Új Algoritmusok és Új Adathalmaz a Spektrum Alapú Hibalokalizáció Támogatásához

Vancsics Béla

Szoftverfejlesztés Tanszék
Szegedi Tudományegyetem

Szeged, 2023

Témavezető:

Dr. Beszédes Árpád

Ph.D. értekezés tézisei

Szegedi Tudományegyetem
Informatika Doktori Iskola

Bevezetés

Nincs olyan, hogy hiba nélküli program! Szinte hetente látnak napvilágot a különböző cikkek arról, hogy valamilyen hibát, sérülékenységet találtak a legnépszerűbb programokban, operációs rendszerekben. Vannak olyanok amik „csak" kellemetlenséget okoznak, de előfordultak olyan esetek is amik nagyon komoly anyagi veszteséggel jártak (például az Ariane 5 rakéta megsemmisülése[1]). És ez csak a jéghegy csúcsa!

Ezek az esetek mind arra világítanak rá, hogy mennyire fontos és erőforrás-igényes részfeladata a fejlesztési folyamatnak valamint a (szoftver)életciklusnak a megbízható és alapos tesztelés. Minél előbb sikerült „elkapnunk a hibát", annál kisebb ráfordítással javíthatjuk azt, ezáltal rengeteg időt és pénzt takaríthatunk meg.

Számos megközelítés és módszer létezik a hibák mielőbbi automatikus, félig automatikus vagy manuális detektálására és a módszerek hatékonyságának mérésére, ezáltal is támogatva az alapos és folyamatos ellenőrzést. Kutatásom is eme két fő irányvonal mentén haladt: felvázoltam egy automatikus, dinamikus futtatással kapott, (unit) teszteredményeken és azok lefedettségén alapuló algoritmust, amely segít automatikusan meghatározni a hiba helyét a szoftver kódjában, valamint előállítottunk egy olyan, JavaScript nyelven írt programokat tartalmazó benchmark-ot (adathalmazt), amely segítségével hatékonyan lehet mérni a különböző hibalokalizáló algoritmusoknak a hatékonyságát. Ezek segítségével megbízható képet kaphatunk az egyes módszerek teljesítményéről, valamint kiindulási alapja lehet további kutatásoknak, amelyek segítségével tovább javítható a hibalokalizáló módszerek teljesítménye.

I. Hibalokalizáló Algoritmusok

Az egyik legnépszerűbb és legtöbbet kutatott algoritmus-család az úgynevezett *spectrum-based fault localization* algoritmusok (SBFL), amelynek a lényege, hogy a (dinamikusan) futtatott unit tesztek lefedettsége és a teszteredmények alapján minden kódelemhez hozzárendel egy gyanúsági értéket és ezen értékek alapján felállít egy gyanússági rangsort, amely meghatározza, hogy potenciálisan melyik kódelemek a leggyanúsabbak, vagyis a módszer alapján melyik kódrészletek tartalmazhatják a hibát. A kódrészlet lehet statement, branch vagy metódus is a granularitástól függően, de a kutatások többsége a hibás metódus meghatározására törekszik, vagyis metódusszintű hibakeresést folytat (a kutatásaimban én is metódus-alapú felbontást használtam).

A lefedettségi adatokat egy bináris mátrixban *(coverage matrix)*, a teszteredményeket pedig egy bináris vektorban *(result vector)* tárolják az SBFL algoritmusok. Ha a lefedettségi mátrix egy eleme egyenlő 1-gyel akkor az azt jelenti hogy az adott teszt végrehajtása során meghívodott az adott kódelem (jelen esetben a metódus), ha az eredményvektor adott eleme 1 akkor pedig az adott teszt bukik, vagyis failed (ha az értéke 0 akkor pedig sikeres, vagyis passed)
Ebből a két adatszerkezetből kinyerhető egy szám-négyes (spektrum):

- $|m_{ef}|$: az m metódus által *futtatott bukó tesztek* száma
- $|m_{ep}|$: az m metódus által *futtatott sikeres tesztek* száma
- $|m_{nf}|$: az m metódus által *nem futtatott bukó tesztek* száma
- $|m_{np}|$: az m metódus által *nem futtatott sikeres tesztek* száma)

amely az alapja az SBFL formuláknak, és amelyek alapján a gyanússági értékek (illetve a rangok) számolódnak. Ezek sok esetben hatékony módszerek, azonban nem veszik figyelembe a tesztek

[1] `https://homepages.inf.ed.ac.uk/perdita/Book/ariane5rep.html`

és a kódelem közötti egyéb kapcsolatokat, amelyek segítségével tovább növelhető a módszer hatékonysága.

Az egyik ilyen lehetséges „extra információ” a hívások gyakorisága. Voltak már kutatások ilyen irányba [7, 1] de ezek váltották be a hozzájuk fűzött reményeket, azonban az új megközelítésünk változtatott ezen.
Az alapötlet az volt, hogy a teszt végrehajtása során keletkező dinamikus, egyedi, leghosszabb hívási stack-ekben (*unique deepest call stack – UDCS*) levő előfordulások számát használjuk fel a coverage mátrixban, vagyis nem csak azt tároljuk el, hogy egy metódus meghívódott-e vagy sem a teszt végrehajtása során, hanem azt is hogy hányszor. Azáltal, hogy a mátrixunk nem csak 0 és 1 értékeket tárol újra kellett definiálnunk az $|m_{ef}|$, $|m_{ep}|$, $|m_{nf}|$ és $|m_{np}|$ értékeket is. Ebben az új analógiában az adott metódus $|m_{ef}|$ értéke azoknak a mátrixelemeknek lesz az összege amelyeknél a teszt eredménye failed, az $|m_{ep}|$ pedig azoknak az elemeknek az összege ahol a passed lett. Az nf és np adaptációja már valamivel bonyolultabb. Ebben az esetben az átlagos lefedettségét használtuk a nem lefedett elemeknek, vagyis kiszámoltuk, hogy a teszt által le nem fedett metódusoknak mekkora volt az átlagos lefedettsége (különvéve a bukó és a nem bukó teszteket) Az így kapott új értékek „behelyettesíthetőek” a SBFL formulákba.

Láthatjuk a példán keresztül (1. táblázat), hogy hogyan számolódnak ki a spektrumértékek a bináris és a gyakoriság-alapú mátrixok esetében. A pirossal jelzett cellák határozzak meg az ef értékét, a zölddel írottak az ep, a narancssárgával az nf és kékkel pedig az np értékét. Láthatjuk továbbá, hogy a bináris mátrix esetében ezek az értékek csak az adott metódushoz tartozó vektortól (és a teszteredményektől) függnek, ezzel szemben a gyakoriság-alapú megközelítés esetében a „teljes” mátrix befolyásolja a négy változó értékét.

1. táblázat. Példa bináris (hit) és gyakorisági (count) mátrixra és az ezekből számolt, **c** metódushoz tartozó spektrumokra

	bináris mátrix				gyakoriság mátrix				teszteredmény
	a	b	**c**	d	a	b	**c**	d	(result)
t1	1	0	1	1	2	0	2	1	0
t2	0	1	1	1	0	3	1	2	0
t3	1	1	0	0	2	2	0	0	0
t4	1	0	1	0	2	0	2	0	1
t5	1	1	0	1	3	2	0	1	1
t6	0	1	1	0	0	1	2	0	1
c metódus	ef	ep	nf	np	ef	ep	nf	np	
spektruma	2	2	1	1	4	3	3	4	

