beszedes_242_24

Kódelemzési módszerek a hibakeresés és szoftverkarbantartás támogatására

Code Analysis Techniques for Debugging and Software Maintenance

MTA Doktora értekezés tézisei

Beszédes Árpád

Szeged, 2024

beszedes_242_24

Tartalomjegyzék

Célkitűzések és tézisek összefoglalása **2**

1. Dinamikus függőségelemzés **4**
1.1. Gráfmentes programszeletelés 4
1.2. Dinamikus csatolás számítása 6

2. Függőségi klaszterek **8**

3. Spektrum-alapú hibalokalizációs algoritmusok **12**
3.1. Kódlefedettség mérése 12
3.2. Eljárás hívási láncok a hibalokalizációban 14
3.3. Eljárás hívási gyakoriságok a hibalokalizációban 15
3.4. Dinamikus programszeletelés alkalmazása a hibalokalizációban 16

Hivatkozások **18**

beszedes_242_24

Célkitűzések és tézisek összefoglalása

Számos kutatás kimutatta, de a fejlesztők is tapasztalják a napi munkájuk során, hogy a szoftverfejlesztési költségek jelentős hányadát, mintegy 50-75%-át a hibakeresés és javítás (debugging) teszi ki [22, 30, 41]. A különböző mesterséges intelligencia alapú segédeszközök elterjedése mellett is, ez a tevékenység többnyire manuális munka, ami nagy szakértelmet és a fejlesztett szoftver nagyon jó ismeretét igényli. Így, minden olyan megoldás, ami részben vagy teljesen automatizálja a hibakeresés és javítás valamely fázisát, jelentős erőforrás-megtakarítást tud eredményezni.

A hibakeresés egy komplex tevékenység, ami magán a programkódon kívül gyakran egyéb adatokra is támaszkodik, mint amilyenek a tesztesetek, hibajegyek és a különböző dokumentációk. Legtöbb esetben azonban a forráskód a legfontosabb információ-forrás, így annak az elemzése (kód szoftveres analízise) nem megkerülhető [23]. A kódelemzés lehet statikus (a program végrehajtása nélküli), amikor általános információkat gyűjtünk, például a kódelemek közötti függőségeket, vagy dinamikus, amikor konkrét végrehajtások során elemezzük a kódelemek egymáshoz való viszonyát vagy a végrehajtás egyéb tulajdonságait.

Ebben a dolgozatban olyan forráskód elemzési módszereket tárgyalok, amik a hibakeresés támogatását szolgálják különböző aspektusok mentén, és a gyakorlatban is alkalmazhatóak. A hibakeresésen túl ezek a megoldások a szoftverkarbantartás egyéb területein is használhatók úgy, mint a hatásanalízis, programmegértés, utólagos dokumentáció vagy a szoftverminőség-mérés.

Amikor egy hibás programviselkedés alapján szeretnénk a hibát beazonosítani, hogy azután kijavíthassuk, az első lépés annak reprodukálása, vagyis a program futtatása valamilyen tesztesettel. Ebben az esetben, a *dinamikus elemzésen* van a legnagyobb hangsúly, és az értekezésben bemutatott módszerek jelentős része is ezt a területet célozza. Mivel ez a fajta analízis a programvégrehajtást dolgozza fel, nagy mennyiségű adattal kell számolni – ha a programlogika minden részletét követjük, például az egyes utasítások vagy elemi adatok futását.

A dolgozatban bemutatott megoldások közös tulajdonsága, hogy valódi, nagyméretű programok esetében is használhatóak, mivel takarékosan bánnak a tárolt adatokkal, illetve az elemzést sok esetben magasabb granularitású elemeken végezzük, például eljárásokon utasítások helyett.

Tézisek

Az értekezés tézispontjait három területre bontottam. A tézisfüzetben az ennek megfelelő alfejezetekben mutatom be az eredmények legfontosabb elemeit.

Az első téziscsoport a dinamikus függőségelemzés témájával kapcsolatos:

T1.1 **Gráfmentes dinamikus programszeletelési algoritmusok.** Ismertetem azt az általános keretrendszert, amely segítségével több kapcsolódó dinamikus programszeletelési algoritmus is megadható, melyek közös alapja a statikus definiálás-használat kapcsolat és a végrehajtási történet lineáris feldolgozása. Megadom a konkrét algoritmusokat is a legfontosabb tulajdonságaikkal és lehetséges alkalmazásaikkal. *Kapcsolódó publikáció:* [2].

T1.2 **Dinamikus csatolás számítása és alkalmazásai.** Bemutatom az eljáráshívások sorrendiségén és távolságán alapuló dinamikus csatolás fogalmát és a kapcsolódó metrikát. Ezután beazonosítom a dinamikus csatolás lehetséges alkalmazásait a hibakeresés és szoftverkarbantartás területén. *Kapcsolódó publikáció:* [1].

Az alábbi tézispont témája a függőségi klaszterek számítása:

T2 **Függőségi klaszterek hatékony számítása.** Meghatározom a statikus eljáráshívási sorrendiség reláción (SEA) alapuló függőségi klaszter fogalmát, majd ismertetem annak összehasonlítását más, magasabb granularitású klaszterekkel. Bemutatom a függőségi klaszterek elméleti és empirikus vizsgálatának eredményeit és a lehetséges alkalmazásokat. *Kapcsolódó publikációk:* [6, 7, 8, 11, 12].

A harmadik téziscsoport a spektrum-alapú hibalokalizáció javítási lehetőségeivel foglalkozik:

T3.1 **Megbízható kódlefedettség mérési megoldások.** A megbízható kódlefedettség alapvetően szükséges különböző dinamikus kódelemzési feladatokhoz, többek között a spektrum-alapú hibalokalizációhoz is. Ebben a tézisben ismertetem, hogy a kódlefedettség mérésekor milyen gyakorlati nehézségekkel kell szembenéznünk. Bemutatom azt a kiértékelési módszertant, amellyel az egyes lefedettségmérő eszközök objektíven és részletesen kiértékelhetők, valamint ismertetem a konkrét kiértékelésünk során talált problémákat és az azokhoz tartozó kezelési javaslatokat. *Kapcsolódó publikációk:* [10, 14].

T3.2 **Eljárás hívási láncok alkalmazása a hibalokalizációban.** A spektrum-alapú hibalokalizációs módszerek legelterjedtebb változatai egyszerű kódlefedettségen alapulnak. Bemutatom, hogy a programvégrehajtás során számolt eljárások közötti dinamikus hívási láncok hogyan használhatók fel kontextus információként a hibalokalizációban. Ismertetem a hívási láncok különböző felhasználási módjait a meglévő algoritmusokban. *Kapcsolódó publikáció:* [5].

T3.3 **Eljárás hívási gyakoriságok alkalmazása a hibalokalizációban.** A hagyományos kódlefedettség alapú hibalokalizáció nem veszi figyelembe azt, hogy az egyes elemek hányszor hívódtak a végrehajtás során. Ebben a tézisben megvizsgálom, hogy az alap algoritmusok milyen módon adaptálhatók az eljárás hívási gyakoriság kezelésére, valamint ismertetem a különböző módszerek mérési eredményeit. *Kapcsolódó publikációk:* [15, 16].

T3.4 **Dinamikus programszeletelés alkalmazása a hibalokalizációban.** A kódlefedettség alapú hibalokalizáció pontossága úgy javítható, hogy a lefedettség helyett a sokkal pontosabb dinamikus szeleteket használjuk az algoritmus alapját képező spektrum mátrixban. Ismertetem azt az elméleti modellt, amellyel igazolom a kétféle spektrum közötti matematikai kapcsolatot, és hogy a szeletelés alapú hibalokalizáció milyen feltételek mellett és milyen mértékben javítja a lefedettség alapú módszert. Bemutatom továbbá az ezzel kapcsolatos empirikus kiértékelésünk eredményeit. *Kapcsolódó publikáció:* [13].

1. Dinamikus függőségelemzés

A forráskód dinamikus elemzése során a kódelemeket (például utasítások, adatok, eljárások) olyan szempontból vizsgáljuk, hogy a program végrehajtása milyen hatással volt rájuk: végre lettek-e hajtva, ha igen, milyen sorrendben, melyik utasítás/adat/eljárás melyik másikra volt hatással, stb. Ezen belül, a függőségelemzés feladata azt meghatározni, hogy adott konkrét futtatás során (például tesztesetek végrehajtásakor) mely programelem mely másikra volt ténylegesen hatással. Fontos különbség a statikus függőségelemzéssel szemben az, hogy amíg a statikus esetben a *lehetséges* kapcsolatokat vizsgáljuk, addig a dinamikusnál a ténylegesen *megvalósult* függőségek vannak a középpontban. A dinamikus függőségelemzés legfontosabb alkalmazásai a hibakeresés, hatásanalízis és a programműködés megértése.

