



Eötvös Loránd Tudományegyetem
Informatikai Kar
Programozási Nyelvek és Fordítóprogramok Tanszék

Erlang programok hatáselemzése

TDK dolgozat

Témavezetők:

Dr. Tóth Melinda, Dr. Bozó István

Egyetemi docens, Egyetemi adjunktus

Szerző:

Góga Andrea

Programtervező informatikus, MSc
2. évfolyam

Budapest, 2023

Kivonat

A statikus elemzési módszerek segítségével különböző aspektusokból vizsgáljuk a szoftvert alkotó forrásfájlokat, anélkül, hogy végrehajtanánk azokat. Támogatást nyújthat a mindennapi szoftverfejlesztői munka során, hiszen fordítási időben felfedezhetjük a szoftver futásidejű hibáit, biztonsági réseit, ellenőrizhetjük, hogy a kód megfelel-e a kitűzött kódolási konvencióknak, valamint hatáselemzést készíthetünk egy kódrészlethez. A hatáselemzés egy olyan módszer, amelynek segítségével azt vizsgáljuk, hogy a szoftverben bekövetkezett változások a szoftver mely további komponenseit érintik, befolyásolják. A programszeletelés egy dekompozíciós technika, amely statikus hatáselemzés eszköze lehet. Programszeleteléssel képesek vagyunk kivonatot készíteni a program azon utasításaiból, amelyek befolyásolják a program egy megadott pontját. Ezen utasítások halmazát programszeletnek (program slice) nevezzük, azt a programpontot, amelyhez a halmazt készítettük pedig szeletelési kritériumnak (slicing criterion).

A programszeletelés fogalmát először 1979-ben adták meg, a témakörben azóta jelentős számú kutatás született. A legnagyobb kihívást a programok vezérési és adatfüggőségeinek megfelelő ábrázolása, ezekből pedig az információ hatékony kinyerése jelenti. Számos elméleti algoritmus hivatott megoldani ezeket a problémákat, azonban elenyésző a gyakorlatban ténylegesen megvalósított programszeletelésre képes keretrendszerek száma. A létező programszeletelők esetén a teljesítményre vonatkozó metrikák sok esetben hiányosak, mivel nagyon nehéz egy programszelet eredményét validálni.

A RefactorErl egy Erlang nyelven íródott, Erlang-forrásfájlokhoz készült statikus forráskódelemző keretrendszer. A betöltött forrásfájlokat egy speciális adatszerkezetben, a szemantikus programgráfban tárolja, amelynek segítségével a kód információi hatékonyan kinyerhetők. A TDK dolgozatomban a RefactorErlben meglévő szeletelő algoritmus működését elemeztem, és azonosítottam azokat a pontokat, amelyek a teljesítményt hátráltatják. Sikeres megoldásokat adnom, amelyek a programszeletben található eredmények pontosítását és a futásidő csökkentését célozzák meg. A programszeletelés gyakorlati alkalmazásai közül a regressziós tesztszelekciót és a párhuzamosítást vizsgáltam meg.

Tartalomjegyzék

1. Bevezető	5
1.1. Problémafelvetés és eredmények	5
1.2. A dolgozat felépítése	6
2. Alapvető fogalmak ismertetése	7
2.1. Erlang	7
2.2. Statikus elemzés	7
2.3. RefactorErl	8
2.3.1. Szemantikus programgráf	8
2.3.2. Lekérdező nyelv	10
2.3.3. Adatfolyam-elemzés	10
2.4. Hatáselemzés	11
2.5. Programszeletelés	11
2.5.1. Programszeletelési típusok	12
3. Programszeletelés a RefactorErlben	14
3.1. Éltípusok a szemantikus programgráfban	14
3.2. Vezérlésfüggőségi gráf építése	15
3.2.1. Vezérlésfolyam-gráf	15

3.2.2.	Posztdominátorok fája	18
3.2.3.	Egyszerű és összetett vezérlésfüggőségi gráf	18
3.3.	Adatfüggőségek felderítése	19
3.4.	Programszelet kiszámítása	20
4.	Meglévő algoritmus elemzése	21
4.1.	Teljesítmény ipari környezetben	21
4.1.1.	Integráció célja	21
4.1.2.	Eredmények értékelése	22
4.2.	Önálló elemzések	22
4.2.1.	Elemzési bemenet	22
4.2.2.	Időigényes komponensek azonosítása	23
4.2.3.	Találatok ellenőrzése	24
5.	Szeletelő algoritmus javítása	28
5.1.	Futásidő racionalizálása	28
5.1.1.	Adatfolyam-elemzés különböző időkorlátokkal	28
5.1.2.	Taskfarm implementáció bevezetése	30
5.1.3.	Nulladrendű és elsőrendű megoldások	31
5.2.	Hamis pozitív találatok szűrése	33
5.2.1.	Adatfolyam-elemzés és hamis pozitív találatok kapcsolata	33
5.2.2.	Függvények utóelemzése adatfolyambeli szűrésekkel	33
5.2.3.	Utóelemző modul implementációja	35
5.2.4.	Utóelemzés eredményeinek kiértékelése	38
5.3.	Kifejezésszintű eredmények pontosítása	40
5.3.1.	Kifejezésszint hatásköre	40