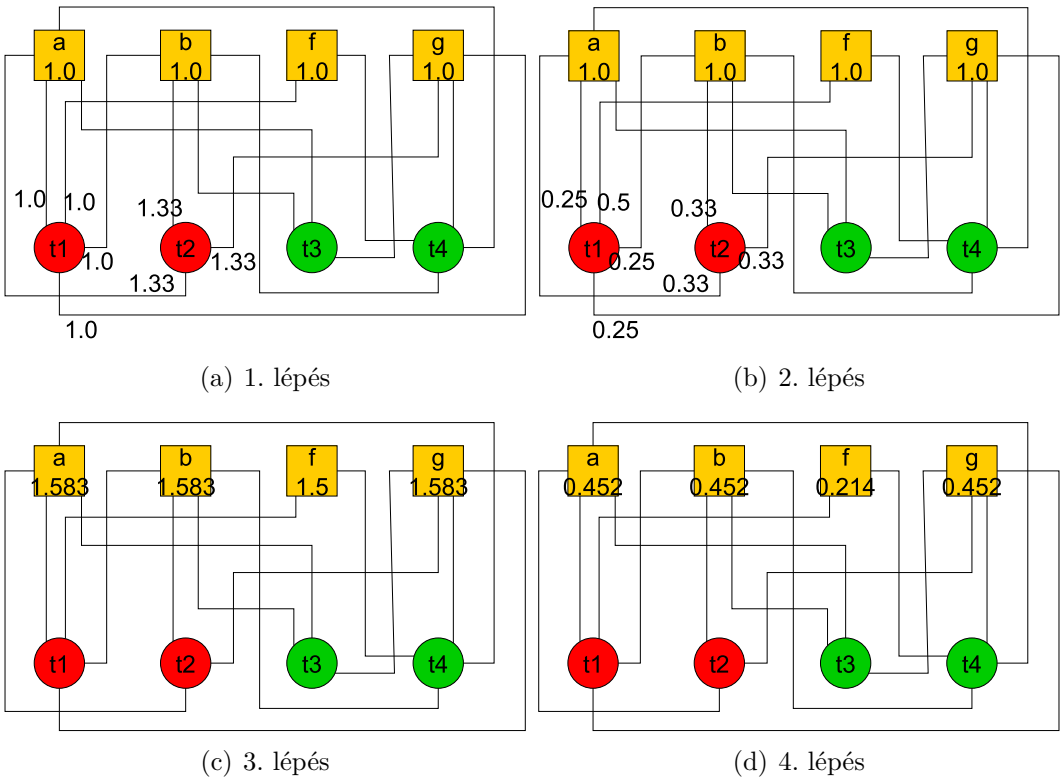
Az SBFL formulák ezt a spektrumot használják fel az egyes metódusok gyanússági értékének a meghatározására. Például ez az érték a c metódusra az Ochiai [4] algoritmus esetében: $\frac{|m_{ef}|}{\sqrt{(|m_{ef}|+|m_{nf}|)\cdot(|m_{ef}|+|m_{ep}|)}} \rightarrow \frac{2}{\sqrt{(2+1)\cdot(2+2)}} = 0.577$, illetve $\frac{4}{\sqrt{(4+3)\cdot(4+3)}} = 0.571$, attól függően hogy melyik típusú mátrixról beszélünk.

Lehetőségünk van csak bizonyos komponenseit kicserélni a képletnek, vagyis a spektrum egyes elemei a bináris, a többi eleme pedig a gyakoriság mátrix alapján számolódik. Ez alapján 4 forgatókönyvet különböztettem meg: (a) $\Delta^U_{ef^{num}}$: csak a számlálóban levő $|m_{ef}|$ esetében használom a gyakoriság-alapú $|m_{ef}|$ értéket, (b) Δ^U_{ef} minden $|m_{ef}|$ értéket „kicserélek”, (c) Δ^U_{e}: az $|m_{ef}|$ és az $|m_{ep}|$ értékeket cserélem ki és (d) Δ^U_{all}: a formulában csak a gyakoriság-alapú spektrumot használom (ahol Δ egy tetszőleges SBFL formulát jelöl). A fenti négy megközelítést hasonlítottam össze az eredeti SBFL formulák eredményével.

Emellett lehetőség van a mátrix struktúrájából is kinyerni további információkat. Az általam megalkotott, gráf-alapú koncepciók a coverage mátrixból képzett páros (lefedettségi) gráfot használják (1. ábra), ahol egy teszt és egy metódus között akkor van él, ha a teszt fedi az adott kódelemet.

Ebből a gráfból kinyert jellemzők segítségével megalkotásra került kettő új algoritmus (*NFL* és a *ENFL*), amelyek 4 illetve 4+3 lépésben határozzák meg a gyanússági értékeket. Az alábbi négy lépés az *NFL* algoritmus alapja (1. ábra):

- a hibás tesztekhez kapcsolódó élek súlyának a meghatározása (1(a). ábra): ez nem más, mint a metódusok számának és a bukó teszt által fedett metódusok számának a hányadosa. Minél nagyobb ez az érték annál inkább „tesztközpontúbb" az adott metódus (mivel csak egy kis részét érinti a teszt a metódusoknak).
- élsúlyok arányosítása (1(b). ábra): az első lépésben kapott (élsúly)értékeket elosztjuk az adott metódus által lefedett tesztek számával, ezáltal kifejezve azt, hogy a kevesebb teszt által fedett metódusok gyanúsabbak mint a több teszt által érintettek.
- aggregálás (1(c). ábra): összeadjuk az éleken levő súlyokat és aggregáljuk a „metódus-csúcsokba"
- arányosítás (1(d). ábra): az előző lépésben kapott értéket megszorozzuk a metódus által fedett bukó tesztek számának és a bukó tesztekből kimenő élek számának a hányadosával, ezáltal jutalmazva azt, ha egy metódus „felelős" a bukó tesztekhez kapcsolódó hívások nagyobb részéért.



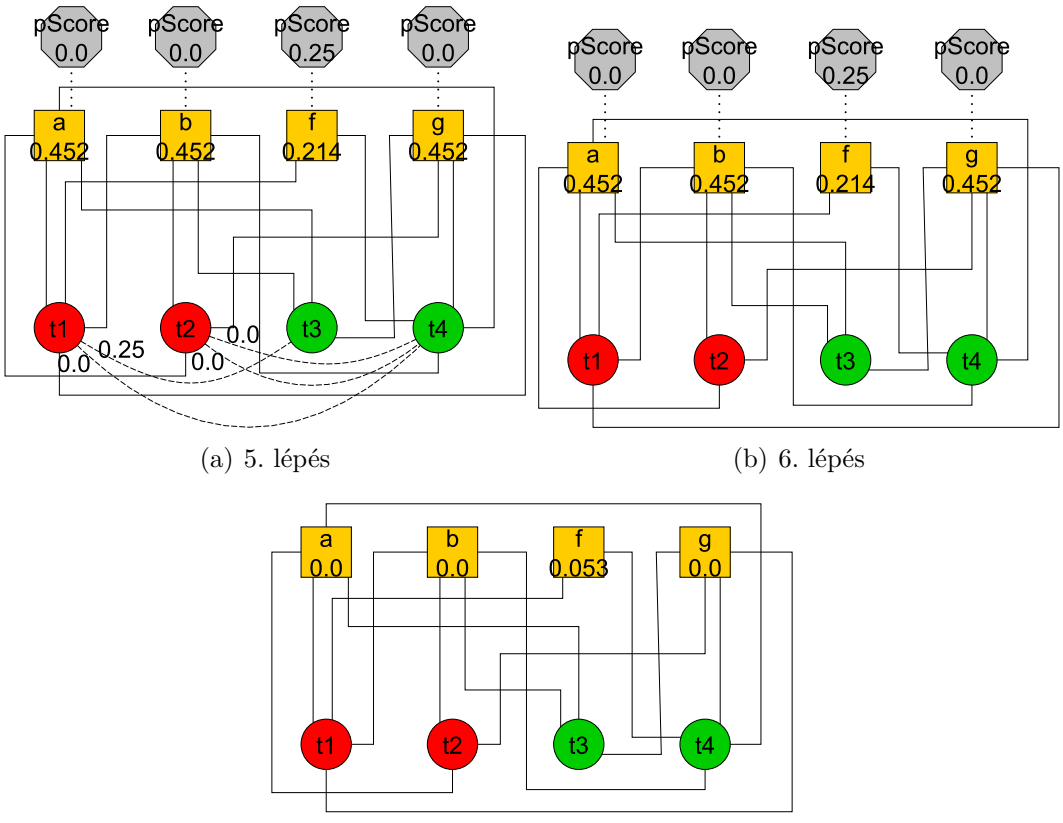
(a) 1. lépés (b) 2. lépés

(c) 3. lépés (d) 4. lépés

1. ábra. *NFL* algoritmus lépései

Az *NFL* algoritmust ki lehet terjeszteni további 3 lépéssel (2. ábra), amely a failed-passed tesztpárok lefedettségének felhasználásával próbálja meg a hiba helyét minél precízebben meghatározni.

- tesztpárok összehasonlítása (2(a). ábra): képezzük az összes tesztpárt ahol az egyik teszt passed, a másik pedig failed, majd kiszámoljuk minden párra a hányadost, amelyeket úgy kapunk, hogy a csak a bukó teszt által fedett (vagyis a passed pár által nem fedett) metódusok számát elosztjuk a bukó teszt által fedett metódusok számával. Ezeket a hányadosokat aggregáljuk azokhoz a metódusokhoz amelyek csak a failed teszt által voltak fedve (és a passed pár által nem)
- átlagolás (2(b). ábra): az előző lépésben kapott értéket átlagoljuk, vagyis elosztjuk a metódus által fedett bukó tesztek számával
- összefésülés ((2(c). ábra)): az *NFL* által adott és a fenti két lépés után kiszámolt értékeket összeszorozzuk, vagyis kombináljuk a teszt-metódus közötti és a teszt-teszt közötti kapcsolatok vizsgálatával kinyert eredményeket.



