Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
FIIT-5220-64370

Bc. Juraj Kostolanský

ANALÝZA EVOLÚCIE ZDROJOVÝCH TEXTOV S VYUŽITÍM ABSTRAKTNÝCH SYNTAKTICKÝCH STROMOV

Diplomová práca

Vedúci práce: Ing. Peter Lacko, PhD.
máj 2014

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
FIIT-5220-64370

Bc. Juraj Kostolanský

ANALÝZA EVOLÚCIE ZDROJOVÝCH TEXTOV S VYUŽITÍM ABSTRAKTNÝCH SYNTAKTICKÝCH STROMOV

Diplomová práca

Študijný program: Softvérové inžinierstvo
Študijný odbor: 9.2.5 Softvérové inžinierstvo
Miesto vypracovania: Ústav informatiky a softvérového inžinierstva, FIIT STU BA
Vedúci práce: Ing. Peter Lacko, PhD.
máj 2014

S T U
F I I T
SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
Fakulta informatiky a informačných technológií

Zadanie diplomovej práce

Meno študenta: **Bc. Juraj Kostolanský**

Študijný program: Softvérové inžinierstvo

Študijný odbor: Softvérové inžinierstvo

Názov práce: **Analýza evolúcie zdrojových textov s využitím abstraktných syntaktických stromov**

Samostatnou výskumnou a vývojovou činnosťou v rámci predmetov Diplomový projekt I, II, III vypracujte diplomovú prácu na tému, vyjadrenú vyššie uvedeným názvom tak, aby ste dosiahli tieto ciele:

Všeobecný cieľ:

Vypracovaním diplomovej práce preukážte, ako ste si osvojili metódy a postupy riešenia relatívne rozsiahlych projektov, schopnosť samostatne a tvorivo riešiť zložité úlohy aj výskumného charakteru v súlade so súčasnými metódami a postupmi študovaného odboru využívanými v príslušnej oblasti a schopnosť samostatne, tvorivo a kriticky pristupovať k analýze možných riešení a k tvorbe modelov.

Špecifický cieľ:

Vytvorte riešenie, zodpovedúce návrhu textu zadania, ktorý je prílohou tohto zadania. Návrh bližšie opisuje tému vyjadrenú názvom. Tento opis je záväzný, má však rámcový charakter, aby vznikol dostatočný priestor pre Vašu tvorivosť.

Riaďte sa pokynmi Vášho vedúceho.

Pokiaľ v priebehu riešenia, opierajúc sa o hlbšie poznanie súčasného stavu v príslušnej oblasti alebo o priebežné výsledky Vášho riešenia alebo o iné závažné skutočnosti, dospejete spoločne s Vaším vedúcim k presvedčeniu, že niečo v texte zadania a/alebo v názve by sa malo zmeniť, navrhnite zmenu. Zmena je spravidla možná len pri dosiahnutí kontrolného bodu.

Miesto vypracovania: Ústav informatiky a softvérového inžinierstva FIIT STU v Bratislave

Vedúci práce: **Ing. Peter Lacko, PhD.**

Termíny odovzdania:

podľa harmonogramu štúdia platného pre semester, v ktorom máte príslušný predmet (Diplomový projekt I, II, III) absolvovať podľa Vášho študijného plánu

Predmety odovzdania:

V každom predmete dokument podľa pokynov na www.fiit.stuba.sk v časti:
home > Informácie o > štúdiu > organizácia štúdia > diplomový projekt

V Bratislave dňa 18. 2. 2013

prof. Ing. Pavol Návrat, PhD.
riaditeľ Ústavu informatiky a softvérového inžinierstva

STU FIIT
SLOVENSKÁ TECHNICKÁ
UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY
A INFORMAČNÝCH TECHNOLÓGIÍ

Návrh zadania diplomovej práce

Finálna verzia do diplomovej práce [1]

Študent:

Meno, priezvisko, tituly:	Juraj Kostolanský, Bc.
Študijný program:	Softvérové inžinierstvo
Kontakt:	juraj.kostolansky@gmail.com

Výskumník:

Meno, priezvisko, tituly:	Peter Lacko, Ing. PhD.

Projekt:

Názov:	Analýza evolúcie zdrojových textov s využitím abstraktných syntaktických stromov
Názov v angličtine:	Analysis of source code evolution using abstract syntax tree
Miesto vypracovania:	Ústav informatiky a softvérového inžinierstva, FIIT STU, Bratislava
Oblasť problematiky:	Analýza zdrojových textov programov

Text návrhu zadania[2]

S neustále sa zväčšujúcou komplexnosťou softvérových systémov narastá aj počet riadkov zdrojových textov, ktoré sú potrebné pre zabezpečenie ich požadovanej funkcionality. Monitorovanie časovej evolúcie takto rozsiahlych systémov je často veľmi obtiažne. Aj preto sa v dnešnej dobe venuje nemalé úsilie snahe získavať rôzne informácie zo zdrojových textov softvérových systémov.

Analyzuje možnosti využitia abstraktných syntaktických stromov pri porovnávaní zdrojových textov programov. Zamerajte sa na porovnávanie rôznych verzií zdrojového textu toho istého programu. Zistite, aké informácie o zmenách zdrojových textov v čase je možné získať s využitím takejto analýzy a ako je možné takto získané informácie následne využiť. Navrhnite spôsob, akým sa dá porovnávanie rôznych verzií zdrojových textov pomocou abstraktných syntaktických stromov aplikovať pri získavaní informácií zo systémov na správu revízií zdrojových textov programov.

Navrhnuté riešenie overte implementovaním softvérového prototypu, ktorý bude slúžiť na analýzu časovej evolúcie zdrojových textov programov. Experimentujte s vytvorenou aplikáciou v kontexte dostatočne veľkých softvérových systémov s otvoreným zdrojovým textom.

---

[1] Vytlačiť obojstranne na jeden list papiera

[2] 150-200 slov (1200-1700 znakov), ktoré opisujú výskumný problém v kontexte súčasného stavu vrátane motivácie a smerov riešenia

Literatúra[3]

- Giger, E., Pinzger, M., Gall, H. C.: Comparing fine-grained source code changes and code churn for bug prediction. In: Proceedings of the 8th Working Conference on Mining Software Repositories. ACM, 2011, pp. 83-92.
- Neamtiu, I., Foster, J. S., Hicks, M.: Understanding source code evolution using abstract syntax tree matching. In: Proceedings of the 2005 international workshop on Mining software repositories. ACM, 2005, pp. 1-5.

Vyššie je uvedený návrh diplomového projektu, ktorý vypracoval(a) Bc. Juraj Kostolanský, konzultoval(a) a osvojil(a) si ho Ing. Peter Lacko, PhD. a súhlasí, že bude takýto projekt viesť v prípade, že bude pridelený tomuto študentovi.

V Bratislave dňa 13.1.2013

Podpis študenta

Podpis výskumníka

Vyjadrenie garanta predmetov Diplomový projekt I, II, III

Návrh zadania schválený: áno / ~~nie~~[4]

Dňa: 1.2.2013

Podpis garanta predmetov

[3] 2 vedecké zdroje, každý v samostatnej rubrike a s údajmi zodpovedajúcimi bibliografickým odkazom podľa normy STN ISO 690, ktoré sa viažu k téme zadania a preukazujú výskumnú povahu problému a jeho aktuálnosť (uveďte všetky potrebné údaje na identifikáciu zdroja, pričom uprednostnite vedecké príspevky v časopisoch a medzinárodných konferenciách)

[4] Nehodiace sa prečiarknite

Anotácia

Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ
Študijný odbor: Softvérové inžinierstvo

Autor: Bc. Juraj Kostolanský
Diplomová práca: Analýza evolúcie zdrojových textov s využitím abstraktných syntaktických stromov
Vedúci diplomovej práce: Ing. Peter Lacko, PhD.

máj 2014

V súčasnosti sa venuje nemalé úsilie získavaniu rôznych informácií zo zdrojových textov programov, ako aj podporných nástrojov pre tvorbu softvéru. Medzi tieto informácie patria aj tie, ktoré súvisia so zmenami zdrojových textov v čase. Tie nachádzajú uplatnenie napríklad pri detekcii miest zdrojového textu náchylných na chyby, pri plánovaní projektu, manažmente rizík, rozdeľovaní ľudských zdrojov, monitorovaní projektu a mnohých ďalších.

V tejto práci sa zaoberáme problémom analýzy časovej evolúcie zdrojových textov softvérových systémov. Hierarchickú štruktúru zdrojových textov reprezentujeme prostredníctvom syntaktických stromov. Analyzujeme existujúce riešenia a prístupy v tejto oblasti a na základe vykonanej analýzy navrhujeme vlastné riešenie. To umožňuje detekciu zmien zdrojového textu porovnávaním dvoch verzií toho istého textu. Toto porovnávanie a následné párovanie príslušných vrcholov stromov je založené na troch rôznych prístupoch – párovanie založené na podstromoch, párovanie vnútorných vrcholov a párovanie listov. Navrhnutú metódu v práci overujeme na zdrojových textoch reálnych softvérov.

Annotation

Slovak University of Technology Bratislava
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES
Degree Course: Software Engineering

Author: Bc. Juraj Kostolanský
Master's Thesis: Analysis of source code evolution using abstract syntax tree
Supervisor: Ing. Peter Lacko, PhD.

2014, May

Nowadays is considerable effort devoted to mining information from various source codes, as well as from supporting tools for creating software systems. This information includes those related to changes over time in source code of programs. There are many possible applications for this information about software system evolution, such as bug prediction, project management, risk management, human resource allocation, project monitoring, and many others.

In this thesis we survey the problem of the source code analysis and its evolution. Syntax trees enable us to retain the hierarchical structure of a source code. We analyze existing solutions and approaches in this research area and we propose our own solution. This allows a detection of changes in source codes by comparing two versions of the same source code of a software system. Our solution is based on three different types of nodes matching – subtree matching, bottom-up matching of inner nodes and leaf matching. We tested our method on source codes from real-world software projects.

Čestné prehlásenie

Čestne prehlasujem, že na diplomovej práci som pracoval samostatne na základe vlastných teoretických a praktických poznatkov, konzultácií a štúdia odbornej literatúry, ktorej prehľad je uvedený v zozname použitej literatúry.

V Bratislave, dňa 13.5.2014 .....

podpis

Poďakovanie

Dovoľujem si poďakovať sa vedúcemu diplomovej práce Ing. Petrovi Lackovi, PhD. za odbornú pomoc a cenné rady, ktoré mi poskytol pri jej vypracovaní.

Obsah

Zoznam obrázkov

Zoznam tabuliek

Zoznam algoritmov

1 Úvod

Skúmaniu zdrojových textov rôznych softvérov sa ľudia venujú už od ich vzniku. So zväčšujúcou sa komplexnosťou softvérových systémov narastá aj počet riadkov zdrojových textov, ktoré sú potrebné pre zabezpečenie ich požadovanej funkcionality. Sledovanie a monitorovanie štruktúry a vývoja takto veľkých systémov je často veľmi obtiažne, najmä ak na nich pracuje väčšie množstvo programátorov. Aj preto sa v súčasnosti venuje nemalé úsilie snahe získavať rôzne informácie zo zdrojových textov programov a ďalších nástrojov, ktoré sa využívajú pri riadení tvorby softvéru (systém na riadenie revízií, systém na zaznamenávanie chýb a pod.). Medzi ne patria aj tie, ktoré súvisia so zmenami zdrojových textov programov v čase.

Analýza procesu tvorby softvéru a evolúcie zdrojových textov v čase v nemalej miere prispieva k porozumeniu vývoja softvéru ako celku. Tieto poznatky nachádzajú uplatnenie v rôznych oblastiach riadenia projektov, napr. v oblasti plánovania projektu, manažmentu rizík, rozdeľovania ľudských zdrojov, monitorovania projektu a mnohých ďalších.

Systémy riadenia revízií zdrojového textu zaznamenávajú množstvo informácií o evolúcii textu v čase. Hoci sa tieto systémy stali neoddeliteľnou súčasťou procesu tvorby rozsiahlych softvérových produktov, komplexných nástrojov na sledovanie a interpretáciu zmien v kontexte vývoja softvéru je stále pomerne málo. Väčšina systémov riadenia revízií využíva jednoduchý nástroj na zistenie zmien medzi jednotlivými verziami zdrojového textu. Ten je založený na porovnávaní verzií zdrojových textov na úrovni textového zápisu, avšak bez hlbšieho syntaktického a sémantického porozumenia. To môže byť užitočné na rýchle prezeranie malých zmien v texte, avšak limitujúce, niekedy až zavádzajúce, ak chceme sledovať zmeny na určitej úrovni abstrakcie. Ak si chceme vytvoriť ucelený pohľad nad zmenami v zdrojovom texte v čase, je preto nevyhnutné využiť iný spôsob porovnávania rôznych verzií zdrojového textu ako na úrovni zmien v texte. Východiskom môže byť využitie tzv. abstraktného syntaktického stromu. Takýto zápis zdrojového textu prostredníctvom stromu prináša viaceré výhody: umožňuje brať do úvahy väzby medzi jednotlivými časťami zdrojových textov a jeho hierarchickú štruktúru, ako aj využiť algoritmy známe v problematike stromových štruktúr.

V prvej časti tejto práce analyzujeme existujúce riešenia v problematike zisťovania zmien zdrojových textov v čase. Následne navrhujeme vlastné riešenie problému, ktoré je založené na porovnávaní dvoch verzií zdrojového textu reprezentovaných prostredníctvom stromových štruktúr. V práci taktiež uvádzame výsledky overovania navrhnutého riešenia a porovnanie s existujúcimi nástrojmi.

2 Systémy riadenia revízií ako zdroj zmien

Systémy riadenia revízií zdrojového textu (VCS, angl. *version control system*) sa v súčasnosti využívajú pri tvorbe takmer všetkých väčších softvérových projektov. Medzi najznámejšie z nich patria *Git*[1], *Apache Subversion*[2], *Perforce*[3], *CVS*[4] a ďalšie.

Okrem samotných zdrojových textov softvérových projektov zaznamenávajú tieto nástroje aj množstvo ďalších informácií o zdrojovom texte, okrem iného aj jeho zmeny v čase. Preto by sa mohli tieto nástroje javiť ako ideálny zdroj dát pre analýzu časovej evolúcie zdrojových textov. To však nemusí platiť vždy.

Rozdiely medzi zmenami, ktoré sú zaznamenané v systéme na riadenie revízií a skutočnými zmenami počas vývoja analyzovali Negara et al. [10]. K zaznamenaniu zmien druhého typu využili nástroj *CodingTracker*[5], ktorý sleduje aktivitu programátora pri písaní zdrojového textu. Autori poukazujú na skutočnosť, že jedna zmena vo VCS môže v skutočnosti znamenať hodiny, prípadne i dni práce, ktoré VCS zaznamenať nedokáže. Ide o tzv. zatienenie (angl. *shadowing*), kedy vývojár pracuje na jednej časti zdrojového textu, pričom do VCS sa dostane iba výsledná (posledná) zmena, ktorá zatieni predchádzajúce. Ďalším príkladom rozdielu je nepresnosť VCS. Jedna zaznamenaná zmena v zdrojovom texte môže byť v skutočnosti spojením napr. úpravy entity s refaktorovaním. VCS tiež nezaznamenáva korelujúce aktivity vývojára počas práce na zdrojovom texte programu (napr. spúšťanie testov, automatizovaných nástrojov pre refaktorovanie a pod.). Z výsledkov ich analýzy:

- 37% zmien nie je zaznamenaných vo VCS z dôvodu zatienenia inými zmenami
- 46% zmien sa týkalo súčasne úpravy entít aj refaktorovania

Vplyv týchto poznatkov na využívanie VCS ako zdroja informácií pre analýzu evolúcie zdrojových textov môže, ale aj nemusí byť významný. Dôležitý je totiž kontext analýzy dát. V prípade, že zmeny zdrojových textov využijeme na analýzu vynaloženého úsilia, hrá zatienenie zmien podstatnú úlohu. Chýbajúce informácie o ďalších aktivitách vývojára v podobe spúšťania testov môžu byť prekážkou pri detegovaní miest zdrojových textov náchylných na chyby. V inom prípade, kedy nás zaujíma celkový pohľad na vývoj softvéru, môžu byť tieto informácie irelevantné a prispievali by k nárastu dát, ktoré je potrebné spracovať. Systémy riadenia revízií však v každom prípade poskytujú dobrý základný zdroj informácií o zmenách zdrojových textov v čase, ktorý môže byť prípadne doplnený o ďalšie zdroje dát.

[1] `http://git-scm.com/`
[2] `http://subversion.apache.org/`
[3] `http://www.perforce.com/`
[4] `http://www.nongnu.org/cvs/`
[5] `http://codingtracker.web.engr.illinois.edu/`

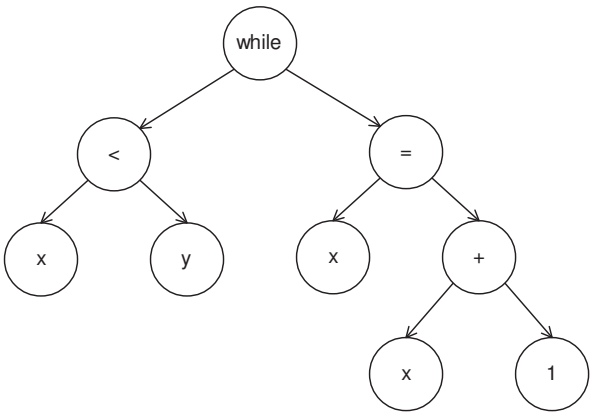
3 Abstraktné syntaktické stromy

Jadrom detekcie zmien v softvérových systémoch v čase je porovnávanie dvoch verzií zdrojového textu a následná detekcia vykonaných zmien. Jedným z možných prístupov je porovnávanie reťazcov textu na základe textovej podobnosti. Takýto prístup však nie je veľmi vhodný, nakoľko sa na zdrojový text pozerá ako na text bez hierarchického usporiadania.

Pre účely porovnávania zdrojových textov je preto vhodná taká jeho reprezentácia, ktorá berie do úvahy aj jeho hierarchické členenie vo forme syntaktických (prípadne aj sémantických) informácií a reprezentuje zdrojový text vo forme stromovej štruktúry. Takýto jeho zápis prináša viaceré výhody – nielenže umožňuje brať do úvahy väzby medzi jednotlivými časťami zdrojových textov, ale aj využiť algoritmy známe v problematike stromových štruktúr.

Takouto reprezentáciou zdrojového textu je syntaktický strom, ktorý je jeho štandardnou reprezentáciou využívanou väčšinou kompilátorov a interpreterov zdrojového textu. Každý uzol takéhoto stromu reprezentuje konštrukciu v pôvodnom zdrojovom texte. Syntaktický strom môže byť abstraktný alebo konkrétny.

Abstraktný syntaktický strom (AST) [1, 2] je (na rozdiel od konkrétneho syntaktického stromu) forma reprezentácie dát, ktorá je nezávislá od strojovo orientovaných štruktúr a od fyzickej reprezentácie dát. AST skrýva určité detailné informácie o zdrojovom texte (napríklad znamienka pre oddeľovanie výrazov, argumentov, zátvorky a pod.). Vďaka tomu je využívaný pre poskytnutie vyššej úrovne opisu programov. Príklad AST sa nachádza na obrázku 3.1.



Obr. 3.1: Príklad AST pre zdrojový text: `while(x < y) x = x + 1`.

3.1 Nástroje na tvorbu AST

Väčšina nástrojov na tvorbu abstraktných syntaktických stromov zo zdrojových textov je obmedzená na jeden konkrétny programovací jazyk, s ktorým dokáže pracovať. Ako sme zistili, nástrojov, ktoré podporujú jazyk Java, s ktorým budeme ďalej pracovať, je v súčasnosti pomerne málo. V tejto kapitole uvádzame opis niektorých vybraných generátorov, ktoré jazyk Java podporujú.

3.1.1 Java Development Tools

JDT[6] je sada nástrojov vývojového prostredia *Eclipse*, medzi ktoré patrí asi najznámejší voľne dostupný generátor abstraktných syntaktických stromov zo zdrojových textov v jazyku Java.

Počas analýzy sme si vyskúšali aj prácu s týmto nástrojom. Veľkou nevýhodou pre naše účely bol prístup k jednotlivým vrcholom generovaných stromov a nepodporovanie operácií, ktoré budeme nevyhnutne potrebovať. Nástroj totiž neumožňuje jednoduchý posun medzi vrcholmi stromu ani priamu iteráciu cez deti daného vnútorného vrchola. Prístup k vrcholom je riešený prostredníctvom návrhového vzoru Návštevník (angl. *Visitor*). Prípadným riešením tohto problému je predspracovanie vytvoreného stromu do vlastnej štruktúry, ktorá by tieto operácie poskytovala.

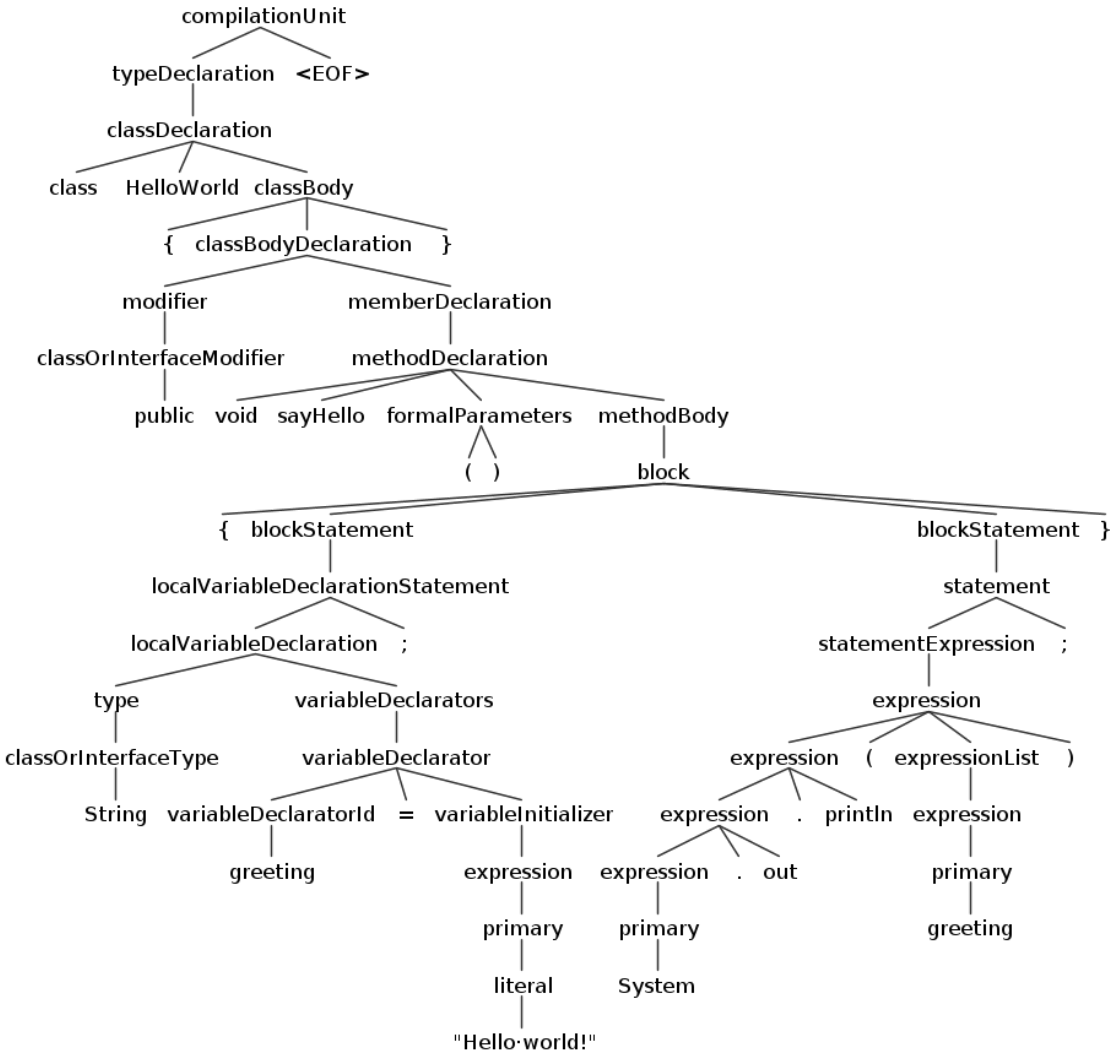
3.1.2 ANTLR

Nástroj ANTLR[7] (*ANother Tool for Language Recognition*) umožňuje na základe definície gramatiky ľubovoľného jazyka vygenerovať triedy v jazyku Java, ktoré okrem iného umožňujú aj vytváranie syntaktických stromov z viet z daného jazyka. Gramatiky najpopulárnejších jazykov sú vytvárané používateľmi tohto nástroja a voľne dostupné na stiahnutie[8].

Ukážka stromu vygenerovaného nástrojom ANTLR sa nachádza na obrázku 3.2, ktorý reprezentuje zdrojový text:

```
class HelloWorld {
  public void sayHello() {
    String greeting = "Hello world!";
    System.out.println(greeting);
  }
}
```

[6] http://www.eclipse.org/jdt/
[7] http://www.antlr.org/
[8] https://github.com/antlr/grammars-v4



Obr. 3.2: Strom vygenerovaný nástrojom ANTLR.

V bežných abstraktných syntaktických stromoch sú informácie ako modifikátor, hodnota alebo typ premennej uložené ako atribút toho istého vrchola reprezentujúceho túto premennú. V stromoch generovaných nástrojom ANTLR sú tieto informácie uložené v samostatných listoch. Tieto stromy taktiež nie sú abstraktné – neskrývajú detailné informácie o zdrojovom texte, akými sú napríklad interpunkčné znamienka pre oddeľovanie výrazov alebo argumentov, zátvorky a pod., ktoré sú v strome samostatnými listami.

Veľkou výhodou tohto nástroja je však jeho univerzálnosť. Nástroj dokáže vytvárať stromy nielen pre rôzne programovacie jazyky, ale aj pre iné jazyky, pre ktoré je možné zostaviť príslušnú gramatiku pre nástroj ANTLR (napr. XML, JSON a pod.). Mnohé z týchto gramatík sú už pritom vytvorené inými používateľmi tohto nástroja a voľne dostupné. Hoci sú tieto vytvárané iba komunitou používateľov, väčšina z nich sa udržiava aktuálna a kontinuálne sa vylepšuje.

3.1.3 JavaCC

Java Compiler Compiler[9] je podobný nástroju ANTLR. Zo špecifikácie jazyka umožňuje vygenerovať program v jazyku Java, ktorý dokáže parsovať vety z daného jazyka. Okrem toho taktiež poskytuje možnosť generovania stromov pre dané vety (pomocou nástroja *JJTree*). Dostupné gramatiky pre tento nástroj sú však vo väčšine prípadov niekoľko rokov staré.