5.3.2.	Adatfolyam-elemzés eredményének kiegészítése	40
5.3.3.	Adatfolyam-elemzés iránya	43
5.3.4.	Vezérlési függőségek korlátozása	44
5.4.	Egyéb megoldások vizsgálata	45
5.4.1.	Dinamikus információk beépítése	46
5.4.2.	Elemzés hatókörének szűkítése	52
5.4.3.	Kifejezések előszűrése	54
5.4.4.	Függvényhívási láncok vizsgálata	57
6.	Javított algoritmus alkalmazásai	59
6.1.	Regressziós tesztelés	59
6.2.	Párhuzamosítás	61
6.2.1.	Szeletelés tulajdonságai	61
6.2.2.	Végrehajtható programkód-részlet létrehozása	61
6.2.3.	Intraprocedurális eset	62
6.2.4.	Interprocedurális eset	63
7.	Kapcsolódó munkák	65
7.1.	Frama-C	65
7.2.	Indus	66
7.3.	Wrangler	66
7.4.	Static Execute After	67
7.5.	Definíció-használat alapú szeletelés	68
8.	Összefoglaló	69
8.1.	Továbbfejlesztési lehetőségek	70

1. fejezet

Bevezető

A statikus elemzési módszerek segítségével különböző aspektusokból vizsgáljuk a szoftvert alkotó forrásfájlokat, anélkül, hogy végrehajtanánk azokat. Támogatást nyújt a mindennapi szoftverfejlesztői munka során, hiszen fordítási időben felfedezhetjük a szoftver futásidejű hibáit, biztonsági réseit, ellenőrizhetjük, hogy a kód megfelel-e a kitűzött kódolási konvencióknak, valamint hatáselemzést készíthetünk egy kódrészlethez. A hatáselemzés [1] egy olyan módszer, amelynek segítségével azt vizsgáljuk, hogy a szoftverben bekövetkezett változások a szoftver mely további komponenseit érintik, befolyásolják. A programszeletelés egy dekompozíciós technika, amely statikus hatáselemzés eszköze lehet [2]. Programszeleteléssel képesek vagyunk kivonatot készíteni a program azon utasításaiból, amelyek befolyásolják a program egy megadott pontját. Ezen utasítások halmazát programszeletnek (program slice) nevezzük, azt a programpontot, amelyhez a halmazt készítettük pedig szeletelési kritériumnak (slicing criterion). A programszeletelésnek számos gyakorlati alkalmazása van a szoftverfejlesztés különböző fázisaiban: alkalmazható regressziós tesztszelekciónál, párhuzamosításnál vagy akár refaktorálás esetén.

A RefactorErl [3, 4] egy Erlang nyelvben [5] írt keretrendszer, amely Erlang-programok statikus forráskód-elemzéséhez és refaktorálásához alkalmazható. A RefactorErlben meglévő programszeletelő algoritmus [6] a programot vezérlésfolyam- és vezérlésfüggőségi gráfokkal ábrázolja, majd a szeletelési kritérium alapján bejárja ezeket a gráfokat, így állítva elő a programszeletet.

TDK dolgozatomban a meglévő algoritmust elemzem, és többféle javítási ötletet adok meg, annak érdekében hogy az algoritmus eredményét pontosítsam, és a futási idejét pedig csökkentsem. Kitérek a szeletelés két leggyakoribb felhasználási területére, a regressziós tesztszelekcióra és a párhuzamosításra.

1.1. Problémafelvetés és eredmények

A RefactorErl szeletelő komponensének javítását egy ipari körülmények között született tanulmány vetette fel, amelyek a Klarna egy belső, *KRED* nevű szoftverén végeztek el [7]. A vizsgált szoftver esetén az egyes pull requesteknél az összes rendelkezésre álló teszt végrehajtásra került.

