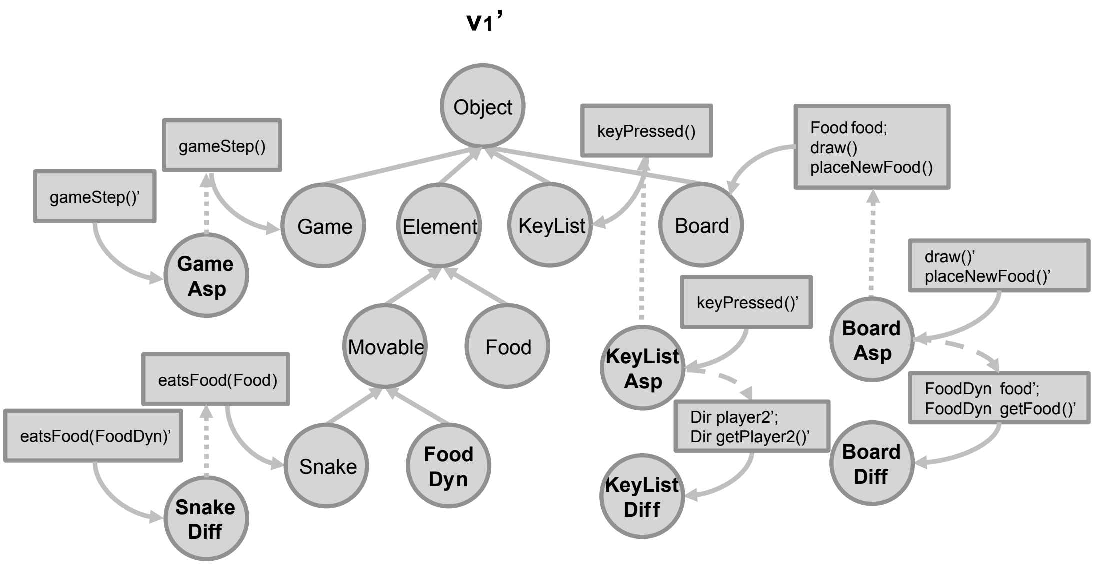


Figure 9.5: *Snake* updated to version with *dynamic* class `Food` that inherits from existing class `Movable`

classes. `BoardDiff` and `SnakeDiff` are used for changed fields and methods with changed parameters, whereas `BoardAsp` and `GameAsp` for methods that use these fields and methods. In comparison to the first example, there is a change in the `KeyList` class, the `player2` field and the `getPlayer2()` method were added to receive keyboard entries for the second player. The `getPlayer2()` method is invoked from the `gameStep` method in class `Game` from v_2. Therefore, advice method `gameStep()'` invokes the `getPlayer2()'` method defined in the `KeyListDiff` *diff* class. Furthermore `gameStep()'` based on the direction of the second player, moves food in the current game step. *Dynamic aspect* class `KeyListAsp` is created to change the body of the `keyPressed()` method to store the keyboard entries for the second player. Advice method `keyPressed()'` uses field `player2'` defined in `KeyListDiff`.

Possible runtime phenomena discussion

Previous use case examples show that dynamic update is performed for two arbitrary program changes. However, dynamic updates can introduce runtime phenomena, as described in Chapter 6. Algorithms to estimate and detect runtime phenomena from Chapter 6 are used to discuss possible runtime phenomena. The used cost function is δ_{RT}, as defined in Algorithm 13. The results are shown in Table 9.9. Dynamic update $v_1 \rightarrow v_{1.1}$ corresponds to the first example and $v_1 \rightarrow v_{1.2}$ to the second example.

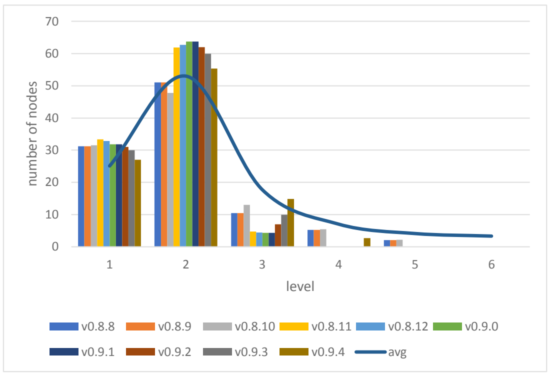
Table 9.9: Detected runtime phenomena for use case example

version update	cost	oblivious	absent state type	absent state field	absent state field (m)
$v_1 \rightarrow v_{1.1}$	4	1	1	1	0
$v_1 \rightarrow v_{1.2}$	4	1	0	2	1

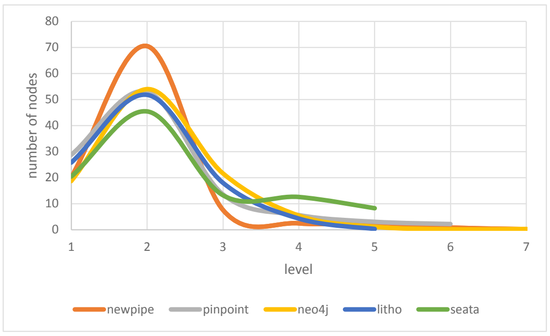
For both examples, the estimated cost is 4. In the first example, Pulsing is added as predecessor to Food in $v_{1.1}$. Class Pulsing contains three fields, therefore *absent state field* is detected and the cost is increased by three. In this scenario, the *absent state* would not occur because an object of the Food class exists on the board until it is eaten by the snake. When the food is eaten then new object is created in the gameStep() method that uses initialized fields from the Pulsing class. However, if the Food object was not created on each snake hit, then a new instance would be created using the *dynamic aspect* and *diff* classes, on the next call to method gameStep(). The state transfer would occur from an existing Food object to the new FoodDyn object by the DSU manager. Furthermore, class Pulsing contains a single constructor, resulting in an increase in cost by 1 and detected *oblivious update*. As described runtime phenomena related to fields would not occur since new object of FoodDyn would be created. *Absent state type* is detected because class Pulsing inherits Element which changes the subtype information for class Element. In $v_{1.1}$ objects of class Element are comparable to objects of class Pulsing. However, as previously mentioned, this type of runtime phenomena is implicitly avoided with support for class hierarchy changes.

In the second example, the Food class changed parent to the Movable class. Since the Movable class contains a single field, *absent state field* is detected, and the cost is increased by 1. Furthermore, Movable contains two constructors, resulting in detected *oblivious update*m thus increasing the cost by 2. Meanwhile, *oblivious update* and *absent state* would not occur because, as in the first example, a new object of class FoodDyn is created with inherited members from the *Movable* class and state transfer is performed from the Food object. Meanwhile, in the KeyList class, a field is added for the moves of the second player. Therefore, *absent state field (m)* is increased by 1 because there may be objects of class Keylist created in v_1. Consequently, *absent state field* also increases by 1. In this particular case, *absent state* would not occur because the player's move is set to enum with the default value none. In the Snake gameplay, this does not produce unwanted behavior because the second player joins the game by pressing the corresponding key and thus changes the value of the field. Therefore, it is not necessary to initialize the value using the custom transformation functions. However, for more complex cases when the program semantic requires, a custom transformation function can be used.

It can be observed that the cost function δ_{RT} should be adjusted for specific DSU approaches and program semantics. The approach presented in this dissertation can implicitly perform state transfer for cases with class hierarchy changes. However, the initialization for the added field in matched classes should be inspected based on the program semantics in the updated version.



(a) Node distribution per level for 10 *NewPipe* program versions and average number of nodes for currently available versions



(b) Node distribution per level for available versions of 5 publicly available programs

Figure 9.6: Node distribution in program versions [98]

9.2 The efficiency of the algorithms

Efficiency evaluation is performed by measuring and comparing the algorithm's execution time where the generated trees are used as input for algorithms. The algorithm for creating a random tree from [98] is suitable for scenarios that require a random connection between nodes. However, domain data usually follows certain patterns. Therefore, the distribution of nodes in several open-source programs is observed. The obtained distribution is used to generate initial trees that reflect the class hierarchy of the real program. A generated class tree is modified by the distortion parameters obtained by analyzing the changes between subsequent versions in open-source programs to reflect real changes between class trees in different versions. Algorithms for generating class trees are based on work in [98]. Therefore, parts of this work are included for completeness.