3.1.4 DMS Software Reengineering Toolkit

Ide o sadu nástrojov[10] pre automatickú analýzu, modifikáciu, preklad a tvorbu softvérových systémov. Okrem iného umožňujú aj tvorbu abstraktných syntaktických stromov. Tieto nástroje dokážu pracovať s mnohými programovacími jazykmi, medzi ktorými je aj Java. Ich výhodou je podľa dostupných informácií možnosť spracúvania nielen kompletných zdrojových textov, ale aj ich podčastí (napr. samostatnej metódy triedy). Vygenerované stromy by mali byť podobné s tými z nástroja ANTLR, avšak bez špeciálnych syntaktických vrcholov ako napríklad bodky alebo zátvorky. Nakoľko však ide o proprietárny softvér, ktorého využitie je platené, podrobnejšie sme sa s ním nezaoberali.

3.2 Reprezentácie odvodené od AST

V analyzovaných existujúcich riešeniach nástrojov pre detegovanie zmien v zdrojových textoch sa okrem AST využívajú aj niektoré odvodené reprezentácie.

Abstraktný sémantický strom je štrukturálne podobný abstraktnému syntaktickému stromu, z ktorého je zvyčajne konštruovaný. Na rozdiel od neho je však rozšírený o špeciálne hrany z abstraktných sémantických grafov, ktoré reprezentujú sémantické informácie získané zo zdrojových textov programov (spájajú premenné s ich typom, referencie na premenné s ich deklaráciami a pod.). Tento typ reprezentácie využíva nástroj *Dex* [13].

FAMIX (*FAMOOS Information ExchangeModel*) je model pre reprezentáciu objektovo orientovaného zdrojového textu nezávislý od programovacieho jazyka. Prevodom AST do tohto modelu sa stráca určitá presnosť v dátach, nakoľko úroveň granularity je vzhľadom na univerzálnosť modelu redukovaná. Výhodou však je spomenutá jazyková nezávislosť. Model FAMIX využili Sager et al., ktorí vytvorili nástroj *Coogle* [4] pre zisťovanie miery podobností zdrojových textov v jazyku Java.

[9] `https://javacc.java.net/`

[10] `http://www.semanticdesigns.com/Products/DMS/DMSToolkit.html`

4 Detekcia zmien v zdrojovom texte

Na detekciu zmien medzi dvoma syntaktickými stromami (resp. dvoma verziami toho istého stromu) T_1 a T_2 sa môžeme pozerať ako na zisťovanie rozdielov medzi nimi. Tieto rozdiely potom reprezentujú zmeny, ktoré sa udiali pri ich úpravách v čase.

Ako formálny zápis rozdielu dvoch stromov sa využíva tzv. *zoznam zmien* (angl. *edit script*) [3].Zoznam zmien S medzi stromami T_1 a T_2 je taká sekvencia operácií zmien, po aplikovaní ktorej na strom T_1 získame strom T_2. Najčastejšie využívané typy operácií zmien v kontexte porovnávania dvoch stromových štruktúr sú:

- odstránenie vrchola
- vloženie nového vrchola
- úprava hodnoty vrchola
- presunutie podstromu

Nech v_1 , v_2 , ..., v_m sú deti vrchola u. Potom v_i nazývame i-tym dieťaťom tohto vrchola, $l(x)$ predstavuje označenie (typ) vrchola x (angl. *label*), $v(x)$ predstavuje jeho hodnotu (angl. *value*), $p(x)$ reprezentuje jeho rodiča v strome (angl. *parent*). Potom môžeme operácie zmien definovať nasledovne [5]:

DEL(x): odstránenie vrchola x zo stromu T_1. Operácia nemení relatívne poradie zvyšných vrcholov. Operácia odstraňuje iba listový vrchol. Pre odstránenie vnútorného vrchola stromu musia byť najprv postupne odstránení všetci jeho nasledovníci.

INS((x, l, v), y, k): vloženie nového vrchola x s označením l ($l = l(x)$) a hodnotou v ($v = v(x)$) do stromu T_1 ako k-te dieťa vrchola y.

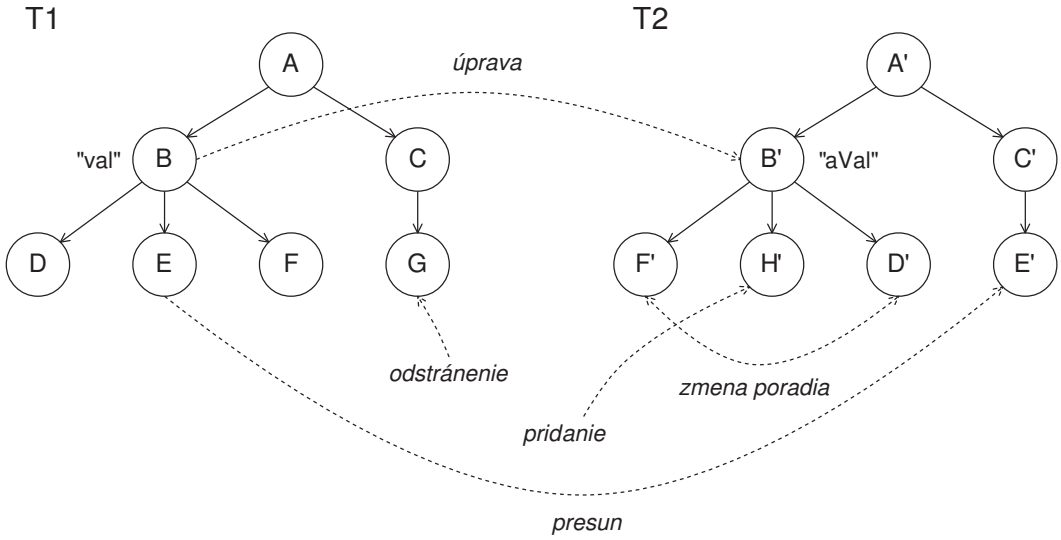
UPD(x, v): úprava hodnoty vrchola x v strome T_1 na hodnotu v (v strome T_2 platí $v = v(x)$).

MOV(x, y, k): presun podstromu s koreňom x. Vrchol x sa stáva k-tym dieťaťom vrchola y (v strome T_2 platí $y = p(x)$).

Tieto operácie znázorňuje obrázok 4.1. Keďže abstraktné syntaktické stromy radíme medzi usporiadané stromové štruktúry, môžeme medzi tieto zmeny zaradiť aj operáciu zmeny poradia vrcholov. Tá sa však v konečnom dôsledku dá reprezentovať operáciou presunutia podstromu, čím sa vytvorí požadované poradie.

Nakoľko takýchto zoznamov zmien pre dva stromy existuje spravidla väčšie množstvo (napr. jedna operácia úpravy vrchola môže byť nahradená operáciou odstránenia vrchola zo stromu T_1 nasledovaná operáciou vloženia „nového“ vrchola do stromu T_2),

zavádza sa pojem *ceny zoznamu zmien*. Jednotlivým operáciám zmien je možné priradiť ich cenu. Cena zoznamu zmien S je potom súčtom cien jednotlivých operácií zmien v S. Zoznam zmien medzi T_1 a T_2 , ktorý má minimálnu cenu, nazývame *optimálny*. Cena tohto zoznamu zmien reprezentuje vzdialenosť stromov T_1 a T_2.



Obr. 4.1: Operácie zmien pre AST [6].

Vzhľadom na malú frekvenciu výskytu operácie presunu podstromu a na veľký nárast dvojíc vrcholov pre porovnávanie kvôli detegovaniu tejto operácie sa autori niektorých analyzovaných nástrojov (kapitola 4.1) rozhodli tieto operácie nepodporovať [13].

4.1 Existujúce riešenia

Najrozšírenejšie nástroje na zistenie zmien medzi dvoma verziami zdrojového textu sú založené na porovnávaní textových reťazcov. Tieto sú často súčasťou systémov riadenia revízií, v ktorých umožňujú vizualizáciu zmien medzi jednotlivými verziami textu. Do tejto kategórie patria nástroje ako *GNU Diffutils*[11], *git-diff*[12], *Meld*[13] a iné. Ide však o univerzálne nástroje detekcie zmien, ktoré sú použiteľné nielen na zdrojové texty programov, ale aj na obyčajný text. Neberú totiž do úvahy syntaktickú ani sémantickú stránku zdrojového textu. Ich použitie je vhodné na rýchle prezeranie menších zmien, avšak ich úspešnosť na úsekoch zdrojových textov, ktoré boli zasiahnuté väčšími úpravami, je viac ako otázna.

V nasledujúcich častiach uvádzame podrobnejší opis niektorých existujúcich riešení, ktoré sa zaoberajú detekciou zmien v hierarchicky štruktúrovaných dátach, akými sú aj syntaktické stromy.

[11] `http://www.gnu.org/software/diffutils/`
[12] `http://git-scm.com/docs/git-diff`
[13] `http://meldmerge.org/`

4.1.1 LaDiff

Asi najznámejšou publikáciou z oblasti detegovania a reprezentácie zmien v hierarchicky štruktúrovaných informáciách je *Change detection in hierarchically structured information* [5]. Hierarchicky štruktúrované informácie môžu byť reprezentované ako usporiadané stromy. Chawathe et al. sa zaoberajú takými stromami, v ktorých majú jednotlivé vrcholy označenie a hodnotu.

Prvým z problémov, s ktorými sa museli vysporiadať, je nájdenie tzv. správneho párovania (angl. *good matching*), teda mapovania vrcholov prvého stromu na vrcholy druhého stromu. V prípade, kedy majú vrcholy stromu jednoznačný unikátny identifikátor, je riešenie tohto problému jednoduché. V prípade vrcholov bez takýchto identifikátorov ide o zložitejší problém.

Autori predpokladajú, že pre každý list v prvom strome sa nachádza maximálne jeden taký list v druhom strome, ktorý vyhovuje podmienke zhodnosti vrcholov. Pri párovaní sa dva vrcholy pokladajú za zhodné na základe týchto podmienok:

- dva listy sú zhodné, ak majú rovnaké označenie a dostatočne podobné hodnoty
- dva vnútorné vrcholy sú zhodné, ak majú rovnaké označenie a dostatočný relatívny počet spoločných spárovaných listov v ich podstromoch

V prvej časti algoritmus páruje listy stromov. Pre každé možné označenie listu stromu sa vytvoria postupnosti vrcholov daného označenia z oboch stromov. Algoritmus potom hľadá najdlhšiu spoločnú podpostupnosť dvojíc vrcholov, ktoré sú na základe stanovených kritérií považované za zhodné a z ktorých sa následne vytvára párovanie. Pre vrcholy, ktoré sa takto nepodarí spárovať, sa ďalej použije lineárne prehľadávanie. V druhej časti sa obdobným prístupom párujú vnútorné vrcholy.

Ďalším krokom po nájdení správneho párovania je vytvorenie zoznamu zmien, ktorý je tvorený štyrmi druhmi operácií: vloženie nového listu, odstránenie listu, úprava hodnoty vrchola a presunutie podstromu.

Navrhnutú metódu autori implementovali v podobe systému na detekciu, označovanie a zobrazovanie zmien dokumentov Latex – *LaDiff*.

4.1.2 ChangeDistiller

Na prácu Chawathe et al. nadviazali Fluri et al. [6]. Aplikovaním navrhnutej metódy detegovania zmien pre porovnávanie abstraktných syntaktických stromov vytvorili algoritmus *Change Distilling*. V ich práci uvádzajú hlavné nedostatky algoritmu, ktorý navrhli Chawathe et al., pre takéto využitie spolu s navrhovanými úpravami.

Hoci výsledný zoznam zmien je pri spracúvaní abstraktných syntaktických stromov vždy správny, nie vždy je aj minimálny.

Prvým nedostatkom je pôsob párovania listov. V pôvodnom algoritme sa využíva porovnávanie vrcholov založené na cene úpravy hodnoty vrchola. Správne porovnávanie hodnôt vrcholov sa pritom ukázalo ako kľúčové. Fluri et al. skúmajú viaceré možnosti výpočtu podobnosti dvoch textových reťazcov v kontexte zdrojových textov. Najlepšie výsledky získali s využitím podobnosti na základe n-gramov (v tomto prípade bigramov) a modifikovaného Jaccardovho koeficientu.

Druhým nedostatkom je spôsob párovania vnútorných vrcholov. Ak sa chybne spáruje list malého podstromu, nastáva šírenie tejto chyby smerom nahor, k rodičom. Táto propagácia je spôsobená tým, ako párujú vnútorné vrcholy stromov. V pôvodnom algoritme sa totiž pri vnútorných vrcholoch berie do úvahy iba počet spoločných listov v príslušných podstromoch. To je v prípade Latex dokumentov opodstatnené, pretože vnútorné vrcholy spravidla nesú iba informáciu o štruktúre dokumentu. V prípade zdrojových textov programov je však situácia iná – vnútorné vrcholy obsahujú aj sémantické informácie.

Ako riešenie tohto nedostatku berú Fluri et al. do úvahy nielen párovanie listov príslušných podstromov, ale aj vnútorných vrcholov. Ďalej navrhujú využitie porovnávania hodnôt aj pre vnútorné vrcholy stromov. Pre malé podstromy (počet listov podstromu ≤ 4) taktiež navrhujú znížiť hranicu pre podobnosť vnútorných vrcholov, nad ktorou sa dva vrcholy považujú za zhodné.

Ďalším nedostatok je predpoklad, že pre každý list prvého (pôvodného) stromu sa v druhom strome nachádza práve jeden zhodný list pre spárovanie. V prípade, že sa takýchto listov nachádza viac, nevyberá sa ten s najvyššou podobnosťou, ale prvý. Hoci sa autori pôvodného algoritmu snažia tento nedostatok odstrániť dodatočným krokom, Fluri et al. uvádzajú jednoduchý príklad, pri ktorom algoritmus zlyhá.

Tento problém riešia vypočítaním podobnosti daného listu z prvého podstromu pre všetky listy druhého podstromu, s ktorými je možné daný vrchol spárovať. Namiesto uspokojenia sa s prvým zhodným listom tak vyberajú taký list, ktorého podobnosť je najvyššia. Tým sa algoritmus stáva výpočtovo zložitejší, no v kontexte zdrojových textov dosahuje lepšie výsledky.

Testovaním pôvodného a upraveného algoritmu na vzorke 1064 manuálne klasifikovaných zmien autori dokázali, že tieto úpravy algoritmu prispievajú k nájdeniu kratšieho zoznamu zmien zdrojového textu. Autori zaznamenali zlepšenie chybovosti algoritmu až o 45% (pôvodný algoritmus priemerne nájde o viac ako tri úpravy navyše, zavedením zmien toto číslo klesá na necelé dve).

4.1.3 Treed

Nguyen et al. [7] predstavujú algoritmus *Treed*. Ten vychádza z nasledovných pozorovaní:

- Ak list AST prislúcha k nezmenenému riadku zdrojového textu, môže byť považovaný za nezmenený.
- Dva vrcholy by nemali byť spárované, ak sa medzi ich potomkami nenachádzajú iné spárované vrcholy (párovanie vrcholov by malo prebiehať zdola nahor).

Samotný algoritmus sa podobá algoritmu *Change Distilling*. Treed však využíva jednoduchý nástroj textového porovnávania dvoch verzií zdrojového textu, ktorý je súčasťou verziovacieho nástroja SVN, aby odhalili nezmenené riadky zdrojového textu (a príslušné nezmenené uzly v AST). Na párovanie vnútorných vrcholov namiesto počtu listov prislúchajúceho podstromu využíva vzdialenosť charakteristických vektorov *Exas* [8]. Tie dokážu zachytiť štruktúru stromov a grafov.

V rámci testovania autori náhodne zvolili 100 revízií zdrojového textu projektu *GEclipse*. Pre každú revíziu vybrali jeden súbor a manuálne kontrolovali výsledný zoznam zmien, ktorý produkuje ich algoritmus. Zistili, že v 92% prípadov dával Treed korektné výsledky. Problémy mal v prípadoch, kedy neboli dva vrcholy dostatočne štrukturálne podobné a z tohto dôvodu ich nedokázal správne spárovať. Výsledkom boli namiesto operácií pre zmeny vrchola (presunutie, úprava) sekvencie zmien odstránenia a vloženia nového vrchola. Algoritmus taktiež nedokáže detegovať presuny častí zdrojových textov medzi súbormi.

4.1.4 DiffCat

Kawrykow a Robillard [9] vychádzajú tiež z algoritmu *Change Distilling*, pritom zavádzajú pojem tzv. podstatných zmien (angl. *non-essential*). Zmeny zaznamenané vo VCS považujú za nízkoúrovňové a navrhujú spôsob, ako sa na zdrojové texty pozerať z vyššej úrovne. Za triviálne zmeny považujú:

- triviálne zmeny typov premenných
- extrakcia lokálnych premenných
- zmeny referencií po premenovaní entity
- triviálne modifikácie kľúčových slov (napr. *this* v jazyku Java)
- premenovanie lokálnych premenných
- úpravy týkajúce sa bielych znakov, prípadne iné úpravy formátovania textu

Autori vytvorili nástroj pre detekciu podstatných (netriviálnych) zmien s názvom *DiffCat*, ktorý dokáže spracovať zdrojové texty v jazyku Java s využitím verziovacích nástrojov CVS a SVN. Ich návrh algoritmu vychádza z poznatku, že tieto triviálne zmeny (najmä premenovania) môžu byť eliminované porovnávaním zdrojových textov. Preto zavádzajú dvojfázové porovnávanie stromov. V prvej fáze využívajú ChangeDistiller spolu s ich vlastnou analýzou na detegovanie premenovaní. Takto detegované zmeny označia a úpravy vrátia späť. V druhej fáze znovu spustia ChangeDistiller, tentoraz sú výsledkom už podstatné (netriviálne) zmeny.

Z testovania je zrejmé, že presnosť takéhoto algoritmu na detekciu podstatných zmien sa pohybuje nad hranicou 93%. Pritom medzi 2.6% až 15.5% všetkých úprav metód pozostáva z triviálnych zmien. Nakoľko nejde o malé číslo, môže tento poznatok hrať významnú úlohu pri interpretácií analýz časovej evolúcie zdrojových textov. Tu však opäť závisí miera významu od kontextu analýzy.

4.1.5 iDiff

Nguyen et al. [11] pri zisťovaní zmien medzi dvoma verziami programov vychádzali z princípu, že entity v zdrojových textoch môžu zmeniť umiestnenie, meno, poradie, prípadne vnútornú reprezentáciu, no ich interakcie s inými entitami zostávajú pomerne stabilné. Na základe tejto domnienky vytvorili nástroj *iDiff*.

iDiff sa zaujíma o dva hlavné druhy entít – triedy a metódy. Medzi tri hlavné interakcie v objektovo orientovaných systémoch autori radia volanie metódy, prístup k dátam a dedenie. Pre každú entitu vytvárajú tri typy atribútov:

- označenie – názov a informácie o rozhraní entity
- vzťahy dedenia – iné entity, od ktorých dedí alebo ktoré dedia od danej entity
- vzťahy spolupráce – iné entity, ktoré sú používané alebo ktoré používa daná entita

Ako kritérium pre porovnávanie entít zdrojových textov sa autori rozhodli využiť práve podobnosť vzťahov entít.

Vytvorený nástroj testovali na Java projektoch a výsledky manuálne kontrolovali. Vo všetkých testoch iDiff vykazoval úspešnosť väčšiu ako 90%, pričom vo väčšine prípadov dosiahol hranicu 99%. Autori však pri dvojiciach súborov, ktoré obsahovali väčší počet zmien, náhodne vybrali 100 párovaní, ktoré skontrolovali (nekontrolovali teda všetky párovania). Hoci autori vo výsledkoch testov uvádzajú počet nesprávnych spárovaní entít pri analýze reálnych dát, chýba opis, o aké chyby išlo.

Výhodou tohto prístupu je vo všeobecnosti lepšie vysporiadanie sa so štrukturálnymi zmenami (napr. premiestnenie podstromu v AST) a premenovaním entít. Problémy

však môžu nastať pri refaktorovaní, kedy sa zvyčajne menia vzťahy medzi entitami.

4.1.6 UMLDiff

Xing a Stroulia [12] vytvorili nástroj *UMLDiff* pre detekciu štrukturálnych zmien medzi dvoma verziami objektovo orientovaného softvéru. Pre tvorbu párovania berú do úvahy entity systému a ich vzťahy extrahované z diagramov tried získaných zo zdrojových textov. Algoritmus ich nástroja začína prechádzať strom zhora, z virtuálnej úrovne, a postupuje nadol po jednotlivých logických úrovniach (balíky, triedy a rozhrania, bloky). Na ďalšiu úroveň postúpi po spracovaní predchádzajúcej. Nástroj vytvára párovania entít na základe ich mien a vzťahov s inými entitami. V každej úrovni sa spracúvajú deti spárovaných vrcholov z predchádzajúcej úrovne. Spracovanie jednej úrovne prebieha v nasledovných krokoch:

1. spárovanie detí s rovnakým názvom
2. spárovanie premenovaných detí
3. spárovanie premiestnených detí

Z testovania nástroja vyplýva jeho presnosť na približnej úrovni 95%. UMLDiff však vyhodnocuje približne o 5% viac zmien, než ich bolo v skutočnosti. Algoritmus pritom dokáže lepšie detegovať premenovania, než presuny entít. Dokáže však detegovať presuny častí zdrojových textov aj medzi súbormi (resp. triedami).

4.1.7 Dex

Raghavan et al. [13] vytvorili nástroj *Dex* pre analýzu syntaktických a sémantických zmien zdrojových textov v jazyku C. Primárny zámer pre vytvorenie tohto nástroja bola analýza opráv chýb vo veľkých projektoch. Zdrojové texty reprezentujú formou abstraktných sémantických stromov. Tie sú štrukturálne podobné abstraktným syntaktickým stromom. Na rozdiel od nich sú však rozšírené o hrany z abstraktných sémantických grafov.

Algoritmus tohto nástroja kombinuje prístupy zhora nadol a zdola nahor. Význam prístupu zhora nadol spočíva v rýchlom vybudovaní hrubého párovania vrcholov, čím sa zníži množstvo porovnaní dvojíc vrcholov v ďalšej fáze. Autori uvádzajú, že počet vrcholov typického AST je v rozpätí od 20.000 do 60.000, avšak počet vrcholov, ktoré prislúchajú zmeneným častiam zdrojového textu, je iba 20 až 200. Párovanie zhora nadol dokáže preto v priemere správne spárovať až 94% zo všetkých vrcholov.

Pri rekurzívnom prechádzaní stromov zhora nadol sa hľadajú potenciálne zhody dvojíc vrcholov na tej istej úrovni stromov. Pri každom rekurzívnom volaní sa porovnávajú deti tej dvojice vrcholov, ktorá už bola spárovaná. Dieťa prvého takéhoto vrchola je spárované s dieťaťom druhého vrchola, ak existuje cesta stanovenej dĺžky (štyri) začínajúca týmito deťmi, ktorej vrcholy majú identické typy a atribúty. V prípade, že sa nájde viacero takýchto párovaní pre jeden vrchol, využije sa heuristika, ktorá berie do úvahy doterajšie zistené párovanie. V prípade, že sa ani takto nenájde jednoznačné párovanie, zostávajú takéto nespárované vrcholy uchované pre ďalšie spracovanie.

Počas fázy prístupu zdola nahor sa pre každú dvojicu vrcholov rovnakého typu vytvára tzv. *matica cien*, ktorá znázorňuje cenu transformácie podstromu prvého vrchola na podstrom druhého vrchola rôznymi spôsobmi. Záznamy v tejto matici s nulovou hodnotou indikujú ich presnú zhodu. Táto druhá fáza sa skladá z troch krokov:

1. nájdenie presných zhôd
2. nájdenie vložených a zmazaných vrcholov
3. nájdenie upravených vrcholov

V prípade, že sa po niektorom z týchto krokov zmení existujúce párovanie, spustí sa opätovné vyhľadanie párov zhora nadol a vytvorí sa nová matica cien.

Vzhľadom na malú frekvenciu výskytu operácie presunu podstromu v AST a na komplikovanosť porovnávania kvôli detegovaniu týchto zmien (veľký nárast dvojíc vrcholov pre porovnávanie) sa autori rozhodli tieto operácie nedetegovať.

Autori poukazujú na skutočnosť, že v abstraktných syntaktických stromoch listy často predstavujú mená premenných a nie príkazy. Porovnávanie takýchto listov je často zbytočné a môže produkovať nechcené operácie presunov vo výslednom zozname zmien. List je preto v tomto algoritme porovnávaný iba s deťmi vrchola, ktorý je spárovaný s rodičom tohto listu, okrem prípadu, kedy rodič definuje rozsah (vtedy list reprezentuje príkaz). Podobne blok spojený s funkciou, príkazom podmienky alebo cyklom by mal byť porovnávaný s ohľadom na jeho rodiča v strome. Tieto uzly sú vynechané z matice cien.

V rámci vyhodnocovania vytvoreného nástroja autori uvádzajú výsledky analýzy zdrojových textov veľkých projektov (frekvencie výskytov jednotlivých detegovaných úprav). Autori náhodne zvolili niektoré úpravy a ručne vyhodnocovali správnosť algoritmu. Úspešnosť ich riešenia presiahla hranicu 95%.

4.1.8 Zhrnutie

Z hľadiska smeru prechádzania stromov existujú dva základné prístupy algoritmov: (1) zhora nadol a (2) zdola nahor.

V kontexte detekcie zmien abstraktných syntaktických stromov je najviac zaužívaný prístup *zdola nahor* (napr. *LaDiff* [5], *ChangeDistiller* [6], *DiffCat* [9]). Pri tomto prístupe sa najprv párujú listy stromov a následne sa pokračuje smerom nahor. Tento spôsob vo všeobecnosti dáva lepšie výsledky, avšak v prvej fáze (párovanie listov) zväčša nevyužíva prirodzenú hierarchickú štruktúru zdrojových textov vyjadrenú prostredníctvom abstraktných syntaktických stromov.

Z analyzovaných existujúcich nástrojov využíva prístup *zhora nadol* iba *UMLDiff* [12]. Pri tomto riešení sú vnútorné vrcholy párované skôr, než je známe párovanie listov príslušných podstromov. Najväčšou slabinou tohto prístupu je nesprávne vytvorené párovanie vrcholov na vyšších úrovniach stromu. Ako ďalej uvádzajú Nguyen et al. [7], dva vrcholy by nemali byť spárované, ak sa medzi ich potomkami nenachádzajú iné spárované vrcholy. Pri prístupe zhora nadol však táto podmienka nebýva dodržaná.