1.1. Gráfmentes programszeletelés

Az egyik legfontosabb elemzési technika ebben a kategóriában a *programszeletelés* (program slicig) [27, 37, 38], amelyet a szoftverfejlesztés számos területén használnak (például hatáselemzésnél vagy a hibakeresés során). A dinamikus szeletelés [34] a program egy konkrét futása közben követi nyomon az utasítások közötti ténylegesen megvalósuló függőségeket. Szemben a statikus szeleteléssel, ahol az összes *lehetséges* vezérlési- és adatfüggőség alapján számoljuk ki, hogy a szeletelés kiindulópontjaként megadott utasítás (szeletelési kritérium) mitől függhet (hátramutató – backward) vagy éppen mire lehet hatással (előremutató – forward), dinamikus szeletelésnél nem kell figyelembe venni a csak potenciálisan, de ténylegesen nem realizálódó függőségeket. A tényleges függőségek alapján összeállítjuk a program dinamikus szeletét, amely tartalmazza az aktuális futás során a szeletelési kritériumtól függő vagy arra hatással lévő utasításokat.

A dinamikus szeletelésnek számos előnye van, de leginkább az, hogy – a mindössze lehetséges függőségek figyelmen kívül hagyása miatt – pontosabb képet ad az aktuális futás során az egymással valamilyen kapcsolatban álló kódelemekről. Azonban ezen eredmények előállítása nagyon költséges folyamat, ha a naiv dinamikus függőségi gráf alapú megoldást alkalmazzuk [20]. Ennek az alapötlete egyszerű: minden végrehajtott utasítás példányhoz hozzunk létre egy csomópontot a függőségi gráfunkban, majd az utasítások és adatok közötti kapcsolatok alapján határozzuk meg a függőségi éleket. A szelet kiszámítása ebben az esetben egy gráfelérési probléma lesz. Mivel az ilyen gráf mérete a programfutás hosszától függ, a gyakorlatban ez nem praktikus.

A mi módszerünk lényege, hogy nem építjük meg a dinamikus függőségi gráfot, hanem helyette a dinamikus függőségeket a végrehajtási történet feldolgozása közben inkrementálisan számoljuk. Korábbi kutatásunkban egy hátramutató szeletelési algoritmust mutattunk be [9], amely az összes lehetséges dinamikus szeletet globálisan számítja ki és kidolgoztuk az algoritmus részleteit C programokhoz [4]. Kijelenthető, hogy ezen eredményünk új lendületet adott a dinamikus szeletelési algoritmusok kutatásának, mivel több kutatócsoport is alkalmazta, illetve továbbfejlesztette a módszerünket (a [9]-es publikáció 133, a [4]-es pedig 61 független hivatkozást kapott és az utóbbit a konferencia legjobb cikke díjjal is jutalmazták).

Az eredeti ötlet alapján lehetséges volt további gráf nélküli algoritmusok megalkotása [2], amelyek alkalmazása jelentős erőforrás-megtakarítással jár a dinamikus függőségi gráfokhoz képest, viszont ugyanazon eredményt adja. A módszer alapja a statikus *definiálás-használat* viszonyok meghatározása a programelemekhez az adat- és vezérlési folyam alapján, majd a *végrehajtási történet* lineáris feldolgozása, miközben a programszeleteket inkrementálisan számoljuk. Több aspektust figyelembe véve megvizsgáltuk az összes gyakorlati módot, ahogyan a fenti alapokon a hasonló dinamikus szeletelő algoritmusok működhetnek.

Konkrétan, három szempont szerint értékeltük ki a módszereket (1. táblázat):

1. Globális vagy igényvezérelt: A hagyományos (igényvezérelt) megközelítés lényege, hogy egyszerre egy szeletet számít ki egyetlen feltételhez. Lehetőség van azonban több szelet kiszámítására különböző kritériumokhoz, mialatt a nyomkövetési információt csak egyszer dolgozzuk fel. Ezt nevezzük globális szeletelésnek.

2. Szeletelés iránya: Ha olyan elemeket keresünk, amelyek korábbi végrehajtása befolyásolta a feltételnél számított értéket, akkor hátramutató szeletről beszélünk. Az előremutató szelet pedig olyan programhelyek halmaza, amelyek későbbi végrehajtása a szeletelési feltételnél számított értékektől függ.

3. Feldolgozás iránya: A végrehajtási történet a programfutás során történt utasítások és adatok végrehajtása, amit mindkét irányban feldolgozhatunk. Az előrefelé történő feldolgozása tűnik a természetesnek (és egyes alkalmazásokban az egyetlen megvalósíthatónak), azonban a történet információ visszafelé történő bejárása is egy lehetőség, amely bizonyos helyzetekben a gyakorlatban is alkalmazható.

Globális/Igényvezérelt	**Szeletelés iránya**	**Feldolgozás iránya**	**Hasznosság**
Igényvezérelt	hátramutató	visszafelé	praktikus
Igényvezérelt	hátramutató	előrefelé	értelmetlen
Igényvezérelt	előremutató	visszafelé	értelmetlen
Igényvezérelt	előremutató	előrefelé	praktikus
Globális	hátramutató	visszafelé	párhuzamos
Globális	hátramutató	előrefelé	praktikus
Globális	előremutató	visszafelé	praktikus
Globális	előremutató	előrefelé	párhuzamos

1. táblázat. Dinamikus szeletelő algoritmusok áttekintése

Összefoglalva az eredményeket és a tapasztalatokat az alábbi megállapításokra jutottunk. A globális vagy igényvezérelt algoritmus közötti választás attól függ, hogy hány kritériumhoz van szükség szeletekre. Kísérleteink alapján, ha néhány tucatnál több szeletre van szükség, a globális algoritmusok összességében jobb teljesítményt nyújtanak – a pontos szám azonban a tényleges programfüggőségek különböző jellemzőitől függ. A globális algoritmusok közötti választásnak két dimenziója van. A gyakorlati algoritmusok kevesebb adatstruktúrát tartanak fenn és kevesebb memóriát használnak. A végrehajtási történet visszafelé való feldolgozása azonban tárolási okokból problémás lehet, ha a futtatási előzmény hossza nagy. Tehát ebben az esetben az előrefelé feldolgozó globális algoritmusok jobb választások lehetnek, mint a visszafelé feldolgozó párjaik.

Főbb eredmények:

- *Alapelv és keretrendszer a gráfmentes dinamikus programszeletelési algoritmusok megadásához (döntően saját eredmény)*
- *Konkrét megvalósítható algoritmusok megadása (közös eredmény)*
- *Algoritmusok kiértékelése praktikusság és alkalmazási területek szerint (közös eredmény)*

Eredmények hatása: Cikkünkre eddig 14 független hivatkozás történt, ezek közül az egyik egy nemzetközi, egy másik pedig az Amerikai Egyesült Államok területére vonatkozó szabadalom.

1.2. Dinamikus csatolás számítása

A függőségelemzés egy fontos alkalmazási területe a hatásanalízis (impact analysis). Ez egy folyamat, amely során meghatározzuk, hogy egy szoftverben végrehajtott módosítások milyen hatással vannak a rendszer többi részére. A hatásanalízis különösen fontos a szoftverfejlesztés, karbantartás és evolúció során, mivel segít azonosítani a változtatások által érintett komponenseket és minimalizálni a hibák kockázatát, de természetesen a hibakeresésnél is sikerrel alkalmazható. A forráskód-elemzéssel történő hatásanalízisre is igaz, hogy a statikus technikák általánosabbak, de pontatlanabb eredményt adnak konkrét programvégrehajtási szituációkban, így sok esetben a dinamikus megközelítés lehet a jobb választás.

Egy egyszerű dinamikus technika a hatáshalmazok eljárásszintű kiszámításához az eljárások (függvények, metódusok) be- és kilépési eseményeiből számított eljáráshívási sorrendiség (Execute After – EA) szekvenciákon alapul [21]. Ennek lényege, hogy egy adott ponton végrehajtott eljárás hatással lesz az összes, utána végrehajtott eljárásra. Ez egy biztonságos megközelítés (mivel nem maradnak ki függőségek), de egyben pontatlan is.