(a) 5. lépés

(b) 6. lépés

(c) 7. lépés

2. ábra. *ENFL* algoritmus lépései

Az *NFL* és a *ENFL* módszer egyik előnye, hogy nem csak súlyozatlan gráfon, hanem súlyozott gráfon is értelmezhető, vagyis lehetőségünk van a gráf-alapú és a gyakoriság-alapú módszerek ötvözésére ahol az él súlya a hívási gyakoriság lesz.

Ahhoz, hogy objektív képet kapjunk az algoritmusok hatékonyságáról három dolog szükséges: „referencia-algoritmusok" amelyekhez hasonlítjuk az új módszerek teljesítményét, megbízható adathalmaz (benchmark) amelyen kiértékeljük a teljesítményt és metrikák, amelyek segítségével számszerűsítjük azt. A megbízható megítélés miatt több, szám szerint 8 SBFL formulát választottam ki (2. táblázat), amelyek a hasonló témájú kutatásokban is rendszeresen kiindulási alapként szolgálnak.

2. táblázat. A kísérlet során használt „referencia-formulák".

Barinel (*B*) [2]: $\frac{|m_{ef}|}{|m_{ef}| + |m_{ep}|}$

DStar (*D*) [10]: $\frac{|m_{ef}|^2}{|m_{ep}| + |m_{nf}|}$

GP13 (*G*) [12]: $|m_{ef}| \cdot \left(1 + \frac{1}{2 \cdot |m_{ep}| + |m_{ef}|}\right)$

Jaccard (*J*) [4]: $\frac{|m_{ef}|}{|m_{ef}| + |m_{nf}| + |m_{ep}|}$

Ochiai (*O*) [4] : $\frac{|m_{ef}|}{\sqrt{(|m_{ef}| + |m_{nf}|) \cdot (|m_{ef}| + |m_{ep}|)}}$

Russell-Rao (*R*) [3]: $\frac{|m_{ef}|}{|m_{ef}| + |m_{nf}| + |m_{ep}| + |m_{np}|}$

Sørensen-Dice (*S*) [8] : $\frac{2 \cdot |m_{ef}|}{2 \cdot |m_{ef}| + |m_{nf}| + |m_{ep}|}$

Tarantula (*T*) [5] : $\frac{\frac{|m_{ef}|}{|m_{ef}|+|m_{nf}|}}{\frac{|m_{ef}|}{|m_{ef}|+|m_{nf}|} + \frac{|m_{ep}|}{|m_{ep}|+|m_{np}|}}$

A módszerek kiértékeléséhez a Defects4J-t [6] használtam, amely Java nyelv írt programokat és a hozzájuk tartozó hibákat tartalmazza. Összesen 17 projektet és 786 hibát használtam fel az összehasonlítás során.

A különböző algoritmusok által adott gyanússági értékek összehasonlíthatóságához a rangokat használtam, amely megadja, hogy hanyadik a vizsgált elem a gyanússági rangsorban. A rang meghatározása a következő:

$$E(f) = \frac{|\{i|s_i > s_f\}| + |\{i|s_i \geq s_f\}| + 1}{2} \tag{1}$$

ahol az s_i és az s_f az i metódushoz és az f hibás metódushoz tartozó gyanússági értékek. Ha több metódus is azonos értékkel rendelkezik akkor ezeknek a rangoknak az átlagát fogjuk minden egyes ilyen metódushoz hozzárendelni. Abban az esetben, ha több metódus is hibás, akkor a legnagyobb gyanússági értékkel rendelkező metódushoz tartozó rang lesz a hibának a rangja. Más szavakkal, a hiba rangja megmutatja, hogy átlagosan hány metódust kell megvizsgálni a fejlesztőnek ahhoz, hogy megtalálja a hiba helyét.

Az összehasonlítás egyik lehetséges aspektus, ami alapján megvizsgáltam az eredményeket az az, hogy hány esetben eredményezett magasabb (jobb) ill. alacsonyabb (rosszabb) rangot az új algoritmus, mint a referenciaként használt SBFL módszerek (3. ábra), mekkora volt az egyes módszerek által adott rangok átlaga (4.ábra) illetve mekkora volt a rangok közötti különbség.

Több tanulmány is bemutatta, hogy a fejlesztők csak az első 5 vagy 10 leggyanúsabb elemet vizsgálják meg a hibajavítás során, a rangsor többi elemét már nem tartják relevánsnak [11]. Ezért megvizsgáltam, hogy az egyes algoritmusok esetében hány hibás metódus van a leggyanúsabb 5 elem között (Top-5).

Láthatjuk, hogy az esetben többségében az új módszerek többször adnak magasabb rangot mint az „klasszikus" SBFL formulák, illetve leolvasható, hogy minden SBFL algoritmus esetében az *WENFL* „győzi le" azt legtöbbször (vagyis a 3. táblázat egy adott sorában a *WENFL* éri el a legmagasabb győz-értéket).

3. táblázat. Hányszor adott alacsonyabb ill. magasabb rangot az új algoritmus (a *veszít* a bináris alapú módszereknek „kedvez”)

	NFL		*ENFL*		$\Delta^U_{ef^{num}}$		Δ^U_{ef}		Δ^U_{e}		Δ^U_{all}		*WNFL*		*WENFL*	
	veszít	nyer	veszít	nyer	veszít	nyer	veszít	nyer	veszít	nyer	veszít	nyer	veszít	nyer	veszít	nyer
B	*56*	150	88	286	293	346	252	334	249	294	249	294	355	316	257	**384**
D	*57*	120	94	252	361	303	361	303	306	348	307	351	372	294	267	**365**
G	*90*	54	121	217	-	-	428	270	435	275	435	275	358	309	262	**372**
J	*49*	131	82	266	295	346	269	342	265	299	262	301	363	306	262	**375**
O	*45*	111	82	243	366	303	273	339	260	296	271	292	370	299	266	**365**
R	90	662	91	663	184	541	182	539	119	631	160	599	116	641	*64*	**693**
S	*49*	131	82	266	300	344	269	342	265	299	262	301	363	306	262	**375**
T	*56*	149	88	285	-	-	104	51	492	117	503	112	355	316	257	**384**

A 4. táblázat mutatja a 786 hibán elért rangok átlagát. Láthatjuk, hogy a legjobb eredményt, vagyis a legalacsonyabb átlagos rangot a *WENFL* módszer érte el (20.59), sokkal jobb eredményeket produkálva mint a referencia-algoritmusok (33.98-135.96). Továbbá, a *Tarantula* és a *GP13* kivételével minden formula esetében a gyakoriság-alapú megközelítés jobban teljesített (legalább az egyik koncepció szerint) mint a bináris, vagyis (átlagosan) hatékonyabban találta meg a hiba helyét a hívási gyakoriság figyelembe vétele mellett. Például: a $B^U_{ef^{num}}$ és a B^U_{ef} is jobban teljesített (24.68 és 24.55), mint a bináris megközelítés (36.01), de a B^U_e és a B^U_{all} rosszabbak voltak (38.63).

4. táblázat. A rangok átlaga

	hit	*NFL*	*ENFL*	$\Delta^U_{ef^{num}}$	Δ^U_{ef}	Δ^U_{e}	Δ^U_{all}	*WNFL*	*WENFL*
B	36.01			24.68	24.55	38.63	38.63		
D	33.99			35.60	35.60	29.08	29.13		
G	43.30			-	66.73	67.19	67.19		
J	36.06	36.42	34.05	24.63	24.40	38.64	38.34	47.19	**20.59**
O	33.98			36.05	24.30	36.44	36.99		
R	135.96			70.80	70.60	35.12	54.95		
S	36.06			24.89	24.40	38.64	38.34		
T	36.01			-	36.87	85.35	74.56		