Do týchto dvoch kategórií môžeme zaradiť aj riešenia, ktoré sa snažia o kombináciu oboch prístupov. Raghavan et al. [13] použili prechádzanie stromu zhora nadol iba pre vytvorenie základného párovania, čím sa snažia dosiahnuť zrýchlenie výpočtu. Nguyen et al. [7] využili tento prístup ako krok dodatočného spracovania.

Nguyen et al. [7] v ich nástroji *Treed* použili jednoduchý nástroj textového porovnávania dvoch verzií zdrojového textu, ktorý je súčasťou verziovacieho nástroja SVN, aby odhalili nezmenené riadky zdrojového textu (a príslušné nezmenené uzly v AST). Z našej skúsenosti však nástroje pre porovnávanie založené na textových reťazcoch môžu chybne označiť riadok za nezmenený aj v prípade, že niektorá jeho časť bola upravená (v prípade, keď tento riadok chybne spárujú s iným riadkom upraveného zdrojového textu). Toto môže vniesť chyby do vytváraného párovania vrcholov už na začiatku výpočtu. Preto takéto využitie iného nástroja pre textové porovnávanie nepovažujeme za správny krok.

Na základe vykonanej analýzy sa nám ako najvhodnejší spôsob prechádzania stromu javí prístup zdola nahor. Pri párovaní listov je však vhodné brať do úvahy nie len samotnú podobnosť listov, ale aj okolitú hierarchickú štruktúru abstraktných syntaktických stromov. Párovanie vnútorných vrcholov stromov je potrebné postaviť nad vytvoreným párovaním listov príslušných podstromov, opäť prístupom zdola nahor.

4.2 Využitie poznatkov o evolúcii

Využitie detekcie zmien v syntaktických stromoch je možné na viacerých úrovniach. Vo svojej podstate sa dá totiž takéto riešenie použiť pri ľubovoľných problémoch porovnávania dvoch stromových štruktúr, v ktorých sú relevantné informácie uložené v ich listoch. V našej práci sa upriamujeme na konkrétny problém porovnávania zdrojových textov.

Jadrom takéhoto riešenia je vytvorenie párovania medzi časťami dvoch verzií zdrojového textu. Jedným z možných využití tohto párovania je uvádzaná tvorba zoznamu zmien. V tomto smere sa dá takéto riešenie použiť namiesto rozšírených nástrojov pre detekciu a vizualizáciu zmien v zdrojovom texte, ktoré sú súčasťou nástrojov pre riadenie revízií. Tie totiž zvyčajne neberú do úvahy hierarchickú povahu zdrojového textu. Pre získanie uceleného pohľadu o zmenách v texte je preto potrebné využiť taký nástroj, ktorý rozumie syntaktickej, prípadne aj sémantickej stránke zdrojových textov. Prepojenie takéhoto nástroja so systémom na riadenie revízií by sa tak dalo využiť napríklad v oblastiach riadenia projektu – pri jeho monitorovaní alebo plánovaní a rozdeľovaní ľudských zdrojov.

Iným využitím poznatkov o časovej evolúcii projektu je detegovanie častí zdrojového textu náchylných na chyby. Riešenia v tejto oblasti totiž často využívajú poznatky o zmenách v zdrojových textoch v čase. Tomuto problému sa venovali aj Kim et al. [14], ktorí si pri návrhu ich algoritmov pre detekciu miest náchylných na chyby všímali tieto fakty:

- časová lokalita – keď sa na určitom mieste vyskytne chyba, je pravdepodobné, že sa na rovnakom mieste vyskytne čoskoro aj iná chyba
- priestorová lokalita – keď sa na určitom mieste vyskytne chyba, je pravdepodobné, že sa vyskytne aj iná chyba na ďalších miestach, ktoré sú spolu často menené
- zmenené a nové časti – časti, ktoré boli nedávno zmenené alebo pridané, sú viac náchylné na chyby

Presnosť predikcie ich algoritmov založených na uvedených predpokladoch sa pohybovala v rozmedzí 73% až 95%. Ich riešenie ďalej skúmali Rahman et al. [15], ktorí zisťujú, že z uvedených troch predpokladov má najväčší prínos pre predpoveď chybovosti práve časová lokalita, pričom prínos zvyšných dvoch predpokladov je vo väčšine prípadov minimálny. Koreláciu medzi zmenami v zdrojových textov a vznikajúcimi chybami potvrdili aj Giger et al. [16]. Experimentami ukázali, že zmeny detegované na nižšej úrovni zdrojového textu vykazujú ako prediktor miest náchylných na chyby lepšie výsledky, než sú zmeny detegované ako riadky textu.

5 Návrh riešenia

Navrhované riešenie *TreeDiff* bude postavené na porovnávaní dvoch verzií zdrojového textu toho istého programu reprezentovaného v podobe syntaktického stromu. Pri porovnávaní týchto dvoch stromov sa budú mapovať vrcholy prvého stromu na vrcholy druhého stromu. Vstupom pre algoritmus budú teda dva súbory – dve verzie toho istého zdrojového textu. Výstupom bude vytvorené mapovanie vrcholov a zoznam zmien transformácií prvého stromu na druhý strom.

Celkový algoritmus sa skladá z nasledovných krokov:

1. Tvorba stromov zo zdrojových textov
2. Párovanie vrcholov stromov
3. Dodatočné spracovanie
4. Tvorba zoznamu zmien

5.1 Tvorba stromu

Na základe analýzy nástrojov pre tvorbu syntaktického stromu zo zdrojového textu (kapitola 3.1) sme sa rozhodli využiť nástroj ANTLR. Ten umožňuje na základe definície gramatiky ľubovoľného jazyka vygenerovať triedy v jazyku Java, ktoré okrem iného umožňujú aj vytváranie stromov z viet daného jazyka. Gramatiky najpopulárnejších programovacích jazykov sú pritom voľne dostupné. Výnimkou nie je ani Java, s ktorou sme sa rozhodli pracovať.

Výhodou využitia tohto nástroja je jeho univerzálnosť. Vzhľadom na nezávislosť navrhovaného riešenia od konkrétneho programovacieho jazyka tak bude v budúcnosti možné zameniť triedy, ktoré sú potrebné pre zostavenie stromov zo zdrojových textov, čím sa umožní spracúvanie aj iného jazyka.

5.2 Párovanie vrcholov stromov

Jadrom tejto práce je vytvorenie párovania vrcholov vygenerovaných stromov, čiže mapovanie vrcholov prvého stromu na vrcholy druhého stromu. Celkový algoritmus tvorby tohto párovania sa v navrhovanom riešení zakladá na troch prístupoch:

- párovanie podstromov
- párovanie listov
- párovanie vnútorných vrcholov

Cieľom párovania založeného na porovnávaní podstromov generovaných zo syntaktických stromov je párovanie väčších častí stromov a vytvorenie základu pre ďalšie párovanie. Navrhovaný algoritmus dokáže zachytiť vnútornú štruktúru častí stromov a na jej základe sa rozhodnúť, ktoré vrcholy majú byť spárované.

Po tomto párovaní podstromov zostávajú nespárované dva druhy vrcholov – niektoré vnútorné vrcholy, ktoré neboli obsiahnuté vo vygenerovaných podstromoch, a časti stromov, ktoré reprezentujú upravené časti zdrojových textov a ktoré sa nepodarilo spárovať pomocou podstromov. Detailnejšie párovanie týchto vrcholov je cieľom ďalších dvoch častí algoritmu.

Doteraz nespárované vnútorné vrcholy sa párujú v druhej časti – párovanie vnútorných vrcholov zdola nahor. Navrhované riešenie spočíva v párovaní takých vnútorných vrcholov, ktoré doteraz neboli spárované a pre ktoré poznáme párovanie dostatočného počtu listov k nim prislúchajúcich podstromov.

Ďalším čiastkovým spôsobom párovania je párovanie listov. V tejto fáze nášho riešenia sa už porovnávajú a párujú samotné listy generovaných syntaktických stromov. Táto časť sa taktiež stará o odhaľovanie upravených vrcholov prostredníctvom výpočtu textovej podobnosti s využitím n-gramov a Jaccardovho koeficientu podobnosti.

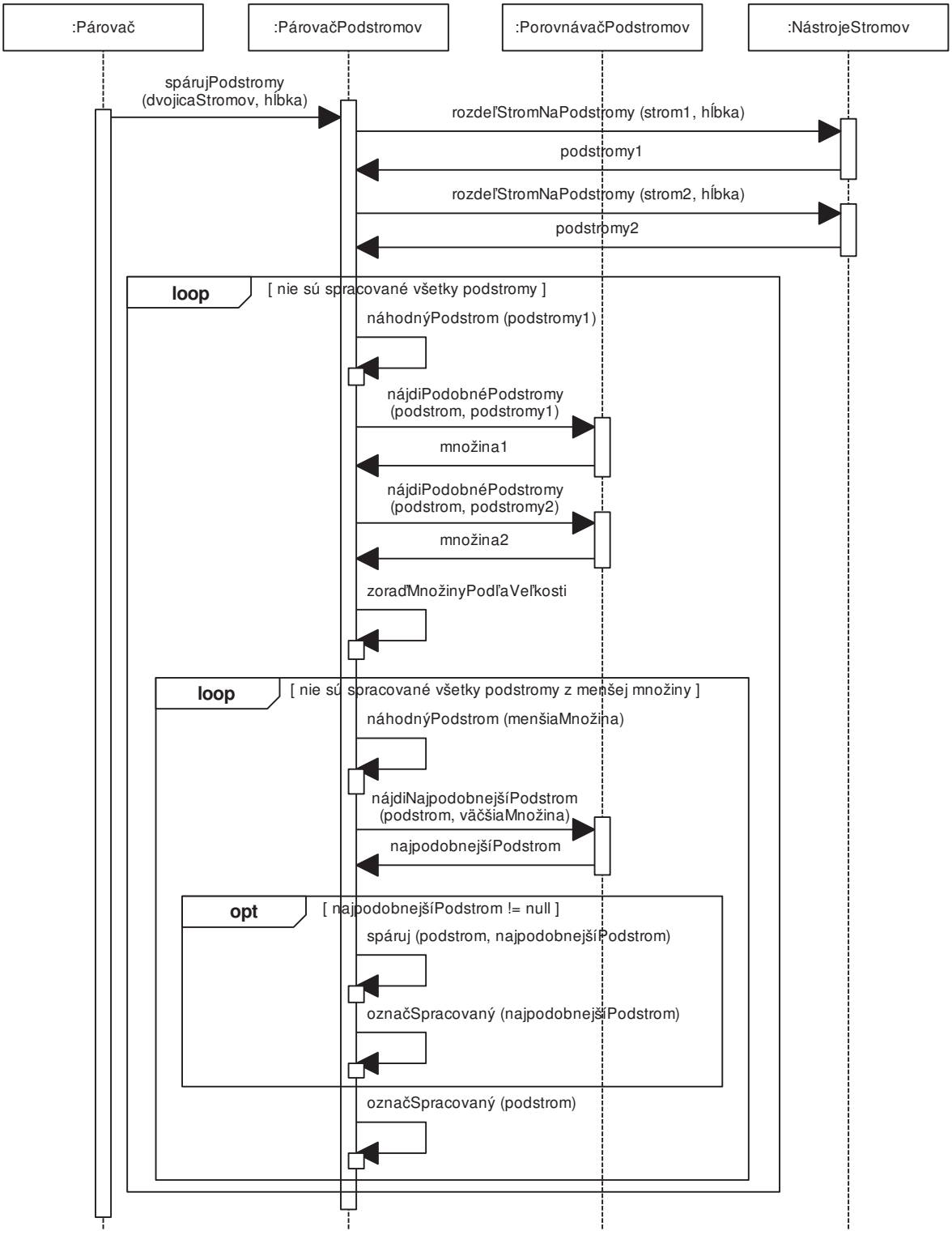
Navrhované čiastkové spôsoby párovania vrcholov budú na seba nadväzovať a navzájom sa dopĺňať. Nakoľko každý z nich dokáže pri svojej činnosti využiť už existujúce párovanie ostatných vrcholov, budú prebiehať vo viacerých iteráciách. Opis jednotlivých čiastkových prístupov uvádzame v nasledujúcich kapitolách.

Na základe takto vytvoreného párovania bude možné zistiť vykonané zmeny medzi dvoma verziami zdrojového textu.

5.2.1 Párovanie podstromov

Vygenerované syntaktické stromy budú rozdelené na menšie podstromy, ktoré budú navzájom porovnávané. Párovanie vrcholov založené na takomto porovnávaní sa bude skladať z nasledovných častí (obr. 5.1):

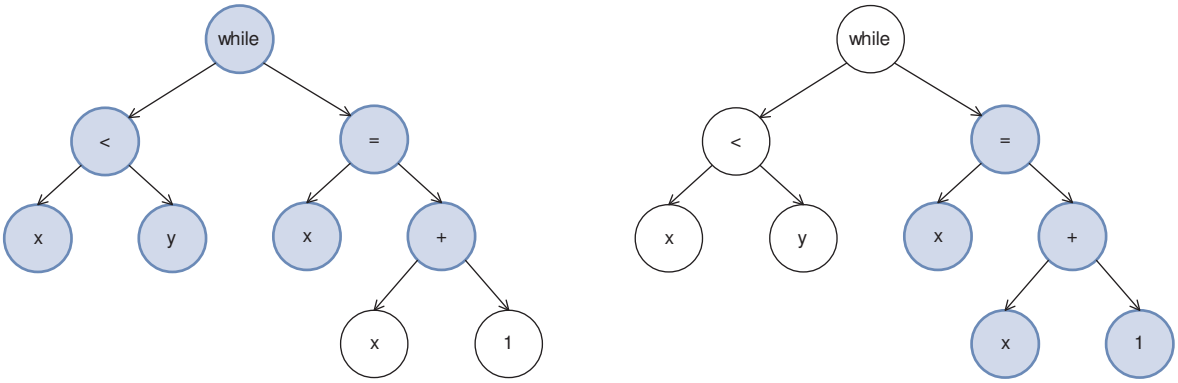
1. generovanie podstromov
2. vytvorenie množín podobných podstromov
3. výber najpodobnejšieho podstromu pre párovanie
4. párovanie vrcholov podstromov



Obr. 5.1: Sekvenčný diagram párovania podstromov.

Generovanie podstromov. V tejto časti sa zo syntaktických stromov generujú menšie podstromy danej maximálnej hĺbky, ktoré budú medzi sebou neskôr porovnávané a ktorých vrcholy budú párované.

Nakoľko dva vnútorné vrcholy by nemali byť spárované bez znalosti párovania ich potomkov [7], musia generované podstromy obsahovať aspoň jeden list. Vytváranie podstromov preto prebieha od listov smerom nahor. Vstupom pre navrhovaný algoritmus generovania podstromov je okrem samotného stromu aj parameter h_{max}, ktorý reprezentuje maximálnu výšku generovaných podstromov. Pre každý list l zo stromu sa nájde taký jeho predchodca x, aby príslušný podstrom, ktorého koreň tvorí vrchol x, mal v hĺbke h_{max} list l. Tento podstrom je následne orezaný tak, aby jeho hĺbka nebola väčšia ako h_{max}. Ukážka generovania podstromov sa nachádza na obrázku 5.2.

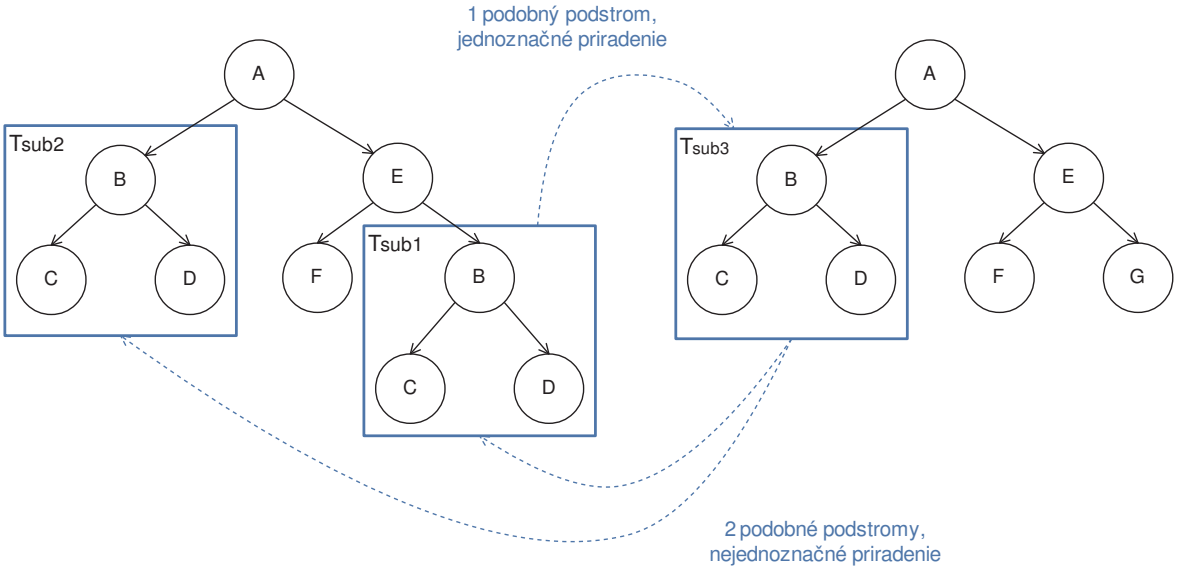


Obr. 5.2: Generovanie podstromov s maximálnou hĺbkou $h_{max} = 2$. Vrcholy dvoch vygenerovaných podstromov sú označené modrou farbou.

Vygenerované podstromy sú pritom unikátne. Ak sa v pôvodnom strome nachádza viacero listov v tej istej hĺbke a pri tvorbe podstromov od každého z týchto listov smerom nahor sa vygeneruje ten istý podstrom, bude sa vo výslednej množine všetkých podstromov nachádzať iba jeden z nich.

Vytvorenie množín podobných podstromov. Pred samotným porovnávaním je potrebné zoskupiť podobné podstromy do dvoch množín. Pre ľubovoľný vytvorený podstrom T_{sub} sa vytvorí množina G_1 takých podstromov zo stromu T_1, ktoré sú mu podobné (majú rovnakú štruktúru a ich vrcholy rovnaké označenie). Obdobne sa vytvorí množina G_2 podobných podstromov vygenerovaných zo stromu T_2. Tento krok je potrebný pre nasledujúcu fázu – výber najpodobnejšieho podstromu, v ktorom potrebujeme poznať počet prvkov pre množiny G_1 a G_2.

Význam generovania a porovnávania veľkostí množín G_1 a G_2 uvádza nasledovný ilustračný príklad (obr. 5.3). Nech strom T_1 obsahuje dva podobné podstromy T_{sub1} a T_{sub2}. Nech druhý strom T_2 obsahuje jeden taký podstrom T_{sub3}, ktorý je podobný s podstromami T_{sub1} a T_{sub2}. Potom $G_1 = \{T_{sub1}, T_{sub2}\}$ a $G_2 = \{T_{sub3}\}$. V prípade, že by sme pre podstrom T_{sub1} hľadali najpodobnejší podstrom v strome T_2, bol by ním jednoznačne podstrom T_{sub3}. To však nemusí byť pravda v prípade, ak porovnávame podstromy opačným smerom. Ak pre podstrom T_{sub3} hľadáme najpodobnejší podstrom zo stromu T_1, rozhodujeme sa už medzi dvoma podstromami T_{sub1} a T_{sub2}. Ak pri voľbe najpodobnejšieho podstromu využijeme aj ďalšie dostupné informácie (napr. okolité vrcholy podstromu v pôvodnom strome, existujúce párovanie vrcholov), môžeme za správne určiť párovanie podstromov T_{sub3} a T_{sub2}. Ako je vidieť, na smere porovnávania záleží. Takto vznikajúcim chybám sa dá predísť vygenerovaním spomínaných množín G_1 a G_2, pričom smer porovnávania bude zvolený od menšej z množín k väčšej z nich.



Obr. 5.3: Význam voľby smeru porovnávania podstromov.

Výber najpodobnejšieho podstromu. Pre ľubovoľný podstrom T_{sub} z menšej z množín podobných podstromov G_1 a G_2 sa v tejto fáze hľadá najpodobnejší podstrom T_{sim} z väčšej z množín G_1 a G_2, s ktorým bude podstrom T_{sub} spárovaný.

Pri výbere najpodobnejšieho podstromu sa berie do úvahy viacero doplňujúcich informácií, ktoré pomôžu rozhodnúť sa, ktorý z množiny podobných podstromov má byť spárovaný so zvoleným podstromom. Medzi ne patria existujúce párovanie vrcholov a štruktúra okolia porovnávaných podstromov v pôvodných stromoch.

Nech vygenerovaná množina G_1 má menší počet prvkov ako množina G_2. Potom sa z množiny G_1 zvolí ľubovoľný podstrom T_{sub}, pre ktorý sa hľadá najpodobnejší podstrom T_{sim} z množiny G_2. Výber podstromu T_{sim} opisuje algoritmus 5.1.

Algoritmus 5.1 Výber najpodobnejšieho podstromu

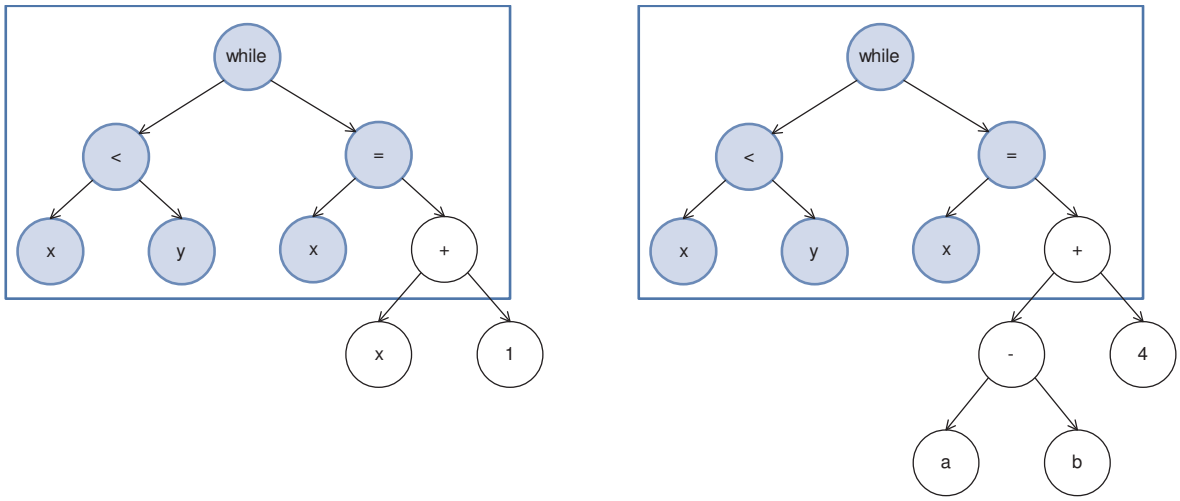
```
1: function NajpodobnejšíPodstrom (podstrom T_sub, množina podstromov G)
2:     if T_sub obsahuje spárované vrcholy then
3:         v G ponechaj iba podstromy čiastočne spárované s T_sub
4:     else
5:         v G ponechaj iba nespárované podstromy
6:     end if
7:     if |G| = 1 then
8:         return G[0]
9:     end if
10:    H ← podstromy z G s najpodobnejšou cestou nahor
11:    if |H| = 1 then
12:        return H[0]
13:    end if
14:    return nil
15: end function
```

Hľadanie podstromu s najpodobnejšou cestou nahor hľadá v tomto algoritme najdlhšiu podobnú cestu od koreňa podstromu smerom nahor v pôvodnom strome. Pritom taktiež berie do úvahy existujúce párovanie predchodcov koreňov oboch podstromov. Ak sa pri hľadaní takejto najpodobnejšej cesty smerom nahor objaví už spárovaný vrchol, skontroluje sa, či je tento vrchol spárovaný s prislúchajúcim predchodcom vrchola druhého podstromu. Ak nie, potom je tento podstrom z množiny podobných vrcholov vylúčený.

Ak sa týmto spôsobom podarí nájsť najpodobnejší podstrom, nasleduje párovanie vrcholov týchto dvoch podstromov.

Párovanie vrcholov zhodných podstromov. Ako už bolo spomenuté, pri vnútorných vrcholoch by nemali byť spárované dva vrcholy, ak sa medzi ich potomkami nenachádzajú iné spárované vrcholy [7]. Pri párovaní podstromov, ktoré boli v predchádzajúcej fáze označené za zhodné, sú preto spárované iba vrcholy, ktoré boli v pôvodnom strome listami, a ich predchodcovia v danom podstrome.

Ukážka párovania zhodných podstromov sa nachádza na obrázku 5.4. Modrý rámik označuje nájdené zhodné podstromy určené pre spárovanie. Zvýraznené vrcholy budú spárované. Vrchol s hodnotou '+' nemôže byť spárovaný, nakoľko nie je známe párovanie jeho detí (podstrom s koreňom '+' mohol byť napr. presunutý z inej časti stromu).



Obr. 5.4: Párovanie podstromov. Zhodné podstromy sú označené rámikom, zvýraznené vrcholy budú spárované.

5.2.2 Párovanie vnútorných vrcholov zdola nahor

Tento prístup párovania vnútorných vrcholov vychádza z návrhu Chawathe et al. [5]. Dva vnútorné vrcholy sú pri tomto algoritme spárované, ak podstromy, ktorých koreň tvoria tieto vrcholy, majú dostatočný počet spoločných, spárovaných detí.

Nech je daný vnútorný vrchol x zo stromu T_1 a vnútorný vrchol y zo stromu T_2. Ďalej označme $|x|$ počet listov podstromu s koreňom x a $common(x, y)$ spoločné (spárované) listy podstromov s koreňmi x a y. Vnútorné vrcholy x a y sú spárované vtedy a len vtedy, ak platí:

$$\frac{|common(x,y)|}{max(|x|,|y|)} > t$$

kde t je hraničná hodnota podobnosti vrcholov. V prípade, že existuje viacero takýchto podobných vrcholov, vyberie sa najpodobnejší.

Ako už bolo spomenuté, stromy generované nástrojom ANTLR nie sú abstraktné (obr. 5.2). Samostatnými listami sú v nich aj časti zdrojového textu ako zátvorky, bodky a pod. Z tohto dôvodu sa pri porovnávaní vnútorných vrcholov do počtu listov príslušných podstromov tieto listy nezarátajú.