A végrehajtandó teszteknek a számát szerették volna csökkenteni a programszeletelés regressziós teszt szelekciós felhasználásán keresztül, annak érdekében, hogy a CI/CD keretrendszert tehermentesítsék. A szeletelő algoritmust öt kisebb pull request esetén alkalmazták. A kísérletek során a szerzők mindegyik esetben többórás futásidőt tapasztaltak, így arra a következtetésre jutottak, hogy az akkori formájában a RefactorErl programszeletelője nem minősült alkalmazhatónak ipari méretű kód bázisokon. A hosszú futásidő mellett pedig további problémát jelentettek a hamis negatív esetek, ezek a domain-specifikus információk hiányából adódtak.

A dolgozatom kezdeti lépései során önálló elemzéseket is végeztem a szeletelés teljesítményét illetően, szintén teszt szelekciós nézőpontból. A saját elemzéseimet túlnyomórészt az Erlang/OTP *mnesia* könyvtárán végeztem el. A hosszú futásidőt magam is tapasztaltam, továbbá felfedeztem, hogy az eredmények magas százalékban hamis pozitívak.

A futásidőt részalgoritmusokra vonatkozó felső időkorlátok megadásával, párhuzamosítással és az elemzés hatókörének szűkítésével racionalizáltam. A hamis pozitív eredményeket a függvényhívási láncokra vonatkozó információk felhasználásával, egy általam implementált utóelemző modul segítségével szűrtem. Az algoritmust finomhangoltam a párhuzamosítás felhasználási céljából. Ez utóbbi esetben a használt reprezentáció és a RefactorErl keretrendszer lehetőségeit használtam ki, így növeltem a valódi pozitív találatokat, és csökkentettem a hamis pozitív találatok számát.

1.2. A dolgozat felépítése

A 2. fejezetben részletezem az algoritmus megértéséhez szükséges fogalmakat, a 3. fejezetben bemutatom az algoritmus alapvető működését. A meglévő és általam készített elemzéseket a 4. fejezetben, ezek alapján a javításokat az 5. fejezetben mutatom be. A javított algoritmus alkalmazásait a 6. fejezet tartalmazza. A 7. fejezetben kitérek kapcsolódó munkákra a témakörben. Végezetül, a 8. fejezetben összegzem az elért eredményeimet és levonom a következtetéseket.

2. fejezet

Alapvető fogalmak ismertetése

Ebben a fejezetben ismertetem a dolgozatom megértéséhez szükséges fogalmakat, technikákat és eszközöket, valamint ezeknek a lehetséges megközelítési módjait.

2.1. Erlang

Az Erlang egy deklaratív paradigmát követő funkcionális programozási nyelv, amelyet valós idejű, elosztott rendszerek tervezéséhez alkottak meg. A nyelv eszközei lehetőséget biztosítanak a konkurens folyamatok, hibatűrés és robusztusság támogatásához, valamint a forráskódon történő módosítások végrehajtásához anélkül, hogy az alkalmazást szüneteltetnénk [5, 8]. Használata leginkább a telekommunikációs infrastruktúrák esetén, az üzenetküldő alkalmazásoknál, illetve a bankszektorban terjedt el, azonban előszeretettel használják más iparágakban is, amennyiben a rendszerkövetelmények nagymértékű rendelkezésre állást (high availability), megbízhatóságot (reliability) és skálázhatóságot követelnek meg [9].

Az Erlang/OTP (Open Telecom Platform) [10] olyan használatra kész (ready-to-use) Erlang-könyvtárakat tartalmaz, amelyek hozzájárulnak a fent említett jellemzőkkel rendelkező rendszerek fejlesztéséhez. Az OTP tartalmaz olyan alapvető eszközöket, mint például a fordító, de – többek között – biztosít adatbázisok kezelésére alkalmas keretrendszert is (*mnesia*). Az Erlang-rendszerek megalkotását előre beépített tervezési alapelvekkel (design principles) segíti [11].

2.2. Statikus elemzés

A statikus elemzés egy módszer, melynek segítségével a szoftvert alkotó forráskódokat vizsgálni tudjuk anélkül, hogy végrehajtsuk azokat. Statikus elemzési módszerekkel elsősorban a szoftver-fejlesztők mindennapi munkáját támogatjuk: elősegítheti a programkód megértését, a kódolási konvenciók betartását vagy a potenciális hibák felderítését. Azonban nem csak a fejlesztők gör-

dülekeny munkáját szolgálja, bizonyos esetekben alkalmazható a minőségbiztosítás eszközeként is (Software Quality Assurance).

Megfelelő forráskód-reprezentáció mellett olyan összetettebb szempontok vizsgálata is lehetséges, mint például a sebezhetőségek felfedése, vezérlésfolyam-elemzés, adatfolyam-elemzés (a szoftver különböző pontjain lévő kifejezések lehetséges értékeinek a kiszámítása), vagy akár a kódban bekövetkezett változások felderítése.