Az 5. és 6. táblázatok mutatják, hogy az egyes algoritmusok esetében hány olyan hiba volt, aminek a rangja kisebb vagy egyenlő mint 5 (# oszlop), ez a teljes adathalmaz mekkora része (% oszlop), hány olyan hiba volt amely a referencia-algoritmus szerint a Top-5 kategóriába tartozik de az új módszer ennél rosszabb rangot eredményezett (*Det.* oszlop), illetve hány olyan hiba volt amely esetében az 5-nél rosszabb rangot az új módszer lecsökkentette és ezáltal a Top-5 kategóriába került (*E.im.* oszlop). Ebben az esetben is hasonló eredményeket látunk, mint az átlagos rang esetében. További megállapítható, hogy a gyakoriság-alapú spektrum használatával növelhető a Top-5 kategóriában levő hibák száma. Az *WENFL* módszer magasabb Top-5 értéket ér el (389 – 49.5%) mint a bináris megközelítést használó módszerek és emellett magas *E.Im.* értékeket is produkál (97-282), vagyis sok olyan eset volt, ahol a referencia-formulák 5-nél nagyobb rangot adtak, de az új algoritmus ezt lecsökkentette 5 vagy annál kisebb értékre. Érdekes tovább, hogy az *NFL* és a *ENFL* is nagyobb # értéket ér el, mint bármelyik SBFL formula, mindezt

kimagaslóan alacsony *Det.* érték mellett (vagyis kevés olyan eset van, hogy az új algoritmus hatására a hiba kikerül a Top-5 kategóriából) és minden esetben (hasonlóan a *WENFL*-hez) többször kerül be a Top-5-be, mint ahányszor kikerül onnan (vagyis $Det - E.Im. < 0$)

5. táblázat. A Top-5 kategóriában levő hibák száma, illetve a kategóriába történő „ki-be mozgás” a gyakoriság-alapú megközelítések esetében

	hit		$\Delta^U_{ef^{num}}$				Δ^U_{ef}				Δ^U_e				Δ^U_{all}			
	#	%	#	%	Det.	E. Im.	#	%	Det.	E. Im.	#	%	Det.	E. Im.	#	%	Det.	E. Im.
B	357	(45.4%)	373	(47.5%)	78	94	376	(47.8%)	65	84	354	(45.0%)	45	42	354	(45.0%)	45	42
D	366	(46.6%)	327	(41.6%)	128	89	327	(41.6%)	128	89	391	(49.7%)	86	111	388	(49.4%)	88	110
G	361	(45.9%)	-	-	-	-	266	(33.8%)	172	77	269	(34.2%)	170	78	269	(34.2%)	170	78
J	358	(45.5%)	372	(47.3%)	81	95	380	(48.3%)	69	91	357	(45.4%)	47	46	356	(45.3%)	48	46
O	367	(46.7%)	323	(41.1%)	135	91	380	(48.3%)	74	87	363	(46.2%)	51	47	361	(45.9%)	54	48
R	111	(14.1%)	249	(31.7%)	12	150	248	(31.6%)	12	149	380	(48.3%)	6	275	356	(45.3%)	10	255
S	358	(45.5%)	366	(46.6%)	87	95	380	(48.3%)	69	91	357	(45.4%)	47	46	356	(45.3%)	48	46
T	357	(45.4%)	-	-	-	-	351	(44.7%)	12	6	258	(32.8%)	111	12	254	(32.3%)	117	14

6. táblázat. A Top-5 kategóriában levő hibák száma, illetve a kategóriába történő „ki-be mozgás” a gráf-alapú megközelítések esetében

	hit		***NFL***				***ENFL***				***WNFL***				***WENFL***			
	#	%	#	%	Det.	E. Im.	#	%	Det.	E. Im.	#	%	Det.	E. Im.	#	%	Det.	E. Im.
B	357	(45.4%)			14	26			15	26			127	96			69	101
D	366	(46.6)%			12	15			13	15			133	93			74	97
G	361	(45.9)%			16	24			19	26			132	97			76	104
J	358	(45.5)%	369	(46.9%)	10	21	368	(46.8%)	11	21	326	(41.5%)	128	96	389	(49.5%)	70	101
O	367	(46.7)%			11	13			12	13			134	93			75	97
R	111	(14.1)%			2	260			2	259			6	221			4	282
S	358	(45.5)%			10	21			11	21			128	96			70	101
T	357	(45.4)%			14	26			15	26			127	96			69	101

A fenti összehasonlítás alapján kijelenthetjük, hogy (i) a gyakorisági információk felhasználásával javítható az SBFL algoritmusok teljesítménye, (ii) a gráf-alapú megközelítés sok esetben jobb (ill. majdnem olyan jó) eredményeket produkál, mint a referencia-algoritmusok és (iii) a súlyozott, hívási gyakoriságot használó, gráf-alapú megközelítés találja meg leghatékonyabban a hiba helyét a kódban.

A szerző hozzájárulása

A disszertáció szerzője alkotta meg a gráf-alapú megközelítést használó, „hagyományos” lefedettségmátrixokon működő algoritmust, valamint ő végezte el a módszer kvantitatív kiértékelését. Meghatározó szerepe volt a hívási láncokon alapuló megközelítés kidolgozásában, annak adaptálásában valamint az eredmények számszerűsítésében. A gráf- és a lánc alapú módszerek ötvözése szintén a szerző munkája, ahogy az eredmények összehasonlítása és kiértékelése is. Ezek mellett a disszertáció szerzője aktívan részt vett a szakirodalom feldolgozásában, a kísérletek valamint a mérések megtervezésében és a publikációk megírásában egyaránt.

Publikációk

- **B. Vancsics**, F. Horváth, A. Szatmári and Á. Beszédes. Fault Localization Using Function Call Frequencies Journal of Systems and Software (JSS), Volume: 193, Elsevier, 2022.
- **B. Vancsics** NFL: Neighbor-Based Fault Localization Technique In Proceedings of the 1th International Workshop on Intelligent Bug Fixing (IBF), Hangzhou, China, pp. 17-22, IEEE, 2019
- **B. Vancsics** Graph-Based Fault Localization In Proceedings of the International Conference on Computational Science and Its Applications (ICCSA), Saint Petersburg, Russia, pp. 372-387, Springer, 2019
- **B. Vancsics**, F. Horváth, A. Szatmári and Á. Beszédes Call Frequency-Based Fault Localization In Proceedings of the 28th International Workshop on on Software Analysis, Evolution, and Reengineering (SANER), Honolulu, HI, USA, pp. 365-376, IEEE, 2021

II. BugsJS: JavaScript Hibák Benchmark-ja

A hibalokalizációs algoritmusok megbízható kiértékeléséhez elengedhetetlen a megfelelő minőségű hiba-adatbázis. Több, hasonló hibagyűjtemény létezik (például: *Defects4J* [6] - Java), azonban ezidáig JavaScript nyelven írt programokra nem állt rendelkezésre ilyen adathalmaz.

Első lépésként meghatároztuk azokat a kritériumokat, amik alapján a programokat kiválasztjuk, ezután automatikusan összegyűjtöttük a bennük levő hibákat, ezeket manuálisan illetve automatikusan (dinamikusan) validáltuk, majd a kialakított infrastruktúrába betöltöttük.

A projektek kiválasztása során több, releváns szempontot vettünk figyelembe. Az egyik ilyen, hogy elérhető legyen a GitHub verziókövető rendszerben a projekt, valamint szerver-oldali Node.js alkalmazásnak kellett lennie. A hibák azonosításához és validálásához nélkülözhetetlen volt, hogy használja a projekt a GitHub *issue-tracker* rendszerét és az tartalmazzon *bug* címkével ellátott elemeket. Ezen kívül népszerűnek (szerzett csillagok száma ≥ 100), „érettnek" (a commit-ok száma > 200) és aktívnak (az utolsó módosítás éve ≥ 2017) kellett lennie ahhoz, hogy a potenciális jelöltek közé kerülhessen.

Minden egyes kiválasztott projekt esetében először lekérdeztük a GitHub hivatalos API-jával, hogy egy adott hibacímkével ellátott issue lezárt bug-e vagy sem. Minden lezárt bug bejegyzéshez automatikusan megkerestük azt a commit-ot, ami a hibát javította és azt amelyik a javítás előtti commit volt (de figyelmen kívül hagytuk azokat az eseteket, amikor kettő vagy több módosítással javították a hibát). Ezáltal azonosítva lett a hibát tartalmazó és a hibát már nem tartalmazó állapota a projektnek.

A manuális validáció esetében 5, a dinamikus validáció esetében 4 további feltételt támasztottunk a hibával illetve a commit-tokkal szemben. A manuális validáció esetében meg kellett felelni az alábbiaknak:

- izoláció: a hibajavítás pontosan 1 hibára vonatkozhat (vagyis ki lettek zárva azok a commit-ok amelyek 2 vagy több issue-t fixáltak)
- komplexitás: a hibajavító változtatásoknak korlátozott számú fájlt ($\leq$3), kódsort ($\leq$50) lehetett magában foglalnia és ésszerű időn belül (max. 5 perc) megérthetőnek kellett lennie
- függőség: ha egy javítás új függőséget (például könyvtárat) tartalmaz, akkor a kód módosításainak és új teszteseteknek is ugyanabban a commit-ban kell lenniük
- relevancia: a hibajavító változtatások csak a kódban végrehajtott változtatásokat érinthetik, amelyek célja a hiba kijavítása (szóközök és megjegyzések megengedettek)
- átalakítás: a fixek nem tartalmazhatják a kód újraírását (vagyis refaktorálását).