Hodnotu hraničnej hodnoty podobnosti vrcholov t sme zvolili dynamickú, podobne ako Fluri et al. [6], ktorí ním riešili tzv. problém malých podstromov. Tento problém sa vyskytuje pri porovnávaní vnútorných vrcholov, z ktorých aspoň jeden príslušný podstrom má príliš malý počet listov. Pri úpravách takýchto podstromov môže jeho koreň (prípadne niektorý jeho predchodca) zostať nespárovaný. Táto chyba je dôsledkom párovania vnútorných vrcholov na základe počtu spárovaných listov príslušného podstromu, ktoré v tomto prípade tvoria príliš malú časť.

Hraničnú hodnotu sme sa preto rozhodli nastaviť nasledovne:

- $t = 0.4$ ak $min(|x|, |y|) \leq 4$
- $t = 0.6$ ak $min(|x|, |y|) > 4$

Ďalší špeciálny prípad nastáva, ak k danému vnútornému vrcholu z jedného stromu existuje takýchto najpodobnejších vrcholov z druhého stromu viacero. Tento prípad môže nastať, ak sa v strome nachádzajú nad sebou dva alebo viaceré vnútorné vrcholy rovnakého typu, pričom prvý z týchto vrcholov (rodič) nemá okrem tohto vrcholu už žiadne iné dieťa. Nakoľko dané vrcholy zdieľajú rovnaké deti, ich podobnosť je rovnaká. Tento problém riešime obdobne ako pri porovnávaní podstromov – do úvahy sa berie podobnosť cesty od tohto vrchola smerom nahor.

5.2.3 Párovanie listov

Táto fáza párovania vrcholov je založená na párovaní listov vygenerovaných stromov. Vzhľadom na povahu stromov generovaných nástrojom ANTLR je vhodné porovnávať a následne párovať nie samotné listy, ale listy spolu s ich rodičmi. Rodič listu totiž v našom prípade vypovedá o type daného listu. To zabezpečí, že nebudú spárované dva listy reprezentujúce rôzne druhy vrcholov, napr. názov metódy s názvom triedy.

Obsahová podobnosť listov. Vstupom pre navrhovaný algoritmus je okrem syntaktických stromov aj hraničná hodnota t_{sim} pre porovnávanie textovej hodnoty listov na základe bigramov. Dva listy x, y budú považované za obsahovo podobné, ak $sim_{txt}(x, y) \geq t_{sim}$, kde $sim_{txt}(x, y)$ je podobnosť založená na základe n-gramov a Jaccardovho koeficientu podobnosti:

$$sim_{txt}(x, y) = \frac{|ngrams(x) \bigcap ngrams(y)|}{|ngrams(x) \bigcup ngrams(y)|}$$

Vytvorenie množín podobných listov. Podobne, ako v prípade porovnávania podstromov (kap. 5.2.1), aj tu je potrebné pred samotným porovnávaním vytvoriť množiny podobných listov. Pre ľubovoľný list sa vytvorí množina G_1 listov zo stromu T_1, ktoré sú mu podobné (dostatočná obsahová podobnosť listov a rovnaký typ rodiča) a sú nespárované. Obdobne sa vytvorí množina G_2 listov zo stromu T_2.

Výber najpodobnejšieho listu. Pre ľubovoľný list L z menšej z množín podobných listov G_1 a G_2 sa v tejto fáze hľadá najpodobnejší list L_{sim} z väčšej z množín G_1 a G_2, s ktorým bude list L spárovaný.

Nech vygenerovaná množina G_1 má menší počet prvkov ako množina G_2. Potom sa z množiny G_1 zvolí ľubovoľný list L, pre ktorý sa hľadá najpodobnejší list L_{sim} z množiny G_2. Výber listu L_{sim} opisuje algoritmus 5.2.

Algoritmus 5.2 Výber najpodobnejšieho listu

```
 1: function NajpodobnejšíList (list L, množina listov G)
 2:     if rodič lista L je spárovaný then
 3:         v G ponechaj iba listy, ktorých rodič je spárovaný s rodičom L
 4:     else
 5:         v G ponechaj iba listy, ktorých rodič nie je spárovaný
 6:     end if
 7:     if |G| = 1 then
 8:         return G[0]
 9:     end if
10:     H ← listy z G s rovnako spárovaným ľavým súrodencom
11:     if |H| = 1 then
12:         return H[0]
13:     end if
14:     H ← listy z G s najpodobnejšou cestou nahor
15:     if |H| = 1 then
16:         return H[0]
17:     end if
18:     return nil
19: end function
```

Hľadanie listu s najpodobnejšou cestou nahor hľadá v tomto algoritme najdlhšiu podobnú cestu od tohto listu smerom nahor v pôvodnom strome, obdobne, ako v prípade hľadania najpodobnejšieho podstromu. Pritom taktiež berie do úvahy existujúce párovanie predchodcov listov oboch stromov. Ak sa pri hľadaní takejto najpodobnejšej cesty smerom nahor objaví už spárovaný vrchol, skontroluje sa, či je tento vrchol spárovaný s prislúchajúcim predchodcom vrchola druhého listu. Ak nie, potom je tento list z množiny podobných vrcholov vylúčený.

Špeciálnym prípadom sú listy s rovnakou podobnosťou. V generovaných stromoch sa môžeme stretnúť s prípadmi takých listov, ktoré majú rovnakú hodnotu, nachádzajú sa v rovnakej hĺbke stromu a majú spoločného rodiča (a teda aj celú cestu ku koreňu stromu). Takým prípadom môže byť napr. podstrom reprezentujúci zápis v jazyku Java `Map<String, String>`, kde sú dva listy s obsahom `String` v rovnakej hĺbke so spoločným rodičom. Iným prípadom, s ktorým sme sa stretli, sú bodky v zápise importu balíka `import sk.kostolansky.dp.treediff;` (obrázok 6.4). V prípade zmeny niektorej časti stromu na úrovni týchto listov nebudú tieto uzly spárované, nakoľko pre jeden z týchto listov v prvom strome sú rovnako podobné oba listy v druhom strome.

V prípade, kedy sa k danému nespárovanému listu L z prvého stromu nachádza viacero rovnako podobných listov z druhého stromu (z množiny G), vyberá algoritmus najpodobnejší list aj na základe jeho ľavého súrodenca. Ak je najbližší ľavý súrodenec tohto listu spárovaný s najbližším ľavým súrodencom niektorého z listov množiny G, budú tieto dva podobné listy spárované.

5.2.4 Dodatočné spracovanie

V tejto záverečnej časti tvorby párovania vrcholov prvého stromu s vrcholmi druhého stromu sa snažíme opraviť niektoré prípady chybného alebo chýbajúceho párovania, ktoré môžu nastať. Skladá sa z dvoch krokov: (1) odstraňovanie chybných presunov a (2) dodatočné párovanie doteraz nespárovaných vnútorných vrcholov.

Odstraňovanie chybných presunov. V niektorých prípadoch môžu byť spárované jednotlivé uzly alebo menšie podstromy, ktoré síce sú zhodné, no intuitívne nereprezentujú tú istú časť zdrojového textu. Dôsledkom takéhoto chybného párovania sú operácie presunov vo výslednom zozname zmien. Tieto prípady nastávajú najmä pri úpravách zdrojového textu, kedy je časť textu odstránená a na inom mieste zasa časť textu pridaná. V prípade, že sa v odstránenej časti nachádzajú rovnaké alebo podobné časti (napr. použité premenné), môžu byť tieto algoritmom spárované. Pre redukciu výskytu týchto chýb sa počas dodatočného spracovania odstraňujú párovania vrcholov, ktorých príslušné podstromy majú:

1. príliš málo spoločných (spárovaných) listov
2. malý počet listov a sú od seba príliš vzdialené

V prvom prípade sa odstraňujú nesprávne detegované presuny častí zdrojového textu, ktoré sú príliš upravené. Hraničnú hodnotu pre pomer spoločných listov s celkovým (maximálnym) počtom listov podstromov sme nastavili na hodnotu 0.4.

V druhom prípade sa algoritmus snaží odstrániť presuny menších častí zdrojového textu, akými môžu byť napr. samostatné premenné alebo volania metód. V prípade, že aspoň jeden z príslušných podstromov má menej ako 4 listy a vzdialenosť presunu je väčšia ako 10, sú tieto párovania odstránené. Vzdialenosť presunu je počítaná ako súčet dĺžky ciest od spárovaných listov po koreň stromu, od ktorého sa odpočíta dvojnásobok dĺžky spoločnej cesty (na základe spárovaných vrcholov) od koreňa stromov nadol.

V oboch prípadoch sa (obdobne ako v prípade párovania vnútorných vrcholov zdola nahor) do počtu listov nerátajú špeciálne syntaktické jednotky – uzly ako bodky, čiarky, zátvorky a pod.

Algoritmus prechádza stromy zdola nahor a hľadá presuny podstromov (také dvojice spárovaných vrcholov, ktoré nemajú spárovaných rodičov). Ak taký prípad nájde, rozhoduje sa na základe vyššie uvedených podmienok, či tento presun ponechá alebo odstráni. Ak sa rozhodne ponechať ho, uloží si túto informáciu o danom presune. Ak ide o chybný presun, nasleduje jeho odstránenie. To sa vykonáva zrušením existujúcich párovaní uzlov príslušných podstromov. V prípade, že sa v rámci tohto podstromu nachádza iný presun, musel už byť spracovaný (algoritmus postupuje zdola nahor). Ak sa nachádza v zozname presunov, ktoré boli doteraz vyhodnotené ako správne, potom párovania uzlov podstromov tohto vnútorného presunu nebudú odstránené.

Dodatočné párovanie vnútorných vrcholov. Napriek zavedeniu dynamickej hraničnej hodnoty podobnosti pri párovaní vnútorných vrcholov zdola nahor môžu niektoré vnútorné vrcholy zostať nespárované. Tieto chyby sa snažíme opraviť v dodatočnom kroku, kedy sa prechádzajú spárované vrcholy, ktoré nemajú spárovaných rodičov. Ak sú títo rodičia rovnakého typu a oba vrcholy zostali nespárované, vytvorí sa ich párovanie v tomto kroku.

5.2.5 Celkový algoritmus

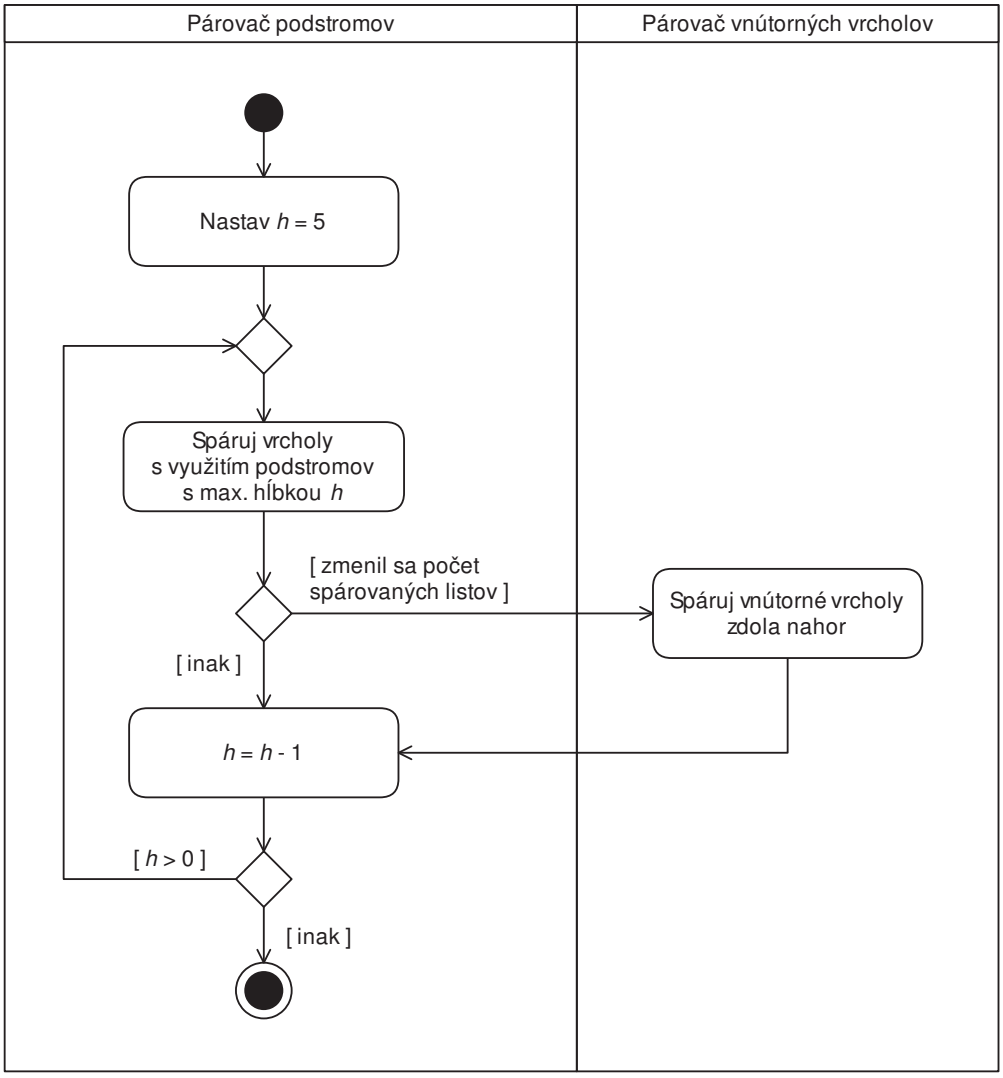
Uvedené spôsoby párovania vrcholov – párovanie založené na porovnávaní podstromov, párovanie vnútorných vrcholov stromov a párovanie listov – na seba nadväzujú a navzájom sa dopĺňajú. Každý z nich totiž dokáže pri svojej činnosti využiť už existujúce párovanie pre výber najpodobnejších častí stromov.

Obr. 5.5: Fázy párovania.

Celkový algoritmus párovania vrcholov sa skladá z troch fáz (obr 5.5), ktoré po sebe nasledujú. V prvej dominuje párovanie podstromov, v druhej párovanie listov a tretiu tvorí dodatočné spracovanie párovania.

Jednotlivé fázy celkového algoritmu bližšie opisujeme v nasledovných častiach tejto práce.

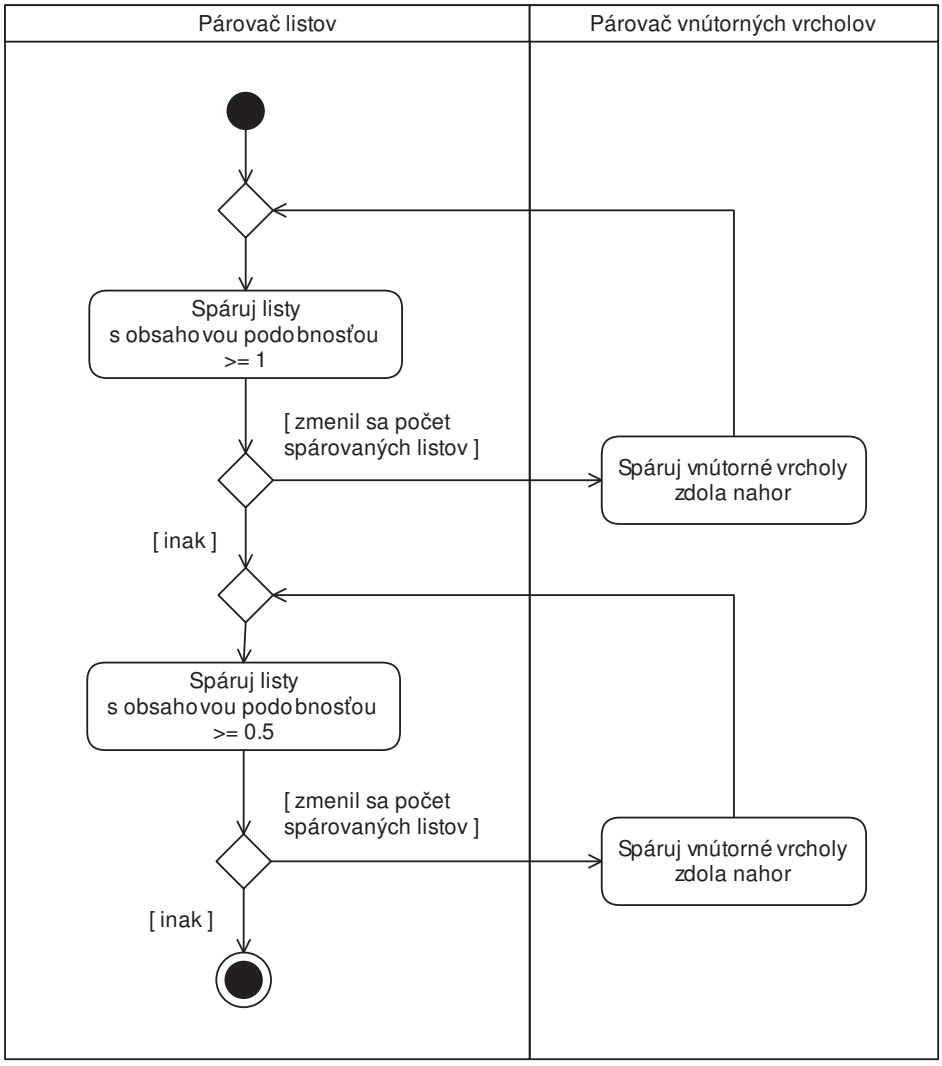
Fáza 1. V prvej fáze prebieha párovanie vrcholov vytvorených syntaktických stromov na základe podstromov postupne pre rôznu maximálnu hĺbku generovaných podstromov, od 5 do 1. Po každej takejto iterácii sa skontroluje, či sa zmenilo celkové párovanie vrcholov. Ak áno, prebehne následne aj párovanie vnútorných vrcholov zdola nahor. Túto fázu znázorňuje obrázok 5.6.



Obr. 5.6: Diagram aktivít pre prvú fázu párovania.

Fáza 2. V druhej fáze sa párujú samostatné listy. Táto fáza sa skladá z dvoch častí. V prvej sa párujú listy s nezmeneným obsahom (hraničná hodnota $t_{sim} = 1,0$), v druhej listy s upravenými hodnotami (hraničná hodnota $t_{sim} = 0,5$). Toto rozdelenie prispieva k rýchlejšiemu výpočtu, nakoľko v prvej časti, v ktorej sa predpokladá spárovanie väčšieho počtu listov ako v druhej časti, nie je potrebné generovať n-gramy a počítať Jaccardovu mieru podobnosti.

Aj v tomto prípade nasleduje po každom párovaní listov párovanie vnútorných vrcholov zdola nahor (v prípade, že sa zmení existujúce párovanie vrcholov). Výpočet sa opakuje, kým je možné spárovať ďalšie (doteraz nespárované) vrcholy. Druhú fázu znázorňuje obrázok 5.7.



Obr. 5.7: Diagram aktivít pre druhú fázu párovania.

Fáza 3. Poslednú fázu tvorí dodatočné spracovanie párovaní. Tvoria ho dve po sebe nasledujúce časti: (1) odstránenie chybných presunov nasledované (2) dodatočným párovaním vnútorných vrcholov. V oboch prípadoch prebehne jediná iterácia.

5.3 Tvorba zoznamu zmien

Po spárovaní príslušných vrcholov nasleduje vytváranie zoznamu zmien. Výsledkom tejto časti bude zoznam tvorený operáciami:

- `INS` – pridanie vrchola
- `DEL` – odstránenie vrchola
- `UPD` – úprava hodnoty listu
- `MOV` – presun podstromu

Nakoľko pri niektorých operáciách zmien je potrebné spracovať upravené vrcholy zdola nahor (napr. odstránenie rodiča môže nasledovať až po odstránení dieťaťa), je v tejto fáze potrebné prechádzať strom rovnakým smerom. Pritom sa na základe vytvoreného párovania vrcholov medzi dvoma stromami budú detegovať uvedené podporované zmeny. Spôsob ich detekcie uvádza tab. 5.1.

Tabuľka 5.1: Spôsob detekcie jednotlivých typov zmien.

Operácia	**Typ úpravy**	**Spôsob detekcie**
INS	pridanie vrchola	nespárovaný vrchol v druhom strome
DEL	odstránenie vrchola	nespárovaný vrchol v prvom strome
UPD	úprava hodnoty listu	spárované listy s odlišnou hodnotou
MOV	presun podstromu	iný rodič spárovaných vrcholov

5.4 Prepojenie so systémom riadenia revízií

V kapitole 2 sme rozoberali možnosti využitia systému riadenia revízií ako zdroja zmien v zdrojovom texte. Ako uvádzame, tieto systémy poskytujú dobrý základný zdroj informácií o zmenách zdrojových textov v čase, ktorý môže byť v prípade potreby doplnený o ďalšie zdroje dát.

Prepojenie týchto systémov s navrhovaným riešením pre účely zistenia zmien v danom súbore so zdrojovým textom je pomerne triviálne. Problém nastáva v prípade, ak by sme chceli vidieť zmeny z vyššej úrovne, než sú dvojice súborov. Návrh riešenia, ktorý sme uviedli v predchádzajúcich častiach práce, nedokáže detegovať zmeny na vyššej úrovni, akými sú presun časti zdrojového textu do iného súboru a podobne. Zmeny medzi dvoma verziami softvéru, ktoré tieto systémy riadenia revízií zaznamenávajú, však väčšinou zasiahnu viac než len jeden súbor zdrojového textu.

Ako možné riešenie sa naskytá možnosť vytvorenia akéhosi virtuálneho vrchola, ktorý by tvoril koreň porovnávaného stromu. Deťmi tohto virtuálneho koreňa by boli korene samotných stromov vygenerovaných nástrojom ANTLR pre jednotlivé súbory zdrojového textu daného projektu. Tým by bol zachytený celý projekt v podobe jedného stromu nezávisle od počtu súborov so zdrojovým textom projektu, čo by umožnilo využitie navrhovaného riešenia aj na sledovanie zmien na úrovni celého projektu. Podobné riešenie daného problému využili aj Xing a Stroulia pri vytváraní nástroja *UMLDiff* [12].

Vzhľadom na rozsiahlosť niektorých projektov by však bolo porovnávanie takto veľkých stromov priveľmi zložité a časovo aj priestorovo náročné, ba priam až nemožné. Ako možnú optimalizáciu riešenia je však možné využiť fakt, že hoci zmeny medzi dvoma verziami zdrojových textov zasahujú viaceré súbory, pomer takýchto upravených súborov ku všetkým súborom projektu býva väčšinou pomerne malý. Purushothaman a Perry [17] zistili, že 10% všetkých zmien v zdrojových textoch sa týka jedného riadku, 50% zmien sa týka menej než 10 riadkov a 95% zmien sa týka menej než 50 riadkov textu. Vzhľadom na množstvo riadkov zdrojového textu analyzovaného projektu – 2 milióny – je zrejmé, že podstatná časť textu sa medzi dvoma najbližšími verziami nemení.

Nástroje na riadenie revízií zvyčajne poskytujú možnosť zistiť, ktoré súbory boli počas danej revízie zmenené. V prípade, že tento poznatok nemáme, je možné využiť jednoduchý a rýchly nástroj na porovnávanie dvojíc textových súborov, ktorý berie do úvahy iba textovú podobnosť daných súborov (niektoré z nich uvádzame v kapitole 4.1). Tie dokážu rýchlo porovnať jednotlivé dvojice súborov projektu a zistiť, či v danom súbore nastala alebo nenastala zmena. Následne je možné zostaviť strom iba z tých súborov zdrojových textov, ktoré boli v danej revízii zmenené. Tým dokážeme výrazne redukovať náročnosť takéhoto odhaľovania zmien.

6 Overenie riešenia

Počas navrhovania uvedeného algoritmu pre analýzu evolúcie zdrojového textu bol priebežne implementovaný prototyp, na ktorom sme overovali správnosť nášho riešenia. Prototyp dokáže podľa návrhu detegovať zmeny medzi dvoma verziami zdrojového textu. Jeho prepojenie s nástrojom na riadenie revízií ponechávame ako súčasť ďalšej práce. Technická dokumentácia sa nachádza v prílohe B.

Vytvorený prototyp sme overovali s využitím ako umelých dát, tak aj dát z reálnych softvérových projektov. Nástroj sme taktiež porovnávali s existujúcimi riešeniami. Postup a výsledky overovania navrhnutého riešenia uvádzame v tejto kapitole.

6.1 Nejednoznačnosť riešenia

Overovanie riešenia daného problému je do istej miery problematické. Rozhodnutie o správnom výsledku – výstupnom párovaní vrcholov – je totiž v mnohým prípadoch nejednoznačné a subjektívne.

V mnohých prípadoch úprav zdrojových textov existuje viacero riešení, ktoré sa dajú označiť za správne. Nasledovný príklad ilustruje konkrétny prípad takéhoto problému, s ktorým sme sa počas overovania stretli. Majme dve verzie časti zdrojové textu. Nech prvá verzia vyzerá nasledovne:

```
public long timeInMillis() {
  return existsTimeInMillis + missingTimeInMillis;
}
public long getTimeInMillis() {
  return timeInMillis();
}
```

Nech druhá verzia zdrojového textu po úprave pôvodnej verzie – spojení týchto dvoch metód do jednej – vyzerá nasledovne:

```
public long getTimeInMillis() {
  return existsTimeInMillis + missingTimeInMillis;
}
```

Na uvedenú úpravu je možné pozerať sa dvoma spôsobmi:

1. ako na odstránenie metódy `getTimeInMillis` a premenovanie metódy `timeInMillis` na `getTimeInMillis`
2. ako na presun obsahu metódy `timeInMillis` do metódy `getTimeInMillis`, odstránenie metódy `timeInMillis` a odstránenie pôvodného obsahu metódy `getTimeInMillis`

V iných prípadoch úprav zdrojových textov je pri overovaní problémom bez podrobnejšieho skúmania hlbšieho kontextu testovacích dát rozhodnúť, či dané dva uzly stromov reprezentujú tú istú entitu (a ide o jej úpravu, premenovanie), alebo ide o odlišné entity s podobným názvom. Ak bol napríklad na mieste volania metódy inej triedy zamenený jej názov, je bez skúmania pridružených úprav v tejto triede nemožné jednoznačne určiť, či ide o premenovanie danej metódy, alebo či bolo volanie zamenené za volanie inej metódy s rovnakými argumentami. Z tohto dôvodu je bezchybné odhaľovanie premenovaní bez dodatočných sémantických informácií iba na základe syntaxe zdrojového textu nemožné.

V každom z týchto pohľadov na dané úpravy vzniká iné párovanie príslušných vrcholov a iný výsledný zoznam zmien. V týchto (a podobných) prípadoch, kedy voľba toho „správnejšieho“ riešenia nie je jednoznačná, považujeme v našom overovaní za správne obe (resp. viaceré) varianty.