9.2.1 Distribution pattern analysis

Tree characteristics such as the number of nodes (n), height (h) together with the level and degree of nodes are used to analyze the shape of a tree. Statistical values based on these measures include the number of nodes for each level; that is, the number of nodes per level - N_l and the average degree of nodes per level D_a.

Figure 9.6a shows how the distribution of nodes per level N_l is similar between class trees in *NewPipe* for 10 subsequent program versions from *0.8.8* to *0.9.12*. The average number of nodes per level for the currently available 62 versions of the *NewPipe* [63] is represented by the blue line in Figure 9.6a. The distribution shape for 10 subsequent versions corresponds to the node distribution in 62 versions. To obtain a more precise approximation of tree shape and node distribution, in addition to *NewPipe*, four more programs with publicly available source code were analyzed, for which the average node distribution is shown in Figure 9.6b. Average values show a similar tendency at the same levels between different programs and correspond to the values shown in Figure 9.6a. Both figures show a pattern for the class hierarchy in the form of a node distribution.

9.2.2 Generating class trees by distribution pattern

The distribution of nodes per level for the class hierarchy of five publicly available programs is discussed in the previous subsection and shown in Figure 9.6b. The node distribution pattern reflects that the nodes are mostly concentrated at the first two levels, with an increase at the third level, while at a higher levels the node concentration decreases. An approach based on distribution pattern obtained from the data is required to generate a tree that reflects real data from a particular domain or observed phenomenon. Therefore, the algorithm given in [98] is introduced to generate a tree based on the given distribution of nodes per level. Algorithm 14 to generate a class tree is based on the distribution of nodes algorithm. Similar to the original algorithm, connections between nodes are generated randomly, where nodes at the current level are randomly selected from a set of nodes without parent node - N and are connected to a randomly selected node from a set of nodes at the previous level - P. Consequently, the algorithm creates a tree level by level. First, the root node is randomly selected - r, and then nodes for each level - u. The number of nodes at each level (C) is determined as a result of the function `getNodeCountPerLevel`. The function returns the number of nodes per level based on the given distribution of nodes in percentage - D, number of nodes – n, and current level i. Set of nodes in the current step in the algorithm – T are predecessor nodes – P in the next step. The difference between Algorithm 14 and the algorithm presented in [98] is in created nodes and input parameters related to the created nodes. Algorithm 14 at line 3 uses function `newClassNode`. The function creates a class node based on the following parameters: number of fields (f_n), executables (e_n), and number of parameters for executables (p_n). Fields and parameters are integer type by default.

9.2.3 Distort original tree

One of the possibilities to compare class trees is to generate two independent trees using Algorithm 14. In the case of the class hierarchy, a comparison is performed between two versions of the program. Therefore, it is necessary to modify the original tree to obtain another tree for comparison. The difference between created trees can be controlled by distortion parameters as the amount of added, deleted, and matched nodes with the changed parent, similar to operations performed on nodes in edit distance algorithms [98]. By changing the specific distortion parameter, tree changes can be analyzed. Generally, parameters can be used in domains where the controlled difference between generated trees is required for comparison (e.g., a domain where

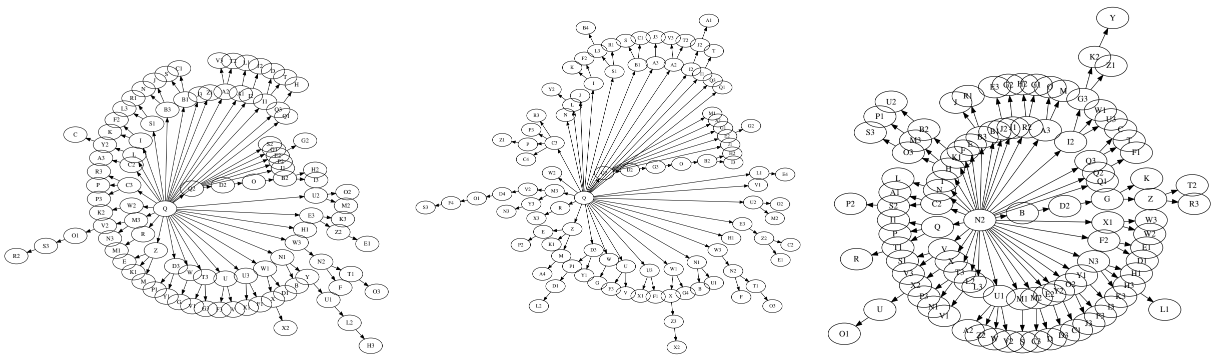
Algorithm 14: Algorithm to generate class tree by distribution

```
input : the number of nodes as classes $c_n$, node distribution per level $D$ (expressed as
        percentages), the number of fields $f_n$ and executables $e_n$ for class, number of
        parameters $p_n$ for executables
output: Class tree $T(V,E)$

$V \leftarrow \varnothing, E \leftarrow \varnothing$;
for $i \leftarrow 1$ to $n$ by $1$ do
    $V \leftarrow V \cup \{$ newClassNode($f_n$, $e_n$, $p_n$) $\}$;
end
$N \leftarrow V$;
$u_r \leftarrow$ getRandomNode($N$);
$N \leftarrow N \setminus \{u_r\}$;
$P \leftarrow \{u_r\}$;
for $i \leftarrow 1$ to $|D|$ by $1$ do
    $T \leftarrow \varnothing$;
    $C \leftarrow$ getNodeCountPerLevel ($D$, $n$, $i$);
    for $j \leftarrow 1$ to $C$ by $1$ do
        $u_c \leftarrow E$ getRandomNode($N$);
        $u_p \leftarrow$ getRandomNode($P$);
        $E \leftarrow E \cup$ newEdge($u_p$, $u_c$);
        $N \leftarrow N \setminus \{u_c\}$;
        $T \leftarrow T \cup \{u_c\}$;
    end
    $P \leftarrow T$;
end
```

the tree can only differ in added or deleted nodes). The algorithm to generate trees based on the given tree and the amount of distortion is given in [98] and is used in this dissertation to modify the initial class tree.

An example of a tree generated by distortion algorithm [98] from the tree shown in Figure 9.7a, as the source tree with 100 nodes, is shown in Figure 9.7b. The distortion parameters used are 10 added and deleted nodes and 10% of matched nodes with changed parents. Note that the tree in Figure 9.7a is generated by the algorithm with the following distribution: 32.5, 50, 10, 5, and 2.5%. The given distribution of nodes corresponds to the distribution of trees representing the class hierarchy of *NewPipe* in *v0.8.9*. Trees in Figures 9.7a and 9.7b are similar since the tree in Figure 9.7b is generated by controllable distortion of the source tree in Figure 9.7a. Meanwhile, the tree in Figure 9.7c is generated by distribution algorithm with equal distribution as the tree in Figure 9.7a. An approach where the node distribution algorithm generates both trees can also be used to generate trees for comparison. In that case, the distribution of nodes in both trees is equal, but distortion arises in the form of a possibly different number of nodes and random connections between nodes.



(a) by node distribution (b) distorted from (a) (c) by node distribution equal to (a)

Figure 9.7: Class trees generated by node distribution and tree distortion

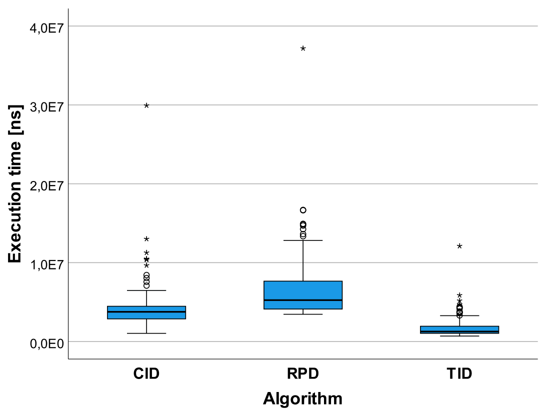
9.2.4 Evaluation setup

The evaluation experiment consists of an experimental setup in which the source class tree (v_1) generated by Algorithm 14 is compared with the target class tree (v_2) obtained by the distortion algorithm from [98]. Based on the analysis of open-source programs from Subsection 9.2.1, the distribution used to generate the class trees representing the v_1 is as follows: 32.5, 50, 10, 5, and 2.5 %. Class trees in v_1 were generated for sizes of 1k, 10k and 100k nodes. Analysis of the program changes as distortion shows that program versions differ on average in the number of added, deleted classes and changed type classes. Therefore, the following parameters are used as input of the distortion algorithm: 6.5% added, 4% deleted, and 0.3% permuted nodes. Nodes as mock classes are created with a single field and method with a single parameter. Furthermore, class trees were generated 100 times, and Algorithm 4 (CID - Class Inheritance Detection) and Algorithm 11 (RPD - Runtime Phenomena Detection) are executed to measure execution time. The results of the experiment are shown as the distribution of execution time in the form of box-whisker graphs shown in Figure 9.8. The tree inheritance algorithm (TID, Algorithm 2) is executed on the same trees as a reference for comparison. The algorithms are not interchangeable considering usage, although the results are presented together.