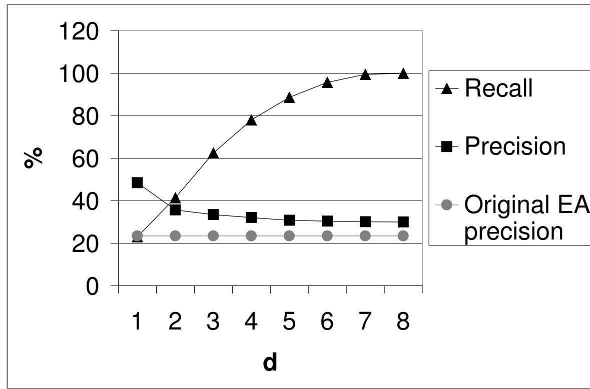
Az EA-relációk azonban egy egyszerű heurisztikával finomíthatók. Az alapötlet azon az intuíción alapul, hogy minél közelebb van két eljárás végrehajtása (a végrehajtási sorrendben) egymáshoz, annál valószínűbb, hogy azok egymástól függenek. A megközelítés a következő: a dinamikus csatolás (Dynamic Function Coupling – DFC) megadja az indirektség minimális értékét két eljárás minden lehetséges előfordulása között a végrehajtási történetben. Más szavakkal, a közvetettség szintje a két eljárás *távolsága*, figyelembe véve a többi, közöttük végrehajtott eljárások számát [1].

Magát a csatolás fogalmát régóta használják a szoftverevolúciós feladatoknál, beleértve a hatásanalízist [28], de a legtöbb megközelítés a program statikus tulajdonságain, különböző típusú függőségein alapul. A mi megoldásunk ezzel szemben a pontosabb dinamikus információkat használja fel.

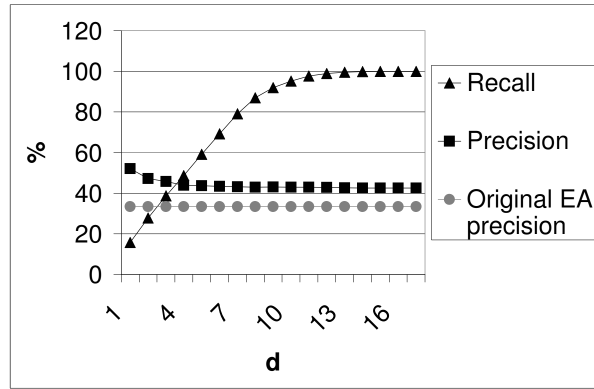
A dinamikus csatolás előnyeit empirikus kísérletben vizsgáltuk. Minden eljáráspárhoz meghatároztuk a DFC metrikát, majd kiszámítottuk egy eljárás hatáshalmazát úgy, hogy azokat a eljárásokat vettük figyelembe, amelyeknek a DFC-je legfeljebb valamilyen rögzített d határértékkel rendelkezik. E heurisztika alapján módszert adtunk a hatáshalmazok fix küszöbértékkel történő kiszámítására, amely elősegíti a hatáshalmazok kinyerését (1) a pontatlan Execute After relációk, (2) a precíz, de költséges dinamikus szeletek és (3) pontatlan és/vagy nem biztonságos statikus függőségi elemzések használata helyett.

A módszer hatékonyságát összehasonlítottuk a dinamikus szeletelés eredményeivel a pontosság és a fedés mérőszámok segítségével, valamint a konzervatív módszerek eredményeivel az elért halmazméret-csökkenés mértéke alapján. Az eredmények számszerűsítéséhez a JSubtitles, a NanoXML és a java2html programokat használtunk a méréseinkben. Az eredmények alapján nagy magabiztossággal kijelenthető, hogy egy kis DFC-érték két eljárás között azt jelzi, hogy nagyobb valószínűséggel van tényleges csatolás közöttük, vagyis az egyik módosítása kihat a másik elemre. A távolság-limit (d) paraméter 5-15 körüli értéke közel 100%-os fedést eredményez a biztonságos módszerhez hasonló (20-30%-os) pontossággal, ezt a pontosságot azonban már sokkal kisebb d értékkel megközelítik (1. ábra), ami után már nem csökken jelentősen.

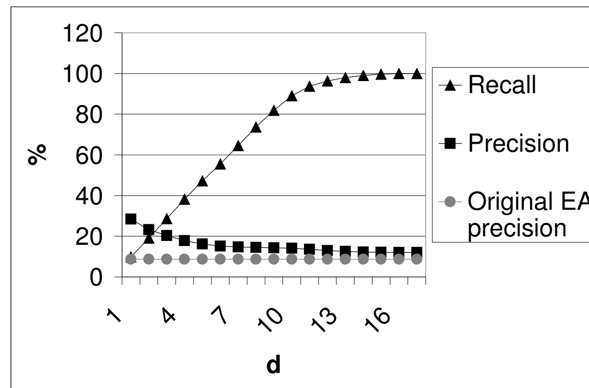
A hatáshalmazok méreteit vizsgálva megállapítottuk, hogy alacsony d értékkel kis halmazok jönnek létre, természetesen alacsony fedés érték mellett. Ugyanakkor, a legjobb pontossági értékek



(a) JSubtitles



(b) NanoXML



(c) java2html

1. ábra. Dinamikus csatolás pontossága (Precision) és fedése (Recall) a konzervatív EA-relációkhoz képest (Original EA precision)

az első vagy a második szinten érhetők el. A hatáshalmazok méretét tekintve a konzervatív megközelítéshez viszonyított relatív nyereség is skálázható a fedés értékekhez hasonló jellemzővel. A legközelebbi 1-es szint ugyanis a biztonságos módszer halmazméreteinek átlagosan 13-15%-át, míg a 2-es szint körülbelül 25-35%-át adja.

Főbb eredmények:

- *Az eljáráshívások sorrendiségén és távolságán alapuló dinamikus csatolás fogalma (döntően saját eredmény)*
- *A DFC metrika kiszámításának módja (közös eredmény)*
- *A módszer mérésekkel történő kiértékelése (közös eredmény)*
- *A dinamikus csatolás lehetséges alkalmazásai a hibakeresés és szoftverkarbantartás területén (döntően saját eredmény)*

Eredmények hatása: Cikkünkre eddig 32 független hivatkozás történt, többek között egy hatásanalízissel és egy programcsatolással foglalkozó összefoglaló (survey) cikkbe is belekerült a munkánk.

2. Függőségi klaszterek

A függőségelemzés egy speciális területe a függőségi klaszterek vizsgálata. Ezek olyan képződmények a szoftverkódokban, melyek egymással szorosan összefüggő elemeket tartalmaznak. Két programelem (pl. utasítás vagy eljárás) közötti függőség azt jelenti, hogy az egyik elem végrehajtása befolyásolhatja a másikat, ezért a fejlesztőknek minden, a két elemet érintő feladatnál tisztában kell lennie ezzel az összefüggéssel. A programkódban a függőségi klaszterek a programelemek maximális halmazai, amelyek egymástól függenek [24], de más definíciók is használatosak, amelyek könnyebben számíthatók. A nagy függőségi klaszterek károsak a szoftverfejlesztési folyamatra; sok folyamatot akadályoznak, beleértve a karbantartást, a tesztelést és a megértést [26, 31]. A fő probléma az, hogy a klaszter bármely elemével való találkozás egy szoftverfejlesztési, hibajavítási vagy karbantartási feladat során azt vonja maga után, hogy az összes többi klasztertagot is meg kell vizsgálnunk, mint vele kapcsolatban álló entitást.

Static Execute After (SEA)-alapú klaszterek

A statikus eljáráshívási sorrendiség reláción (Static Execute After – SEA) alapuló függőségek olyan magasszintű kapcsolatok a progam eljárásai között, amelyek a lehetséges végrehajtási sorrenddel helyettesítik a pontosabb programszelet alapú függőségeket [3], és amelyek a dinamikus EA relációk statikus megfelelői. A SEA relációk számítása nagyságrendekkel kisebb erőforrás igényű a programszeletekéhez képest, és azok konzervatív közelítései. Ez azt jelenti, hogy a SEA függőségek tartalmazzák a hozzájuk tartozó programszeleteket, és a méréseink alapján csak kis mértékben pontatlanabbak azoknál.