6.2 Overovanie na umelých dátach

Prototyp bol už počas jeho implementácie priebežne testovaný na vzorke nami vytvorených umelých dát. Tieto testovacie dáta zahŕňali nasledovné zmeny v zdrojovom texte:

- Deklarácia balíka: pridanie, odstránenie, úprava
- Trieda: zmena modifikátora, premenovanie
- Globálna premenná: pridanie, odstránenie, premenovanie, zmena modifikátora
- Premenná inštancie: pridanie, odstránenie, premenovanie, presunutie, úprava, zmena typu, oddelenie definície a deklarácie
- Podmienka `if`: pridanie `else`-vetvy, úprava podmieneného výrazu
- Metóda: pridanie, odstránenie, presunutie, premenovanie, zduplikovanie obsahu do inej metódy, zmena modifikátora, zmena návratovej hodnoty

Testovaním prototypu na týchto dátach sme priebežne overovali funkčnosť jednotlivých navrhovaných častí celkového algoritmu a podľa odhalených chýb párovania postupne vylepšovali návrh do konečnej podoby.

Nakoľko išlo iba o syntetické dáta slúžiace pre overovanie čiastkových riešení stanoveného problému, výsledky z overovania prototypu na týchto dátach v práci neuvádzame.

6.3 Overovanie na reálnych dátach

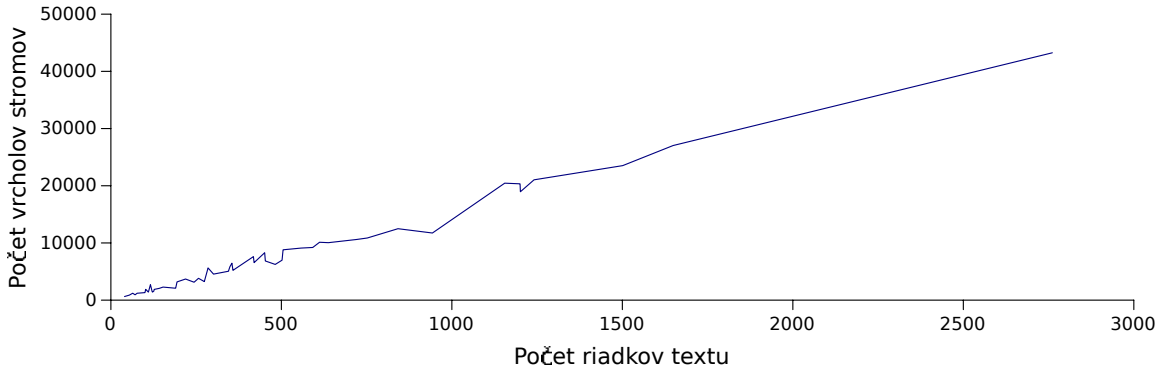
Po dokončení implementácie prototypu sme vyhodnocovali správnosť jeho výsledkov pri použití na zisťovanie zmien v zdrojových textoch reálnych projektov.

6.3.1 Opis testovacích dát

Pre účely overovania prototypu sme zostavili množinu testovacích dát, ktorú tvorilo spolu 50 dvojíc súborov, ktoré reprezentovali vždy dve verzie tej istej triedy v jazyku Java. Testovacie dáta pochádzali z troch projektov s otvoreným zdrojovým textom: (1) Elasticsearch[14] (20 dvojíc zdrojových textov), (2) Apache Hadoop[15] (20 dvojíc zdrojových textov) a (3) Spring Framework[16] (10 dvojíc zdrojových textov).

Dvojice súborov boli vyberané tak, aby zachytili rôzne prípady úprav textu od malých zmien až po zmeny, ktoré zasiahli takmer 70% riadkov. Nakoľko väčšina zmien medzi dvoma nasledovnými verziami zdrojových textov zasahuje menšie množstvo textu, niektoré vybrané dvojice sú od seba vzdialené viac než len jednu verziu, aby zachytili väčšie množstvo zmien.

Priemerný počet uzlov vygenerovaných stromov pripadajúci na jednu dvojicu testovacích súborov bol 7803, medián 5461. Priemerný počet riadkov oboch verzií zdrojových textov (bez komentárov a prázdnych riadkov) bol 497, medián 348. Priemerný pomer zmenených riadkov (opäť bez komentárov a prázdnych riadkov) ku všetkým riadkom bol podľa nástroja *GNU diff* 13%, medián 9%. Závislosť počtu riadkov od počtu vrcholov príslušných stromov sa nachádza na obr. 6.1. Podrobnejšie informácie uvádzame v prílohe A.1.



Obr. 6.1: Závislosť počtu vrcholov od počtu riadkov.

[14] `https://github.com/elasticsearch/elasticsearch`
[15] `https://github.com/apache/hadoop-common`
[16] `https://github.com/spring-projects/spring-framework`

Treba však prihliadať na to, že počet zmenených riadkov nemusí vždy odzrkadľovať počet zmien na úrovni vrcholov stromov. Rôzne riadky totiž obsahujú rôzny počet syntaktických jednotiek, z ktorých môže byť zmenená iba určitá časť.

6.3.2 Výsledky testovania

Párovanie uzlov jednotlivých stromov a príslušné zoznamy zmien vygenerované našim nástrojom pre jednotlivé dvojice súborov sme následne manuálne vyhodnocovali. Všímali sme si pritom:

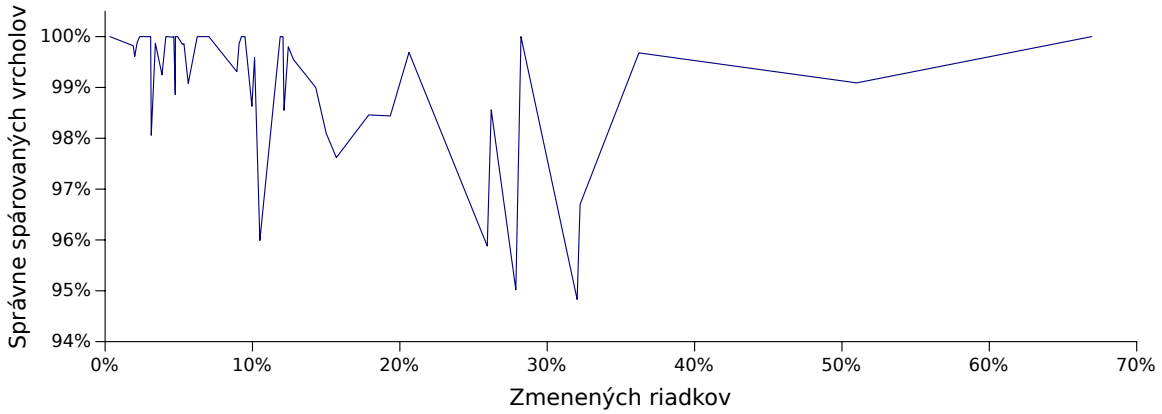
- celkový počet uzlov vygenerovaných stromov
- počet spárovaných a nespárovaných uzlov
- počet uzlov, ktoré nástroj chybne spároval (FP, *false positives*)
- počet uzlov, ktoré nástroj chybne ponechal nespárované (FN, *false negatives*)

Priemerný počet chybne spárovaných uzlov bol 8 (medián 0), priemerný počet uzlov, ktoré nástroj ponechal chybne nespárované, bol 33 (medián 6). Nástroj dokázal v priemere správne spárovať 99,5% vrcholov. Priemerná hodnota presnosti riešenia sa nachádzala na úrovni 99,9%, priemerná hodnota pokrytia dosiahla 99,5%. Výsledná hodnota F-štatistiky bola 99.7%. Hodnoty pomeru správne spárovaných vrcholov, presnosti a pokrytia sa pritom vždy nachádzali nad hranicou 94%, pričom neraz dosiahli 100%. Podrobnejšie výsledky sa nachádzajú v prílohe A.2 tejto práce.

Zo získaných výsledkov vyplýva, že väčšina z pozorovaných chýb párovania bola chybou typu FN, kedy spoločné uzly zostali nespárované. Zvyčajne išlo o časti textu, ktoré boli veľmi upravené, o premenovania entít, o špeciálne listy (bodky, čiarky, zátvorky) a podobne. To má zá následok vygenerovanie dvojíc operácii odstránenia uzla z prvého stromu (DEL) a pridania nového uzla do druhého stromu (INS) namiesto požadovaných operácií úpravy hodnoty vrchola (UPD) alebo presunu podstromu (MOV).

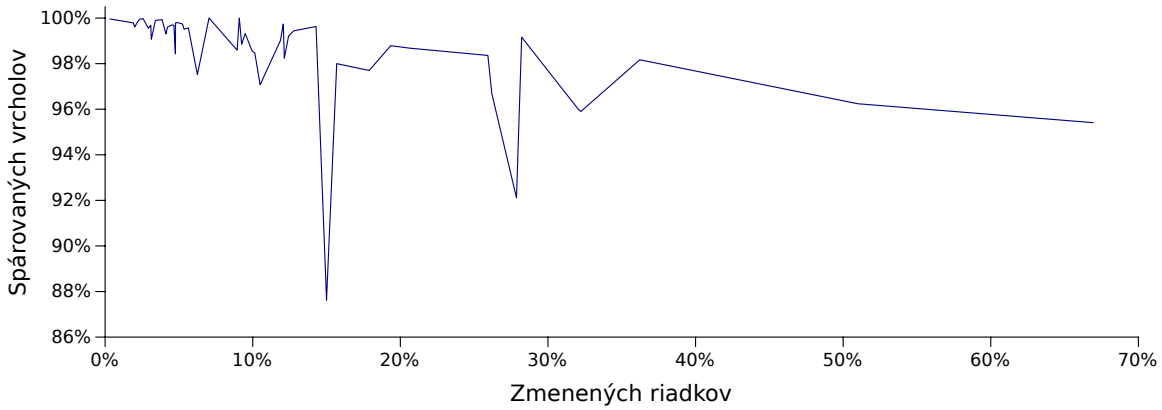
Menej častými chybami boli chyby typu FP, kedy prototyp chybne spároval dve rôzne entity. Tieto chyby väčšinou reprezentovali problém presunov podstromov pri deklaráciách importov balíkov. Ďalším prípadom bolo spárovanie dvoch rôznych entít, napr. vygenerovanie operácie UPD pre uzly *Exception* a *IOException*. Najčastejšie sa opakujúce a najväčšie chyby párovania uvádzame v kapitole 6.4.

Závislosť pomeru správne spárovaných vrcholov od veľkosti zmien je znázornený na obrázku 6.2. Ako je vidieť, táto závislosť sa pri daných testovacích dátach prejavila v menšej miere. Môžeme usudzovať, že správnosť vytvoreného párovania závisí skôr od konkrétnych typov zmien obsiahnutých v danom texte.



Obr. 6.2: Závislosť pomeru správne spárovaných vrcholov od veľkosti zmien.

Zaujímavý je poznatok, že v priemere až 98,7% všetkých párovaní vzniklo párovaním podstromov. Pri párovaní vnútorných vrcholov vzniklo 0,7% zo všetkých párovaní a pri párovaní listov 0,6%. Závislosť pomeru vrcholov spárovaných v prvej fáze výpočtu k vrcholom spárovaným v druhej fáze od veľkosti zmien v zdrojovom texte zobrazuje obrázok 6.3. Z výsledkov je zrejmé, že so vzrastajúcim počtom zmien tento pomer pomaly klesá.



Obr. 6.3: Závislosť pomeru vrcholov spárovaných v prvej fáze od veľkosti zmien.

Toto pozorovanie nám potvrdilo účel prvých dvoch fáz výpočtu. Kým prvá fáza založená na porovnávaní podstromov slúži na spárovanie vôbec alebo málo zmenených častí zdrojového textu, účelom druhej fázy založenej na porovnávaní listov je spárovanie tých častí textu, ktoré podliehali väčším zmenám a nepodarí sa ich kvôli zmenenej štruktúre okolia v strome spárovať pomocou podstromov.

Zo značného rozptylu v uvádzanom pomere spárovaných vrcholov sa dá taktiež usudzovať, že počet spárovaných vrcholov v prvej a druhej fáze sú ovplyvnené aj typom zmien, ktoré sa v danom zdrojovom texte nachádzajú.

6.4 Zistené nedostatky

Počas overovania prototypu sme identifikovali niektoré nedostatky tohto nástroja a navrhovaného riešenia. V tejto kapitole uvádzame typy chybného párovania vrcholov generovaných stromov, ktoré sme počas testovania spozorovali.

6.4.1 Problémy premenovania entít.

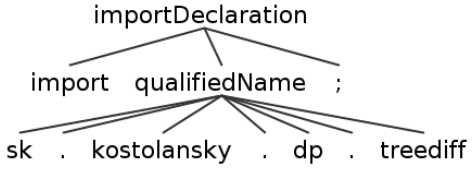
Pri úpravách zdrojového textu (najmä s využitím integrovaného vývojového prostredia) sa často stretávame s prípadom premenovania entity (premennej, metódy a pod.). Táto zmena sa v zdrojovom texte prejaví na viacerých miestach – všade tam, kde bola daná entita definovaná, deklarovaná alebo použitá. V prípade, kedy je názov premenovanej entity natoľko odlišný od pôvodného, že textová podobnosť názvov (obsahová podobnosť príslušných listov stromov) je väčšia ako stanovená hraničná hodnota, zostanú tieto vrcholy stromu nespárované, čo vedie vo výslednom zozname zmien k dvojiciam operácií odstránenia a následného vloženia vrchola namiesto operácie úpravy hodnoty tohto vrchola. Tento problém sa vo všeobecnosti nedá riešiť zvolením lepšej hraničnej hodnoty podobnosti, nakoľko premenovaná entita nemusí v názve niesť žiadne znaky po pôvodnom mene.

Možným riešením je dodatočný krok párovania vrcholov stromov. Ak sa v stromoch častokrát nachádzajú dvojice podobných nespárovaných listov, ktoré majú rovnakého rodiča a obsah prvého a obsah druhého listu z dvojice je vždy rovnaký, môže ísť o hromadné premenovanie entity. Takto by však vznikla nová skupina chýb, kedy by vznikali chybné párovania vrcholov. Ak sa napríklad volanie istej metódy na viacerých miestach zamení za volanie inej metódy, boli by príslušné uzly spárované a zmena v zdrojovom texte vyhodnotená ako premenovanie. Z tohto dôvodu sme sa rozhodli ponechať tento problém otvorený.

Ďalšou skupinou chýb, ktoré súvisia s premenovaniami, je chybné spárovanie nesúvisiacich entít, ktoré algoritmus vyhodnotí ako upravené. V mnohých prípadoch je však problémom aj pre človeka bez podrobnejšieho skúmania testovacích dát rozhodnúť, či dané dva uzly stromov reprezentujú tú istú entitu (a ide o jej úpravu), alebo ide o odlišné entity s podobným názvom. Aj z tohto dôvodu je bezchybné odhaľovanie premenovaní bez dodatočných sémantických informácií iba na základe syntaxe zdrojového textu nemožné.

6.4.2 Problém rovnakej podobnosti listov.

V generovaných stromoch sa môžeme stretnúť s prípadmi takých listov, ktoré majú rovnakú obsahovú podobnosť, nachádzajú sa v rovnakej hĺbke stromu a majú spoločného rodiča. Takým prípadom môže byť napríklad podstrom reprezentujúci zápis `import sk.kostolansky.dp.treediff;`, kde sú tri listy s obsahom `.` (bodka) v rovnakej hĺbke so spoločným rodičom. Príslušný podstrom je znázornený na obrázku 6.4. Iným príkladom je zápis `Map<String, String>`, kde sú takýmito vrcholmi dva listy s obsahom `String`. Pôvodný navrhnutý algoritmus v prípade zmeny niektorej časti stromu na úrovni týchto listov nedokáže spárovať tieto uzly, nakoľko pre jeden z týchto listov v prvom strome sú rovnako podobné oba listy v druhom strome.



Obr. 6.4: Podstrom reprezentujúci zápis `import sk.kostolansky.dp.treediff;`.

Tento problém sme čiastočne odstránili výberom najpodobnejšieho listu na základe jeho ľavého súrodenca (kapitola 5.2.3). V prípade, že by bola premenovaná entita `treediff`, dokázal by nástroj bodky spárovať správne. V prípade, že by boli upravené iné (alebo viaceré) entity, prípadne by bola niektorá entita nahradená inou (napr. `sk` na `com`), zostali by bodky nespárované (a to aj v prípade, ak by ich počet zostal nezmenený).

Problém je riešiteľný dodatočným prechádzaním stromov a párovaním podobných listov metódou „prvý s prvým“, kedy je prvý nespárovaný list z prvého stromu spárovaný s prvým podobným nespárovaným listom z druhého stromu. Takýto prístup by však mohol do výsledného párovania vniesť väčšie množstvo chýb, kedy by boli spárované nesúvisiace entity. Nakoľko sa tento problém vo väčšine prípadov týka iba menej dôležitých listov (bodky, čiarky a pod.), rozhodli sme sa ponechať nástroj v tomto ohľade bez zmeny.

6.4.3 Problémy súvisiace s povahou ANTLR stromov.

Vo väčšine abstraktných syntaktických stromov sú informácie ako modifikátor, hodnota alebo typ premennej uložené ako atribút daného vrchola reprezentujúceho túto premennú. Stromy generované nástrojom ANTLR nie sú v tomto ohľade abstraktné – tieto informácie sú v nich uložené v samostatných listoch.

Z tohto dôvodu vzniká pri párovaní stromov našim nástrojom skupina chýb podobného charakteru. Takýmito môže byť napr. zmena typu premennej z `boolean` na `int`. Nakoľko sú tieto informácie uložené v samostatnom liste, ich porovnávanie a párovanie prebieha iba na základe textovej podobnosti. Keďže nástroj nevie, že ide o vrcholy rovnakého typu, zostávajú tieto nespárované. Podobným spôsobom vznikajú aj chyby súvisiace so zmenou názvu triedy, metódy, premennej, so zmenou typu premennej, zmenou typu návratovej hodnoty metódy, zmenou modifikátora premennej, metódy alebo triedy atď.

Vo výslednom zozname zmien sa tak namiesto operácie úpravy vrchola objavujú dvojice operácií odstránenia vrchola a vloženia nového vrchola. V kontexte samotného párovania vrcholov stromov nejde o chyby v pravom zmysle slova, nakoľko ide o odlišné vrcholy, ktoré reprezentujú inú entitu (vrchol `public` a vrchol `private` nie sú tie isté entity), a teda ich nespárovanie je správne. Z pohľadu očakávaného výsledného zoznamu zmien by sme však uvítali, aby sa zmena prístupového modifikátora prejavila ako úprava hodnoty a nie ako odstránenie modifikátora a následné vloženie iného.

Problém sa dá do istej miery riešiť vytvorením množín podobných listov. V jednej takejto množine by sa nachádzali prístupové modifikátory (`private`, `public`, `protected`), v inej zasa typy premenných (`String`, `Integer`, ...) a pod. V prípade, že sa v stromoch nachádzajú na rovnakom mieste dva nespárované listy, pričom hodnoty oboch listov sa nachádzajú v niektorej z týchto množín podobných listov, môžeme prehlásiť, že ide o úpravu vrchola a tieto vrcholy spárovať. Pre toto riešenie je však potrebné ručne vytvoriť spomínané podobnostné množiny, pričom samotné riešenie je silne závislé od programovacieho jazyka zdrojových textov. Vzhľadom na to, že nejde o chyby párovania a riešenie problému nezachováva univerzálnosť celkového algoritmu, rozhodli sme sa navrhované riešenie neimplementovať do nášho prototypu.

6.4.4 Problém veľmi upravených podstromov.

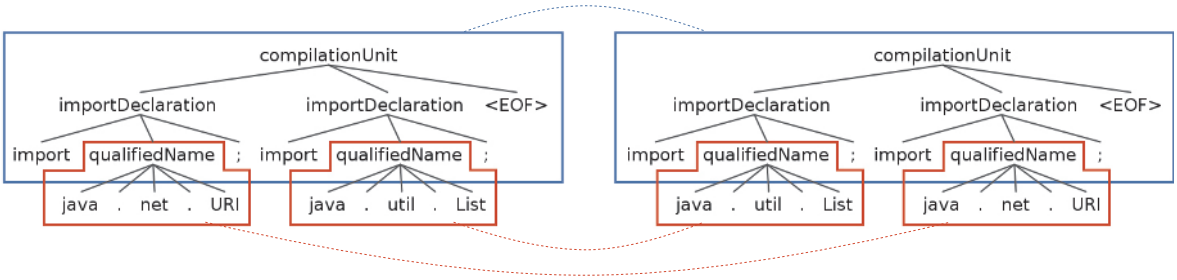
Ak sa v zdrojovom texte nachádzajú bloky, ktoré sú príliš upravené, môže ich nástroj ponechať nespárované. Tento problém sa najviac prejavil v prípade tzv. malých podstromov. Ak sa v texte nachádzali napr. malé metódy, do ktorých sú následne pridané ďalšie riadky zdrojového textu, koreň (prípadne niektorý jeho predchodca) zostával nespárovaný. Táto chyba bola dôsledkom párovania vnútorných vrcholov na základe počtu spárovaných listov príslušného podstromu, ktoré v tomto prípade tvorili príliš malú časť. Výskyt chyby bol však podmienený ďalšími úpravami v podobnej hĺbke stromu, v ktorom sa nachádzal nespárovaný koreň (v opačnom prípade by bol tento koreň sprárovaný pri párovaní založenom na porovnávaní

podstromov). V prípade malých podstromov sa nám tento problém podarilo eliminovať zavedením dynamickej hraničnej hodnoty pri párovaní vnútorných vrcholov zdola nahor (kapitola 5.2.2) a krokom dodatočného párovania vnútorných vrcholov (kapitola 5.2.4).

V menšej miere sa tento problém prejavuje aj pri väčších podstromoch. V takýchto prípadoch ide zväčša o kombináciu viacerých faktorov, ktoré prispejú k chybnému ponechaniu nespárovaných vrcholov. Zväčša ide o príliš malé alebo veľké hraničné hodnoty (napr. pre podobnosť vnútorných vrcholov pri párovaní zdola nahor, pre obsahovú podobnosť listov, pre vzdialenosť pri odstraňovaní chybných presunov, ...). Pri zmene týchto hodnôt by sa však podobné chyby párovania prejavili v iných častiach zdrojových textov. Výsledkom je namiesto množiny operácií, ktoré by zachytávali jemné úpravy danej časti textu, vygenerovanie dvojíc operácií odstránenia celého podstromu, ktorý reprezentuje upravenú časť zdrojového textu, a následne vloženia nového podstromu. S týmto problémom sa stretávame pri všetkých analyzovaných existujúcich riešeniach, či už vo väčšej, alebo menšej miere. Ukážka takejto časti zdrojového textu sa nachádza v prílohe A.4.

6.4.5 Problém presunov podstromov.

Tento problém je priamym dôsledkom párovania založeného na porovnávaní podstromov. Problém ilustuje obrázok 6.5. Nech sa ako prvý porovnáva podstrom o hĺbke 2, ktorý obsahuje dva listy `import` (na obrázku znázornené modrým rámikom). Koreňom takéhoto podstromu je vrchol `compilationUnit`. Vygenerovaný podstrom je následne spárovaný s rovnakým podstromom z druhého stromu (všetky vrcholy okrem vrcholov `qualifiedName`, pre ktoré nepoznáme párovanie príslušných listov). Pri následnom párovaní menších podstromov alebo listov sa spárujú vrcholy `qualifiedName` a ich deti. Takéto párovanie má za následok vygenerovanie operácií MOV pre vrcholy `qualifiedName` – nie pre vrcholy `importDeclaration`, ako by sme mohli očakávať.

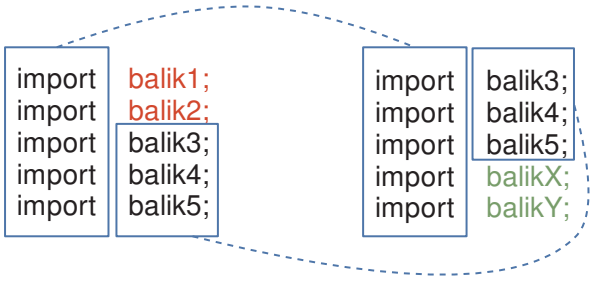


Obr. 6.5: Príklad na problém presunov podstromov.

S týmto problémom sme sa stretli iba pri importoch balíkov. Vznik takéhoto párovania je podmienený viacerými faktormi, ktoré musia byť splnené:

- v stromoch sa musí nachádzať v rovnakej hĺbke viacero vnútorných vrcholov rovnakého typu (v príklade vrcholy `qualifiedName`)
- v tej istej hĺbke stromu sa musia nachádzať aj listy (v príklade `import`), aby boli dané podstromy vygenerované a aby tieto vrcholy rovnakého typu tvorili listy vygenerovaných podstromov
- tieto vygenerované podstromy musia byť nezmenené, aby boli príslušné vrcholy spárované (v príklade vrcholy v modrých rámikoch)
- podstromy sa musia porovnávať v správnom poradí (v príklade najprv podstromy v modrých rámikoch, potom v červených rámikoch)
- počet týchto vrcholov rovnakého typu musí byť zachovaný aj po úprave (v opačnom prípade by podstromy v príklade znázornené modrým rámikom neboli spárované), napríklad zmenou poradia podstromov alebo prídaním a odstránením rovnakého počtu podstromov

Uvedený problém sa pri testovacích dátach vyskytoval najmä pri odstránení určitého počtu deklarácií importov balíkov a pridania rovnakého počtu iných takýchto deklarácií na iné miesto v celkovom zozname. Výsledné spárovanie spôsobilo vygenerovanie operácií MOV pre všetky príslušné podstromy tých deklarácií, ktoré sa nachádzali medzi pôvodnými a novými deklaráciami. Problém je znázornený na obrázku 6.6, kde boli odstránené dve prvé deklarácie balíkov a pridané dve nové na koniec zoznamu. Pre balíky 3 až 5 budú vygenerované operácie presunov.



Obr. 6.6: Príklad na posun párovania podstromov. Rámiky označujú zhodné podstromy.

Pri týchto prípadoch je opäť diskutabilné, či ide o skutočnú chybu párovania. Z pohľadu párovania častí zdrojových textov nastala v uvedenom príklade chyba pri párovaní vrcholov `import`, `importDeclaration` a `;` (bodkočiarka), nakoľko sú tieto vrcholy vo vzťahu k podstromu s koreňom `qualifiedName`. Ak sa však na problém pozeráme z pohľadu párovania vrcholov stromov, potom môžeme výsledné párovanie považovať za správne. Pri vyhodnocovaní nástroja sme však dané párovanie považovali za chybné.

6.5 Porovnanie s existujúcimi riešeniami

Vytvorený prototyp sme porovnávali s inými existujúcimi riešeniami pre detekciu zmien v zdrojových textoch. Ako testovacie dáta sme opäť použili množinu dvojíc zdrojových textov z reálnych projektov.