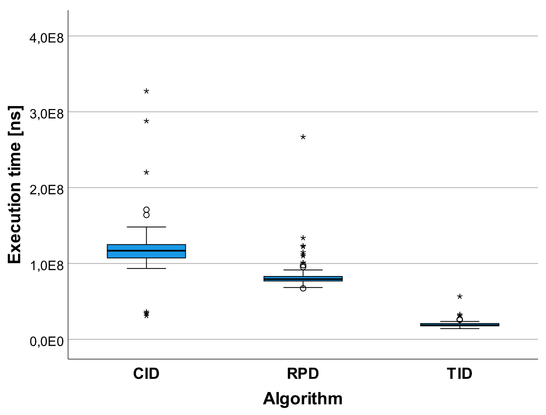
Evaluation tests are performed on Windows 8.1 Pro 64-bit running on the computer with Intel i7-3610QM@2.3GHz processor power and 24GB of system memory.

9.2.5 Results

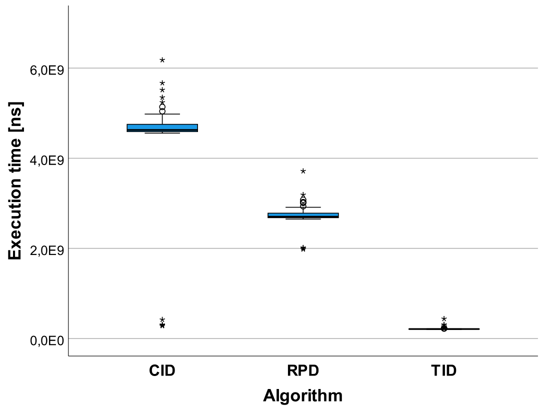
The evaluation results are shown in Figure 9.8. The execution time for each algorithm is expressed as nano-seconds. For a class tree with the initial size of 1000 nodes, algorithms to detect class hierarchy changes and runtime phenomena are slower than tree the inheritance distance (Figure 9.8a). CID and RPD algorithms perform additional comparisons of nodes by class members, whereas TID only detect structural changes. Furthermore, RPD is slightly slower than the CID algorithm, which is related to the `costRT` procedure to estimate runtime phenom-



(a) 1k nodes



(b) 10k nodes



(c) 100k nodes

Figure 9.8: Execution time for algorithms by various initial class tree sizes (CID - class inheritance changes detection, RPD - runtime-phenomena detection, TID - tree inheritance distance)

ena. As described in Subsection 9.1.2, RPD iterates through changed predecessor classes and calculates the cost based on the number of class members. Meanwhile, Figure 9.8b shows the results for the class tree with an initial size of 10000 nodes. The CID and RPD algorithms are slower than TID because of the additional node processing. However, in comparison to the case with 1000 nodes, the CID algorithm is slower than the TID algorithm. Moreover, the difference in execution time is more pronounced with a larger number of nodes, as shown in Figure 9.8c. The main difference between these algorithms is in detecting the overriding methods in CID by the procedure `detectOverridingMethods`. Therefore, for a large number of classes, changes in classes affect the method overriding because for the evaluation setup, each class contains single method with the same signature.

Based on the results in Figure 9.8, it can be concluded that algorithms perform calculations relatively quickly compared to the size of generated class trees. For example, for large trees as 100k nodes in Figure 9.8c, CID as the slowest executed algorithm is below 6 seconds on the evaluation computer. However, in real cases, execution corresponds to the number of class members in the program version. The results presented serve as an estimation for comparison, as they are based on the case where each class contains a single field and method.

9.3 DSU performance

9.3.1 Methodology

According to the proposed measurement methodology in Chapter 7, several Java DSU approaches are selected for comparison [9, 15, 22, 27]. The main criterion for selection was public availability. Evaluation of the DSU approaches in this section is based on the work in [54]. As an extension, the prototype system (eDAOP) presented in Chapter 8 is evaluated and compared to the selected DSU approaches. Therefore, parts of the work are included for completeness. Based on the categorization in Section 3.1, Jvolve [9] and DCEVM [15] belong to the modified JVM, whereas Prose [27], Jooflux [22], and prototype system (eDAOP) belong to the DAOP category implemented as JVMa. eDAOP as an extended dynamic aspect system is evaluated in two implementations, using Prose and built-in prototype, as described in Chapter 8. Currently, pure JVM agents are not publicly available. Javeleon [3] has become a commercial product, and Javadaptor [16] has not been made public. Microbenchmark tests for Prose and Jooflux are based on work in [24]. Because the Cech [24] implementation is not publicly available, tests are implemented as an approximation based on the described concept. For example, `ext` field is used in the class to reference the object containing the changed class members.

According to the methodology described in Chapter 7, the developed benchmark tool is used to measure resource demands. As aforementioned, the tool consists of macro and implemented micro benchmark tests. In this dissertation, open-source test suite DaCapo 2006MR2 [92] is used as a macro benchmark, with selected tests that are executable on evaluated approaches: *bloat*, *chart*, *hsqldb*, *jython*, *luindex*, and *lusearch*.

Update duration, execution duration and memory overhead are measured by the implemented micro benchmarks with the developed test suite. Table 9.10 contains tests from the suite with the associated categorization. For each program change, besides for method body and multiple modifications, the results for two tests, add and remove (A/R), are shown. Tests in the basic category perform simple modifications; for example, in the original version of `Fact` class in Figure 7.2 of Chapter 7, test *AddField* (AF) represents the addition of the field. To perform changes on DAOP beyond the method body modification, the extended model (eDAOP) described in this dissertation is used and compared to the client-supplier model from [24] (Prose-Cech'). In Table 9.10, tests in the compound category consist of multiple changes. For example, *AddFieldChangeMethod* (AF+MB) in Example 5.3 consists of added field `result` and the modified body of the `calculate` method that uses the newly added field in the updated version. Figure 5.3 shows the *Multiple* test (Table 9.10) that includes various modifications of class members (level 2) with supertype change (level 3). Furthermore, in Table 9.10, column *Lev* represents the modification level as the maximum level used in a particular test.

Evaluation tests are performed on Ubuntu 12.04.5 LTS kernel 3.13.0-32-generic virtual machine running on Hyper-V in Windows Server 2012 R2 Datacenter. The virtual machine is set to 25% of the overall 2xIntel Xeon E5-2640 processor power, with up to 8 GB of system memory. Furthermore, the *server* compiler configuration is used in the HotSpot VM for eDAOP, Jooflux, Prose and DCEVM whereas the *fast adaptive* compiler configuration in the Jikes RVM for Jvolve.

9.3.2 Results

The tests are performed 50 times with the DaCapo warm-up option and in separate VM instances to measure overhead due to the presence of the DSU in program execution (steady-state). Each test run is performed on the two test cases, with (DSU) and without dynamic updating (standard). Overhead is calculated as a mean value of the difference between execution time with and without DSU expressed as a percentage. The results in Figure 9.9 represent overhead across the entire test suite for the selected approaches, besides for eDAOP, *hsqldb* test for Jooflux and Prose, and *jython* test for DCEVM. The difference in these cases is due to separate runs of the tests and is negligible. For DAOP approaches, the overhead in most tests is below the standard deviation, besides for Prose in the *lusearch* test. Moreover, the results for *lusearch* demonstrate a high steady state overhead for all selected approaches except Jooflux and eDAOP. This shows that DSU approaches introduce a significant overhead for multithreaded tasks that process large memory objects [92]. The results show that eDAOP does not introduce steady state overhead. Meanwhile, eDAOP uses the HotSwap feature to replace the bytecode of the method body, which does not affect VM execution when there are no dynamic updates. Furthermore, the results of eDAOP implementation using Prose correspond to the results of Prose.