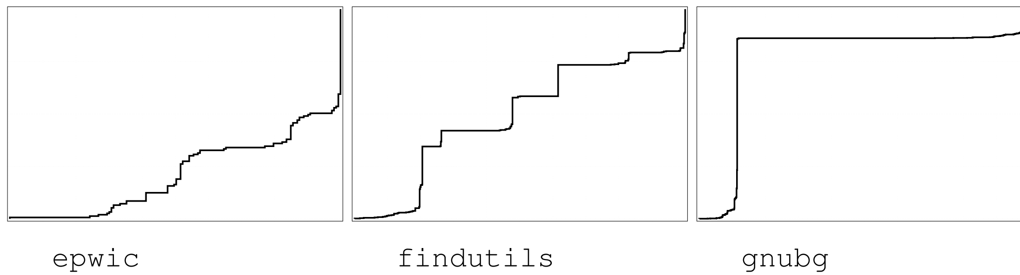
A SEA függőségeken alapuló klaszterek fogalmát úgy határozzuk meg, mint a program eljárásainak olyan maximális halmazát, amelyben bármely két eljárás reflexív SEA függőségi halmazai megegyeznek [11, 12]. Ez a gyakorlatban azt jelenti, hogy egy függőségi klaszterben minden eljárás minden másikkal SEA reláció szerinti kapcsolatban lesz. A függőségi klaszterek vizsgálatánál gyakran a függőségi halmazok méreteinek összehasonlításával közelítik a klasztereket, mivel ez sokkal egyszerűbb elemzést jelent, és viszonylag nagyméretű klaszterek esetében kis valószínűséggel ad csak hibás eredményt [24].

Klaszterek tulajdonságainak vizsgálata

A függőségi klaszterek jelenlétének kimutatása egy adott programban nem egyértelmű feladat, mert az, hogy hány elemszámtól számít jelentősnek egy klaszter program- és klaszterfüggő. A teljes program általános klaszterezettsége is fontos jellemző lehet, de ezt a tulajdonságot sem egyszerű objektíven mérni. Kidolgoztunk ezért egy empirikus kiértékelési módszertant, amely manuális (vizuális) klaszterezettség vizsgálatot és objektív metrikákon alapuló mérést is magában foglal [6, 7]. A 2. ábrán láthatunk egy-egy példát alacsony, közepes és magas klaszterezettségű programra az MSG (Monotone Size Graph) vizualizáció használatával [24]. Egy program MSG-je az összes függő halmazának grafikus ábrázolása úgy, hogy a halmazok mérete monoton növekvő sorrendben van megrajzolva az x tengely mentén.

Kiterjedt empirikus vizsgálatban 29 közepes és nagy C programban megállapítottuk a klaszterezettségi szintet kézi vizsgálattal, majd azt összehasonlítottuk a metrika-alapú osztályozással. Négyféle metrikát használtunk: MSG alatti terület, klaszterek entrópiája és a klaszter méretek regularitása alapján. Azt kaptuk, hogy különböző klaszterezettségi szintek mellett más metrikák alkalmasak azok kimutatására: az entrópia alapú az alacsony és közepes klaszterezettség esetében, a magas klaszterezettségnél inkább a regularitás bizonyult jobbnak.

beszedes_242_24



2. ábra. Példa három különböző szintű klaszterezettségre az MSG vizualizáció alapján (alacsony, közepes, magas)

SEA- és programszelet-alapú klaszterek összehasonlítása

A függőségi klaszterek alapjául szolgáló programfüggőségek számítása különböző algoritmusokkal történhet. Kutatásunkban megvalósítottuk a rendszerfüggőségi gráfokat (System Dependence Graph – SDG) alkalmazó statikus programszeletelés-alapú klasztereket is, majd összehasonlítottuk azokat a SEA-alapúakkal. Mivel a programszeleteket (definíció szerint) tartalmazzák a SEA függőségi halmazok, felmerül a kérdés, hogy maguk a függőségek között mekkorák a különbségek. Azt kaptuk, hogy eljárás szinten átlagosan 11%-al nagyobbak a SEA halmazok. A klaszterek összehasonlítására itt is mindkét módszert alkalmaztuk, a kézi és a metrika-alapú megközelítést. Eredményeink azt mutatják, hogy a klaszterezettségi szint sok esetben megegyezik a két granularitás esetében, és ahol eltérés van, ott általában a SEA-alapú klaszterek lesznek jelentősek. Ez egyrészt igazolja a SEA függőségek alkalmazhatóságát a függőségi klaszterek kimutatására, de azok esetenkénti pontatlanságára is rámutat.

Linchpin elemek azonosítása

Korábbi kutatások kimutatták, hogy sok esetben a szoftver egy viszonylag kis része felelős a függőségi klaszterek kialakulásáért [25]. A függőségi viszonyok szempontjából központi szerepet töltenek be az úgynevezett „linchpin” elemek, amelyek gyakran összetartják az egész programot. Ha a függőségek követésekor figyelmen kívül hagyjuk ezeket, a klaszterek eltűnnek, vagy legalábbis jelentősen csökkennek. A linchpinek definiálásának általános megközelítése egy olyan programelem megtalálása, amelynek eltávolítása a klaszterek legnagyobb csökkenését eredményezi egy adott metrika szerint.

A klaszteren belüli elemeket összekötő linchpin entitások vizsgálatához elkészítettünk két manuális elemzést (*gold* és *silver standard*), valamint a metrika-alapú meghatározását is [8]. Manuálisan megvizsgáltuk a releváns metódusokat és osztályoztuk azokat aszerint, hogy mennyire tartják egyben a klasztert, vagyis mennyire esne szét, ha az adott elem eltávolításra kerülne a függőségi klaszterből. Amennyiben egyértelműen és jelentősen csökkenne a klaszter mérete, akkor ez az elem a gold standard része, ha a csökkenés nem ilyen nagy, de még jelentős mértékű, akkor a silver standardba kerül.

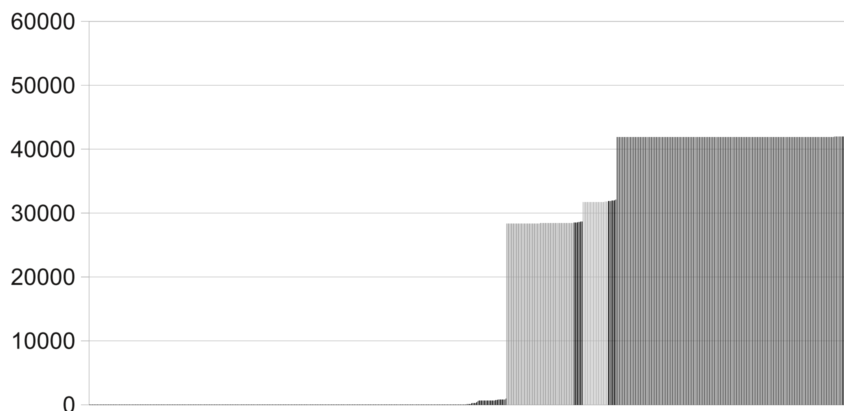
Emellett a klaszterezettségi metrikákat [6, 7] felhasználva kidolgoztuk a metrika-alapú linchpin azonosítási módszert. A nyers erővel történő beazonosítás azt jelenti, hogy a programelemeket egyesével eltávolítjuk a programból, újraszámoljuk a függőségeket és a klasztereket, majd meghatározzuk a klaszterezettségi metrikákat. Azok az elemek, amelyek a legnagyobb metrika-csökkenést eredményezik, azokat tekintjük linchpinnek. Ez a módszer azonban nem alkalmazható a gyakorlatban a költségei miatt, ezért különböző heurisztikus megközelítésekkel kísérleteztünk. Azt kaptuk, hogy néhány hagyományos statikus kódmetrika jól korrelál az elemek linchpin jelle-

gével, így a továbbiakban azokat használtuk. Ilyen például az eljárásokból kimenő hívások száma (NOI). A kézi és a metrikával történő beazonosítási módszer összehasonlításából kiderült, hogy a metrika alapú módszer kiemelkedően jól teljesít, különösen a gold standard linchpinek esetében. Más szóval, ha a linchpin létezése egyértelmű, az metrikus alapú megközelítéssel nagy sikerrel megtalálható.

Függőségi klaszterek használata a hatásanalízisben

Kutatásokat végeztünk, amelyekben megvizsgáltuk a SEA-alapú klaszterek tulajdonságait, valamint kiértékeltük a kapott eredményeket a WebKit rendszeren alkalmazva a módszert [11, 12]. A WebKit egy webböngésző motor, amit többek között a Safari és Chrome böngészők is használnak. Ez a rendszer több mint 2 millió C++ nyelvű programsort tartalmaz, a programelemek közötti bonyolult kapcsolatrendszerrel. A verziókövető rendszerében sok százezer revízió található, ami naponta akár 100 új változattal gyarapszik. Emiatt a WebKit egy nagyméretű valódi ipari alkalmazásnak minősül és remekül alkalmas a kísérleteink elvégzésére ebben a témában.