V prvej časti porovnávame nami vytvorený nástroj s inými riešeniami založenými na textovom porovnávaní. V ďalšej časti porovnávame jeho správnosť voči riešeniam, ktoré taktiež využívajú reprezentáciu formou syntaktických stromov.

6.5.1 Riešenia založené na porovnávaní textu

Bežne používané nástroje pre zisťovanie zmien medzi dvoma verziami zdrojového textu sú založené na porovnávaní textu. Tieto univerzálne nástroje však neberú do úvahy syntax programovacieho jazyka a na zdrojové texty sa pozerajú ako na obyčajný text bez hierarchickej štruktúry.

Nami vytvorený nástroj sme porovnali s viacerými nástrojmi tejto kategórie, ktoré uvádzame v kapitole 4.1. Výsledky zo všetkých nástrojov boli podobné. Ako sme predpokladali, tieto nástroje vo všeobecnosti vykazovali zlé párovanie častí zdrojových textov, špeciálne na miestach, ktoré obsahovali väčšie zmeny. Nakoľko využívajú textové porovnávanie, snažia sa podobné textové reťazce párovať prístupom „prvý s prvým“, čo vo veľa prípadoch vnáša do výsledného párovania chyby. Tieto riešenia tiež spárujú nesúvisiace entity, napr. názov premennej s podobným názvom metódy a pod. Ukážka porovnania párovania vygenerovaného našim prototypom a nástrojom *Meld* sa nachádza v prílohe A.3.

6.5.2 Riešenia založené na porovnávaní stromov

Existujúce nástroje pre detegovanie zmien v zdrojových textoch, ktoré berú do úvahy syntaktickú stránku programovacieho jazyka, sú menej rozšírené a zväčša určené pre zdrojové texty konkrétnych programovacích jazykov, čo redukuje množinu nástrojov, s ktorými je možné nami vytvorený prototyp priamo porovnať na rovnakých dátach. Autori vedeckých článkov často nezverejňujú nimi vytvorené nástroje (pre účely porovnania výsledkov na našich testovacích dátach), ani neposkytujú nimi využité testovacie dáta (pre porovnanie nimi uvádzaných výsledkov). Výnimku tvorí program *ChangeDistiller* [6], ktorý je určený pre zdrojové texty v jazyku Java a ktorého zdrojové texty autori sprístupnili[17].

[17] `https://bitbucket.org/sealuzh/tools-changedistiller`

Nástroje sme porovnávali na (už spomenutej) vytvorenej množine testovacích dát z reálnych softvérových projektov. Ako sa ukázalo, takéto porovnanie nie je jednoduché. Oba nástroje generovali odlišné stromy, čo malo vplyv aj na výsledné zoznamy zmien. ChangeDistiller (na rozdiel od nášho prototypu) nedokázal detegovať zmeny v importovaniach balíkov ani v anotáciách. Dokázal však nájsť zmeny v komentároch a v dokumentácii (*JavaDoc*). Nakoľko nami použitý nástroj pre generovanie stromov (ANTLR) komentáre a dokumentáciu v zdrojovom texte ignoruje, nedokážeme detegovať príslušné zmeny.

ChangeDistiller taktiež vytvára abstraktné syntaktické stromy, v ktorých sú informácie ako modifikátor, hodnota alebo typ premennej uložené ako atribút toho istého vrchola reprezentujúceho túto premennú. Ako sme už spomenuli, stromy generované nástrojom ANTLR majú tieto informácie uložené v samostatných listoch. Z tohto dôvodu sa v zoznamoch zmien často vyskytujú operácie súvisiace nielen s takýmito atribútami, ale aj so syntaktickými jednotkami, akými sú bodky, čiarky, zátvorky a pod. V prípade nástroja *ChangeDistiller* sa s týmito úpravami nestretávame.

Zoznamy zmien týchto nástrojov mali rôznu úroveň granularity. Vo viacerých prípadoch, kedy sa v rámci jednej časti zdrojového textu nachádzalo viacero úprav, vrátil ChangeDistiller ako upravenú entitu tú, pod ktorou sa tieto úpravy nachádzali. Ak napríklad volanie metódy obsahovalo zmenené viaceré parametre, nástroj vygeneroval jednu operáciu UPD pre celé volanie. Náš prototyp v týchto prípadoch vracal síce dlhší, no detailnejší zoznam zmien (Ukážka 4 v prílohe A.4). Aj tento rozdiel pravdepodobne súvisí s rozdielnou reprezentáciu zdrojových textov.

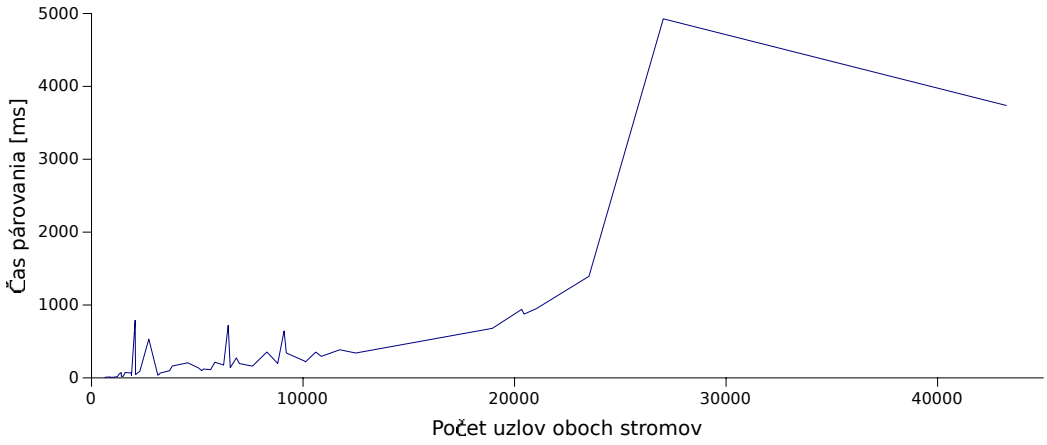
Ďalšie rozdiely sme spozorovali pri blokoch zdrojového textu, ktoré obsahovali väčší relatívny počet úprav. Nástroj *ChangeDistiller* mal v porovnaní s našim nástrojom väčšie problémy pri párovaní malých podstromov. Tie častokrát nedokázal spárovať, hoci šlo o jednoduché úpravy zdrojového textu. Výsledkom boli operácie odstránenia danej časti textu a vloženia upravenej časti (Ukážka 1 a Ukážka 3 v prílohe A.4). V prípade blokov, ktoré obsahovali naozaj veľký počet zmien zdrojového textu, sa tento problém začal prejavovať aj pri našom nástroji. Napríklad v prípade veľmi upravenej metódy, ktorú oba nástroje správne spárovali, náš prototyp už nespároval samotné telo tejto metódy (Ukážka 2 v prílohe A.4).

Z pozorovaní však nedokážeme určiť, ktorý nástroj dáva lepšie výsledky. To totiž závisí od testovacích zdrojových textov a úprav v nich obsiahnutých. Vo všeobecnosti sa tieto dva nástroje správali podobne a navzájom sa dopĺňali. Pozorované rozdiely zväčša pramenili z rozdielnej reprezentácie zdrojového textu. Ukážky rozdielov párovania nástrojov uvádzame v prílohe A.4.

6.6 Rýchlosť prototypu

V tejto kapitole uvádzame vyhodnotenie rýchlosti výpočtu prototypu. Uvádzané časy výpočtu boli merané na počítači Lenovo R400 s procesorom Intel Core 2 Duo T6670 (2.2GHz) a so 4GB pamäte RAM. Testovanie prebiehalo pod operačným systémom Linux Ubuntu 14.04 s využitím Java Development Kit 1.7.0_45. Testovacími dátami boli jednotlivé dvojice zdrojových textov z reálnych softvérových projektov.

Na obrázku 6.7 je znázornený graf závislosti dĺžky trvania párovania stromov a veľkosti porovnávaných stromov. Dva najdlhšie trvajúce testovacie vstupy pochádzajú z projektu Elasticsearch – RobinEngine a MapperService. V prvom prípade ide o nadpriemerne veľký strom (spolu viac než 40000 uzlov, takmer 2800 riadkov zdrojového textu bez prázdnych riadkov a komentárov), v druhom prípade obsahujú vygenerované stromy až takmer 5000 nespárovaných vrcholov (pridaných alebo odstránených), čo je vysoko nadpriemerné. Obrázok 6.8 zobrazuje tú istú závislosť po vynechaní týchto dvoch odľahlých hodnôt (angl. *outliers*).



Obr. 6.7: Graf závislosti času párovania a veľkosti stromov.

Obr. 6.8: Graf závislosti času párovania a veľkosti stromov bez odľahlých hodnôt.

Z nameraných hodnôt vyplýva, že priemerný čas párovania vrcholov stromov pre celý testovací dataset bol 434ms. Medián z času párovania dosiahol hodnotu 170ms. Ak k tomu prirátame čas generovania stromov nástrojom ANTLR, dostávame sa na priemernú hodnotu 526ms a medián 201ms. Podrobnejšie informácie o nameraných hodnotách uvádzame v prílohe A.5.

Z pohľadu spôsobov párovania vrcholov stromu zaberá 60% všetkého času práve párovanie podstromov. V tejto fáze je pritom vytvorených až takmer 99% zo všetkých párovaní. Párovanie vnútorných vrcholov a párovanie listov trvajú približne rovnako, 20% času párovania.

Ďalej sme si všímali dobu výpočtu pre jednotlivé fázy párovania vrcholov. Ako sme uviedli v návrhu, fázu 1 tvoria iterácie párovania podstromov pre rôznu maximálnu hĺbku týchto podstromov (od 5 do 1) nasledované párovaním vnútorných vrcholov zdola nahor. Fázu 2 predstavujú iterácie párovania listov nasledované párovaním vnútorných vrcholov zdola nahor. Vo fáze 3 prebieha dodatočné spracovanie.

Približne 79% všetkého času zaberá práve prvá fáza založená na porovnávaní podstromov, ktorá má za úlohu spárovať najväčšiu časť syntaktických stromov. Druhá v poradí je fáza založená na párovaní listov, tá trvá v priemere 20% času. Fáza dodatočného spracovania zaberá menej než 1% času.

V porovnaní s nástrojom *ChangeDistiller* sú namerané časy dosiahnuté našim prototypom dlhšie. Tento nástroj dosiahol na rovnakých testovacích dátach priemerný čas 67 ms a medián 25 ms. To môže byť spôsobené viacerými faktormi. Oproti spomínanému nástroju je nami navrhovaný algoritmus výpočtovo náročnejší. Pri párovaní listov neberieme do úvahy iba samotné listy, ale aj ich predchodcov v stromoch. Okrem párovania listov a párovania vnútorných vrcholov zavádzame aj párovanie založené na podstromoch.

Ďalším ovplyvňujúcim faktorom je povaha generovaných stromov. Stromy vytvorené nástrojom ANTLR (vzhľadom na ich už spomenutý charakter) obsahujú väčšie množstvo uzlov, čo znamená aj viac porovnaní. O tomto svedčí aj fakt, že samotná tvorba stromov týmto nástrojom je pomalšia, než generovanie a párovanie stromov nástrojom ChangeDistiller. Generovanie oboch podstromov nástrojom ANTLR trvalo v priemere 92ms, medián z týchto hodnôt bol 31ms.

7 Optimalizácia

Pri testovaní implementovaného prototypu sme sa pokúsili o optimalizáciu parametrov, ktoré vstupujú do procesu párovania vrcholov.

Optimalizácia hraničných hodnôt. Parametre ako hraničné hodnoty pre podobnosti vnútorných vrcholov alebo listov nemajú vplyv na rýchlosť párovania, ale na jeho kvalitu. Niektoré z týchto parametrov sme prebrali z existujúcich riešení, iné sme nastavili na začiatku intuitívne. Po manuálnom vyhodnocovaní prototypu sme sa vzhľadom na charakter vznikajúcich chýb v párovaní rozhodli ponechať ich nezmenené, nakoľko ich úpravy by už neviedli k znateľne lepším výsledkom.

Optimalizácia veľkostí podstromov. Ďalším parametrom, ktorý ovplyvňuje ako výsledok párovania, tak aj rýchlosť tohto procesu, je veľkosť generovaných a porovnávaných podstromov. Na začiatku sme proces párovania podstromov nastavili na viaceré iterácie párovania s rôznymi veľkosťami generovaných podstromov – od hĺbky 5 až do 1 (5 iterácií). Ak by sa dal celkový počet iterácií znížiť bez straty kvality výsledného párovania (prípadne s jej vylepšením), potom by bol celkový čas výpočtu kratší. Tento parameter sme sa pokúsili optimalizovať testovaním rôznych kombinácií veľkostí podstromov. Keďže z testovania prototypu vyplýva, že väčšina chýb párovania je typu FN (chybné nespárovanie vrcholov), ako pozorovaný faktor vypovedajúci o kvalite výsledného párovania sme si všímali celkový počet spárovaných vrcholov. Z pozorovaní vyplýva, že zníženie počtu iterácií vynechaním niektorej veľkosti generovaných podstromov síce v vedie k rýchlejšiemu výpočtu, avšak k horším výsledkom párovania (nižšiemu počtu vytvorených párovaní). Pri zvýšení počtu iterácií s využitím podstromov s hĺbkou väčšou ako 5 sme, naopak, zaznamenali zvýšenie času párovania. Niektoré vybrané príklady, ako zmena veľkosti podstromov vplýva na čas výpočtu a počet spárovaných uzlov, uvádzame v tabuľke 7.1.

Tabuľka 7.1: Vplyv veľkostí podstromov na čas výpočtu a počet párovaní oproti referenčým veľkostiam 5,4,3,2,1.

Veľkosti podstromov	**Čas výpočtu** [%]	**Spárované vrcholy** [%]
6,5,4,3,2,1	+16,34	-0.00
5,4,3,2	-7,10	-0,01
5,4,3	-16,50	-0,10
5,3	-25,24	-0,14

Vzhľadom na výsledky pozorovania sme veľkosti generovaných a porovnávaných podstromov ponechali nezmenené. Ak by sme však oželeli znížený počet vytvorených párovaní (väčší počet chýb párovania), potom je možné týmto spôsobom urýchliť celkový výpočet až o desiatky percent.

Optimalizácia párovania upravených listov. Párovanie založené na porovnávaní listov sme rozdelili na dve iterácie. V prvej sme nastavili hraničnú hodnotu pre podobnosť listov na 1,0, čo znamená presnú zhodu, takže algoritmus nemusel generovať n-gramy a počítať Jaccardovu mieru podobnosti. V druhej sme túto hranicu nastavili na 0,5. Sledovali sme, či takéto rozdelenie porovnávania do dvoch iterácií prispieva k zrýchleniu alebo k spomaleniu celkového výpočtu oproti jedinej iterácií s hraničnou hodnotou 0.5. Ako sme predpokladali, takéto rozdelenie prispieva k zrýchleniu celkového výpočtu bez zmeny výsledného párovania.

Optimalizácia náväzností párovania vnútorných vrcholov. Každá iterácia párovania vrcholov na základe podstromov a párovania listov je (v prípade, že sa počas nej zmení celkové párovanie vrcholov) nasledovaná párovaním vnútorných vrcholov zdola nahor. Počas testovania sme overovali, či sa zmení výsledné párovanie v prípade vynechania niektorej iterácie párovania vnútorných vrcholov. Vynechanie ktorejkoľvek takejto iterácie však viedlo k spárovaniu menšieho celkového počtu vrcholov. Vzhľadom na túto skutočnosť sme sa rozhodli ponechať algoritmus v tomto ohľade bez zmeny.

Optimalizácia nižšej úrovne. Implementovaný prototyp bol optimalizovaný aj na nižšej, implementačnej úrovni. Toto zahŕňalo nielen voľbu vhodných programátorských prístupov, ale aj pamätanie a znovupoužívanie výsledkov výpočtov, ktoré prebiehali často a boli výpočtovo náročnejšie (napr. pamätanie si listov k prislúchajúcemu vnútornému uzlu, ukladanie a znovupoužitie textových podobností založených na generovaní n-gramov a Jaccardovho koeficientu a pod.). Tieto kroky viedli k zrýchleniu celkového algoritmu oproti prvej verzii prototypu o viac ako 60%.

8 Záver

Cieľom tejto práce bolo navrhnutie a implementácia nástroja pre analýzu časovej evolúcie zdrojových textov softvérových systémov. V úvode práce (kapitola 4) analyzujeme aktuálny stav v problematike odhaľovania zmien v zdrojových textoch, kde uvádzame spôsoby prístupu k porovnávaniu abstraktných syntaktických stromov a analyzujeme niektoré existujúce riešenia.

Na základe vykonanej analýzy navrhujeme vlastnú metódu odhaľovania zmien v zdrojových textoch súvisiacich s časom. Ako reprezentáciu textov využívame syntaktické stromy generované nástrojom ANTLR. K jeho prednostiam patrí najmä nezávislosť od programovacieho jazyka. Pre dve verzie zdrojového textu sa tak vytvoria stromy, ktorých zhodné vrcholy je potrebné spárovať. Na základe vytvoreného párovania je ďalej možné vygenerovať tzv. zoznam zmien, ktorý reprezentuje transformáciu prvého stromu na druhý, teda samotné zmeny v zdrojovom texte.

Nami navrhovaná metóda sa skladá z troch čiastkových prístupov k párovaniu vrcholov stromov: (1) párovanie podstromov, (2) párovanie vnútorných vrcholov a (3) párovanie listov. Tie na seba nadväzujú a prebiehajú vo viacerých iteráciách. Podrobný návrh jednotlivých algoritmov a ich nadväznosť uvádzame v kapitole 5.

Na základe vytvoreného návrhu bol implementovaný prototyp, ktorý zahŕňa fázy tvorby stromov zo zdrojových textov, párovania ich vrcholov a generovania zoznamu zmien. Úspešnosť navrhovanej metódy sme manuálne overovali na testovacích dátach, ktoré tvorili dvojice zdrojových textov z reálnych softvérových projektov. Prototyp sme taktiež porovnávali s existujúcimi riešeniami. Postup a výsledky overovania podrobnejšie opisujeme v kapitole 6.

Na základe experimentov sme zistili, že takmer 99% zo všetkých párovaní vzniká pri párovaní podstromov. Prototyp dokáže v priemere správne spárovať viac než 99% vrcholov stromov. Pri daných testovacích dátach sa táto hodnota pohybovala vždy nad hranicou 94%. Väčšina z pozorovaných chýb párovania bola chybou typu FN (*false negative*), kedy zostali nespárované uzly reprezentujúce tú istú časť zdrojového textu. Najčastejšie chyby v párovaní vrcholov uvádzame v kapitole 6.4.

Z výsledkov overovania vyplýva, že navrhovaná metóda má oproti riešeniam založeným na porovnávaní textu výrazne lepšie výsledky. Čo sa týka iných riešení, ktoré využívajú reprezentáciu textov vo forme stromov, porovnávali sme naše riešenie s nástrojom ChangeDistiller. Navrhovaná metóda dosahovala podobné výsledky, avšak generovaný zoznam zmien obsahoval vo viacerých prípadoch podrobnejšie informácie o zmenenej časti textu.

Ako priestor pre ďalšiu prácu vidíme overenie univerzálnosti nášho riešenia. Hoci implementovaný prototyp dokáže pracovať iba so zdrojovými textami v jazyku Java, po menších úpravách, ktoré uvádzame v prílohe B.2, sa otvára priestor pre jeho využitie nielen pre ďalšie programovacie jazyky, ale aj pre iné jazyky, pre ktoré je možné zostaviť gramatiku pre nástroj ANTLR (napr. XML, JSON a pod.). Mnohé z nich sú už pritom vytvorené inými používateľmi a voľne dostupné.

Zaujímavé by tiež bolo experimentovanie s inými nástrojmi pre generovanie abstraktných syntaktických stromov a následné overovanie ich vplyvu na správnosť výsledného párovania a rýchlosť celého výpočtu. Hoci je nami použitý nástroj univerzálny, generovaným stromom chýba abstraktnosť, čo prispelo k viacerým problémom, ktorým sme museli čeliť. Ako sme počas overovania prototypu zistili, tento nástroj taktiež nepatrí medzi tie rýchlejšie.

Nakoľko dnešné počítače už bežne obsahujú procesory s viacerými jadrami a podporujú súbežný výpočet vo viacerých vláknach, otvára sa tiež možnosť urýchlenia tohto riešenia vhodnou paralelizáciou výpočtu jednotlivých čiastkových prístupov celkového párovania vrcholov stromov.

Ako priestor pre zrýchlenie algoritmu vidíme aj voľbu vhodnejšej reprezentácie podstromov pre účely ich porovnávania. V súčasnom riešení porovnávame tieto podstromy na úrovni vrcholov, bez žiadneho predspracovania. Ako sme ukázali, takmer 99% zo všetkých párovaní vzniká práve pri párovaní podstromov, pričom táto časť zaberá približne 60% z celkového času párovania. Pre zrýchlenie tohto procesu je možné využiť reprezentáciu vo forme vektorov Exas [8], ktorá dokáže zachytiť štruktúru takýchto podstromov pre zrýchlenie ich porovnávania. Takúto reprezentáciu podstromov využil aj Súkeník [18] pre porovnávanie podstromov za účelom detekcie zduplikovaných častí zdrojového textu. Ten vo svojej práci taktiež využil redukciou dimenzionality týchto vektorov pomocou ich k-násobného hešovania a ich následné klastrovanie prostredníctvom algoritmu Birch.

V neposlednom rade je tu možnosť prepojenia vytvoreného nástroja so systémom na riadenie revízií. Návrh možných úprav prototypu, ktoré by umožňovali sledovanie zmien v softvérovom projekte s využitím takéhoto systému, uvádzame v kapitole 5.4.

Literatúra

[1] Howe, D. 1998. *Free On-line Dictionary of Computing - abstract syntax.* http://foldoc.org/abstract+syntax

[2] Jones, J.: *Abstract Syntax Tree Implementation Idioms.* In: Proceedings of the 10th Conference on Pattern Languages of Programs. 2003.

[3] Bille, P.: *A Survey on Tree Edit Distance and Related Problems.* In: Theoretical Computer Science, vol. 337, 2005, pp. 217-239.

[4] Sager, T. et al.: *Detecting Similar Java Classes Using Tree Algorithms.* In: Proceedings of the 2006 international workshop on Mining software repositories, ACM, 2006, pp. 65-71.

[5] Chawathe, S. S. et al.: *Change detection in hierarchically structured information.* In: Proceedings of the 1996 ACM SIGMOD international conference on Management of data, ACM Press, 1996, pp. 493-504.

[6] Fluri, B. et al.: *Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction.* In: IEEE Transactions on Software Engineering, IEEE, 2007, vol. 33, no. 11, pp. 725-743.

[7] Nguyen, T. T. et al.: *Clone-aware Configuration Management.* In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, IEEE, 2009, pp. 123-134.

[8] Nguyen, H. A. et al: *Accurate and Efficient Structural Characteristic Feature Extraction for Clone Detection.* In: Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, Springer-Verlag, 2009, pp. 440-455.

[9] Kawrykow, D., Robillard, M. P.: *Non-Essential Changes in Version Histories.* In: Proceeding of the 33rd international conference on Software engineering, ACM, 2011, pp. 351-360.

[10] Negara, S. et al.: *Is It Dangerous to Use Version Control Histories to Study Source Code Evolution?* In: Proceedings of the 26th European conference on Object-Oriented Programming, Springer-Verlag, 2012, pp. 79-103.

[11] Nguyen, H. A. et al.: *iDiff: Interaction-based program differencing tool.* In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, IEEE, 2011, pp. 572-575.

[12] Xing, Z., Stroulia, E.: *UMLDiff: An Algorithm for Object-Oriented Design Differencing*. In: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, ACM, 2005, pp. 54-65.

[13] Raghavan, S. et al.: *Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases*. In: Proceedings of the 20th IEEE International Conference on Software Maintenance, IEEE, 2004, pp. 188-197.

[14] Kim, S. et al.: *Predicting Faults from Cached History*. In: Proceedings of the 29th international conference on Software Engineering, IEEE, 2007, pp. 489-498.

[15] Rahman, F. et al.: *BugCache for inspections: hit or miss?*. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ACM, 2011, pp. 322-331.

[16] Giger, E., Pinzger, M. a Gall, H. C.: *Comparing Fine-Grained Source Code Changes And Code Churn For Bug Prediction*. In: Proceedings of the 8th Working Conference on Mining Software Repositories, ACM, 2011, pp. 83-92.

[17] Purushothaman, R., Perry, D. E.: *Towards Understanding the Rhetoric of Small Changes*. In: Towards Understanding the Rhetoric of Small Changes, IEEE, 2005, vol. 31, no. 6, pp. 511-526.

[18] Súkeník, J.: *Analýza zdrojových kódov s využitím abstraktných syntaktických stromov*. FIIT STU BA, 2013.

A Výsledky overovania

V tejto prílohe uvádzame podrobnejšie výsledky k jednotlivým overovaniam prototypu implementovaného podľa navrhnutého riešenia.

A.1 Testovacie dáta

Tabuľka A.1 obsahuje informácie o množine testovacích dát, ktoré sme použili pri vyhodnocovaní prototypu. Tvorilo ju spolu 50 dvojíc súborov, ktoré reprezentovali vždy dve verzie tej istej triedy v jazyku Java a ktoré pochádzali z troch rôznych projektov - *Apache Hadoop* (H), *Elasticsearch* (E), *Spring Framework* (S).

Legenda: Počet vrcholov – počet vrcholov oboch stromov, LOC – počet riadkov oboch verzií zdrojového textu bez komentárov a prázdnych riadkov (angl. *lines of code*), Zmenených – podiel zmenených riadkov zdrojových textov ku všetkým riadkom (opäť bez komentárov a prázdnych riadkov).

Tabuľka A.1: Testovacie dáta.