2.3. RefactorErl

A RefactorErl [3, 4] egy Erlang programozási nyelvhez készült, Erlang nyelven íródott statikus forráskód-elemző keretrendszer. A betöltött forrásfájlok belső reprezentációjának köszönhetően lehetővé teszi, hogy hatékonyan tudjuk ezeket különböző aspektusokból elemezni. A következőkben azok a funkcionálisok kerülnek bemutatásra, amelyek a TDK kutatásom szempontjából a leginkább relevánsnak bizonyultak.

2.3.1. Szemantikus programgráf

A forráskódokat reprezentáló szemantikus programgráfot (Semantic Program Graph, röviden SPG) Tóth Melinda és Bozó István adták meg [12].

A szemantikus programgráf egy gyökérrel rendelkező, irányított, címkézett és indexelt gráf, amely az Erlang-forrásfájlok három különböző rétegét egyszerre képes tárolni. Tartalmazza a kód lexikális rétegét, a szintaktikus rétegét a szintaxisfa által és a szemantikus réteget is, például függvényhivatkozásokat és változókra vonatkozó információkat. A kutatásom során vizsgált algoritmus szempontjából kevésbé lényeges az SPG lexikális rétege.

A betöltött fájl reprezentációjában a leglényegesebb elemek a függvények (function), formok, clause-ok és kifejezések (expression), amelyek gráfban csúcsként vannak jelen. Ezek között építjük fel a szintaktikus és szemantikus kapcsolatokat: az egyes csúcsokat a köztük levő összefüggésnek megfelelő címkéjű éllel látjuk el. Az összefüggések túlmutatnak az absztrakt szintaxisfa megszozott keretein, megadhatunk például expression-expression típusú szemantikus kapcsolatokat is. A konstrukciónak köszönhetően hatékonyan és egyszerűen tudjuk azonosítani a kapcsolatokat a gráf csúcsai között.

Példa

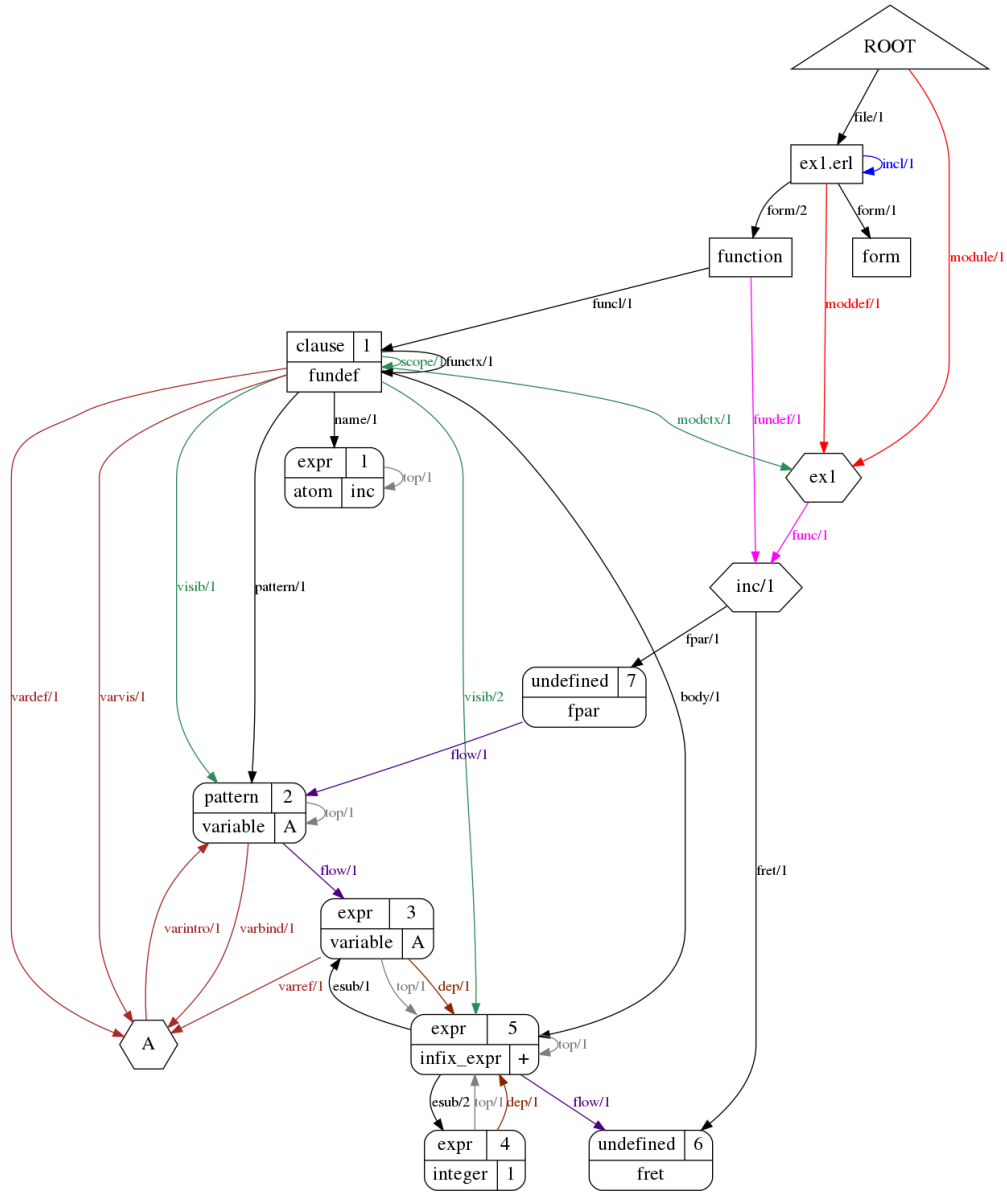
A 2.1 ábrán egy egyszerű Erlang-fájl forráskódja található. Az *inc* függvény a függvényparaméterben kapott számot növeli eggyel. (A függvény nem tér ki a hibakezelésre, ugyanis azzal a céllal lett megírva, hogy egy könnyen átlátható szemantikus programgráf készülhessen belőle.) A kódhoz tartozó szemantikus programgráfot a 2.2 ábra mutatja.

```

1 -module(ex1).
2 -export([inc/1]).
3
4 inc(A) -> A + 1.