Hatáshalmaznak ebben a kutatásban az adott programelem (eljárás) SEA-alapú függőségeinek halmazát tekintettük. Ehhez megvizsgáltuk, hogy a WebKit-ben lévő hibákat bevezető kódváltoztatások hatáshalmazai tartalmazzák-e a módosításokat a javított revízióknál (ezt nevezzük a hatásanalízis előrejelző képességének). Kiszámoltuk az összes eljárásra vonatkozó SEA hatáshalmazt és MSG vizualizációikat a Webkit programra. A 3. ábra a WebKit r91555 változatának grafikonját mutatja, ahol három, vizuálisan is jól elkülöníthető, szürke színekkel jelölt klasztert azonosítottunk. Ezen felül 4, a klaszterhez nem tartozó régiót lehet beazonosítani az ábrán, amiket sötétebb színnel jelöltünk.



3. ábra. SEA-alapú függőségi klaszterek a WebKit rendszerben (y tengely: SEA halmaz mérete)

A 2. táblázat második és harmadik oszlopa ezeket a méreteket mind a hatáshalmazainak számában, mind az összes hatáshalmazhoz viszonyított százalékban mutatja. A legnagyobb klaszter az összes hatáshalmaz csaknem 1/3-át (29.46%) veszi fel, míg mindhárom összege 41,53%, ami azt jelenti, hogy az összes hatáshalmaz majdnem fele néhány nagy klaszterben található. A táblázat utolsó oszlopa a három klaszter hatáshalmazának méretét mutatja. Mivel a WebKitben körülbelül 92.000 eljárás található, megállapíthatjuk, hogy bár sok nagy hatáshalmaz létezik, a legnagyobb önmagában az egész programnak körülbelül a felét tartalmazza.

Megvizsgáltuk, hogy mi a jellemző azokra a metódusokra, amelyek hibajavítást tartalmaznak, vagyis volt olyan commit, ami változtatott a metóduson és ennek hatására egy addig bukó (failed)

teszt eredménye sikeres (passed) lett. Azt szerettük volna megtudni, hogy mi jellemző ezekre a hibákat javító változtatásokra: többnyire klaszterhez tartozó hatáshalmazok elemei-e vagy sem? Ehhez az összes hatáshalmaz listáját leszűrtük azokra, amelyek legalább egy javítási változást tartalmaztak a megfelelő verzióban. Összesen 2021 ilyen eljárás volt.

A 2. táblázat negyedik és ötödik oszlopa azt mutatja, hogy a szűrt hatáshalmazok közül hány tartozott az MSG-ben azonosított régiókhoz. Megfigyelhető, hogy százalékos arányban minden klaszter több hatáshalmazt tartalmazott a szűrt halmazból, de leginkább a Klaszter3, amely megduplázódott. Összességében a hiba-előrejelző hatáshalmazok 77,93%-a tartozik néhány nagy klaszterhez.

Összefoglalva, az összes hatáshalmaz közel fele valamely nagy klaszterbe tartozik, míg a meghibásodást előrejelző hatáshalmazoknak lényegesen nagyobb része, több mint 3/4-e, tartozik ebbe a kategóriába. Ez azt mutatja, hogy a SEA-alapú függőségi klaszterek alkalmasak a program hibára hajlamos részeinek előrejelzésére.

	Össz eljárás	Százalék	Előrejelzett eljárások	Százalék	SEA méret
Régió_1	49623	54.42%	360	17.81%	–
Klaszter_1	8028	8.80%	180	8.91%	28395
Régió_2	1127	1.24%	28	1.39%	–
Klaszter_2	2981	3.27%	156	7.72%	31720
Régió_3	1040	1.14%	15	0.74%	–
Klaszter_3	26864	29.46%	1239	61.31%	41892
Régió_4	1530	1.68%	43	2.13%	–
Össz	91193	100.00%	2021	100.00%	–

2. táblázat. Függőségi klaszterek mérete és hatáshalmazok

Főbb eredmények:

- *A statikus eljáráshívási sorrendiség reláción (SEA) alapuló függőségi klaszter fogalma (döntően saját eredmény)*
- *A függőségi klaszterek tulajdonságainak vizuális és metrika-alapú vizsgálata (közös eredmény)*
- *A SEA-alapú klaszterek összehasonlítása más, magasabb granularitású klaszterekkel (közös eredmény)*
- *Linchpin elemek heurisztikus beazonosítása (közös eredmény)*
- *A SEA-alapú klaszterek használata a hatásanalízisben (közös eredmény)*

Eredmények hatása: Kapcsolódó cikkeinkre eddig 12 független hivatkozás történt, külföldi társszerzőinkkel azóta további kutatásokat folytatunk. A [6] sorszámú publikációnk elnyerte a konferencia legjobb cikke díjat.

3. Spektrum-alapú hibalokalizációs algoritmusok

A hibakeresés (debugging) egyik első lépése annak megállapítása, hogy adott hibás viselkedésért (ami bukó tesztesetek formájában manifesztálódik) mely programrészek a felelősek, ami jelenthet utasítást, eljárást vagy egyéb kódblokkot. Ezt a tevékenységet legtöbbször hibalokalizációnak (Fault Localization – FL) szokták hívni. Ezután következik a hibajavítás, majd az ellenőrző és regressziós tesztelés teszi teljessé a folyamatot.

A hibalokalizációs eljárásoknak nagy a szakirodalma, és számos különböző megközelítést fejlesztettek ki, amik részben vagy teljesen automatizálják a folyamatot [40]. Sajnos nincs teljesen pontos módszer, hiszen a hibák is nagyon sokfélék lehetnek, és különböző szituációkban más-más megközelítés lehet az előnyös. Ebben a dolgozatban elsősorban a *spektrum-alapú hibalokalizáció-val* foglalkozunk (SBFL – Spectrum Based Fault Localization) [29].

Ez az algoritmus-család a részletes kódlefedettség eredményeken alapszik és ezen információk felhasználásával határozza meg a hiba szempontjából leggyanúsabb elemeket. A módszerek lényege, hogy a programra meghatározzuk az úgynevezett *spektrumot*, ami egy lefedettségi mátrixból és egy eredményvektorból áll. A mátrix sorai a teszteseteket, oszlopai pedig a kódelemeket reprezentálják (pl. utasítások vagy eljárások) és egy elem értéke 1, ha az adott teszt végrehajtása során az adott kódelem meghívásra kerül, különben 0. Az eredményvektor megadja, hogy a tesztesetek eredménye sikeres (0) vagy bukó (1) volt.

Ezekből az információkból kiszámításra kerül minden kódelemhez, hogy mi az ő spektrum metrikája, vagyis az a szám-négyes, amely megadja, hogy hány olyan sikeres ill. sikertelen teszt volt, amely a (teszt)futtatás során végrehajtotta az adott kódot (*ef - executed failed, ep - executed passed*), valamint mennyi olyan sikeres ill. sikertelen teszt volt, amely nem érintette a vizsgált elemet (*nf - not executed failed, np - not executed passed*). Ezekből különböző képletek segítségével meghatározható a gyanússági érték, amely aztán a (gyanússági) rangsor alapja lesz (pl. Ochiai [19], Tarantula [32], DStar [39]). Az alap intuíció abból adódik, hogy egy kódelem annál gyanúsabb lesz, minél több bukó teszteset érinti és minél kevesebb sikeres.

3.1. Kódlefedettség mérése

Az SBFL módszerek hatékony működéséhez elengedhetetlen a megfelelő minőségű kódlefedettség-információ megléte, ezért megvizsgáltuk, hogy a különböző módszerek és eszközök mennyire precízen tudják meghatározni ezeket az adatokat és milyen gyakorlati nehézségekkel kell szembenéznünk a használatuk során [10, 14]. A legelterjedtebb forráskód és bájtkód alapú megközelítést használó eszközöket elemeztük (Clover [17] ill. JaCoCo [18]) 8 referencia-programon keresztül, kvantitatívan és kvalitatívan egyaránt, ami részletes és objektív kiértékelést tesz lehetővé. A kiértékelésnél meghatároztuk a különbségeket (mennyiben térnek el az eredmények), azok fő okait (mi miatt alakultak ki az eltérések), valamint ezek javítási lehetőségeit is. Részletesen megvizsgáltuk a lefedettségmérés hiányosságainak lehetséges alkalmazásokra gyakorolt hatását is.

Az eredmények azt mutatták, hogy jelentős eltérés mutatkozhat a lefedettség-eredményekben a két koncepció között (3. táblázat). Megállapítottuk, hogy az összesített lefedettségi különbségek mindkét irányban változhatnak, a nyolc programból hét esetében legfeljebb 1,5% volt az eltérés (a checkstyle esetében azonban rendkívül nagy, 40%-os eltérést mértünk, amit a generált kód eltérő kezelésének tulajdonítottunk).