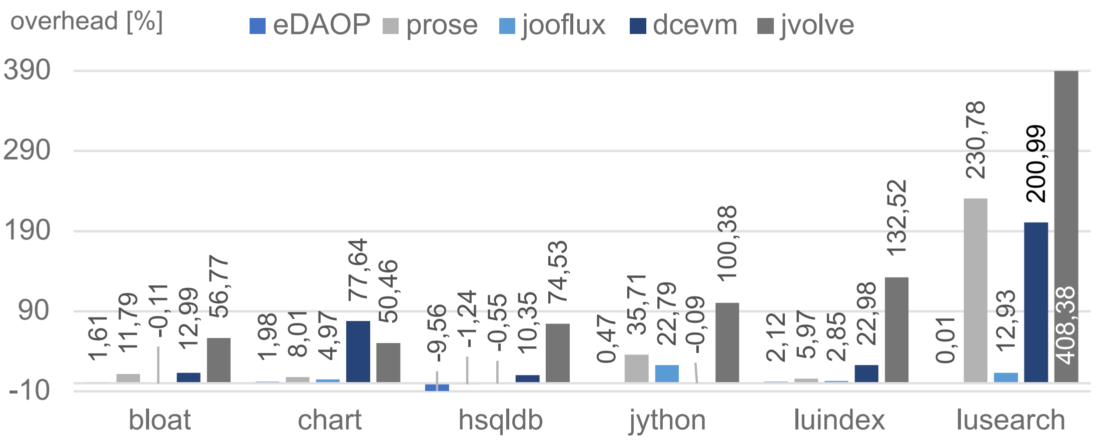


Figure 9.9: Steady state overhead

Dynamic update duration is measured in 50 test runs on separate VM instances and 100k test class objects. The results are shown as the mean update time in milliseconds for each dynamic change test from Table 9.10. DAOP approaches, eDAOP, Jooflux (Cech†), and Prose

(Cech†), support level 2 changes only indirectly through compound changes ("**"). Jooflux lacks support for field changes because it does not currently support the join-point activation on the field level, required for dynamic evolution based on Cech. Meanwhile, eDAOP, the approach presented in this dissertation, supports class hierarchy changes, and therefore *Multiple* test. DCEVM supports all microbenchmark tests, while Jvolve partially supports supertype change, but only as a compound change ("***"). The Jvolve comparison tool lacks the functionality to detect class supertype change. The results show that values are lowest for the Jooflux, and highest for the Prose approach. Jooflux switches call site using the invokedynamic [22], while Prose modifies the bytecode with method inlining technique [27]. DCEVM and Jvolve show similar results, as both are based on the modified JVM garbage collector. In addition, eDAOP shows similar results for various program changes. For built-in implementation, the results are lower than Prose and comparable to the DCEVM and Jvolve as a modified VM. eDAOP (Prose) results are slightly higher than Prose (Cech†) which can be related to using an aspect register implemented in the prototype. Overall, the time to perform dynamic updates for different tests within the same approach does not vary significantly. However, it can be observed that DCEVM has higher values for the *AddField* and *AddSupertype* tests because the dynamic updating in DCEVM involves memory and class hierarchy rearrangement. Meanwhile, Prose results show that field updates are slower, which indicates that join-point activation is slower for class fields. Finally, eDAOP shows the most stable values across microbenchmark tests.

Table 9.10: Dynamic update duration

Category		DSU duration [ms]					
Level	Test	eDAOP (built-in)	eDAOP (Prose)	Prose (Cech[†])	Jooflux (Cech[†])	DCEVM	Jvolve
1	Method body (MB)	183,92	703,051	685,58	6,78	115,49	73,01
2	Method (M*)	**	**	**	**	116,08/115,32	143,26/137,85
	Constructor (Co*)	**	**	**	**	113,34/116,29	142,76/142,45
	Field (F*)	**	**	**	NA	154,12/123,86	143,38/148,57
	Class (C*)	****	***	****	****	****	****
3	Supertype (S*)	NA	NA	NA	NA	154,91/135,48	***
	Interface (I*)	NA	NA	NA	NA	117,40/126,73	142,65/140,66
2	M* + MB	175,63/179,74	702,534/702,908	685,91/705,47	3,98/6,62	121,92/122,77	169,73/164,62
	F* + MB	189,23/186,16	705,880/700,456	1125,74/685,88	NA	170,49/124,81	167,75/170,36
	Co* + MB	179,75/173,57	703,046/705,618	686,66/684,68	5,37/5,22	122,19/122,37	169,82/167,22
	C* + MB	191,81/185,96	709,845/709,149	693,02/690,29	6,56/6,33	115,63/115,23	69,31/74,59
3	Multiple	178,22	730,266	NA	NA	121,31	142,05

[†] tests are based on [24]
* Add and remove modification
** Indirectly supported
*** Partially supported
**** Dependent modification

Table 9.11 shows the differences in memory usage before and after the dynamic update. The test setup is the same as for the update duration (50 runs and 100k objects), and the values denote the mean difference in megabytes (MB). Values were measured only for method body and compound changes because most approaches support them. The results for some approaches show a decrease in memory consumption. For example, Jvolve, if required, performs garbage collection after each update and then executes the transformation function. To perform a memory rearrangement, DCEVM activates a modified garbage collector to add a field. Meanwhile, DAOP approaches have low memory usage, besides Prose for the *AddField* test, related to field join-point activation. The results for eDAOP as built-in implementation shows decrease in memory usage, which is related to the JVM class redefinition functionality. In these cases, JVM performs garbage collection after dynamic update for *AddFieldChangeMethod* and *AddClassChangeMethod* microbenchmark tests.

Table 9.11: Difference in memory usage in MB

Test	eDAOP (built-in)	eDAOP (Prose)	Prose[†]	Jooflux[†]	DCEVM	Jvolve
Method Body	-1,26	1,04	0,47	0,41	1,18	-5,66
M + MB (A/R)	-0,05/-0,04	2,14/2,15	1,35/1,15	0,97/1,00	1,28/1,30	2,81/2,82
F + MB (A/R)	-1,25/-1,18	1,04/1,03	5,42/1,16	NA	-0,39	2,76/2,13
C + MB (A/R)	-1,17/-1,17	1,09/1,06	0,56/-0,07	0,48/-0,1	1,18/1,18	-2,31
Multiple	-0,02	2,4	NA	NA	1,25	1,87

[†] based on Cech [24]

Execution overhead in the short-term is measured over 50 runs on separate VM instances with a single method call. Long-term overhead is measured with multiple method calls (i.e. 1000) on the same VM instance. Figure 9.10 contain results expressed as mean execution time in milliseconds in an environment without DSU (standard) and a DSU environment. The *AddChangeMethod* (AM + MB) test is performed, where the non-recursive method from Example 5.3 is added to the class and called 10M times from the method changed in another class. The results are shown for eDAOP implemented as a built-in dynamic weaver since results for Prose are similar to eDAOP that uses Prose as a dynamic aspect system. Results show that short-term overhead is lower for DAOP approaches. Prose and Jooflux introduce small overhead in short-term execution related to DSU implementation. In the long-term, execution times in a DSU environment are almost equal to execution times in an environment without DSU. Jvolve is an exception because it introduces a long-term execution overhead, meaning that some JVM optimizations must be discarded to perform dynamic updates. Furthermore, the built-in dynamic weaver (eDAOP) shows no overhead in execution time, similar to the DCEVM, conforming to the previous results. eDAOP replaces bytecode in the method body by using VM HotSwap feature that does not affect execution performance, similar to the modified VM as in DCEVM. The changed bytecode in the method body is optimized as the original bytecode, compatible with internal VM optimizations.