```

2.1. ábra. Példa Erlang-forráskódra



2.2. ábra. Egy egyszerű SPG.

2.3.2. Lekérdező nyelv

A lekérdező nyelv [13] célja, hogy a felépített szemantikus programgráfból információt tudjunk kinyerni. A lekérdezések eredménye használható a fejlesztői munka során és összetettebb statikus elemzési módszerek bemeneteként is.

A RefactorErl felhasználói felületein keresztül lehetőségünk van olyan lekérdezéseket megadni, amelyekhez nem szükséges az SPG ismerete, a kód részeihez tartozó megfeleltetés. A 2.3 ábrán egy, az adatbázisba betöltött fájlban található függvényből induló függvényhívási láncokat kérdezzük le, az illető függvényt tartalmazó modul nevének, a függvény nevének, illetve a függvény aritásának megadásával. A lekérdezés eredményének értelmezéséhez szintén nem szükséges a reprezentáció ismerete, ugyanis betöltött fájlokban előforduló *modul:függvénynév/aritás* sorozatokból áll.

```
1 ri:q("mods[name=mnesia_lib].funs[name=db_match_object,arity=2].(calls)+").
```

2.3. ábra. Példa egy felhasználói felületről adott lekérdezésre.

Az általam vizsgált elemzési módszerhez szükséges lekérdezéseket többnyire a keretrendszer *referl_lib* nevű könyvtára tartalmazza. Ezek a lekérdezések az előbbiekkal ellentétben bemenetként a szemantikus programgráf egy konkrét csúcsát várják, illetve egy kapcsolódási pontot vagy ezeknek listáját, ami alapján az SPG csúcsai között keresni tudunk. Egy lekérdezés eredménye tipikusan az SPG csúcsainak a listája. A RefactorErl fájljaiban megfelelő makrók használata mellett a lekérdezések könnyen olvashatóak. A 2.4 ábrán található lekérdezés az SPG-ben `{'$gn', expr, 1133}`-ként azonosított kifejezésre visszaadja annak a függvénynek az SPG-beli azonosítóját, amely a forráskódban a kifejezést tartalmazza.

```
1 ?Query:exec({'$gn', expr, 1133}, ?Query:seq([?Expr:clause(),
2                                     ?Clause:form(), ?Form:func()])).
```

2.4. ábra. Példa RefactorErl-forrásfájlban adott lekérdezésre.

2.3.3. Adatfolyam-elemzés

Az adatfolyam-elemzés módszere segítségünkre lehet abban, hogy a program különböző pontjain levő kifejezések lehetséges értékeit kiszámoljuk. Ilyenfajta elemzéseket többek között a fordítóprogramok is használnak [14]. Az elemzésre több megközelítés létezik, ezek közé tartozik a Reaching Definition Analysis is, amelynek lényege, a program minden pontjára meghatározzuk, hogy amikor a végrehajtás valamilyen útvonal mentén eljut ezen pontok egyikére, akkor melyek azok az értékadások, amelyek ezt megelőzően végrehajtásra kerülhettek, és a hatásuk nincs felülírva [15].