Az eltérések okait megvizsgálva azt találtuk, hogy vannak olyan eszközspecifikus tulajdonságok, mint az eszközök eltérő almodul-kezelése, vagy a setup- és teardown-kezelés; ezek függetlenek a kiválasztott lefedettségi módszertől. Ezek kiküszöbölhetők az eredmények szűrésével, bár ez nem

beszedes_242_24

Program	JaCoCo	Clover	különbség
checkstyle	53.77%	93.82%	-40.05%
commons-lang	92.92%	93.28%	-0.36%
commons-math	84.92%	84.65%	+0.27%
joda-time	89.52%	89.94%	-0.42%
mapdb	74.64%	76.06%	-1.42%
netty	40.92%	40.18%	+0.74%
orientdb	27.01%	28.01%	-1.00%
oryx	29.51%	28.75%	+0.76%
átlag	**61.65%**	**66.84%**	**-5.19%**

3. táblázat. Lefedettség-értékek és azok eltérése

teljesen automatizált. Más eszközspecifikus jellemzők – mint például az instrumentálás hatása a vizsgált rendszer tesztjeinek a viselkedésére – az eszközök szerves részét képezik, és általában nem kerülhetők el, nem befolyásolhatók, nem vagy csak minimálisan változtathatók.

Összességében megállapítottuk, hogy a lefedettségi eredményekben lévő különbségek egy része kiküszöbölhető további információk felhasználásával, de nem mindegyik. Mivel az eszköz használójától eleve nem várható el ezen korrekciós műveletek megtétele, általános tanácsként az eszköz használóinak meg kell vizsgálniuk az eszköz konkrét elveit és lehetőségeit, valamint tisztában kell lenniük a korlátaival. A különbségek lehetséges okait tartalmazó listánk útmutatóként használható arra vonatkozóan, hogyan lehet elkerülni és megkerülni a bájtkód alapú eszközök pontatlanságait a forráskód alapú megközelítésekhez képest.

Főbb eredmények:

- *Kiértékelési módszertan, amellyel az egyes lefedettségmérő eszközök objektíven és részletesen kiértékelhetők (döntően saját eredmény)*
- *A forráskód és a bájtkód alapú lefedettségmérés összehasonlítása konkrét eszközökkel és referencia-programokkal (közös eredmény)*
- *A különbségek kvantitatív elemzése (közös eredmény)*
- *Az eltérések okainak kvalitatív feltárása (közös eredmény)*
- *A pontatlan lefedettség-mérés hatásainak vizsgálata konkrét alkalmazási területeken (közös eredmény)*

Eredmények hatása: Kapcsolódó cikkeinkre eddig 24 független hivatkozás történt, leginkább a lefedettség mérés alkalmazói oldaláról, ami eredményeink hasznosságát mutatja.

3.2. Eljárás hívási láncok a hibalokalizációban

A hagyományos SBFL spektrum az egyszerű kódlefedettségre épít, és a kapcsolódó gyanússági képletekkel való kutatás, amely nem használ további információkat, mára telítődni látszik. Ezeknek az alapvető megközelítéseknek a további módosítása, finomítása csak elenyésző javulást eredményezhet, és a probléma megközelítésében általánosabb változtatásokra van szükség a számottevőbb javulás elérése érdekében, például további információk bevonása a FL-folyamatba.

Kísérletet tettünk arra, hogy a dinamikus programvégrehajtási adatokból kinyerhető további információkkal okosítsuk fel az SBFL algoritmusokat, ezáltal növelve azok hatékonyságát [5]. Bevezettük az *eljárás hívási lánc* (call chain) fogalmát, amely az eljárások egymáshoz viszonyított dinamikus hívási információival, mint kontextus információval egészíti ki az SBFL spektrumot. A hívási lánc eljárások olyan leghosszabb sorozata, amelyben az eljárások a programvégrehajtás során egymást közvetlenül meghívták. Ez a hívási verem egy konkrét állapotának is megfeleltethető. Ez az adat könnyen felépíthető, és fontos információt hordoz a hibalokalizációhoz: adott bukó tesztesetek milyen konkrét hívási verem állapotokat eredményeztek.

Az új módszer szerint az eljárás hívási láncokból felépíthető egy speciális kódlefedettségi mátrix, ahol az oszlopok nem a kódelemeket, hanem az egyes láncokat reprezentálják. Így, a hívási láncokra kapjuk meg a gyanússági értékeket, amiket további ötletes algoritmusokkal visszavetítünk egyedi kódelemekre, konkrétan eljárásokra.

Az algoritmust a Defects4J nevű Java hibaadatbázison értékeltük ki [33] és azt találtuk, hogy a kombinált módszert használó SBFL eljárások hatékonyabban azonosítják a hibás metódust, mint a hagyományos kódelemeket tartalmazó spektrum-alapú megközelítések. Két kiugró érték kivételével (Chart és Closure programoknál) a javasolt megközelítés jelentős, mintegy 19-48%-os rang javulást érhet el azokban az esetekben, amikor a hagyományos SBFL eljárások rosszul teljesítenek (4. táblázat - Rang-változás oszlop). Ezen kívül összesen 44 olyan eset volt, amikor az eredeti Ochiai módszer a rangsorban a tizediknél rosszabb pozícióba sorolta a hibás elemet, de a kombinált módszer hatására bekerült a 10 leggyanúsabb elem közé és ezzel átlagosan 43 pozícióval került előrébb a rangsorban az adott hibás elem (4. táblázat – utolsó két oszlop).

Program	**Hibák**	**Ochiai átl. rang**	**Kombinált átl. rang**	**Rang-különbség**	**Rang-változás**	**Ochiai > 10**	**Javítás**	**Rang-javítás**
Chart	25	8.3	10.8	2.4	29%	5	2	-19.0
Closure	173	99.5	131.4	31.9	32%	106	16	-58.8
Lang	60	4.7	3.5	-1.1	-24%	7	4	-15.4
Math	92	11.0	7.3	-3.7	-34%	27	17	-28.1
Mockito	28	25.6	20.6	-5.0	-19%	9	3	-92.0
Time	26	18.3	9.5	-8.8	-48%	7	2	-49.2
Össz / Átlag	**404**	**49.3**	**61.1**	**11.9**	**24%**	**161**	**44**	**-43.0**

4. táblázat. Hívási lánc alapú hibalokalizációs módszer mérési adatai

A hívási láncok további hasznos információkkal szolgálnak a hiba kontextusának jobb megértéséhez (az esetek 69%-ában a hibás elem a legnagyobb gyanússági értékű láncokban volt). Ezek az eredmények mind azt támasztják alá, hogy további információk hozzáadásával jelentősen javítható az SBFL módszerek hatékonysága.

Főbb eredmények:

- *Eljárás hívási láncok fogalmának definiálása és kapcsolata a hívási vermekkel (döntően saját eredmény)*
- *Eljárás hívási láncok kontextusként való alkalmazása a spektrum-alapú hibalokalizációban (közös eredmény)*
- *Hívási lánc alapú SBFL algoritmusok kidolgozása, melyek eljárás szintű lokalizációt adnak eredményül a kapcsolódó empirikus mérésekkel (közös eredmény)*

Eredmények hatása: Kapcsolódó cikkünkre eddig 5 független hivatkozás történt.

3.3. Eljárás hívási gyakoriságok a hibalokalizációban

A hibás program végrehajtásakor meghívott eljárások további fontos információt szolgáltathatnak a hibalokalizációhoz: ez a hívási gyakoriságuk különböző kontextusokból. Az alap intuíció az, hogy ha egy hibás kódelem több különböző kontextusból is meghívódik egy hibás teszteset futtatásakor, akkor ez az elem nagyobb valószínűséggel felelős a hibáért. A hívási gyakoriság meghatározásának legegyszerűbb módja az, hogy a teszteset futtatásakor összeszámoljuk, hogy az adott eljárás összesen hányszor lett meghívva, függetlenül attól hogy melyik másik eljárás által (ezt hívjuk *naiv* módszernek). Ez az információ több, mint a hagyományos lefedettség-alapú SBFL megközelítés, hiszen nem csak azt a tényt rögzítjük, hogy meghívódott-e az adott eljárás, hanem azt is, hogy hányszor. Ugyanakkor, mint kiderült, ez nagyon pontatlan eredményt ad a legtöbb esetben, aminek a legfőbb oka az, hogy egy eljárást gyakran hívjuk ciklusokból, és így könnyen olyan nagy gyakoriságszámok adódnak, amik nem adnak hasznos kontextust, sőt, sok esetben eltorzítják az eredményt.