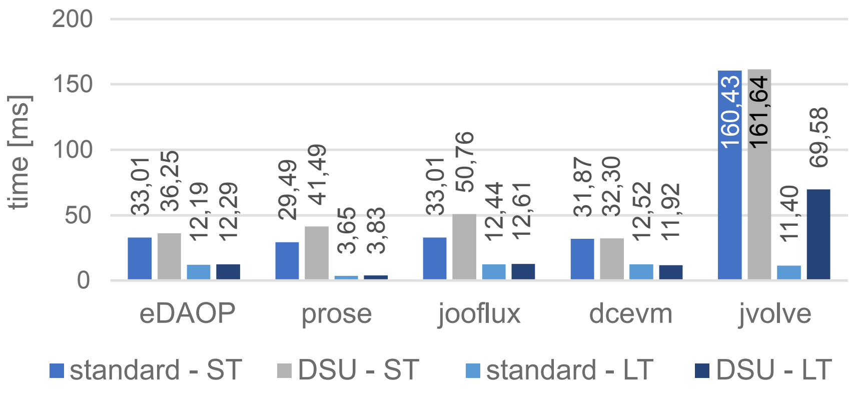


Figure 9.10: Short-term (ST) and long-term (LT) execution time

The results show that eDAOP with a built-in dynamic weaver does not introduce overhead in a steady state nor in the execution of dynamically updated methods. Dynamic update duration is comparable to the modified VM. Furthermore, there is no additional memory usage because VM performs garbage collection, resulting in less memory usage. Other approaches introduce higher memory consumption for specific microbenchmark tests. The built-in dynamic weaver utilizes the HotSwap feature of VM compatible with VM optimizations and does not introduce overhead in join point activation because method bodies are replaced with the new bytecode. Meanwhile, eDAOP based on Prose, for particular tests introduces an overhead in steady state, memory usage, and execution. However, the results are comparable to other DAOP approaches, which in general introduce less overall overhead. Jooflux performs best in the dynamic update duration. Therefore, for future work, eDAOP could be implemented with Jooflux based join point activation by using dynamic invoke feature. However, the evaluation results show that Jooflux introduces certain overheads in steady state and execution performance. Meanwhile, eDAOP implemented as a prototype conforms to the requirement of minimal performance impact on the executing environment, as described in Chapter 2.

Chapter 10

Concluding remarks

Several of the DSU challenges considered in this dissertation resulted in the proposed system for dynamic updating. In order to realize an approach for an object-oriented paradigm with a large set of possible changes, correct state after the update, and a small impact on execution performance, changes between program versions are considered from a class hierarchy perspective and applied by dynamic aspects. This chapter summarizes the contributions and discusses the presented approach and solutions. Furthermore, current open issues are discussed, together with recommendations for future work.

10.1 Summary of contributions

In this dissertation, changes between program versions are considered as changes in the class hierarchy, which is the basis for the extended DAOP update model supporting the class hierarchy changes. Runtime phenomena that occur after dynamic updating due to changes in class inheritance are considered, and algorithms for detection and estimation are presented. Furthermore, the performance evaluation methodology enables the development of the DSU approach with performance impact assessment. Therefore, the prototype system is developed based on the extended model, runtime phenomena detection algorithms, and performance evaluation methodology. By using the prototype system, the currently running program can be updated dynamically with a program version containing class members and inheritance changes with estimated runtime phenomena and minor impact on execution performance.

- **Extended DAOP update model** Analysis of the relationship between classes allows the detection of changes in the class hierarchy between program versions and the dynamic update of these changes, as shown in Chapter 5. Classes are related through inheritance relationships, forming a class hierarchy. DAOP with cross-cutting concerns property, as an indirection level, provides changes in class inheritance for dynamic updating. To enable class hierarchy changes, a *dynamic* class is introduced. By using the *dynamic* class and the client/supplier pattern, the existing class hierarchy is extended, thus enabling type changes regarding hierarchy. Furthermore, *dynamic aspect* and *diff* classes allow changes to class members.

- **Run-time phenomena detection algorithms** Algorithms to detect and estimate runtime phenomena based on changes between program versions are proposed in Chapter 6. Runtime phenomena are analyzed from the perspective of changes in the class hierarchy and call dependency. In this dissertation, the focus is on the impact of changes in inheritance on runtime phenomena. The algorithm to estimate runtime phenomena assesses the risk of performing the dynamic update. Meanwhile, the algorithm for detecting runtime phenomena results in information about program changes that can cause runtime phenomena. This information can be used to make adjustments to an updated version of the program to perform the dynamic update without detected runtime phenomena. Furthermore, to perform dynamic updates correctly in the model, a state transfer is performed by a procedure that initializes the new state of the object using the current state.
- **Prototype system** The prototype system is implemented based on the extended DAOP model and runtime phenomena algorithms. The system is described in Chapter 8 consisting of an offline and online tool. The offline tool analyzes source code of the currently running and updated version, and extracts changes in the form of Java classes. The online tool loads the created classes and applies the changes to the running program using DAOP. As Prose as DAOP does not provide a redefinition of the constructor body, dynamic weaver intended for dynamic updating and based on the Prose definition of aspects is implemented. However, in addition to Prose, other aspect languages can be used. The prototype system is evaluated in Chapter 9 on the use case example and empirical study. Results show that changes in class hierarchy and class members as consequence of software evolution can be applied by dynamic aspects. Furthermore, the estimated runtime phenomena, together with performance evaluation, reflect the applicability and effectiveness of the proposed approach.
- **Benchmark methodology for performance evaluation** One of the requirements of DSU is the minimal impact of the implementation on the system's resources and program execution performance. The methodology to evaluate and compare approaches was proposed in Chapter 7, based on which the benchmark tool was implemented. The methodology contains the measurement of DSU impact on performance without dynamic updates, with the performed dynamic update, duration of dynamic update, and impact on memory usage. With the implemented test cases representing changes in programs, the evaluation in Chapter 9 shows the advantages and disadvantages of the evaluated approaches, conforming to the implementation of a particular approach.

10.2 Open issues and recommendations for future work

The current implementation of the dynamic update prototype is constrained to executing `.class` files. Therefore, future work should include defining program changes in `.jar` files. Furthermore, the current implementation lacks support for the anonymous and inner class changes. Special cases for such classes can be implemented in the change detection algorithm and handled by *DSU manager* in the online tool. In the case of constructor changes in the v_2, when in version v_1 there is no default constructor in the parent class, the update process will create statements that use a non-existent constructor. As a result, runtime exceptions or errors will occur. In future work, this limitation can be overcome by replacing the affected classes with the *dynamic* classes. The implementation of the online tool, in general, uses JVM reflection API to manipulate objects. In further prototype development, the instrumentation API can be used to improve performance because the reflection discards some of the JVM optimizations.

Runtime phenomena detection algorithms detect phenomena related to inheritance changes. However, the algorithms are based on static analysis and estimate possible runtime phenomena. Further work should include dynamic analysis, where the online tool can be extended by heap analysis to determine currently active objects. Moreover, call dependency analysis can be performed by analyzing the current state of the stack. Dynamic analysis should improve the detection of runtime phenomena, where dynamic updates that can cause runtime phenomena can be performed if the analysis determines that the affected objects are not active. Furthermore, the DSU manager can postpone the update to analyze the current state within a predefined time frame and perform the update when possible.

A benchmark tool to evaluate the performance of the DSU approach could use unified micro-benchmark tests. In the current implementation of the benchmark, tests are manually adapted to the evaluated approach. However, implementation with unified tests requires an appropriate interface for each approach to transform the test from a unified format to a suitable format for a particular approach. In addition, future work may include new measurements, such as the impact of dynamic updating on memory usage in steady-state.

Bibliography

[1] Chen, H., Yu, J., Chen, R., Zang, B., Yew, P.-C., “POLUS: A POwerful Live Updating System”, in Software Engineering, 2007. ICSE 2007. 29th International Conference on, May 2007, str. 271–281.

[2] Fabry, R. S., “How to design a system in which modules can be changed on the fly”, in Proceedings of the 2nd international conference on Software engineering. San Francisco, California, USA: IEEE Computer Society Press, 1976, str. 470–476.

[3] Gregersen, A. R., Rasmussen, M., Jørgensen, B. N., “State of the Art of Dynamic Software Updating in Java”, in Software Technologies, Cordeiro, J., van Sinderen, M., (ur.). Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, str. 99–113.

[4] Gupta, D., Jalote, P., “On-line software version change using state transfer between processes”, Software: Practice and Experience, Vol. 23, No. 9, 1993, str. 949–964, available at: http://dx.doi.org/10.1002/spe.4380230903

[5] Kramer, J., Magee, J., “The evolving philosophers problem: Dynamic change management”, Software Engineering, IEEE Transactions on, Vol. 16, No. 11, Nov. 1990, str. 1293–1306.