A dinamikus validációhoz négy feltételnek kellett teljesülnie:

- a hibajavítással módosított vagy hozzáadott tesztnek buknia kellett a javítást nem tartalmazó állapotban
- nem fordulhatott elő olyan, hogy hiányzó függőségek miatt nem lehet lefuttatni a teszteket
- nem lépett fel hiba a tesztek futtatása közben
- Mocha (teszt)framework-öt használ a projekt

A 7. táblázatban láthatóak a kiválasztott projektek, valamint a manuális és a dinamikus validáció statisztikái. Összességében 795 commit lett ellenőrizve manuálisan, amelyek közül 542 (68%) felelt meg a kritériumoknak. A 7. táblázatban (Manuális) szemlélteti ennek a lépésnek az eredményét az egyes projektekre és az összes alkalmazásra vonatkozóan. A hiba kizárásának leggyakoribb oka az, hogy a javítást túl bonyolultnak ítélték (136). Más gyakori forgatókönyvek közé tartoztak azok az esetek, amikor a hibajavítási egynél több hibát kezelt (32), vagy amikor a

javítás nem tartalmazott production code módosítást (29), vagy refaktorálást hajtott végre (39). Négy olyan esetet is találtunk, amikor a javítás nem a tényleges teszt forráskódját, hanem inkább megjegyzéseket vagy konfigurációs fájlokat tartalmazott.

A dinamikus elemzés után végül 453 commit lett megtartva (az előző lépésből származó 542 hibajelölt 84%-a). A 7. táblázatban a dinamikus validáció eredményeit a Dinamikus sorok tartalmazzák. 22 esetben nem tudtuk futtatni a teszteket, mert a függőségek eltávolításra kerültek a tárolókból. 15 esetben nem Mocha-t használt a projekt, 12 esetben nem sikerült végrehajtani a teszteket, míg 40 esetben pedig a javító commit-ban levő változtatott vagy hozzáadott teszt eredménye a javítást nem tartalmazó állapotban is passed volt.

7. táblázat. A manuális és a dinamikus validáció statisztikái

		Bower	Eslint	Express	Hessian.js	Hexo	Karma	Mongoose	Node-redis	Pencilblue	Shields	**Total**
	A kiindulási hibák száma	10	559	39	17	24	37	56	25	18	10	**795**
Manuális	✘ Több issue-t javít	0	18	1	0	1	5	2	5	0	0	32
Manuális	✘ Túl komplex	0	94	0	4	8	4	8	7	9	2	136
Manuális	✘ Dependency gond	1	9	0	0	1	0	2	0	0	0	13
Manuális	✘ Nincs prod. code módosítás	0	20	4	0	1	1	2	0	0	1	29
Manuális	✘ Nincs teszt módosítás	1	0	1	0	0	0	0	1	1	0	4
Manuális	✘ Refactoring	0	36	0	0	0	1	1	1	0	0	39
	Manuális validáció után	8	382	33	13	13	26	41	11	8	7	**542**
Dinamikus	✘ A teszt nem bukik a V_{bug} verzión	1	11	6	4	1	2	8	3	1	3	40
Dinamikus	✘ Hiányzó dependecia	3	17	0	0	0	1	1	0	0	0	22
Dinamikus	✘ Teszt-hiba	1	7	0	0	0	0	3	1	0	0	12
Dinamikus	✘ Nem Mocha használata	0	14	0	0	0	1	0	0	0	0	15
	✔ ***A hibák végső száma***	3	333	27	9	12	22	29	7	7	4	**453**

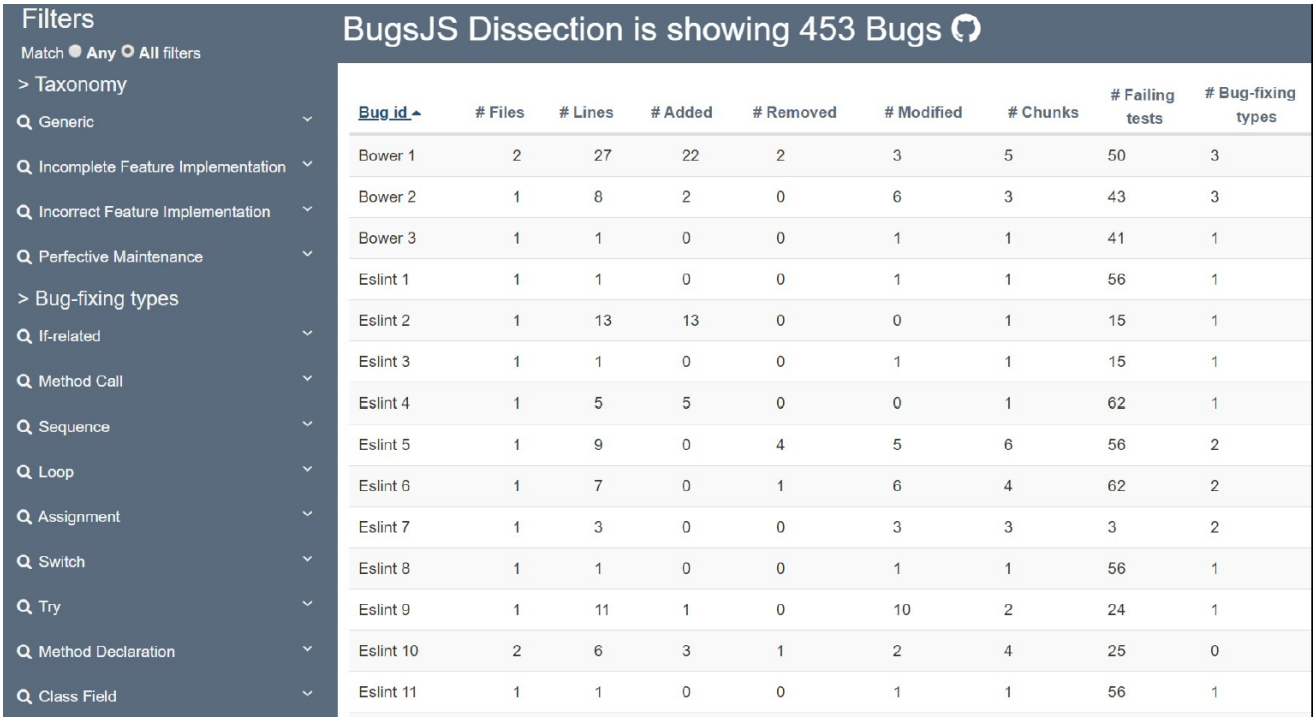
Kézi tisztítást végeztünk a módosításokon, hogy megbizonyosodjunk arról, hogy csak a hibajavításokhoz kapcsolódó változtatásokat tartalmazzák. Eltávolítottuk az irreleváns módosításokat (pl. *.md vagy .gitignore módosítása, komment-változtatásokat). Továbbá a könnyebb elemzés érdekében a javításokat két külön fájlra bontottuk, az első a tesztek módosításait, a második pedig a production code változtatásait tartalmazza.

Az így kapott változtatásokat rögzítettük a BugsJS infrastruktúrájában, valamint a felhasználók rendelkezésére bocsátottunk egy parancssori „felületet”, ami (jelenleg) 4 utasítást tud fogadni: (i) *info*: kiírja az adott hibára vonatkozó (alap)információkat, (ii) *checkout*: letölti az adott hiba adott állapotát, (iii) *test*: végrehajtja a teszteket és kódlefedettséget mér és (iv) *per–test*: hasonló az előzőhöz, de ez egyenként futtatja a teszteket és méri a lefedettséget.

Minden egyes hibához 5 állapotot rendeltünk hozzá, attól függően, hogy milyen módosításokat (hibajavításokat) tartalmaz.

- Bug-X: a javítást nem tartalmazó (azaz a hibás) állapot
- Bug-X-original: változtatás nélküli, a hiba javítását tartalmazó állapot (az eredeti hibajavító commit)
- Bug-X-test: csak a (validált és tisztított) teszt-változtatásokat tartalmazó állapot
- Bug-X-javítás: csak a (validált és tisztított) production code változtatást tartalmazó állapot
- Bug-X-full: a (validált és tisztított) teszt és a production code változtatásokat is tartalmazó állapot

Ezen felül megalkottunk a Defects4J Dissection[2] mintájára a BugsJS Dissectiont (3. és 4. ábrák) amely az egyes hibákhoz tartozó alapvető információkat tartalmazza és az alábbi linken keresztül érhető el: `https://bugsjs.github.io/dissection/`.