Ezért, kidolgoztunk egy hívási láncokon alapuló hívási gyakoriság mérést, aminek az a lényege, hogy egy eljárás hívási gyakoriságát a különböző hívási vermekben való előfordulási gyakoriságaként értelmezzük (ami a láncokból meghatározható) [15, 16]. Ez tulajdonképpen a különböző hívási kontextusok számát jelenti, és nem számol a ciklusokból származó ismételt hívásokkal, ami sokkal hasznosabb információ a hibalokalizáció szempontjából. Ezután, a hívási gyakoriságokkal lecseréljük a hagyományos lefedettségi értékeket a spektrum mátrixban, majd adaptáljuk a négy spektrum metrikát és a gyanússági formulákat, hogy kezeljék ezt az új információt.

Az új módszerünket empirikusan kiértékeltük, amihez a Defects4J hibaadatbázist használtuk. Mindkét megközelítést (naivot és hívási lánc-alapút) megvizsgáltuk 9 különböző formulával és azok 4-4 variánsával, és megmértük a hibalokalizáció sikerességét különböző mérőszámokkal. Az eredmények alapján egyértelműen megállapítható, hogy a naiv módszer hatékonysága elmarad a hagyományos lefedettség-alapú mátrixon alapuló módszerek hatékonyságától. Azonban a hívási láncok alapján számított gyakoriság-értékek használatával javítható az SBFL hatékonysága.

Az 5. táblázatban láthatjuk a hibás elemre elért átlagos rang pozíciókat a hagyományos lefedettség-alapú hibalokalizáció esetében („eredeti" sor), valamint a két gyakoriság-alapú megközelítést minden formulára (oszlopokban) és mind a négy variációra (sorokban). A naiv megközelítés minden esetben rosszabbul teljesít, mint az hagyományos megoldás, de a hívási lánc alapú módszernél 9 formulából 6 esetben átlagosan 5-101 pozícióval javítottuk a hibás metódus gyanússági rangsorban elfoglalt pozícióját.

`beszedes_242_24`

		B	D	G	J	N	O	R	S	T
eredeti		36.01	33.99	**43.30**	36.06	**43.30**	33.98	135.96	36.06	**36.01**
naiv	Variáns-1	63.46	97.94		63.55		94.76	167.78	64.11	
	Variáns-2	62.60	97.94	153.43	62.89	153.43	62.59	167.41	62.89	37.09
	Variáns-3	42.73	66.14	153.98	43.03	153.60	41.85	73.15	43.03	44.34
	Variáns-4	42.73	66.38	153.98	43.10	153.45	41.74	126.12	43.10	44.80
lánc-alapú	Variáns-1	24.68	35.60		24.63		36.05	70.80	24.89	
	Variáns-2	**24.55**	35.60	66.73	**24.40**	66.73	**24.30**	70.60	**24.40**	36.87
	Variáns-3	38.63	**29.08**	67.19	38.64	67.04	36.44	**35.12**	38.64	85.35
	Variáns-4	38.63	29.13	67.19	38.34	67.00	36.99	54.95	38.34	74.56

5. táblázat. A hibák átlagos rangja az eredeti és a gyakoriság-alapú módszerek alapján (formulák: B - Barinel, D - DStar, G - GP13, J - Jaccard, N - Naish, O - Ochiai, R - Russel-Rao, S - Sorensen-Dice és T - Tarantula)

Főbb eredmények:

- *Eljárások kétféle hívási gyakoriságának definíciója, naiv gyakoriság továbbfejlesztése hívási verem felhasználásával (döntően saját eredmény)*
- *Hívási lánc-alapú gyakoriság felhasználása az FL algoritmusok javítására (közös eredmény)*
- *SBFL spektrumok, spektrum metrikák és formulák adaptálása az új, gyakoriság-alapú módszerekhez a kapcsolódó empirikus mérésekkel (közös eredmény)*

Eredmények hatása: Kapcsolódó cikkeinkre eddig 6 független hivatkozás történt.

3.4. Dinamikus programszeletelés alkalmazása a hibalokalizációban

A kódlefedettséget használó hagyományos spektrum-alapú hibalokalizáció spektrum mátrixa viszonylag könnyen meghatározható, hiszen a kódlefedettség elvi szinten egy egyszerű dinamikus kódelemzés típus (de mint azt korábban láthattuk, sok gyakorlati nehézséggel kell szembenéznünk). A módszer eredendő problémája viszont az, hogy attól, hogy egy teszteset futtatásában egy kódelemet érintettünk, nem biztos, hogy az a kimenetre hatással is volt, vagyis, hogy hozzájárult a tapasztalt hibához. A megoldás az, hogy kódlefedettség információ helyett hátramutató dinamikus szeleteket használjunk, amelyek a program kimenetből számított pontos függőségeket tartalmazzák. Ezt a hiányosságot felismerték már a kutatók, viszont annak a hibalokalizáció pontosságára való hatásáról kevés tanulmány létezik [35, 36].

A kapcsolódó kutatások áttekintése alapján azt találtuk, hogy nagyon fontos lehet a további spektrum-alapú hibalokalizáció kutatásokhoz az, ha pontosabb képünk lenne arról, hogy a lefedettség pontatlansága a programszeletekhez képest milyen elméleti és gyakorlati különbségeket eredményez a hibalokalizáció tekintetében [13].

Mivel a megfelelően előállított dinamikus programszelet a kódlefedettség részhalmaza, felállítható egy olyan elméleti modell, ahol a kettő különbsége a spektrum formulák alapján levezethető. A modell segítségével sikerült néhány legfontosabb formula esetében kimutatni, hogy a kódlefedettség alapú hibalokalizáció minden esetben szükségszerűen pontatlanabb gyanússági rangsort eredményez a programszelet alapúhoz képest. Továbbá, megadtuk az összefüggést a várható különbségekre, feltételezve a programszelet a kódlefedettséghez képesti pontatlanságát/arányát.

beszedes_242_24

Megvalósítottunk egy kísérleti programszelet-alapú SBFL elemzőt, amelyben először kiszámoljuk a tesztekben levő assert-ekből induló hátramutató dinamikus szeleteket. Ebben a módszerben a spektrum mátrix sorai nem a teszteket, hanem a tesztekben levő assert-eket reprezentálják és egy elem a mátrixban akkor kap 1-es értéket, ha az adott kódelem benne volt az adott assert-hez tartozó programszeletben. Maga a hibalokalizáció innentől identikus a lefedettség-alapú módszeréhez, így az elért eredmények is összehasonlíthatók.

A kvalitatív és kvantitatív vizsgálatot a Defects4J-ben levő JodaTime nevű programon végeztük. Az eredmények azt mutatják, hogy jelentősen csökken a keresési tér a programszelet alapú spektrumban, vagyis az egyes assertekhez tartozó függőség adatok mérete jóval kisebb, mint a kódlefedettséghez tartozóké (kevesebb, mint a fele). Ennek megfelelően a hibalokalizációs képessége is jobb ennek a módszernek, mint a tesztlefedettségre épülő eljárásoknak. A 6. táblázatban láthatjuk a felhasznált formulák esetében (oszlopokban), hogy az egyes megvizsgált hibákhoz milyen rangok tartoznak a lefedettség és a szelet spektrumokkal. Átlagosan 10-32 pozícióval rosszabb eredményeket kapunk a lefedettség alapú módszerrel, mint a programszeleteléssel. A kapott különbségeket kvalitatívan is tételesen kiértékeltük. Eredményünk azt mutatja, hogy mivel elméleti szinten biztos hátrányos a lefedettség alkalmazása, de a gyakorlatban ilyen jelentős eltérések lehetnek, érdemes további kutatásokat folytatni ezen a területen.