[6] Neamtiu, I., Hicks, M., Stoyle, G., Oriol, M., “Practical Dynamic Software Updating for C”, SIGPLAN Not., Vol. 41, No. 6, Jun. 2006, str. 72–83, available at: http://doi.acm.org/10.1145/1133255.1133991

[7] Stoyle, G., Hicks, M., Bierman, G., Sewell, P., Neamtiu, I., “Mutatis Mutandis: Safe and Predictable Dynamic Software Updating”, ACM Transactions on Programming Languages and Systems, Vol. 29, No. 4, Aug. 2007, available at: http://doi.acm.org/10.1145/1255450.1255455

[8] Vandewoude, Y., Ebraert, P., Berbers, Y., D’Hondt, T., “Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates”, Software Engineering, IEEE Transactions on, Vol. 33, No. 12, Dec. 2007, str. 856–868.

[9] Subramanian, S., Hicks, M., McKinley, K. S., Dynamic Software Updates: A VM-centric Approach, 2009.

[10] Bashar Gharaibeh, Hridesh Rajan, J. Morris Chang, "Towards Efficient Java Virtual Machine Support for Dynamic Deployment of Inter-type Declarations", Iowa State University, Computer Science Technical Report 321, 2010.

[11] Hicks, M., Nettles, S., "Dynamic Software Updating", ACM Transactions on Programming Languages and Systems, Vol. 27, No. 6, Nov. 2005, str. 1049–1096, available at: http://doi.acm.org/10.1145/1108970.1108971

[12] Gregersen, A. R., Jørgensen, B. N., "Dynamic Update of Java Applications-Balancing Change Flexibility vs Programming Transparency", The Journal of Software: Evolution and Process, Vol. 21, No. 2, Mar. 2009, str. 81–112, available at: http://dx.doi.org/10.1002/smr.v21:2

[13] Bierman, G., Parkinson, M., Noble, J., "UpgradeJ: Incremental Typechecking for Class Upgrades", in ECOOP 2008 – Object-Oriented Programming: 22nd European Conference Paphos, Cyprus, July 7-11, 2008 Proceedings, Vitek, J., (ur.). Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, str. 235–259, available at: http://dx.doi.org/10.1007/978-3-540-70592-5_11

[14] Gu, T., Cao, C., Xu, C., Ma, X., Zhang, L., Lu, J., "Javelus: A Low Disruptive Approach to Dynamic Software Updates", in Software Engineering Conference (APSEC), 2012 19th Asia-Pacific, Vol. 1, Dec. 2012, str. 527–536.

[15] Würthinger, T., Wimmer, C., Stadler, L., "Unrestricted and safe dynamic code evolution for Java", Science of Computer Programming, Vol. 78, No. 5, 2013, str. 481 – 498, special section: Principles and Practice of Programming in Java 2009/2010 &amp; Special section: Self-Organizing Coordination, available at: http://www.sciencedirect.com/science/article/pii/S0167642311001456

[16] Pukall, M., Grebhahn, A., Schröter, R., Kästner, C., Cazzola, W., Götz, S., "JavAdaptor: Unrestricted Dynamic Software Updates for Java", in Proceedings of the 33rd International Conference on Software Engineering, ser. ICSE '11. New York, NY, USA: ACM, 2011, str. 989–991, available at: http://doi.acm.org/10.1145/1985793.1985970

[17] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J., "Aspect-oriented programming", in ECOOP'97 — Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings, Akşit, M., Matsuoka, S., (ur.). Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, str. 220–242, available at: http://dx.doi.org/10.1007/BFb0053381

[18] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G., “An Overview of AspectJ”, in Proceedings of the 15th European Conference on Object-Oriented Programming, ser. ECOOP ’01. London, UK, UK: Springer-Verlag, 2001, str. 327–353, available at: http://dl.acm.org/citation.cfm?id=646158.680006

[19] Bockisch, C., Haupt, M., Mezini, M., Ostermann, K., “Virtual Machine Support for Dynamic Join Points”, in Proceedings of the 3rd International Conference on Aspect-oriented Software Development, ser. AOSD ’04. New York, NY, USA: ACM, 2004, str. 83–92, available at: http://doi.acm.org/10.1145/976270.976282

[20] Popovici, A., Alonso, G., Gross, T., “Just-In-Time Aspects: Efficient Dynamic Weaving for Java”, in In Proceedings of the 2nd international conference on Aspect-oriented software development. ACM Press, 2003, str. 100–109.

[21] Ansaloni, D., Binder, W., Moret, P., Villazón, A., “Dynamic Aspect-Oriented Programming in Java: The HotWave Experience”, in Transactions on Aspect-Oriented Software Development IX, Leavens, G. T., Chiba, S., Haupt, M., Ostermann, K., Wohlstadter, E., (ur.). Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, str. 92–122, available at: http://dx.doi.org/10.1007/978-3-642-35551-6_3

[22] Ponge, J., Mouël, F. L., “JooFlux: Hijacking Java 7 InvokeDynamic To Support Live Code Modifications”, CoRR, Vol. abs/1210.1039, 2012, available at: http://arxiv.org/abs/1210.1039

[23] Suvée, D., Vanderperren, W., Jonckers, V., “JAsCo: An Aspect-oriented Approach Tailored for Component Based Software Development”, in Proceedings of the 2Nd International Conference on Aspect-oriented Software Development, ser. AOSD ’03. New York, NY, USA: ACM, 2003, str. 21–29, available at: http://doi.acm.org/10.1145/643603.643606

[24] Cech Previtali, S., Gross, T. R., “Aspect-based Dynamic Software Updating: A Model and Its Empirical Evaluation”, in Proceedings of the Tenth International Conference on Aspect-oriented Software Development, ser. AOSD ’11. New York, NY, USA: ACM, 2011, str. 105–116, available at: http://doi.acm.org/10.1145/1960275.1960289

[25] Marija Katic, “Dynamic Evolution of Aspect Oriented Software”, PhD thesis, Faculty of Electrical Engineering and Computing, University of Zagreb, 2013.

[26] Previtali, S. C., “Dynamic Updates: Another Middleware Service?”, in Proceedings of the 1st Workshop on Middleware-application Interaction: In Conjunction with Euro-Sys 2007, ser. MAI ’07. New York, NY, USA: ACM, 2007, str. 49–54, available at: http://doi.acm.org/10.1145/1238828.1238841

[27] Nicoara, A., Alonso, G., Roscoe, T., "Controlled, Systematic, and Efficient Code Replacement for Running Java Programs", ACM SIGOPS Operating Systems Review, Vol. 42, No. 4, Apr. 2008, str. 233–246, available at: http://doi.acm.org/10.1145/1357010.1352617

[28] Gregersen, A. R., Jørgensen, B. N., "Run-time Phenomena in Dynamic Software Updating: Causes and Effects", in Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution, ser. IWPSE-EVOL '11. New York, NY, USA: ACM, 2011, str. 6–15, available at: http://doi.acm.org/10.1145/2024445.2024448

[29] Seifzadeh, H., Abolhassani, H., Moshkenani, M. S., "A survey of dynamic software updating", Journal of Software: Evolution and Process, Vol. 25, No. 5, 2013, str. 535–568, available at: http://dx.doi.org/10.1002/smr.1556

[30] Mlinarić, D. *et al.*, "Challenges in dynamic software updating", TEM Journal, Vol. 9, No. 1, 2020, str. 117–128.

[31] Frieder, O., Segal, M. E., "On Dynamically Updating a Computer Program: From Concept to Prototype", The Journal of Systems and Software, Vol. 14, No. 2, Feb. 1991, str. 111–128, available at: http://dx.doi.org/10.1016/0164-1212(91)90096-O

[32] Sprenkels, R., Pras, A., "Service level agreements", Internet NG D, Vol. 2, 2001, str. 7.

[33] Neamtiu, I., Bardin, J., Uddin, M. R., Lin, D.-Y., Bhattacharya, P., "Improving Cloud Availability with On-the-fly Schema Updates", in Proceedings of the 19th International Conference on Management of Data, ser. COMAD '13. Mumbai, India, India: Computer Society of India, 2013, str. 24–34, available at: http://dl.acm.org/citation.cfm?id=2694476.2694487

[34] "Debugging with the Eclipse Platform", available at: http://www.ibm.com/developerworks/library/os-ecbug/ May 2007.

[35] Microsoft, "Edit and Continue", available at: https://msdn.microsoft.com/en-us/library/bcew296c.aspx 2015.

[36] Payer, M., Bluntschli, B., Gross, T. R., "DynSec: On-the-fly code rewriting and repair", Presented as part of the 5th Workshop on Hot Topics in Software Upgrades, 2013, str. 115–126.