Bug id	# Files	# Lines	# Added	# Removed	# Modified	# Chunks	# Failing tests	# Bug-fixing types
Bower 1	2	27	22	2	3	5	50	3
Bower 2	1	8	2	0	6	3	43	3
Bower 3	1	1	0	0	1	1	41	1
Eslint 1	1	1	0	0	1	1	56	1
Eslint 2	1	13	13	0	0	1	15	1
Eslint 3	1	1	0	0	1	1	15	1
Eslint 4	1	5	5	0	0	1	62	1
Eslint 5	1	9	0	4	5	6	56	2
Eslint 6	1	7	0	1	6	4	62	2
Eslint 7	1	3	0	0	3	3	3	2
Eslint 8	1	1	0	0	1	1	56	1
Eslint 9	1	11	1	0	10	2	24	1
Eslint 10	2	6	3	1	2	4	25	0
Eslint 11	1	1	0	0	1	1	56	1

3. ábra. BugsJS Dissectiont (nyitó oldal)

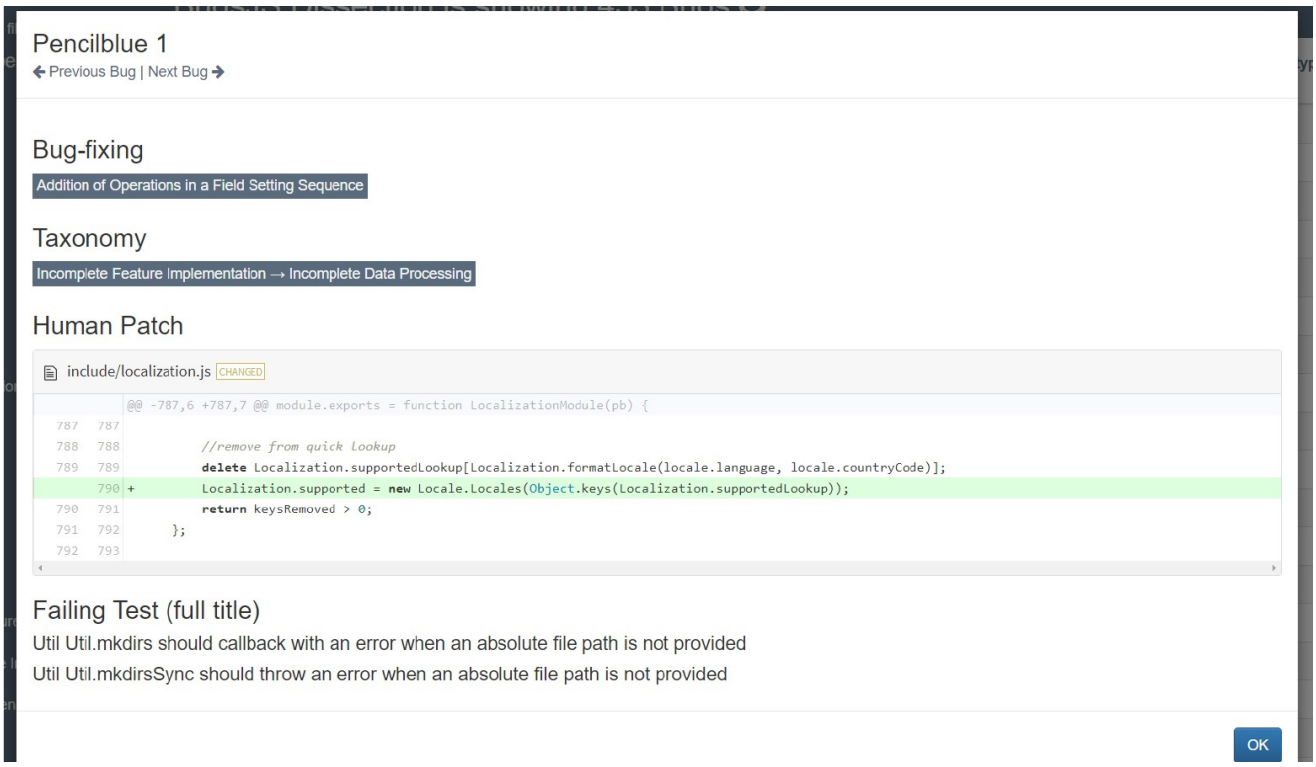
Nemcsak az infrastruktúra lett megalkotva, hanem az abban levő hibákat kategorizáltuk a hiba természete (taxonómiája), valamint a hibajavítás típusa szerint is. Személyes megbeszélések során minden minden hiba esetében minden résztvevő kutató áttekintette az issue-kat, és azonosította azokat a ekvivalencia osztályokat, amelyekhez leíró címkéket rendeltek. Az alulról felfelé irányuló megközelítést követve először kategóriákba csoportosítottuk a hasonló fogalmaknak megfelelő címkéket. Ezután szülőkategóriákat hoztunk létre, amelyekben ezek a kategóriák és alkategóriáik kapcsolódtak. Ezáltal létrejött egy manuálisan, több kutató által validált, 3 mélységű hierarchikus taxonómia, amely az azonos okból létrejövő hibákat tartalmazza.

Továbbá kategorizáltuk a hibajavításokat aszerint, hogy milyen kódváltozással történt a fix. Ehhez a Pan és társai által megalkotott rendszert [9] használtuk és egészítettük ki néhány (szám szerint 3) új csoporttal. Ez a csoportosítás kettő mélységig történt meg.

Miután a két szempont szerinti csoportosítás megtörtént és minden bug kapott legalább egy taxonómia-címkét illetve bug-fix type címkét, megvizsgáltuk, hogy milyen kapcsolat áll fenn a két halmazrendszer között. Ez alapján megállapíthatóvá váltak az erősebb (*missing input validation* és az *if-related fix* között) és a lazább (az *incomplete data processing* és a *sequence-related fix* között) kapcsolatok a csoportok között. A kettő csoportosítás megalkotása és a köztük levő kapcsolatok vizsgálata jó kiindulása alap lehet a további, hasonló témájú kutatásoknak.

Ezen kívül egy lehetséges felhasználási területen, a hibalokalizáción keresztül bemutattuk a BugsJS használhatóságát és létjogosultságát. Megvizsgáltunk 7 projektet és 336 hibát és már korábban leírt algoritmusok közül három (*Tarantula*, *Ochiai* és a *DStar*) módszerrel kiértékeltük

[2] `http://program-repair.org/defects4j-dissection/`



4. ábra. BugsJS Dissectiont (egy bug-ra vonatkozó adatok)

azokat. Az eredmények megerősítették azt, amit a többi benchmark-on tapasztaltak a kutatók (8. táblázat): a három eljárás hasonlóan teljesített, 7 projektből 6 esetben a *DStar* teljesített a legjobban (kivétel a Hessian.js), de összességében legjobb eredményt az *Ochiai* érte el.

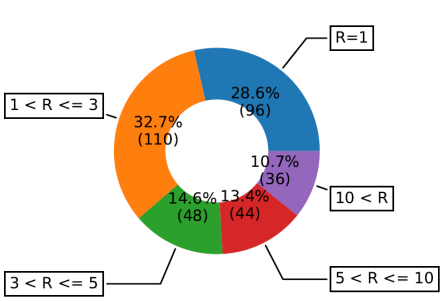
8. táblázat. Átlagos rangok

Projekt	Bower	Shields	Hexo	Hessian.js	Express	Pencilblue	Eslint	All
Tarantula	25.83	5.83	3.25	4.81	8.10	1.83	20.39	18.24
Ochiai	19.17	5.83	3.00	3.88	7.94	1.67	19.90	17.73
DStar	17.50	5.17	80.88	3.00	7.94	1.67	19.90	20.47

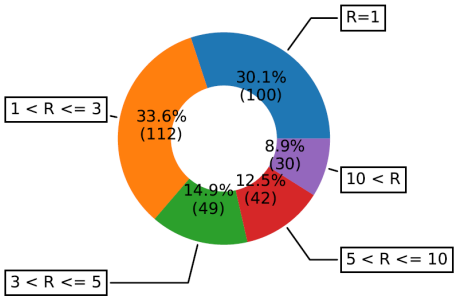
Amennyiben a rangok megoszlását nézzük akkor is hasonló következtetésre jutunk (5. ábra): az esetek 28-30%-ban a rang értéke 1 (vagyis a leggyanúsabb elem a hibás metódus), a hibák közel kétharmada 3 vagy annál kisebb rangot kap (61-63%) és csupán 9-11% között van azoknak a hibáknak az aránya, amelyek esetében a rang nagyobb mint 10.