Hiba	B		D		J		O		S		T	
	lef.	szel.	lef.	szel.	lef.	szel.	lef.	szel.	lef.	szel.	lef.	szel.
4	23.5	1.5	20.5	1.5	23.5	1.5	23.5	1.5	23.5	1.5	23.5	1.5
9	3	11	2	11	3	11	3	11	3	11	3	11
10	21.5	9	11.5	10	19.5	10	15.5	10	19.5	10	21	9
12	2.5	2.5	29.5	2.5	8	2.5	5	2.5	8	2.5	2.5	2.5
16	8	10	8	11	8	11	8	11	8	11	8	10
17	5	5	5	5	5	5	5	5	5	5	5	5
22	23.5	14.5	23.5	13.5	23.5	14.5	23.5	13.5	23.5	14.5	23.5	14.5
23	25.5	20.5	25.5	20.5	25.5	20.5	25.5	20.5	25.5	20.5	25.5	20.5
26	270.5	20.5	60.5	10.5	132.5	19.5	71.5	15.5	132.5	19.5	156.5	20.5
átlag	42.6	10.5	20.7	9.5	27.6	10.6	20.1	10.1	27.6	10.6	29.8	10.5

6. táblázat. Hibás utasítások rangja a JodaTime programban (formulák: B - Barinel, D - DStar, J - Jaccard, O - Ochiai, S - Sorensen-Dice és T - Tarantula)

Főbb eredmények:

- *Elméleti modell, amely leírja a lefedettség és a programszelet által meghatározott spektrum-alapú hibalokalizációk közötti kapcsolatot, valamint a különbségek mértékét a lefedettség pontatlansága függvényében (döntően saját eredmény)*
- *A tesztesetekben lévő assertekből indított hátramutató dinamikus szelet alapján felépített spektrum mátrix felhasználása a hibalokalizációhoz (közös eredmény)*
- *A két módszer empirikus összehasonlítása, a különbségek kvantitatív és kvalitatív kiértékelése (közös eredmény)*

Eredmények hatása: Friss megjelenése ellenére a kapcsolódó publikációnknak már van egy független hivatkozása. Mivel a kapott eredményeink a lefedettség alapú hibalokalizáció jelentős hátrányát igazolják, és ezt eddig nem igazolták, úgy gondoljuk, hogy a programszeletek vizsgálata ezen a területen fellendülhet.

Hivatkozások

A szerző hivatkozott publikációi

[1] Árpád Beszédes, Tamás Gergely, Szabolcs Faragó, Tibor Gyimóthy, and Ferenc Fischer. The Dynamic Function Coupling metric and its use in software evolution. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 103–112, March 2007.

[2] Árpád Beszédes, Tamás Gergely, and Tibor Gyimóthy. Graph-less dynamic dependence-based dynamic slicing algorithms. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'06)*, pages 21–30, September 2006.

[3] Árpád Beszédes, Tamás Gergely, Judit Jász, Gabriella Tóth, Tibor Gyimóthy, and Václav Rajlich. Computation of Static Execute After relation with applications to software maintenance. In *Proceedings of the 2007 IEEE International Conference on Software Maintenance (ICSM'07)*, pages 295–304, 2007.

[4] Árpád Beszédes, Tamás Gergely, Zsolt Mihály Szabó, János Csirik, and Tibor Gyimóthy. Dynamic slicing method for maintenance of large C programs. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR 2001)*, pages 105–113. IEEE Computer Society, March 2001.

[5] Árpád Beszédes, Ferenc Horváth, Massimiliano Di Penta, and Tibor Gyimóthy. Leveraging contextual information from function call chains to improve fault localization. In *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'20)*, pages 468–479, February 2020.

[6] Árpád Beszédes, Lajos Schrettner, Béla Csaba, Tamás Gergely, Judit Jász, and Tibor Gyimóthy. Empirical investigation of SEA-based dependence cluster properties. In *Proceedings of the 13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'13)*, pages 1–10, September 2013.

[7] Árpád Beszédes, Lajos Schrettner, Béla Csaba, Tamás Gergely, Judit Jász, and Tibor Gyimóthy. Empirical investigation of SEA-based dependence cluster properties. *Science of Computer Programming*, 105(0):3 – 25, 2015. Special Issue on SCAM'13.

[8] David Binkley, Árpád Beszédes, Syed Islam, Judit Jász, and Béla Vancsics. Uncovering dependence clusters and linchpin functions. In *Proceedings of the 31th IEEE International Conference on Software Maintenance and Evolution (ICSME'15)*, pages 141–150, September 2015.

[9] Tibor Gyimóthy, Árpád Beszédes, and István Forgács. An efficient relevant slicing method for debugging. *Lecture Notes in Computer Science*, 1687:303–321, 1999.

[10] Ferenc Horváth, Tamás Gergely, Árpád Beszédes, Dávid Tengeri, Gergő Balogh, and Tibor Gyimóthy. Code coverage differences of Java bytecode and source code instrumentation tools. *Software Quality Journal*, 27(1):79–123, 2019.

[11] Lajos Schrettner, Judit Jász, Tamás Gergely, Árpád Beszédes, and Tibor Gyimóthy. Impact analysis in the presence of dependence clusters using Static Execute After in WebKit. In *Proceedings of the 12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'12)*, pages 24–33, September 2012.

[12] Lajos Schrettner, Judit Jász, Tamás Gergely, Árpád Beszédes, and Tibor Gyimóthy. Impact analysis in the presence of dependence clusters using Static Execute After in WebKit. *Journal of Software: Evolution and Process*, 26(6):569–588, June 2014. Special Issue on SCAM'12.

[13] Péter Attila Soha, Tamás Gergely, Ferenc Horváth, Béla Vancsics, and Árpád Beszédes. A case against coverage-based program spectra. In *Proceedings of the 16th IEEE International Conference on Software Testing, Verification and Validation (ICST'23)*, pages 13–24, April 2023.

[14] Dávid Tengeri, Ferenc Horváth, Árpád Beszédes, Tamás Gergely, and Tibor Gyimóthy. Negative effects of bytecode instrumentation on Java source code coverage. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, pages 225–235, March 2016.

[15] Béla Vancsics, Ferenc Horváth, Attila Szatmári, and Árpád Beszédes. Call frequency-based fault localization. In *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'21)*, pages 365–376, March 2021.

[16] Béla Vancsics, Ferenc Horváth, Attila Szatmári, and Árpád Beszédes. Fault localization using function call frequencies. *The Journal of Systems and Software*, 193:111429, 2022.

További hivatkozások

[17] Clover Java and groovy code coverage tool homepage. https://www.atlassian.com/software/clover/overview. last visited: 2024-07-25.

[18] JaCoCo Java code coverage library homepage. http://eclemma.org/jacoco/. last visited: 2024-07-25.

[19] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.*, 82(11):1780–1792, November 2009.

[20] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, number 6 in SIGPLAN Notices, pages 246–256, White Plains, New York, June 1990.

[21] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the 27th international conference on Software engineering*, pages 432–441, 2005.

[22] Boris Beizer. *Software Testing Techniques.* John Wiley and Sons Inc., 605 Third Ave. New York, NY, United States, 1990.

[23] David Binkley. Source code analysis: A road map. *Future of Software Engineering (FOSE'07)*, pages 104–119, 2007.

[24] David Binkley and Mark Harman. Locating dependence clusters and dependence pollution. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 177–186. IEEE, 2005.

[25] David Binkley and Mark Harman. Identifying'linchpin vertices' that cause large dependence clusters. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 89–98. IEEE, 2009.

[26] David Binkley, Mark Harman, Youssef Hassoun, Syed Islam, and Zheng Li. Assessing the impact of global variables on program dependence and dependence clusters. *Journal of Systems and Software*, 83(1):96–107, 2010.

[27] David W Binkley and Mark Harman. A survey of empirical results on program slicing. *Adv. Comput.*, 62(105178):105–178, 2004.

[28] Lionel C. Briand, Jürgen Wüst, and Hakim Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, pages 475–482, Aug 1999.

[29] Higor A de Souza, Marcos L Chaim, and Fabio Kon. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *arXiv preprint arXiv:1607.04347*, 2016.

[30] Brent Hailpern and Peter Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, December 2001.

[31] Mark Harman, David Binkley, Keith Gallagher, Nicolas Gold, and Jens Krinke. Dependence clusters in source code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(1):1–33, 2009.

[32] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proc. of International Conference on Automated Software Engineering*, pages 273–282. ACM, 2005.

[33] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 2014.

[34] Bogdan Korel and Janusz W. Laski. Dynamic slicing in computer programs. *The Journal of Systems and Software*, 13(3):187–195, 1990.

[35] Xiaoguang Mao, Yan Lei, Ziying Dai, Yuhua Qi, and Chengsong Wang. Slice-based statistical fault localization. *Journal of Systems and Software*, 89:51–62, 2014.

[36] Sofia Reis, Rui Abreu, and Marcelo d'Amorim. Demystifying the combination of dynamic slicing and spectrum-based fault localization. In *IJCAI*, pages 4760–4766, 2019.

[37] Frank Tip. *A survey of program slicing techniques.* Centrum voor Wiskunde en Informatica Amsterdam, 1994.

[38] Mark Weiser. Program slicing. *IEEE Transactions on software engineering*, SE-10(4):352–357, 1984.

[39] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The DStar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, 2013.

[40] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[41] Andreas Zeller. *Why Programs Fail, Second Edition: A Guide to Systematic Debugging.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2009.