Tekintettel arra, hogy az Erlangban egy változónak csak egyszer adhatunk értéket (single assignment), így az adatfolyam-elemzés megközelítései közül a Reaching Definition Analysis megfelel számunkra, és arra vagyunk kíváncsiak, hogy a program mely pontjai egyeznek meg egy adott kifejezéssel vagy változóval [12].

A RefactorErl *referl_core* könyvtárában található a már meglévő adatfolyam-elemző algoritmus. Ennek implementációja szintén a Reaching Definition Analysis módszerén alapszik, és képes az SPG egy expressionjéhez megtalálni azokat az SPG-beli expressionöket, melyekre teljesülnek a fent leírt feltételek. Az adatfolyam-elemző algoritmus kulcsszereppel bírt az általam kutatott témakörben, így a későbbi fejezetek során még visszatérek erre.

2.4. Hatáselemzés

A hatáselemzés során azt vizsgáljuk, hogy a program egy pontja mely további programpontokat befolyásol, melyek azok a programkomponensek, amikre kiterjed a hatása. Leggyakrabban a változások hatáselemzését (Change Impact Analysis) szeretnénk elvégezni. A szoftver karbantartása a leginkább költséges és legtöbb erőforrást igénylő folyamat a szoftverfejlesztési ciklus élettartama során [1]. Schneidewind [16] szerint a karbantartás azért bonyolult, mert nehéz megállapítani, hogy egy változtatás kihatással van-e a kód többi részére.

A karbantartás alatt álló (vagy karbantartás fázis felé hajló) szoftvertermék esetén kiemelt figyelemmel kell kísérnünk a beérkezett hibajelentések javításával (bug report) vagy módosult vevői igények megvalósításával járó következményeket. Gyakran bevett szokás, hogy ezekben az esetekben a regressziós tesztelés technikáját alkalmazzuk. Regressziós tesztelés esetén arról szeretnénk meggyőződni, hogy a szoftver változásban nem érintett részei továbbra is megfelelnek az elvárt viselkedésnek, teljesítménynek. Ezt a módszert olyan esetekben alkalmazzuk, amikor valamilyen okból kifolyólag (például idő- vagy erőforráshiány miatt) nem lehetséges az összes teszt végrehajtása. A módszer, amivel kiválasztjuk, hogy milyen tesztesetek végrehajtására van szükség regressziós tesztelés esetén, megfontolást igényel és további kérdéseket vet fel [17]. A változtatások hatáselemzése megadja a megfelelő tesztszelekciót.

A hatáselemzést leggyakrabban a regressziós teszteléssel együtt említi a szakirodalom, de egyéb szoftveres tevékenységekben is segítségünkre lehet, például megkönnyítheti a kód refaktorálását. Egy másik példa lehet, amikor - memóriahasználat szempontjából kritikus - beágyazott rendszerek esetén bizonyos komponenseket el kell távolítanunk a szoftverből, és pontosan szeretnénk tudni, hogy ez az rendszert mely pontokon befolyásolja.

2.5. Programszeletelés

A programszeletelés (program slicing) fogalmát Mark Weiser [18] vezette be. A programszeletést magas szinten úgy értelmezte, hogy a program egy adott viselkedését tekintve meg kell határoznunk a program utasításainak azon részhalmazát, amelyben pontosan azok utasítások szerepelnek, amelyek előállítják ezt a viselkedést [2]. A programszeletelés először a debugginggal kapcsolatosan merült fel (annak a vizsgálatánál, hogy melyik utasítások vezethettek el ahhoz, hogy a program egy adott pontján hiba keletkezik), de a szoftverfejlesztés egyéb tevékenységeiben is jól hasznosítható, többek között szoftvertesztelésnél, a szoftver karbantartása esetén, párhuzamosításnál [19]. A programszeletelés a hatásanalízis eszköze.

A programszeletelés egy dekompozíciós technika, melynek segítségével kivonatot készítünk a

program azon utasításaiból, amelyek relevánsak a program ez adott pontján számolt vagy felhasznált értékekre nézve. A halmazt, amely ezeket az utasításokat tartalmazza a program szeletének (program slice) nevezzük. Azt a programbeli pontot, amelyre nézve a programszeletet előállítottuk szeletelési kritériumnak (slicing criterion) hívjuk.