[37] Pukall, M., Kästner, C., Götz, S., Cazzola, W., Saake, G., "Flexible Runtime Program Adaptations in Java - A Comparison", School of Computer Science, University of Magdeburg, Germany, Tech. Rep. 14, Nov. 2009.

[38] Hayden, C., Smith, E., Hardisty, E., Hicks, M., Foster, J., “Evaluating Dynamic Software Update Safety Using Systematic Testing”, Software Engineering, IEEE Transactions on, Vol. 38, No. 6, Nov. 2012, str. 1340–1354.

[39] Cook, R. P., Lee, I., “DYMOS: A Dynamic Modification System”, SIGSOFT Softw. Eng. Notes, Vol. 8, No. 4, Mar. 1983, str. 201–202, available at: http://doi.acm.org/10.1145/1006140.1006188

[40] Dmitriev, M., “Safe evolution of large and long-lived java applications”, Ph.D. Thesis, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, 2001.

[41] Neamtiu, I., Hicks, M., “Safe and Timely Updates to Multi-threaded Programs”, SIGPLAN Not., Vol. 44, No. 6, Jun. 2009, str. 13–24, available at: http://doi.acm.org/10.1145/1543135.1542479

[42] Segal, M. E., “Dynamic Program Updating in a Distributed Computer System”, Doktorski rad, University of Michigan, Ann Arbor, MI, USA, 1989, uMI Order No: GAX90-01706.

[43] Ebraert, P., Schippers, H., Molderez, T., Janssens, D., “Safely Updating Running Software: Tranquility at the Object Level”, in Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution, ser. RAM-SE ’10. New York, NY, USA: ACM, 2010, str. 2:1–2:6, available at: http://doi.acm.org/10.1145/1890683.1890685

[44] Wernli, E., Lungu, M., Nierstrasz, O., “Incremental Dynamic Updates with First-Class Contexts”, in Objects, Models, Components, Patterns, ser. Lecture Notes in Computer Science, Furia, C., Nanz, S., (ur.). Springer Berlin Heidelberg, 2012, Vol. 7304, str. 304–319, available at: http://dx.doi.org/10.1007/978-3-642-30561-0_21

[45] Duggan, D., “Type-based Hot Swapping of Running Modules (Extended Abstract)”, SIGPLAN Not., Vol. 36, No. 10, Oct. 2001, str. 62–73, available at: http://doi.acm.org/10.1145/507546.507645

[46] Wernli, E., “Theseus: Whole Updates of Java Server Applications”, in Proceedings of the 4th International Workshop on Hot Topics in Software Upgrades, ser. HotSWUp ’12. Piscataway, NJ, USA: IEEE Press, 2012, str. 41–45, available at: http://dl.acm.org/citation.cfm?id=2664350.2664359

[47] Hashimoto, M., “A Method of Safety Analysis for Runtime Code Update”, in Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, ser. Lecture Notes in Computer Science, Okada, M., Satoh, I., (ur.). Springer Berlin Heidelberg, 2007, Vol. 4435, str. 60–74, available at: http://dx.doi.org/10.1007/978-3-540-77505-8_6

[48] “TIOBE Index | TIOBE - The Software Quality Company”, available at: https://www.tiobe.com/tiobe-index/

[49] Noubissi, A., Iguchi-Cartigny, J., Lanet, J., “Hot updates for Java based smart cards”, in Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on, Apr. 2011, str. 168–173.

[50] Bhattacharya, P., Neamtiu, I., “Dynamic Updates for Web and Cloud Applications”, in Proceedings of the 2010 Workshop on Analysis and Programming Languages for Web Applications and Cloud Applications, ser. APLWACA ’10. New York, NY, USA: ACM, 2010, str. 21–25, available at: http://doi.acm.org/10.1145/1810139.1810143

[51] Lin, D.-Y., Neamtiu, I., “Collateral Evolution of Applications and Databases”, in Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops, ser. IWPSE-Evol ’09. New York, NY, USA: ACM, 2009, str. 31–40, available at: http://doi.acm.org/10.1145/1595808.1595817

[52] Kim, D. K., Tilevich, E., “Overcoming JVM HotSwap Constraints via Binary Rewriting”, in Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades, ser. HotSWUp ’08. New York, NY, USA: ACM, 2008, str. 5:1–5:5, available at: http://doi.acm.org/10.1145/1490283.1490290

[53] Neamtiu, I., Hicks, M., Foster, J. S., Pratikakis, P., “Contextual Effects for Version-consistent Dynamic Software Updating and Safe Concurrent Programming”, SIGPLAN Not., Vol. 43, No. 1, Jan. 2008, str. 37–49, available at: http://doi.acm.org/10.1145/1328897.1328447

[54] Mlinaric, D., Mornar, V., “Dynamic Software Updating in Java: Comparing Concepts and Resource Demands”, in Companion to the First International Conference on the Art, Science and Engineering of Programming, ser. Programming ’17. New York, NY, USA: Association for Computing Machinery, 2017, event-place: Brussels, Belgium, available at: https://doi.org/10.1145/3079368.3079389

[55] Katić, M., Fertalj, K., “Model for Dynamic Evolution of Aspect-Oriented Software”, in Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on, Mar. 2011, str. 377–380.

[56] “JDPA Enhancements 1.4”, available at: https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/enhancements1.4.html#hotswap

[57] Gregersen, A. R., Simon, D., Jørgensen, B. N., “Towards a Dynamic-update-enabled JVM”, in Proceedings of the Workshop on AOP and Meta-Data for Software Evolution, ser. RAM-SE ’09. New York, NY, USA: ACM, 2009, str. 2:1–2:7, available at: http://doi.acm.org/10.1145/1562860.1562862

[58] Battista, G. D., Eades, P., Tamassia, R., Tollis, I. G., Graph Drawing: Algorithms for the Visualization of Graphs, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1998.

[59] Herman, I., Melancon, G., Marshall, M. S., “Graph visualization and navigation in information visualization: A survey”, IEEE Transactions on Visualization and Computer Graphics, Vol. 6, No. 1, Jan. 2000, str. 24–43.

[60] “Graphviz - Graph Visualization Software”, available at: http://www.graphviz.org/ Oct. 2019.

[61] Himsolt, M., “GML: A portable graph file format”, Universität Passau, 1997, available at: http://svn.bigcat.unimaas.nl/pvplugins/GML/trunk/docs/gml-technical-report.pdf

[62] “TGF”, available at: http://docs.yworks.com/yfiles/doc/developers-guide/tgf.html

[63] “A libre lightweight streaming front-end for Android.: TeamNewPipe/NewPipe”, available at: https://github.com/TeamNewPipe/NewPipe Original-date: 2015-09-03T23:39:26Z.

[64] Mlinarić, D., Milašinović, B., Mornar, V., “Tree Inheritance Distance”, IEEE Access, Vol. 8, 2020, str. 52 489–52 504, publisher: IEEE.

[65] Singh, G. B., “Single Versus Multiple Inheritance in Object Oriented Programming”, SIGPLAN OOPS Messenger, Vol. 6, No. 1, Jan. 1995, str. 30–39, available at: http://doi.acm.org/10.1145/209866.209871

[66] Sakkinen, M., “Disciplined Inheritance.”, in ECOOP, Vol. 89, 1989, str. 39–56.

[67] Shasha, D., Wang, J. T., Kaizhong Zhang, Shih, F. Y., “Exact and approximate algorithms for unordered tree matching”, IEEE Transactions on Systems, Man, and Cybernetics, Vol. 24, No. 4, Apr. 1994, str. 668–678.

[68] Chawathe, S. S., Rajaraman, A., Garcia-Molina, H., Widom, J., “Change Detection in Hierarchically Structured Information”, SIGMOD Rec., Vol. 25, No. 2, Jun. 1996, str. 493–504, available at: http://doi.acm.org/10.1145/235968.233366

[69] Daelemans, W., De Smedt, K., Gazdar, G., “Inheritance in Natural Language Processing”, Computer Linguistics, Vol. 18, No. 2, Jun. 1992, str. 205–218, available at: http://dl.acm.org/citation.cfm?id=142235.142243

[70] Hashimoto, M., Mori, A., “Diff/TS: A Tool for Fine-Grained Structural Change Analysis”, in 2008 15th Working Conference on Reverse Engineering, Oct. 2008, str. 279–288.