Továbbá kíváncsiak voltunk arra, hogy vannak-e olyan bug-fix típusok, amelyeket hatékonyabban találnak meg a fent említett algoritmusok. Az 9. ábrán látható, hogy az egyes algoritmusok az esetek hány százalékában eredményeztek 1 rangot (Top-1), 3 vagy annál jobb (Top-3), 5 vagy annál jobb (Top-5), 10 vagy annál jobb (Top-10) vagy 10-nél rosszabb (Other) értéket.

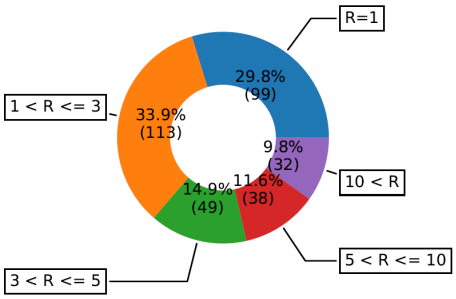
Látható, hogy az *if-related* hibajavítások esetében az átlagosnál hatékonyabban található meg a hiba helye, a metódus hívással kapcsolatos javításoknál (MC) az átlagosnál gyakrabban lesz a hibás metódus a gyanússági lista élén, és a metódus-deklarációval összefüggő esetekben (MD) is valamivel jobban teljesítenek az algoritmusok. Ellentétes következtetés vonható le a *sequence-related*-ek (SQ) esetében. Ezeket a hibákat nehezebben találják meg a módszerek, rosszabb rangot érnek mindegyik algoritmus esetében. A többi esetben az alacsony esetszám (# oszlop a táblázatba) miatt nehéz lenne megalapozott következtetéseket levonni.



(a) *Tarantula*



(b) *Ochiai*



(c) *DStar*

5. ábra. A rangok (R) megoszlása a három algoritmus esetében

9. táblázat. A rangok megoszlása a három algoritmus és a bug-fix típusnak a függvényében

Type	#	**Top-1 (%)**			**Top-3 (%)**			**Top-5 (%)**			**Top-10 (%)**			**Other (%)**		
		T	*O*	*D*	*T*	*O*	*D*	*T*	*O*	*D*	*T*	*O*	*D*	*T*	*O*	*D*
IF	176	32.4	34.1	33.0	65.3	67.6	67.6	80.7	83.5	83.0	93.2	94.3	93.2	6.8	5.7	6.8
AS	102	28.4	30.1	29.1	56.9	61.2	60.2	71.6	73.8	71.8	85.3	89.3	86.4	14.7	10.7	13.6
CF	2	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0
MD	24	29.2	29.2	29.2	70.8	75.0	79.2	79.2	83.3	83.3	87.5	91.7	91.7	12.5	8.3	8.3
MC	65	36.9	38.5	38.5	61.5	63.1	63.1	70.8	72.3	72.3	89.2	87.7	87.7	10.8	12.3	12.3
SQ	29	10.3	10.3	10.3	48.3	48.3	55.2	65.5	69.0	72.4	79.3	82.8	82.8	20.7	17.2	17.2
SW	5	20.0	20.0	20.0	60.0	60.0	60.0	60.0	80.0	80.0	80.0	80.0	80.0	20.0	20.0	20.0
LP	8	0.0	0.0	0.0	50.0	50.0	50.0	87.5	87.5	87.5	87.5	87.5	87.5	12.5	12.5	12.5
TY	1	0.0	0.0	0.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	0.0	0.0	0.0

Összefoglalva, megalkotásra került egy JavaScript nyelvű programokat tartalmazó, validált benchmark, amelyet a hiba természete és a hibajavítás típusa alapján kategorizáltunk, valamint bemutattunk egy lehetséges forgatókönyvet arra, hogy hogyan lehet használni a framework-öt a különböző kutatások során.

A szerző hozzájárulása

A disszertáció szerzője aktívan részt vett a már létező hiba-adatbázisok feltérképezésében, az új data set megtervezésében és kivitelezésében (beleértve egyaránt a hibák validálását és a szoftveres architektúra kiépítését). Mind a taxonómia megalkotása, mind a hibajavítások kategorizálása több kutató konszenzusos döntését követelte, amiben a szerző is aktív szerepet vállalt. Továbbá ő volt az, aki megtervezte azt a kísérletet (és implementálta a szükséges programokat), amin keresztül bemutatásra került, hogy a BugsJS hogyan használható hibalokalizációs algoritmusok hatékonyságának kiértékelésére és összehasonlítására.

Publikációk

- P. Gyimesi, **B. Vancsics**, A. Stocco, D. Mazinanian, Á Beszédes, R. Ferenc and A. Mesbah BugsJS: a Benchmark and Taxonomy of JavaScript Bugs Software Testing, Verification and Reliability (STVR), Volume 31, Issue 4, John Wiley & Sons, 2021

- P. Gyimesi, **B. Vancsics**, A. Stocco, D. Mazinanian, Á Beszédes, R. Ferenc and A. Mesbah BugsJS: a Benchmark of JavaScript Bugs In Proceedings of the 12th IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 90-101, IEEE, 2019

- **B. Vancsics**, A. Szatmári and Á. Beszédes Relationship Between the Effectiveness of Spectrum-Based Fault Localization and Bug-fix Types in JavaScript Programs In Proceedings of the 27th International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 308-319, IEEE, 2020

- A. Szatmári, **B. Vancsics** and Á. Beszédes Do Bug-Fix Types Affect Spectrum-Based Fault Localization Algorithms' Efficiency? In Proceedings of the 3th International Workshop on Validation, Analysis and Evolution of Software Tests (VST), pp. 16-23, IEEE, 2020

- **B. Vancsics**, P. Gyimesi, A. Stocco, D. Mazinanian, Á Beszédes, R. Ferenc and A. Mesbah Supporting JavaScript Experimentation with BUGSJS In Proceedings of the 12th International Conference on Software Testing, Verification and Validation (ICST), pp. 375-3378, IEEE, 2019

További Publikációk

- Horváth, F., Beszédes, Á., **Vancsics, B.**, Balogh, G., Vidács, L. and Gyimóthy, T. Using Contextual Knowledge in Interactive Fault Localization Empirical Software Engineering (EMSE), Volume: 27, Number: 150, Published by Springer, 2022
- Sarhan, Q. I., **Vancsics, B.** and Beszédes, Á Method Calls Frequency-Based Tie-Breaking Strategy For Software Fault Localization In Proceedings of the 21st International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 103-113, IEEE, 2021
- Horváth, F., Beszédes, Á., **Vancsics, B.**, Balogh, G., Vidács, L. and Gyimóthy, T. Experiments with Interactive Fault Localization Using Simulated and Real Users In Proceedings of the 36th International Conference on Software Maintenance and Evolution (ICSME), Adelaide, Australia, pp. 290-300, IEEE, 2020
- **Vancsics, B.**, Gergely, T. and Beszédes, Á. Simulating the Effect of Test Flakiness on Fault Localization Effectiveness In Proceedings of the 3rd International Workshop on Validation, Analysis and Evolution of Software Tests (VST), London, Ontario, Canada, pp. 28-35, IEEE, 2020
- Gergely, T., Balogh, G., Horváth, F., **Vancsics, B.**, Beszédes, Á. and Gyimóthy, T. Differences Between a Static and a Dynamic Test-to-Code Traceability Recovery Method Software Quality Journal (SQJ), Volume 27, Number 2, pp. 797-822, Springer, 2018
- Gergely, T., Balogh, G., Horváth, F., **Vancsics, B.**, Beszédes, Á. and Gyimóthy, T. Analysis of Static and Dynamic Test-to-code Traceability Information Acta Cybernetica, Volume 23, Number 3 (Special Issue In Memoriam Csanád Imreh), pp. 903-919, Institute of Informatics, University of Szeged, 2018
- Tengeri, D., Vidács, L., Beszédes, Á., Jász, J., Balogh, G., **Vancsics, B.** and Gyimóthy, T. Relating Code Coverage, Mutation Score and Test Suite Reducibility to Defect Density In Proceedings of the International Conference on Software Testing, Verification and Validation Workshops: 11th International Workshop on Mutation Analysis (MUTATION), pp. 174-179, IEEE, 2016
- Horváth, F., **Vancsics, B.**, Vidács, L., Beszédes, Á., Tengeri, D., Gergely, T. and Gyimóthy, T. Test Suite Evaluation using Code Coverage Based Metrics In Proceedings of the 14th Symposium on Programming Languages and Software Tools (SPLST), CEUR Workshop Proceedings. Vol. 1525., pp. 46–60, 2015
- Binkley, D., Beszédes, Á., Islam, S., Jász, J. and **Vancsics, B.** Uncovering Dependence Clusters and Linchpin Functions In Proceedings of the 31th International Conference on Software Maintenance and Evolution (ICSME), pp. 141-150, IEEE, 2015
- Vidács, L., Horváth, F., Mihalicza, J., **Vancsics, B.** and Beszédes, Á. Supporting Software Product Line Testing by Optimizing Code Configuration Coverage In Proceedings of the 8th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 1-7, IEEE, 2015