Projekt/Trieda	**Počet vrcholov**	**LOC**	**Zmenených** [%]
E/AbstractFsBlobContainer	2062	142	4,23
E/BytesStreamsTests	2708	116	12,07
E/FetchSearchResult	1584	126	4,76
E/GatewayService	8290	451	4,66
E/GeoDistanceRangeFilter	6461	355	26,20
E/GetStats	3818	257	20,62
E/IndicesClusterStateService	21036	1241	2,01
E/IndicesFieldDataCache	5831	349	2,58
E/InjectorImpl	20449	1155	2,34
E/JvmStats	23518	1501	3,13
E/Lucene	10120	612	0,33
E/MapperService	27033	1649	12,43
E/MatchNoDocsQuery	1406	122	27,87
E/RestCountAction	3190	194	4,12
E/RobinEngine	43247	2761	4,53
E/SearchContext	11738	943	4,77
E/SimpleIdCache	8802	505	5,35
E/SortParseElement	7608	418	1,91

Projekt/Trieda	Počet vrcholov	LOC	Zmenených [%]
E/SourceScoreOrderFragmentsBuilder	1210	64	6,25
E/Uid	5632	285	3,86
H/AMLauncher	6849	453	12,14
H/CachedHistoryStorage	5209	358	8,94
H/ClientCache	1877	127	36,22
H/ClientRMService	10601	720	12,78
H/CompletedJob	10865	752	2,93
H/ContainerManagerPBClientImpl	2284	153	32,03
H/HistoryClientService	9209	592	10,14
H/HsLogsPage	632	40	15,00
H/HsView	1317	100	51,00
H/JAXBContextResolver	1888	102	15,69
H/JHSDelegationTokenSecretManager	1507	121	66,94
H/JobReportPBImpl	6244	482	28,22
H/LocalClientProtocolProvider	936	71	7,04
H/LocalJobRunner	18952	1201	3,41
H/LogDumper	5290	346	9,25
H/MRApps	9104	558	19,35
H/MRClientProtocolPBServiceImpl	6559	420	5,24
H/NotRunningJob	5054	345	11,88
H/PartialJob	3244	274	9,49
H/YARNRunner	12490	842	3,09
S/AbstractResource	2081	190	17,89
S/BeanUtils	10047	638	5,64
S/CollectionFactory	4550	301	32,23
S/ConcurrentReferenceHashMap	20338	1200	2,17
S/DefaultResourceLoader	1409	110	9,09
S/DefaultValueStyler	3692	219	10,50
S/FileCopyUtils	3134	244	4,92
S/MethodParameter	6987	502	9,96
S/OrderComparator	1206	77	14,29
S/StandardReflectionParameterNameDiscoverer	874	54	25,93

A.2 Overovanie na reálnych dátach

Tabuľka A.2 obsahuje výsledky manuálneho vyhodnocovania vytvoreného nástroja na množine testovacích dát z reálnych projektov. Vyhodnotenie výsledkov sa nachádza v kapitole 6.3.

Legenda: T – podiel správne spárovaných vrcholov ku všetkým vrcholom (angl. *true*), P – presnoť (angl. *precision*), R – pokrytie (angl. *recall*).

Tabuľka A.2: Výsledky overovania na reálnych dátach.

Projekt/Trieda	**T** [%]	**P** [%]	**R** [%]
E/AbstractFsBlobContainer	100,00	100,00	100,00
E/BytesStreamsTests	100,00	100,00	100,00
E/FetchSearchResult	98,86	98,82	100,00
E/GatewayService	100,00	100,00	100,00
E/GeoDistanceRangeFilter	98,56	100,00	98,21
E/GetStats	99,69	100,00	99,66
E/IndicesClusterStateService	99,61	99,93	99,67
E/IndicesFieldDataCache	100,00	100,00	100,00
E/InjectorImpl	100,00	100,00	100,00
E/JvmStats	98,06	100,00	97,97
E/Lucene	100,00	100,00	100,00
E/MapperService	99,80	99,84	99,92
E/MatchNoDocsQuery	95,02	98,45	94,95
E/RestCountAction	100,00	100,00	100,00
E/RobinEngine	99,99	100,00	99,99
E/SearchContext	100,00	100,00	100,00
E/SimpleIdCache	99,86	99,95	99,91
E/SortParseElement	99,82	100,00	99,81
E/SourceScoreOrderFragmentsBuilder	100,00	100,00	100,00
E/Uid	99,25	99,24	100,00
H/JobReportPBImpl	100,00	100,00	100,00
H/LogDumper	100,00	100,00	100,00
H/AMLauncher	98,55	100,00	98,43
H/ClientRMService	99,55	100,00	99,50
H/MRClientProtocolPBServiceImpl	99,85	100,00	99,84
H/ContainerManagerPBClientImpl	94,83	93,80	99,65

Projekt/Trieda	**T** [%]	**P** [%]	**R** [%]
H/MRApps	98,44	99,73	98,35
H/LocalJobRunner	99,87	100,00	99,87
H/LocalClientProtocolProvider	100,00	100,00	100,00
H/HistoryClientService	99,59	99,59	99,98
H/JHSDelegationTokenSecretManager	100,00	100,00	100,00
H/PartialJob	100,00	100,00	100,00
H/CachedHistoryStorage	99,31	100,00	99,27
H/CompletedJob	100,00	100,00	100,00
H/HsView	99,09	100,00	98,58
H/JAXBContextResolver	97,62	100,00	97,56
H/HsLogsPage	98,10	98,05	100,00
H/NotRunningJob	100,00	100,00	100,00
H/YARNRunner	100,00	100,00	100,00
H/ClientCache	99,68	100,00	99,60
S/CollectionFactory	96,70	98,32	97,51
S/MethodParameter	98,63	99,91	98,62
S/OrderComparator	99,00	100,00	98,90
S/StandardReflectionParameterNameDiscoverer	95,88	100,00	94,63
S/DefaultValueStyler	95,99	100,00	95,84
S/ConcurrentReferenceHashMap	99,88	100,00	99,88
S/FileCopyUtils	100,00	100,00	100,00
S/DefaultResourceLoader	99,86	100,00	99,85
S/AbstractResource	98,46	100,00	98,11
S/BeanUtils	99,08	100,00	99,07

A.3 Porovnanie s nástrojom *Meld*

V tejto prílohe sa nachádza porovnanie nášho nástroja s existujúcim riešením *Meld*. Tento nástroj je založený na textovom porovnávaní zdrojových textov, pričom neberie do úvahy syntax programovacieho jazyka. Vyhodnotenie výsledkov sa nachádza v kapitole 6.5.1

Testovací úsek zdrojového textu (1. a 2. verziu) spolu s párovaním, ktoré vygeneroval nástroj Meld, sú zobrazené na obrázku A.1. Z výsledku je jasne vidieť, že algoritmus tohto programu neberie do úvahy žiadnu syntaktickú ani sémantickú informáciu o danom zdrojovom texte, čo vedie k mnohým chybám párovania.

Nami vytvorený nástroj dokázal tieto úseky spárovať správne. Problémový bol iba reťazec výnimky (`"Could not copy properties "`), pri ktorom zostali vrcholy nespárované a namiesto operácie UPD boli vygenerované operácie DEL a INS. Výsledný zoznam zmien vyzeral nasledovne:

```
[   6,   8] DEL [ExpressionContext]: targetPd.getWriteMethod()
[   7,   8] INS [ExpressionContext]: writeMethod
[  11,   9] MOV [  12,   9] [ExpressionContext]: sourcePd != null
[  11,  26] DEL [TerminalNodeImpl]: &&
[  11,  29] DEL [ExpressionContext]: sourcePd.getReadMethod() != null
[  11,  63] MOV [  17,  39] [StatementContext]:
  {try{ Method readMethod = sourcePd.getReadMethod(); if(!Modifier.isPublic(
  readMethod.getDeclaringClass().getModifiers())){readMethod.setAccessible(true);}
  Object value = readMethod.invoke(source); Method writeMethod = targetPd.
  getWriteMethod(); if(!Modifier.isPublic(writeMethod.getDeclaringClass().
  getModifiers())){writeMethod.setAccessible(true);} writeMethod.invoke(target,
  value);} catch(Throwableex){throw new FatalBeanException(
  "Could not copy properties "+"from source to target",ex);}}
[  12,  27] INS [TerminalNodeImpl]: {
[  13,   7] MOV [  13,   6] [BlockStatementContext]:
  Method readMethod = sourcePd.getReadMethod();
[  14,   6] INS [TerminalNodeImpl]: if
[  14,   9] INS [ParExpressionContext]:
  (readMethod != null && ClassUtils.isAssignable(writeMethod.getParameterTypes()[0],
  readMethod.getReturnType()))
[  20,   7] MOV [   6,   4] [BlockStatementContext]:
  Method writeMethod = targetPd.getWriteMethod();
[  30,   9] DEL [ExpressionContext]: "Could not copy properties "
[  31,   9] UPD [TerminalNodeImpl]: "' from source to target"
[  34,  10] INS [ExpressionContext]: "Could not copy property '"+targetPd.getName()
[  39,   5] INS [TerminalNodeImpl]: }
```

Obr. A.1: Výstup nástroja Meld.

A.4 Porovnanie s nástrojom *ChangeDistiller*

V tejto prílohe uvádzame príklady rozdielov párovania nášho prototypu *TreeDiff* a nástroja *ChangeDistiller*. Vyhodnotenie výsledkov sa nachádza v kapitole 6.5.2.

Ukážka 1

Projekt Hadoop, trieda YARNRunner. ChangeDistiller nedokázal spárovať výraz, kým TreeDiff správne upravil iba parametre volania metódy.

Verzia 1 zdrojového textu

```
[315] rsrc.setType(LocalResourceType.FILE);
```

Verzia 2 zdrojového textu

```
[315] rsrc.setType(type);
```

TreeDiff

```
[ 315, 18] DEL [ExpressionListContext]: LocalResourceType.FILE
[ 315, 18] INS [ExpressionListContext]: type
```

ChangeDistiller

```
[12150] STATEMENT_DELETE - Delete: rsrc.setType(LocalResourceType.FILE);
[12159] STATEMENT_INSERT - Insert: rsrc.setType(type);
```

Ukážka 2

Projekt Apache Hadoop, trieda NotRunningJob. ChangeDistiller napriek rozsiahlym úpravám metódy `getUnknownApplicationReport` ju v tomto prípade dokázal správne spárovať. TreeDiff metódu taktiež spároval, no jej obsah (*MethodBody*) namiesto úpravy nahradil novým blokom. Irelevantné časti zdrojových textov boli vynechané.

Verzia 1 zdrojového textu

```
[74] private ApplicationReport getUnknownApplicationReport() {
[75]   ApplicationReport unknown =
[76]     recordFactory.newRecordInstance(ApplicationReport.class);
[77]   unknown.setUser("N/A");
[78]   unknown.setHost("N/A");
[79]   unknown.setName("N/A");
[80]   unknown.setQueue("N/A");
[81]   unknown.setStartTime(0);
[82]   unknown.setFinishTime(0);
[83]   unknown.setTrackingUrl("N/A");
[84]   unknown.setDiagnostics("N/A");
[85]   LOG.info("getUnknownApplicationReport");
[86]   return unknown;
[87] }
```

Verzia 2 zdrojového textu

```
[78] private ApplicationReport getUnknownApplicationReport() {
[79]   ApplicationId unknownAppId = recordFactory
[81]       .newRecordInstance(ApplicationId.class);
[82]   ApplicationAttemptId unknownAttemptId = recordFactory
[83]       .newRecordInstance(ApplicationAttemptId.class);
       ...
[88]   return BuilderUtils.newApplicationReport(unknownAppId, unknownAttemptId,
[89]     "N/A", "N/A", "N/A", "N/A", 0, "", YarnApplicationState.NEW, "N/A",
[90]     "N/A", 0, 0, FinalApplicationStatus.UNDEFINED, null, "N/A");
[91] }
```

TreeDiff

```
[ 74, 59] ( 203) DEL [MethodBodyContext]:
  {ApplicationReport unknown=recordFactory.newRecordInstance(
  ApplicationReport.class); unknown.setUser("N/A"); unknown.setHost("N/A");
  unknown.setName("N/A"); unknown.setQueue("N/A"); unknown.setStartTime(0);
  unknown.setFinishTime(0); unknown.setTrackingUrl("N/A");
  unknown.setDiagnostics("N/A"); LOG.info("getUnknownApplicationReport");
  return unknown;}
[ 78, 59] ( 159) INS [MethodBodyContext]:
  {ApplicationId unknownAppId=recordFactory.newRecordInstance(ApplicationId.class);
  ApplicationAttemptId unknownAttemptId=recordFactory.newRecordInstance(
  ApplicationAttemptId.class); return BuilderUtils.newApplicationReport(
  unknownAppId,unknownAttemptId,"N/A","N/A","N/A","N/A",0,"",
  YarnApplicationState.NEW,"N/A","N/A",0,0,FinalApplicationStatus.UNDEFINED,
  null,"N/A");}
```

ChangeDistiller

```
[4132] STATEMENT_DELETE - Delete: unknown.setUser("N/A");
[4160] STATEMENT_DELETE - Delete: unknown.setHost("N/A");
[4188] STATEMENT_DELETE - Delete: unknown.setName("N/A");
[4216] STATEMENT_DELETE - Delete: unknown.setQueue("N/A");
[4245] STATEMENT_DELETE - Delete: unknown.setStartTime(0);
[4274] STATEMENT_DELETE - Delete: unknown.setFinishTime(0);
[4304] STATEMENT_DELETE - Delete: unknown.setTrackingUrl("N/A");
[4339] STATEMENT_DELETE - Delete: unknown.setDiagnostics("N/A");
[4374] STATEMENT_DELETE - Delete: LOG.info("getUnknownApplicationReport");
[4419] STATEMENT_DELETE - Delete: unknown;
[4033] STATEMENT_UPDATE - Update:
  ApplicationReport unknown = recordFactory.newRecordInstance(
  ApplicationReport.class);
[4467] STATEMENT_INSERT - Insert:
  ApplicationAttemptId unknownAttemptId = recordFactory.newRecordInstance(
  ApplicationAttemptId.class);
[4701] STATEMENT_INSERT - Insert:
  BuilderUtils.newApplicationReport(unknownAppId,unknownAttemptId,"N/A",
  "N/A","N/A","N/A",0,"",YarnApplicationState.NEW,"N/A","N/A",0,0,
  FinalApplicationStatus.UNDEFINED,null,"N/A");
```

Ukážka 3

Projekt Elasticsearch, trieda SimpleIdCache. Kým TreeDiff dokázal zmeny detegovať, ChangeDistiller vrcholy nespároval a situáciu riešil nahradením celej metódy.

Verzia 1 zdrojového textu

```
[86] @Override
[87] public void onClose(SegmentReader owner) {
[88]   clear(owner);
[89] }
[90]
[91] @Override
[92] public void clear(IndexReader reader) {
[93]   SimpleIdReaderCache removed = idReaders.remove(reader.getCoreCacheKey());
[94]   if (removed != null) onRemoval(removed);
[95] }
```

Verzia 2 zdrojového textu

```
[85] @Override
[86] public void onClose(Object coreCacheKey) {
[87]   clear(coreCacheKey);
[88] }
[89]
[90] @Override
[91] public void clear(Object coreCacheKey) {
[92]   SimpleIdReaderCache removed = idReaders.remove(coreCacheKey);
[93]   if (removed != null) onRemoval(removed);
[94] }
```

TreeDiff

```
[ 87, 25] DEL [FormalParameterListContext]: SegmentReader owner
[ 86, 25] INS [FormalParameterListContext]: Object coreCacheKey
[ 88, 15] DEL [ExpressionListContext]: owner
[ 87, 15] INS [ExpressionListContext]: coreCacheKey
[ 92, 23] DEL [FormalParameterListContext]: IndexReader reader
[ 91, 23] INS [FormalParameterListContext]: Object coreCacheKey
[ 93, 56] DEL [ExpressionListContext]: reader.getCoreCacheKey()
[ 92, 56] INS [ExpressionListContext]: coreCacheKey
```

ChangeDistiller

```
[3175] REMOVED_FUNCTIONALITY - Delete:
  org.elasticsearch.index.cache.id.simple.SimpleIdCache.onClose(SegmentReader)
[3119] ADDITIONAL_FUNCTIONALITY - Insert:
  org.elasticsearch.index.cache.id.simple.SimpleIdCache.onClose(Object)
[3265] REMOVED_FUNCTIONALITY - Delete:
  org.elasticsearch.index.cache.id.simple.SimpleIdCache.clear(IndexReader)
[3216] ADDITIONAL_FUNCTIONALITY - Insert:
  org.elasticsearch.index.cache.id.simple.SimpleIdCache.clear(Object)
```

Ukážka 4

Projekt Elasticsearch, trieda GatewayService. Kým ChangeDistiller vygeneroval jedinú operáciu UPD pre celé volanie, TreeDiff vrátil detailnejšie informácie o zmenách. Nezmenené časti zdrojových textov boli vynechané.

Verzia 1 zdrojového textu

```
[240] clusterService.submitStateUpdateTask("local-gateway-elected-state",
        new ProcessedClusterStateUpdateTask() {
        ...
[242]   public ClusterState execute(ClusterState currentState) {
          ...
[251]     MetaData.Builder metaDataBuilder = newMetaDataBuilder()
[252]       .metaData(recoveredState.metaData());
          ...
[264]     ClusterState updatedState = newClusterStateBuilder()
            .state(currentState)
[265]       .blocks(blocks)
[266]       .metaData(metaDataBuilder)
[267]       .build();
          ...
[270]     RoutingTable.Builder routingTableBuilder = RoutingTable.builder()
            .routingTable(updatedState.routingTable());
          ...
[278]     RoutingAllocation.Result routingResult = allocationService
            .reroute(newClusterStateBuilder().state(updatedState)
            .routingTable(routingTableBuilder).build());
[279]
[280]    return newClusterStateBuilder().state(updatedState).routingResult(
           routingResult).build();
[281]  }
[282]
[283]   @Override
[284]   public void clusterStateProcessed(ClusterState clusterState) {
[285]     logger.info("recovered [{}] indices into cluster_state",
            clusterState.metaData().indices().size());
[286]     latch.countDown();
[287]   }
[288] });
```

Verzia 2 zdrojového textu

```
[237] clusterService.submitStateUpdateTask("local-gateway-elected-state",
        new ProcessedClusterStateUpdateTask() {
        ...
[239]   public ClusterState execute(ClusterState currentState) {
          ...
[248]     MetaData.Builder metaDataBuilder = MetaData.builder(
            recoveredState.metaData());
          ...
[260]     ClusterState updatedState = ClusterState.builder(currentState)
[261]       .blocks(blocks)
[262]       .metaData(metaDataBuilder)
[263]       .build();
          ...
[266]     RoutingTable.Builder routingTableBuilder = RoutingTable.builder(
```

```
            updatedState.routingTable());
          ...
[274]     RoutingAllocation.Result routingResult = allocationService.reroute(
            ClusterState.builder(updatedState).routingTable(routingTableBuilder)
            .build());
[275]
[276]     return ClusterState.builder(updatedState).routingResult(routingResult)
            .build();
[277]   }
[278]
[279]   @Override
[280]   public void onFailure(String source, Throwable t) {
[281]     logger.error("unexpected failure during [{}]", t, source);
[282]   }
[283]
[284]   @Override
[285]   public void clusterStateProcessed(String source, ClusterState oldState,
          ClusterState newState) {
[286]     logger.info("recovered [{}] indices into cluster_state",
            newState.metaData().indices().size());
[287]     latch.countDown();
[288]   }
[289] });
```

TreeDiff

```
[ 248,  56] INS [ExpressionContext]: MetaData
[ 248,  65] INS [TerminalNodeImpl]: builder
[ 251,  56] DEL [ExpressionContext]: newMetaDataBuilder()
[ 252,  30] DEL [TerminalNodeImpl]: metaData
[ 260,  49] INS [ExpressionContext]: ClusterState
[ 260,  62] INS [TerminalNodeImpl]: builder
[ 264,  49] DEL [ExpressionContext]: newClusterStateBuilder()
[ 264,  74] DEL [TerminalNodeImpl]: state
[ 270,  64] MOV [ 266,  64] [ExpressionContext]: RoutingTable.builder
[ 270,  84] DEL [TerminalNodeImpl]: (
[ 270,  85] DEL [TerminalNodeImpl]: )
[ 270,  86] DEL [TerminalNodeImpl]: .
[ 270,  87] DEL [TerminalNodeImpl]: routingTable
[ 274,  88] INS [ExpressionContext]: ClusterState
[ 274, 101] INS [TerminalNodeImpl]: builder
[ 276,  28] INS [ExpressionContext]: ClusterState
[ 276,  41] INS [TerminalNodeImpl]: builder
[ 278,  88] DEL [ExpressionContext]: newClusterStateBuilder()
[ 278, 113] DEL [TerminalNodeImpl]: state
[ 279,  17] INS [ClassBodyDeclarationContext]:
  @Override public void onFailure(String source,Throwable t){ logger.error(
  "unexpected failure during [{}]", t, source); }
[ 280,  28] DEL [ExpressionContext]: newClusterStateBuilder()
[ 280,  53] DEL [TerminalNodeImpl]: state
[ 284,  51] DEL [FormalParameterListContext]: ClusterState clusterState
[ 285,  51] INS [FormalParameterListContext]:
  String source, ClusterState oldState, ClusterState newState
[ 285,  78] DEL [ExpressionContext]: clusterState
[ 286,  78] INS [ExpressionContext]: newState
```

ChangeDistiller

```
[12709] STATEMENT_UPDATE - Update: clusterService.submitStateUpdateTask
```

A.5 Nameraná rýchlosť prototypu

Táto príloha obsahuje informácie o nameraných rýchlostiach výpočtu prototypu. Uvádzané rýchlosti boli merané na notebooku Lenovo R400 s procesorom Intel Core 2 Duo T6670 (2.2GHz) a so 4GB pamäte RAM. Testovanie prebiehalo pod operačným systémom Linux Ubuntu 14.04 s využitím Java Development Kit 1.7.0_45. Testovacími dátami boli jednotlivé dvojice zdrojových textov z reálnych projektov. Vyhodnotenie údajov uvádzame v kapitole 6.6.

Tabuľka A.3 obsahuje informácie o nameranej rýchlosti generovania stromov nástrojom *ANTLR* a rýchlosti implementovaného prototypu *TreeDiff*.

Tabuľka A.3: Rýchlosť prototypu TreeDiff.

Projekt/Trieda	**ANTLR [ms]**	**TreeDiff [ms]**
E/AbstractFsBlobContainer	695	790
E/BytesStreamsTests	362	532
E/FetchSearchResult	189	73
E/GatewayService	322	355
E/GeoDistanceRangeFilter	127	721
E/GetStats	46	163
E/IndicesClusterStateService	270	951
E/IndicesFieldDataCache	291	215
E/InjectorImpl	642	875
E/JvmStats	137	1394
E/Lucene	30	221
E/MapperService	201	4927
E/MatchNoDocsQuery	5	72
E/RestCountAction	7	46
E/RobinEngine	252	3738
E/SearchContext	28	386
E/SimpleIdCache	56	198
E/SortParseElement	64	161
E/SourceScoreOrderFragmentsBuilder	6	14
E/Uid	24	114
H/AMLauncher	45	274
H/CachedHistoryStorage	32	100
H/ClientCache	6	68

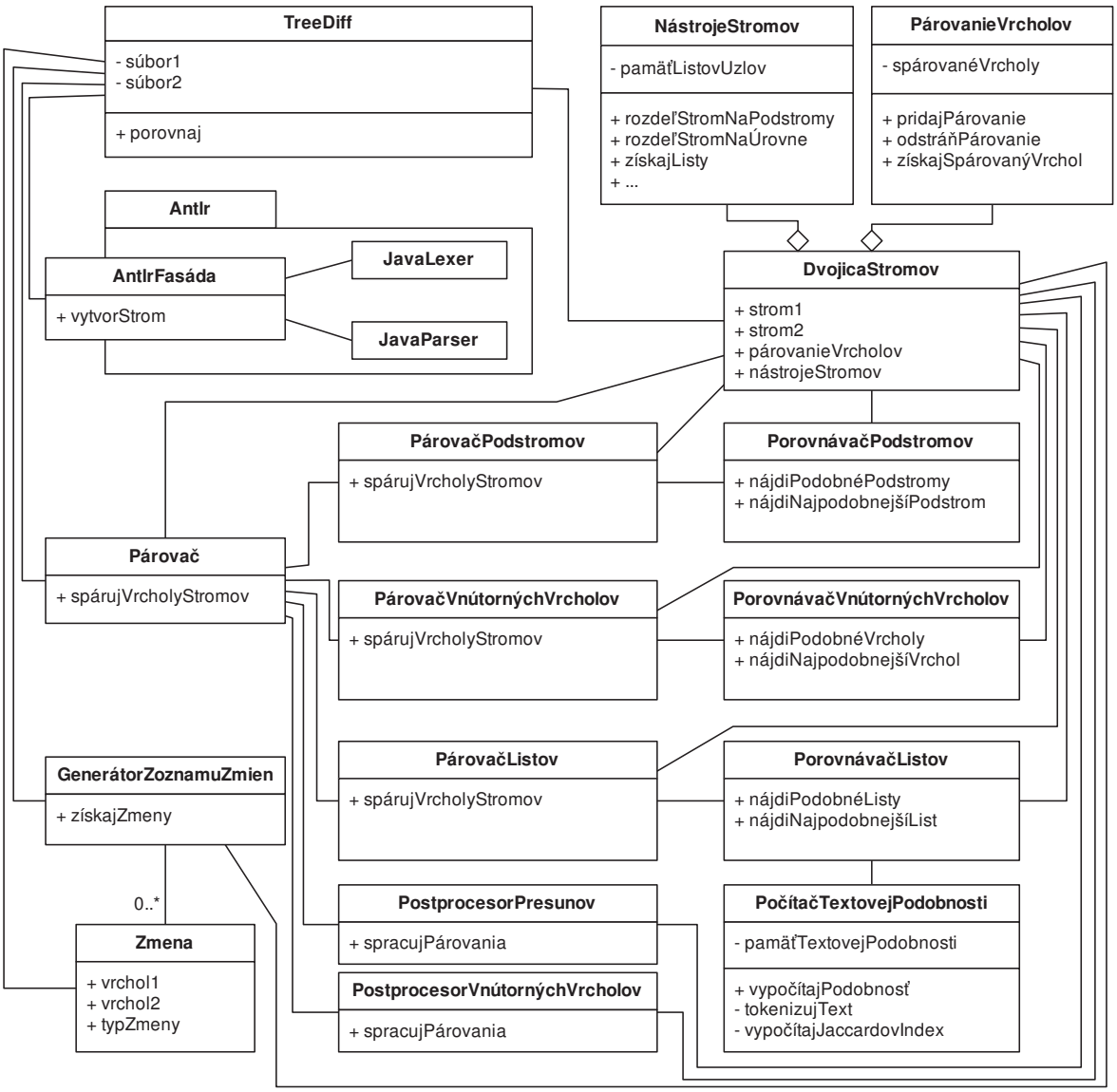
Projekt/Trieda	**ANTLR [ms]**	**TreeDiff [ms]**
H/ClientRMService	61	354
H/CompletedJob	44	295
H/ContainerManagerPBClientImpl	9	91
H/HistoryClientService	47	342
H/HsLogsPage	6	8
H/HsView	7	58
H/JAXBContextResolver	4	31
H/JHSDelegationTokenSecretManager	6	23
H/JobReportPBImpl	20	176
H/LocalClientProtocolProvider	4	6
H/LocalJobRunner	78	680
H/LogDumper	23	121
H/MRApps	46	644
H/MRClientProtocolPBServiceImpl	6	141
H/NotRunningJob	8	137
H/PartialJob	9	66
H/YARNRunner	46	341
S/AbstractResource	7	45
S/BeanUtils	116	234
S/CollectionFactory	19	207
S/ConcurrentReferenceHashMap	139	940
S/DefaultResourceLoader	4	21
S/DefaultValueStyler	28	99
S/FileCopyUtils	6	37
S/MethodParameter	34	197
S/OrderComparator	3	15
S/StandardReflectionParameterNameDiscoverer	5	13

B Technická dokumentácia

Podľa návrhu riešenia, ktorý uvádzame v kapitole 5, sme v jazyku Java implementovali prototyp, na ktorom boli overované jednotlivé spôsoby párovania vrcholov generovaných stromov a detegovania zmien medzi dvoma verziami zdrojového textu.