Ahhoz, hogy a program egy szeletét valamilyen kritérium mentén előállítsuk, szükségünk van arra, hogy a programot különböző függőségi gráfokkal ábrázoljuk, például adatfolyamokat és vezérlési függőségeket leíró gráfokkal. Ezeket a gráfokat megfelelően bejárva lehetséges a szelet előállítása.

2.5.1. Programszeletelési típusok

A programszeletelést egymástól független szempontok szerint kategorizálhatjuk. Megkülönböztethetjük az irány szerint, ekkor beszélhetünk visszafelé/hátrafelé menő (backward slicing) és előre felé menő (forward slicing) szeletelésről. De megkülönböztethetjük őket a szeleteléshez felhasznált információk típusa szerint is, például fordítási és futási időben keletkezett információk mentén. Természetesen léteznek egyéb típusok is, például intraprocedurális vagy interprocedurális szeletelés.

Visszafelé menő és előre felé menő szeletelés

Visszafelé menő szeletelés esetén arra vagyunk kíváncsiak, hogy egy adott programpontot mely utasítások befolyásolnak, tehát melyek azok az utasítások, amelyek a vezérlés során már végrehajtásra kerültek, és valamilyen formában befolyásolják a kiválasztott programpontot. A 2.5 ábra 5. sorában található Z változójára nézve a visszafelé menő szelet a második, harmadik és ötödik sorok értékadásai.

Előrefelé menő szeleteléskor egy programponthoz azokat a programbeli utasításokat szeretnénk megkapni, amelyekre ennek a programpontnak kihatása, befolyása van, és a programpont után következnek a végrehajtásban. A 2.5 ábra 5. sorában található Z változójára nézve az előre felé menő szelet az ötödik és hatodik sorok értékadásai.

```
1 example() ->
2     X = 1,
3     Y = 2,
4     A = 5,
5     Z = X + Y,
6     W = Z + 1.
```

2.5. ábra. Elemi példa programszeletelés irányainak bemutatásához.

Az, hogy a programszeletelést milyen irányban kell elvégeznünk, attól függ, hogy mi a szelet felhasználásának célja, kontextusa.

Statikus és dinamikus szeletelés

A programszeletelést rendszerezhetjük a rendelkezésünkre álló információk alapján is. Statikus szeletelésről beszélünk akkor, ha a szeleteléshez felhasználható információkat csak a forráskódból vagyunk képesek kinyerni. A dinamikus szeletés esetében rendelkezésünkre áll valamilyen futás-időben előálló információ, például egy adott bemenetre futtatott végrehajtási történet [20] vagy tesztesetek futtatásból keletkezett adat.

Végrehajtható és nem végrehajtható szeletelés

A szeletelést kategorizálhatjuk az elvárt eredmények és ezeknek végrehajthatósága viszonylatában. Végrehajtható szeletelés esetén a programkódnak egy olyan értelmes részletét állítjuk elő, amely önállóan fordítható, és futtatása reprodukálja a szeletelési kritériumban szereplő kifejezésre vonatkozó viselkedést, a kifejezés értékét. Nem végrehajtható szeletelés esetén a szeletben szereplő kifejezéseknek nem kell egy önmagában értelmezhető egységet alkotniuk, például egy változó értékadásának bal és jobb oldalai külön is szerepelhetnek az eredmények halmazában.

3. fejezet

Programszeletelés a RefactorErlben

A RefactorErl keretrendszerben implementált algoritmus [12, 21] a beolvasott forrásfájlokból épített szemantikus programgráf alapján hozza létre a szeleteléshez szükséges vezérlésfüggőségi gráfokat (Control Dependency Graph, röviden CDG), melyeknek csúcsain történt adatfolyam-elemzéssel a programszelet előállítható. Ebben a fejezetben az algoritmus főbb részei kerülnek bemutatásra.

3.1. Éltípusok a szemantikus programgráfban

Ebben a fejezetben a szemantikus programgráfban található élek közül azok kerülnek bemutatásra, amelyek a programszeletelő algoritmushoz a legfontosabbnak bizonyultak. Emlékeztetőül: a szemantikus programgráf kizárólag **irányított** éleket tartalmaz.