Összefoglalás

A disszertációmban a szoftverfejlesztés egy nagyon fontos részfeladatára, a szoftvertesztelésre, azon belül is annak egy kis szegmensére, a hibalokalizálásra fókuszáltam. Dolgozatomban a unit tesztekre épülő hibadetektáló algoritmusok egyik legismertebb és legnépszerűbb családjával, a lefedettség/spektrum-alapú eljárásokkal foglalkoztam, a dolgozat második felében pedig bemutattam az általunk megalkotott benchmark-t, amely segítségével az algoritmusok hatékonysága objektíven mérhetővé és ezáltal összehasonlíthatóvá válik.

Részletesen bemutattam a lefedettség-alapú hibakereső eljárások (úgynevezett *SBFL* algoritmusok) működési elvét, a legismertebb módszereket valamint az egyéb releváns szakirodalmakat. Ezt követően bemutattam kettő, gráf-jellemzőkre épülő megközelítést, amelyek az SBFL-hez hasonló bemeneti adatok mellett próbálják minél precízebben meghatározni a hiba helyét. Ehhez egy úgynevezett lefedettségi-gráfot (coverage graph) építettem, amelyből különböző szomszédsági információkat kinyerve azonosítja a leggyanúsabb forráskód-elemeket.

Ezután bevezetésre került az *egyedi, leghosszabb hívássorozatok* fogalma (unique, deepest call stack - UDCS), amelyet kis példán keresztül szemléltettem. Részletesen leírásra került, hogy a UDCS-ből kinyerhető hívási gyakoriság-információk (call frequency) hogyan adaptálhatóak az SBFL által használt, korábban leírt formulákban.

Végezetül pedig bemutattam, hogy a fentebb leír két (gráf- és gyakoriság-alapú) koncepció hogyan ötvözhető, ezáltal kihasználva a két módszerben levő potenciális lehetőségeket.

A kvantitatív kiértékelést a *Defects4J* (Java) benchmark-on végeztem, és az eredmények azt mutatták, hogy (i) a két gráf-alapú módszer sok esetben hatékonyabb, mint a vizsgált SBFL módszerek, (ii) a gyakorisági információkat felhasználó eljárások jelentősen jobb eredményt érnek el, mint a bináris lefedettségre épülő módszerek és (iii) a legjobb eredményt a kettő koncepció ötvözésével kapott, *WENFL* algoritmus eredményezte.

A hibalokalizációs algoritmusok megbízható kiértékeléséhez elengedhetetlen a megfelelő minőségű hiba-adatbázis. Több, hasonló hibagyűjtemény létezik (például: *Defects4J* - Java), azonban ezidáig JavaScript nyelven írt programokból nem állt rendelkezésre ilyen adathalmaz. A disszertáció második felében felvázoltam, hogy hogyan is történt ennek az új adathalmaznak (BugsJS) a megalkotása, részletesen leírom, hogy milyen lépések mentén zajlott a projektek kiválasztása (10 projekt), a bennük levő hibák (453 hiba) detektálása, validálása, „tisztítása" valamint különböző szempontok szerinti elemzése. Ezek mellett megalkottuk az adathalmazban levő hibáknak a rendszertanát (amely egy hierarchikus hiba-kategorizálást takar) és az így kapott taxonómiában létrehozott hiba-csoportokat részletesen, valós példákon keresztül bemutattuk. Továbbá elemeztük és kategorizáltuk az egyes hibákat aszerint is, hogy hogyan történt azok javítása, valamint megvizsgáltuk, hogy milyen összefüggés van a taxonómia és a hibajavítás kategóriái között. Végezetül, pedig egy lehetséges felhasználási területen, a hibalokalizációs algoritmusok vizsgálatán keresztül bemutattuk, hogy a BugsJS hogyan használható a kutatók számára.

Bízom benne, hogy ezek a kutatási eredmények hozzájárulnak ahhoz, hogy további, az eddigieknek hatékonyabb algoritmusok kerüljenek megalkotásra és a teljesítmények értékelése is megbízhatóbb legyen a jövőben.

Köszönetnyilvánítás

A disszertáció a szerző munkáját hangsúlyozza, az ő eredményeire fókuszál, ugyanakkor nem jöhetett volna létre mások segítsége nélkül. Először is szeretnék köszönetet mondani témavezetőmnek, Dr Beszédes Árpád a sokéves közös munkáért. Rengeteget tanultam tőle az éves során. Segítsége, tanácsai és folyamatos motiválása nélkül nem jutottam volna el idáig. Köszönettel tartozom még Gergely Tamásnak, Horváth Ferencnek, Balogh Gergőnek és Szatmári Attilának akikkel éveken át együtt dolgoztam és dolgozok jelenleg is. Ők nemcsak kiváló kollégák, hanem remek kutatók is, akik folyamatosan támogattak az elmúlt évek kutatásai során.

Szeretném köszönetet mondani Szűcs Editnek, aki stilisztikai és nyelvtani észrevételeivel segítette disszertáció létrejöttét.

És természetesen köszönöm a családomnak és a páromnak, hogy végig támogattak és bíztak bennem, megteremtették a feltételek ahhoz, hogy csak a kutatásomra kelljen koncentrálnom. Nem lehetek elég hálás nekik érte!

A disszertációban szereplő kutatást a Magyar Innovációs és Technológiai Minisztérium által a Nemzeti Kutatási, Fejlesztési és Innovációs Alapból a TKP2021-NVA támogatási konstrukció keretében finanszírozott TKP2021-NVA-09 projekt támogatta.

Továbbá, a kutatást, amelyet az SZTE valósított meg, az Innovációs és Technológiai Minisztérium és a Nemzeti Kutatási, Fejlesztési és Innovációs Hivatal támogatta a Mesterséges Intelligencia Nemzeti Laboratórium keretében (RRF-2.3.1-21-2022-00004). A kutatás továbbá a TKP2021-NVA-09 számú projekt az Innovációs és Technológiai Minisztérium Nemzeti Kutatási Fejlesztési és Innovációs Alapból nyújtott támogatásával, a TKP2021-NVA pályázati program finanszírozásában valósult meg.

Vancsics Béla, 2023

Hivatkozások

[1] Rui Abreu, Alberto Gonzalez-Sanchez, and Arjan JC van Gemund. Exploiting count spectra for bayesian fault localization. Proceedings of the 6th International Conference on Predictive Models in Software Engineering, pages 1–10, New York, NY, USA, 2010. ACM, Association for Computing Machinery.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 88–99. IEEE Computer Society, 2009.

[3] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11), 2009.

[4] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE, 2007.

[5] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, New York, NY, USA, 2005. ACM.

[6] 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*, ISSTA 2014, page 437–440, New York, NY, USA, 2014. Association for Computing Machinery.

[7] H. J. Lee, L. Naish, and K. Ramamohanarao. Effective software bug localization using spectral frequency weighting function. Proceedings of the 34th Annual Computer Software and Applications Conference, pages 218–227. IEEE, IEEE Press, 2010.

[8] Lucia Lucia, David Lo, Lingxiao Jiang, Ferdian Thung, and Aditya Budi. Extended comprehensive study of association measures for fault localization. *Journal of software: Evolution and Process*, 26(2):172–219, 2014.

[9] Kai Pan, Sunghun Kim, and E James Whitehead. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[10] W. E. Wong, V. Debroy, R. Gao, and Y. Li. The DStar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, 2014.

[11] Xin Xia, Lingfeng Bao, David Lo, and Shanping Li. „Automated Debugging Considered Harmful" considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems. Proceedings of the 32nd International Conference on Software Maintenance and Evolution (ICSME), pages 267–278. IEEE, IEEE Press, 2016.

[12] Shin Yoo. Evolving human competitive spectra-based fault localisation techniques. Proceedings of the 2012 International Symposium on Search Based Software Engineering, pages 244–258. Springer, Springer, 2012.