Prototyp bol implementovaný a testovaný na operačnom systéme Linux Ubuntu 14.04 s využitím Java Development Kit 1.7.0_45.

Na obrázku B.1 sa nachádza diagram tried vytvoreného prototypu. Ten obsahuje implementované triedy a ich hlavné metódy. Niektoré menej dôležité metódy a asociácie boli vynechané kvôli celkovej prehľadnosti diagramu.



Obr. B.1: Diagram tried prototypu.

B.1 Používateľká príručka

Na priloženom médiu sa nachádza okrem samotných zdrojových textov prototypu aj skompilovaný archív `treediff.jar`. Ten je možné použiť na detekciu zmien medzi dvoma verziami zdrojového textu v jazyku Java. Požiadavkou pre jeho použitie je Java Development Kit verzie 1.6 alebo vyššej.

Príklad použitia prototypu ilustruje nasledovná ukážka zdrojového textu:

```
import java.io.IOException;
import java.util.List;
import sk.kostolansky.dp.treediff.TreeDiff;
import sk.kostolansky.dp.treediff.editscript.TreeChange;

class Demo {
  public static void main(String[] args) throws IOException {
    String src1path = "src1.java";
    String src2path = "src2.java";

    TreeDiff diff = new TreeDiff();
    List<TreeChange> changes = diff.compare(src1path, src2path);

    for(TreeChange change : changes)
      System.out.println(change);
  }
}
```

V ukážke vytvárame objekt triedy *TreeDiff* a následne voláme metódu *compare* tohto objektu. Tá berie ako argumenty dve cesty k zdrojovým textom, ktoré chceme porovnávať. Metóda vracia zoznam detegovaných zmien, ktoré v ukážke následne vypisujeme. Výstup má nasledovný tvar (pre každú detegovanú zmenu):

```
[riadok, pozicia] operacia [typ_vrchola]: suvisiaci_text
```

- `riadok` – číslo prvého riadka v zdrojovom texte, ktorého sa zmena týka
- `pozicia` – pozícia začiatku podreťazca zdrojového textu v riadku
- `operacia` – detegovaný typ zmeny (INS, DEL, UPD, MOV)
- `typ_vrchola` – typ vrchola, ktorého sa zmena týka (napr. *ExpressionContext*)
- `suvisiaci_text` – časť zdrojového textu, ktorého sa zmena týka

Nakoľko však prototyp poskytujeme ako knižnicu, je možné si tento výstup upraviť podľa potreby, prípadne iným spôsobom manipulovať s detegovanými zmenami. Objekty vráteného zoznamu zmien sú typu *TreeChange* (v diagrame *Zmena*), ktorý okrem typu a opisu zmeny poskytuje aj prístup k dvojiciam spárovaných stromov. Informácie o práci so stromami sú dostupné v dokumentácii nástroja ANTLR.

B.2 Pridanie podpory ďalších jazykov

V tejto časti uvádzame postup, akým je možné rozšíriť podporu prototypu o detekciu zmien v ďalších jazykoch.

1. Vytvoríme gramatiku daného jazyka. Gramatiky niektorých jazykov sú dostupné v repozitári *grammars-v4*[18] projektu ANTLR.
2. Z gramatiky vytvoríme triedy v jazyku Java. Návod sa nachádza v dokumentácii[19] nástroja ANTLR (sekcia *Getting Started*). Takto vzniknú štyri triedy: **Lexer*, **Parser*, **Listener* a **BaseListener* (hviezdička znázorňuje prefix názvov tried, ktorý je tvorený názvom daného jazyka).
3. Vygenerované triedy premiestnime do priečinka zdrojových textov prototypu `sk/kostolansky/dp/treediff/antlr/base`.
4. Ako prvý riadok vložíme do vytvorených tried deklaráciu balíka: `package sk.kostolansky.dp.treediff.antlr.base;`
5. Upravíme príslušné riadky triedy *AntlrFacade* tak, aby používali vytvorené triedy (nahradením typov *JavaLexer* a *JavaParser*).

[18] `https://github.com/antlr/grammars-v4`
[19] `https://theantlrguy.atlassian.net/wiki/display/ANTLR4/ANTLR+4+Documentation`

C Článok IIT.SRC

V tejto prílohe sa nachádza článok, ktorý bol prijatý a prezentovaný na študentskej vedeckej konferencii IIT.SRC 2014. Taktiež prikladáme prezentovaný poster (obrázok C.1).

Analysis of Source Code Evolution Using Abstract Syntax Tree

Juraj KOSTOLANSKÝ*

Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies
Ilkovičova 2, 842 16 Bratislava, Slovakia
juraj@kostolansky.sk

Abstract. Source code repositories provide a large amount of information containing the changes introduced in the software system throughout its evolution. However, we lack effective tools to mine repositories for key facts about the software evolution process. In this paper, we analyze existing approaches and propose our own solution for detection of source code changes by comparing two versions of the same code represented by abstract syntax trees. We discuss the strengths and weaknesses of our tool and the direction of our future work.

1 Introduction

More and more effort is devoted to mining various information from source code repositories, as well as from other supporting tools for software development. This information includes those related to source code changes over time. A deep knowledge about software evolution can be useful for bug prediction, project management, risk management, human resource allocation, project monitoring, and so on.

In this paper we analyze existing solutions and approaches in this research area and we propose our own solution. It allows the detection of source code changes by comparing two versions of the same code of a software system. To hide the specific representation of a source code we decided to use abstract syntax trees (AST). Our solution is based on three different approaches to match nodes between trees – subtree matching, leaf matching and bottom-up matching of inner nodes. We show where it works well and where it fails and outline the direction of our future work.

2 Related work

A number of tools for identifying source code changes have been developed. Commonly used tool for code changes detection is *GNU diff*. However, it is a text-based tool which deals with flat information by comparing two files line by line and ignores the hierarchical structure of a software system.

---

* Master study programme in field: Software Engineering
Supervisor: Dr. Peter Lacko, Institute of Informatics and Software Engineering, Faculty of Informatics and Information Technologies STU in Bratislava

IIT.SRC 2014, Bratislava, April 29, 2014, pp. 1–6.

Chawathe et al. [2] studied the problem of detecting and representing changes in hierarchically structured documents. They implemented *LaDiff* – a system to detect, mark and display changes in LaTex documents. Their algorithm is based on a bottom-up traversal and the assumption that when comparing two labeled trees T_1 and T_2, any leaf in T_1 is at most matched by one leaf in T_2. However, it does not perform well for documents with duplicated parts like source code files.

Fluri et al. [3] enhanced the existing tree differencing algorithm by Chawathe et al. to classify source code changes with some improvements (dynamic tresholds, bigrams instead of largest common subsequences, inner node similarity weightening, etc.) and created the *Change Distiller*.

Raghavan et al. [5] developed *Dex* for analyzing syntactic and semantic changes in C-language code bases. They decided to use abstract semantic graphs as a representation of source codes. Their algorithm combines a top-down matching for establishing a rough correspondence between trees, to reduce the number of pairwise node comparisons that must be done in the next bottom-up matching phase.

Nguyen et al. [4] proposed the algorithm *Treed*, which is another example of a combined node matching. The bottom-up process of their algorithm might be unable to map some nodes, so they introduced the top-down part of mapping. For this purpose, they use *Exas* characteristic vectors to measure structural similarity between subtrees.

If we take a closer look at the ways existing approaches traverse trees, they can be divided into two groups: top-down and bottom-up methods (even those that tries to combine them: *Dex* uses a top-down matching only to establish a rough correspondence, *Treed* uses it as a post-processing step). Both of them have some pros and cons. In the top-down approach, inner nodes are matched before the matching of leaves is known. This conflicts with the observation by Nguyen et al. [4]: two inner nodes should not be matched if they do not have any two matched descendant nodes in corresponding subtrees. The majority of analyzed algorithms uses the bottom-up method, which generally gives better results. It starts with a matching leaf and continues with a matching of inner nodes. Hovewer, the internal structure of an AST is ignored in the leaf matching part.

3 Approach overview

In this Section we describe our method for fine-grained source code changes detection. Our approach is based on a comparsion of two versions of the same code represented by two abstract syntax trees. The whole process can be devided into three steps: (1) ASTs building, (2) nodes matching and (3) edit script generating.

As an powerful and universal AST generator we decided to use *ANTLR*[1]. From a grammar, it generates a parser that can build and walk trees. A collection of grammars for many different programming languages is available online[2].

The core of the process consists of comparing and matching nodes between two generated ASTs. Our matching approach is composed from three parts: (1) subtree matching, (2) leaf matching and (3) bottom-up inner node matching. The subtree matching is our way how to combine top-down and bottom-up methods. When some parts of an AST are significantly changed and we can not match them as subtrees, we try to do it in the fine-grained leaf and inner node matching. We describe these methods deeper in the next Sections.

The last part of the overall process is the edit script generation. An edit script is a sequence of edit operations (inserting, deleting, moving and updating nodes) turning one tree into another [1]. Assume that each edit operation has a cost. The main goal of the previous node matching part is to keep the sum of the costs of the operations in the edit script as small as possible. This sum represents tree edit distance.

[1] `http://www.antlr.org`

[2] `https://github.com/antlr/grammars-v4`

If the matching of nodes between two trees is known, it is relatively simple and straightforward to create the appropiate edit script by traversing the ASTs. If there are matched nodes with different values, we can generate an update operation. If two matched nodes have different (unamtched) parents, the node was moved. If there is an unmatched node in first tree, it was deleted and if there is an unmatched node in the second tree, it was newly insterted.

3.1 Subtree matching

Firstly, the ASTs are needed to be sliced into multiple subtrees. Our slicing method creates subtrees with a given maximal depth, so they do not have to contain all of the corresponding leaf nodes from the original AST. However, each of these subtrees has to contain at least one of the leaf nodes from the original tree. This constraint ensures the observation by Nguyen et al. [4] – we should not match two inner nodes if they do not have any two matched descendant nodes.

When these subtrees are created, they are assigned to sets of subtrees with the same structure. After that, there are two corresponding sets of similar subtrees for a given subtree: one set for each of the original ASTs. This step is necessary for choosing the right direction of most similar subtree searching in the next part of the approach.

For a subtree from the smaller set of similar subtrees we are looking for the most similar subtree from the bigger set. The similarity between these subtrees is calculated from predecessors of the subtree root in the original AST. The existing node matching is taken into account in this process.

If we can identify the two most similar subtrees, they are matched together in the last step. As we have already mentioned above, two inner nodes should not be matched if they do not have any two matched descendant nodes [4]. For this reason only the nodes representing leaves in the original ASTs and their predecessors in the subtrees are matched.

3.2 Bottom-up inner node matching

In this step of the overall matching process we got inspired by Chawathe et al. [2] and Fluri et al. [3]. For inner tree nodes we use a measure of how many leaves the subtrees have in common:

$$\frac{|common(x, y)|}{max(|x|, |y|)} \geq t \tag{1}$$

where x, y are compared inner nodes, $|x|$ denotes the number of leaves contained by x, $common(x, y)$ represents the number of common (matched) leaves of appropiate subtrees rooted at nodes x, y and t is a similarity treshlod usually between 0.5 and 1. We set this treshold to 0.5. In case of multiple such node pairs we create a matching from the most similar one.

3.3 Leaf matching

Matching leaves is similar to matching subtrees. They are assigned to sets of leaves with similar values. The value of a leaf node can be, for example, a variable or a method name. In this case, the similarity is based on the Jaccard index of two bigram sets created from the values of leaves. If the similarity coeficient is greater than a givent value, they are considered similar. This way we can match nodes with changed values (e.g. renamed variables).

Then, for a leaf node from the smaller set we are looking for the most similar leaf from the bigger set. The similarity is calculated in the same way as for subtrees – from predecessors of the leaves – and the existing matching is taken into account, too. Finally, if we can identify the two most similar leaves, they are matched together.

When we talk about leaves in the context of comparing and matching nodes, we mean these leaves together with their parents. The reason for this is that a parent of a leaf node determines the type of this leaf in the AST generated by ANTLR. If there is a method with the same name as the name of a variable, the nodes which represents these names will not be matched together.

3.4 Overall algorithm

Each of the mentioned methods can build a new node matching on an existing one. For this reason, it is useful to run them in multiple iterations until there are no more nodes that can be matched together.

The overall process is shown in Figure 1. It starts with iterations of subtree matching with a decreasing maximum subtree depth. Each of these iterations is followed by the bottom-up inner node matching. Then, the algorithm continues with the leaf matching followed by the inner node matching. At first, the treshold for the leaf value similarity is set to 1, so we are looking for leaves with the same value. This step iterates until there is no change in the node matching. After that, the leaf matching iterations repeat with the treshold set to 0.5, so leaves with changed values are matched.

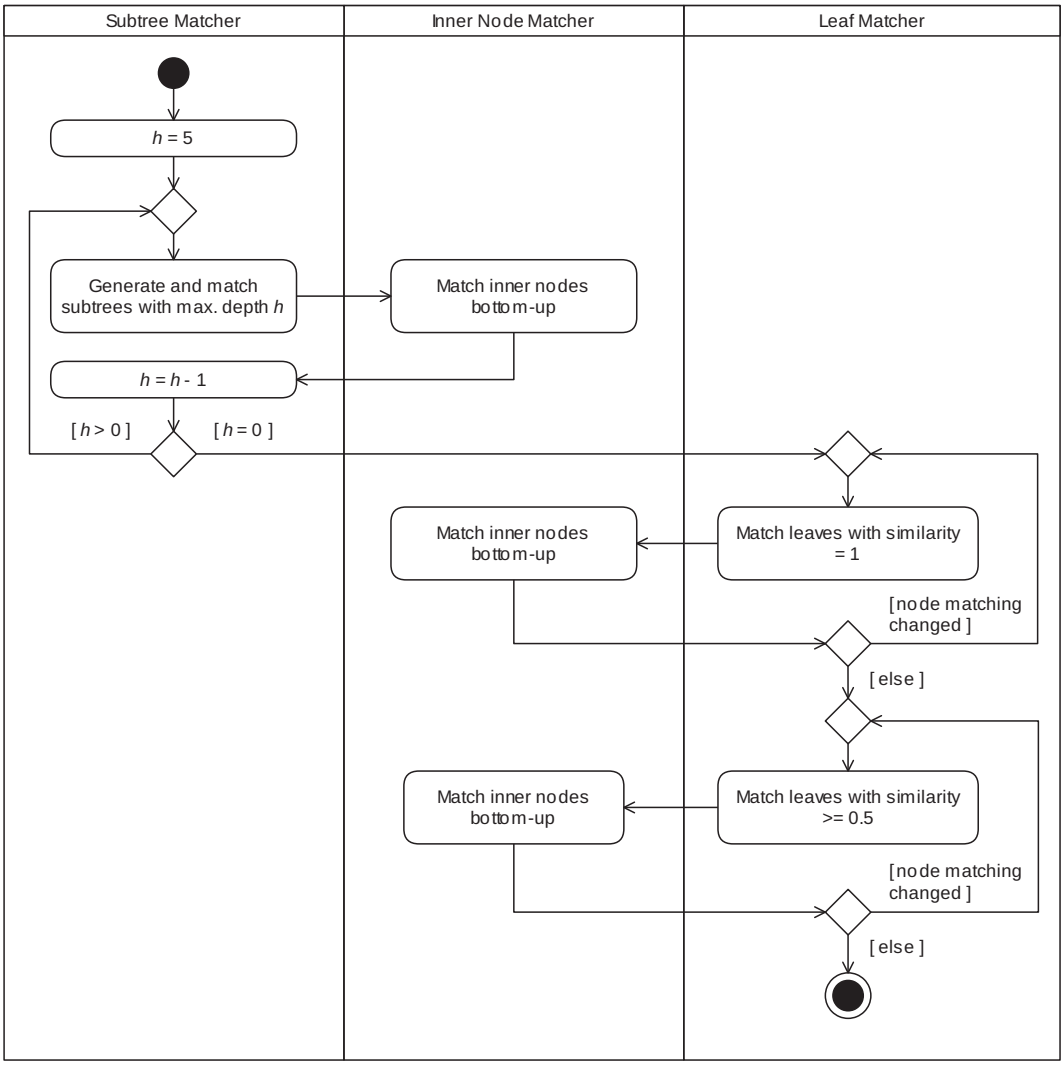


Figure 1. Activity diagram of the overall algorithm.

4 Evaluation

To evaluate our approach we developed a prototype to detect changes in Java language source codes. We tested our tool on a dataset consisting of 25 manually created pairs of source code files. These pairs represented state before and after a code change and covered many different changes which can be seen in real-world software projects. We manually evaluated the output of our prototype for this dataset and summarized the recognized weaknesses of this approach. For each of them we propose a possible solution in this Section.

Based on this evaluation, we can say that about 99% of all matched nodes were matched in the subtree matching part. The precision of our algorithm in this testing was 100%. There were not any false positives, what means that there was not a case of bad matched nodes. All of the observed errors were false negatives – unmatched nodes, that were supposed to be matched. This introduced pairs of deletions and insertions instead of moves in the final edit script. The computed recall varied between 91% and 100% with the average value of 99.3%. The lowest value was caused by an unmatched variable, which was renamed and used in multiple places. The overall f-measure was 99.6%.

However, these values are based on the synthetic dataset and real-world percentages may differ. We are going to perform a deeper evaluation on a dataset based on source codes from real software projects after we implement improvements for the recognized weaknesses and optimize the algorithm.

4.1 Renaming

A common change type in real-world projects is a class, method or variable rename. After renaming, multiple changes occure typically in many places in a source code. In a case of a bigger name change, when the calculated similarity between old and new name is lower than a given treshold, many nodes stay unmatched.

A possible solution for this weakness is a postprocessing step, in wich the ASTs are traversed again. If there are multiple pairs of unmatched leaf nodes with the same "before and after" values, it is probably a rename change in the source code and we can match these nodes together.

4.2 Small subtrees

If there are, for example, small methods in the source code (like setters and getters), into which more lines of code are inserted, the subtree root representing this method can stay unmatched. This happens due to our approach of inner node matching, wich is based on the number of common leaves.

Fluri et al. [3] proposed a solution for these errors – a dynamic treshold for inner nodes similarity. They experienced adequate results for $t = 0.6$ if $n > 4$ and $t = 0.4$ if $n \leq 4$, where n is the number of leaf descendants of the inner node.

4.3 ANTLR trees related weaknesses

Most ASTs built by other tools save a variable-related information (e.g. value, type, modifier) as node attributes. In the case of ANTLR, it is represented by separate leaves. Tree nodes generated by ANTLR do not have attributes, they have only values. For this reason, there is a couple of unmatched nodes. For example, if someone change a variable type from `int` to `double`, corresponding nodes stay unmatched, because their values (strings representing types) are not sufficiently similar.

In this case, the correct matching is uncertain. Should the nodes representing `int` and `double` be matched? Do they represent the same part of a code? Do we want to generate a change operations of removing a variable type and adding another one instead of a change operation of updating a node?

If we want to solve it, we can manually create sets of related node values (e.g. a set for variable types, such as `int`, `double`, `String`) and match node pairs, which values are from the same set. However, this solution is language-specific.

4.4 Leaves with the same similarity

In some cases, two or more leaf nodes in one AST can have the same value, type (specified by their parents) and also the same predecessors. For example, in Java language it can be leaves with value `String` in a subtree representing the code snippet `Map<String, String>`. When these nodes are compared to the nodes in the second AST representing the same source code, they have the same similarity. And because we can not tell which one of them is more similar, they stay unmatched.

We can eliminate this weakness in the leaf matching part. If there are two or more leaves with the same value, type and predecessors, so we can not indentify the most similar one, they will be matched in the order of their occurences in the original source code.

5 Conclusion

In this paper, we propose a method for source code changes detection by comparing two versions of the same code. We use abstract syntax trees as a code representation. The core of our work is comparing and matching nodes between two ASTs. Our method combines three matching approaches: (1) subtree matching, (2) leaf matching and (3) bottom-up inner node matching.

To evaluate this approach, we implemented a prototype and created a dataset, which covered many different code changes. Based on the obtained results, we can say that it is possible to correctly identify the majority of code changes with this method. We describe some recognized weaknesses of our prototype and propose a solution for each of them.

In the future we are planning to implement the mentioned solutions and optimize the parameters of the algorithm (e.g. maximal subtree depth, tresholds for leaf similarity). We are going to perform deeper experiments with our tool on a dataset based on codes from real open-source software projects and compare the results with other available tools.

Acknowledgement: This contribution is the partial result of the Research & Development Operational Programme for the project Research of methods for acquisition, analysis and personalized conveying of information and knowledge, ITMS 26240220039, co-funded by the ERDF.

References

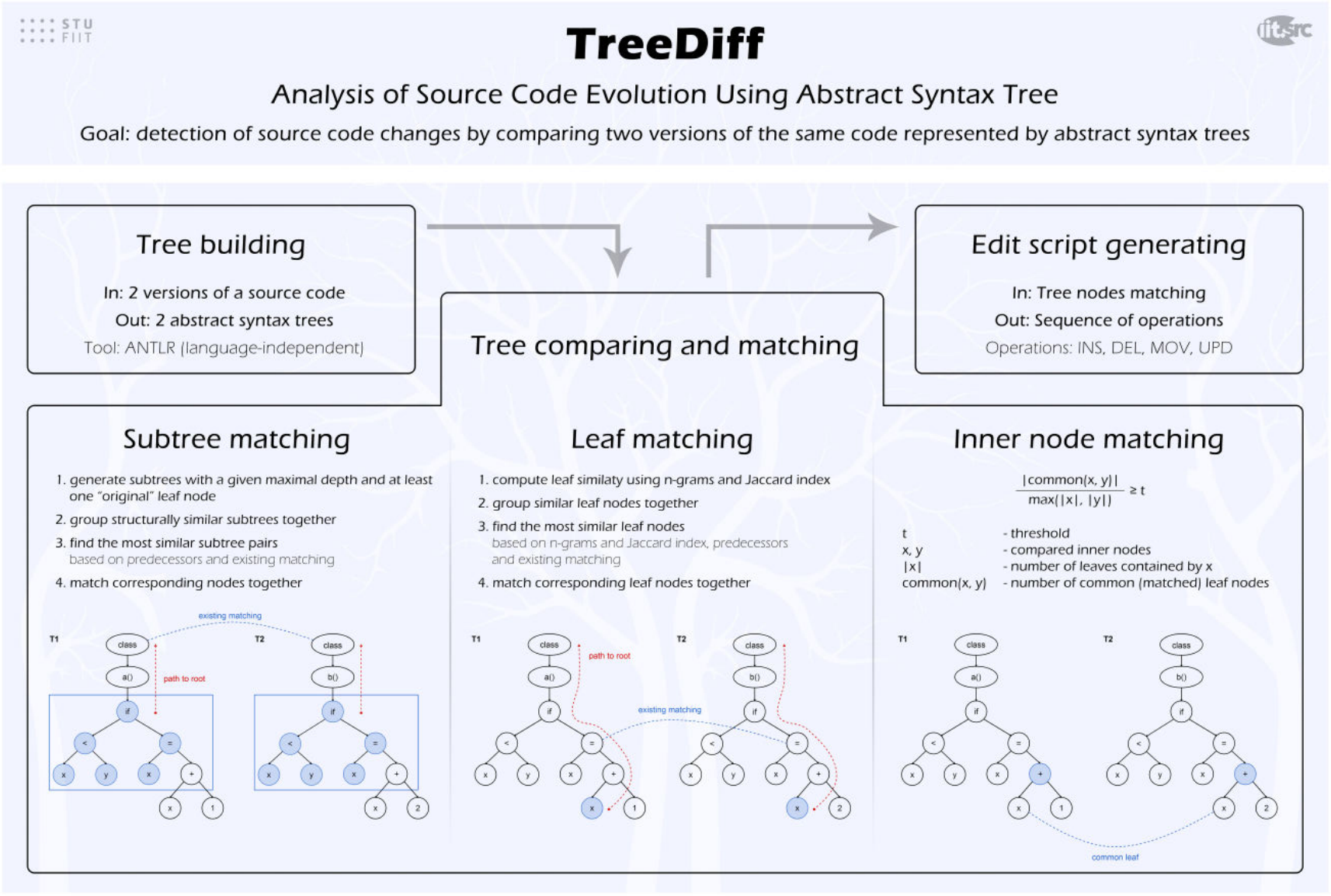
[1] Bille, P.: A Survey on Tree Edit Distance and Related Problems. *Theor. Comput. Sci.*, 2005, vol. 337, no. 1-3, pp. 217–239.

[2] Chawathe, S.S., Rajaraman, A., Garcia-Molina, H., Widom, J.: Change Detection in Hierarchically Structured Information. In: *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. SIGMOD '96, New York, NY, USA, ACM, 1996, pp. 493–504.

[3] Fluri, B., Wuersch, M., PInzger, M., Gall, H.: Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Trans. Softw. Eng.*, 2007, vol. 33, no. 11, pp. 725–743.

[4] Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J.M., Nguyen, T.N.: Clone-Aware Configuration Management. In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. ASE '09, Washington, DC, USA, IEEE Computer Society, 2009, pp. 123–134.

[5] Raghavan, S., Rohana, R., Leon, D., Podgurski, A., Augustine, V.: Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases. In: *Proceedings of the 20th IEEE International Conference on Software Maintenance*. ICSM '04, Washington, DC, USA, IEEE Computer Society, 2004, pp. 188–197.



Obr. C.1: Poster IIT.SRC.

D Obsah priloženého média

Adresár / súbor:	Obsah:
/readme.txt	Obsah média
/document.pdf	Tento dokument
/bin	Skompilované zdrojové súbory
/data	Testovacie dáta
/simple	Umelé dáta
/real	Dáta z reálnych softvérových systémov
/lib	Knižnice využívané programom
/src	Zdrojové súbory programu