[71] Shapiro, B. A., Zhang, K., “Comparing multiple RNA secondary structures using tree comparisons”, Bioinformatics, Vol. 6, No. 4, 1990, str. 309–318, available at: https://doi.org/10.1093/bioinformatics/6.4.309

[72] Dulucq, S., Tichit, L., “RNA secondary structure comparison: Exact analysis of the Zhang–Shasha tree edit algorithm”, Theoretical Computer Science, Vol. 306, No. 1, 2003, str. 471 – 484, available at: http://www.sciencedirect.com/science/article/pii/S0304397503003232

[73] Yang, W., “Identifying syntactic differences between two programs”, Software: Practice and Experience, Vol. 21, No. 7, 1991, str. 739–755, available at: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380210706

[74] Neamtiu, I., Foster, J. S., Hicks, M., “Understanding Source Code Evolution Using Abstract Syntax Tree Matching”, SIGSOFT Software Engineering Notes, Vol. 30, No. 4, May 2005, str. 1–5, available at: http://doi.acm.org/10.1145/1082983.1083143

[75] Sager, T., Bernstein, A., Pinzger, M., Kiefer, C., “Detecting Similar Java Classes Using Tree Algorithms”, in Proceedings of the 2006 International Workshop on Mining Software Repositories, ser. MSR ’06. New York, NY, USA: ACM, 2006, str. 65–71, available at: http://doi.acm.org/10.1145/1137983.1138000

[76] Roy, C. K., Cordy, J. R., Koschke, R., “Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach”, Science of Computer Programming, Vol. 74, No. 7, May 2009, str. 470–495, available at: http://dx.doi.org/10.1016/j.scico.2009.02.007

[77] Böhme, M., Roychoudhury, A., Oliveira, B. C. d. S., “Chapter 2 - Regression Testing of Evolving Programs”, ser. Advances in Computers, Memon, A., (ur.). Elsevier, 2013, Vol. 89, str. 53 – 88, available at: http://www.sciencedirect.com/science/article/pii/B9780124080942000023

[78] Valiente, G., Algorithms on Trees and Graphs. Berlin, Heidelberg: Springer-Verlag, 2002.

[79] Riesen, K., Structural Pattern Recognition with Graph Edit Distance: Approximation Algorithms and Applications, 1st ed. Cham, Switzerland: Springer Publishing Company, Incorporated, 2016.

[80] Kobler, J., Schöning, U., Torán, J., The Graph Isomorphism Problem: Its Structural Complexity. Springer Science & Business Media, 2012.

[81] Gao, X., Xiao, B., Tao, D., Li, X., "A survey of graph edit distance", Pattern Analysis and Applications, Vol. 13, No. 1, Feb. 2010, str. 113–129, available at: https://doi.org/10.1007/s10044-008-0141-y

[82] Wang, J. T. L., Zhang, K., "Finding similar consensus between trees: An algorithm and a distance hierarchy", Pattern Recognition, Vol. 34, No. 1, 2001, str. 127 – 137, available at: http://www.sciencedirect.com/science/article/pii/S0031320399001995

[83] Zhang, K., Shasha, D., "Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems", SIAM Journal on Computing, Vol. 18, No. 6, Dec. 1989, str. 1245–1262, available at: http://dx.doi.org/10.1137/0218082

[84] Tai, K.-C., "The Tree-to-Tree Correction Problem", Journal of the ACM, Vol. 26, No. 3, Jul. 1979, str. 422–433, available at: http://doi.acm.org/10.1145/322139.322143

[85] Wagner, R. A., Fischer, M. J., "The String-to-String Correction Problem", Journal of the ACM, Vol. 21, No. 1, Jan. 1974, str. 168–173, available at: http://doi.acm.org/10.1145/321796.321811

[86] Selkow, S. M., "The tree-to-tree editing problem", Information Processing Letters, Vol. 6, No. 6, 1977, str. 184 – 186, available at: http://www.sciencedirect.com/science/article/pii/0020019077900643

[87] Bille, P., "A survey on tree edit distance and related problems", Theoretical Computer Science, Vol. 337, No. 1, 2005, str. 217 - 239, available at: http://www.sciencedirect.com/science/article/pii/S0304397505000174

[88] Chidamber, S. R., Kemerer, C. F., "A metrics suite for object oriented design", IEEE Transactions on Software Engineering, Vol. 20, No. 6, Jun. 1994, str. 476–493.

[89] Šelajev, O., Gregersen, A., "Using runtime state analysis to decide applicability of dynamic software updates", in Proceedings of the 12th International Conference on Software Technologies, 2017, str. 38–49.

[90] Šelajev, O., Gregersen, A. R., "Genrih, a runtime state analysis system for deciding the applicability of dynamic software updates", in International Conference on Software Technologies. Springer, 2017, str. 135–159.

[91] Gharaibeh, B., Rajan, H., Chang, J.M., "A quantitative cost/benefit analysis for dynamic updating", Iowa State University, Technical Report, 2009.

[92] Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M.,

Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B., “The DaCapo Benchmarks: Java Benchmarking Development and Analysis”, SIGPLAN Not., Vol. 41, No. 10, Oct. 2006, str. 169–190, available at: http://doi.acm.org/10.1145/1167515.1167488

[93] “APM, (Application Performance Management) tool for large-scale distributed systems written in Java. : naver/pinpoint”, available at: https://github.com/naver/pinpoint Original-date: 2014-10-20T09:27:22Z.

[94] “Javassist by jboss-javassist”, available at: https://www.javassist.org/

[95] “Welcome to NetBeans”, available at: https://netbeans.org/

[96] Commons, A., “Bcel: Byte code engineering library”, URL http://commons. apache. org/bcel, 2011.

[97] “ASM”, available at: https://asm.ow2.io/

[98] Mlinarić, D., Mornar, V., Milašinović, B., “Generating Trees for Comparison”, Computers, Vol. 9, No. 2, 2020, str. 35, publisher: Multidisciplinary Digital Publishing Institute.

Biography

Danijel Mlinarić was born in 1985 in Zagreb, Croatia. He received his M.Sc. degree in computing from the Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia, in 2009. From 2009 to 2014, he was a Research Associate with the same institution at the Department of Applied Computing, working on national information systems of application and enrollment in secondary schools and higher education institutions. From 2014, he has worked as a Research and Teaching Assistant with the Faculty of Electrical Engineering and Computing, University of Zagreb. His research interests include software engineering focused on software evolution, program analysis, and dynamic software updating. He has contributed to publications in international peer-reviewed journals and to international conferences. He is a member of the IEEE society.

List of published papers

1. Mlinarić, Danijel, Mornar Vedran, “Dynamic Software Updating in Java: Comparing Concepts and Resource Demands”, Companion to the first International Conference on the Art, Science and Engineering of Programming, April, 2017, pp. 12:1-12:6.

Journal papers

1. Mlinarić, Danijel, Mornar, Vedran, Milašinović, Boris, ”Generating Trees for Comparison”, Computers, Vol. 9, Issue 2, April, 2020, pp. 35.
2. Mlinarić, Danijel, Milašinović, Boris, Mornar, Vedran, ”Tree Inheritance Distance”, IEEE Access, Vol. 8, March, 2020, pp. 52489-52504.
3. Mlinarić, Danijel, “Challenges in Dynamic Software Updating”, TEM Journal, Vol. 9, Issue 1, February, 2020, pp. 117-128.
4. Brčić, Mario, Mlinarić, Danijel, “Tracking Predictive Gantt Chart for Proactive Rescheduling in Stochastic Resource Constrained Project Scheduling”, Journal of Information and Organizational Sciences, Vol. 42, No 2, December, 2018, pp. 179-192.

Životopis

Danijel Mlinarić rođen je 1985. u Zagrebu. Diplomirao je na Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu 2009. i stekao zvanje diplomirani inženjer računarstva. Od 2009. do 2014. zaposlen je kao znanstveni suradnik na istoj instituciji na Zavodu za primijenjeno računarstvo, gdje radi na nacionalnim informacijskim sustavima prijave i upisa u srednje škole i fakultete. Od 2014., radi kao asistent na Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu. Njegov istraživački interes uključuje programsko inženjerstvo fokusirano na evoluciju softvera, analizu programa i dinamičko ažuriranje softvera. Objavio je nekoliko radova u časopisima s međunarodnom recenzijom i sudjelovao na međunarodnoj konferenciji. Član je IEEE organizacije.