Ezek az élek a következők:

- Függvények elemzéséhez szükséges élek. A függvények feldolgozása az SPG-építésnél úgy történik, hogy amikor a kódfeldolgozás során egy függvényhez érünk, akkor felveszünk az SPG-be egy új *function* csúcsot. Ezt követően, a kód további részeinek feldolgozásánál a függvényre vonatkozó hivatkozásokat kezelni tudjuk. A lokális hívások *funlref*, a külső hívások *funeref* éllel vannak jelölve, a függvény formjából pedig *fundef* él vezet a függvénybe.
- Kontextust reprezentáló élek. Ezeknek az éleknek szerepük van a lekérdezések gyorsításában is, de elsősorban meghatározzák a szemantikus programgráfban szereplő csúcsok közötti hierarchikus függőségeket. Egy kifejezést és a hierarchiában ez alá tartozó részkifejezéseket *esub* címkéjű élekkel kötjük össze, a részkifejezéseket pedig a hierarchiában felettük szereplő kifejezéssel a *top* éllel.
- Változók elemzéséhez szükséges élek. A változó hatókörét *vardef* és *varvis* címkéjű élekkel

jelöljük, a változóra vonatkozó referenciákat *varref*, a változókötést (variable binding) pedig *varbind* élekkel.

- Adatfolyambeli élek. Az adatfüggőségekhez négy különböző éltípus került definiálásra.
 - **flow**: két SPG-csúcs közötti *flow* él azt jelenti, hogy a két csúcs értéke megegyezik.
 - **constructor**, röviden **cons**: az $n_1 \xrightarrow{c_i} n_2$ él azt jelenti, hogy az n_2 csúcs egy több részből álló adatszerkezet, amelynek az n_1 csúcs i -edik komponense.
 - **selector**, röviden **sel**: az $n_1 \xrightarrow{sel_i} n_2$ él azt jelenti, hogy az n_2 csúcs az n_1 összetett adatszerkezetet jelölő csúcs i -edik komponense. A constructor éleket a kifejezésekben, a selector éleket pedig a mintákban használjuk. Például, egy párt (tuple, rendezett kettes) leíró mintaillesztő kifejezésből a pár első elemére sel_1 él, a pár második mintaelemére pedig sel_2 mutat.
 - **dependency**, röviden **dep**: $n_1 \xrightarrow{dep} n_2$ azt jelenti, hogy az n_2 értéke közvetlenül függ az n_1 eredményétől. Például, egy $\mathbf{A} + \mathbf{1}$ alakú kifejezés esetén a $+$ eredménye közvetlenül függ az $\mathbf{A}$ és $\mathbf{1}$ kifejezések kiértékeléseitől.

Egy SPG-beli csúcsból induló éleket a *refcore_graph:links/1* függvényével tudjuk lekérdezni, ami visszaad egy olyan listát, amely $\langle \text{kapcsolattípus, SPG-csúcs} \rangle$ rendezett párokat tartalmaz. A *refcore_graph:path/2* függvénnyel le tudjuk kérdezni egy SPG-csúcsból induló, adott címkéjű élekkel elérhető csúcsok listáját.

3.2. Vezérlésfüggőségi gráf építése

A vezérlésfüggőségi gráfokat függvényenként (pontosabban a függvényekhez tartozó *formok* alapján) építjük, majd ezek összesített eredménye alapján készítjük el a programszeletet. Bár az interprocedurális (függvények közötti) kapcsolatok egy része ebben a lépésben is kiderül, a szeletelés szempontjából sok fontos függvény-függvény kapcsolat az adatfolyam-elemzésnek köszönhetően derül ki. Az algoritmus összes lépésében az SPG belső reprezentációjából nyerjük ki az adatokat.

3.2.1. Vezérlésfolyam-gráf

A vezérlésfolyam-gráf (Control Flow Graph, röviden CFG) megadja az összes függvényen belüli végrehajtási és kiértékelési utat a belépési ponttól kezdve. A CFG-t felépítő algoritmus heurisztikája, hogy figyelembe veszi az Erlangban leggyakrabban előforduló programkonstrukciókat, például az *if*, *case*, *try*, *catch*, *send*, *receive* stb. blokkokat, amelyekre vonatkozóan előzetesen ismerjük a vezérlés lehetséges alakulását.

A CFG-ben két csúcs között lévő címke nélküli él a szekvenciális kapcsolatot jelzi. A címkézett élek valamilyen kitüntetett vezérlésfüggőségi kapcsolatot jelölnek a csúcsok között, ilyen például a függvényhívás (ezt *funcall* címkéjű élen keresztül jelöljük), vagy a *send* és *receive* általi kapcsolat. A mintaillesztések és elágazások esetén *yes* és *no* éleket kaphatunk a megfelelő csúcsok között.

A 3.1 ábrán egy egyszerű függvény és az őt tartalmazó modul forráskódja található, ezen a példán keresztül szeretném bemutatni a CFG-építés egy konkrét esetét. A hozzá tartozó gráf grafikus megjelenítése a 3.2 ábrán található. Ez az egyszerű függvény megnézi a bemenetként kapott szám kettővel vett osztási maradékát, majd visszaadja a paritásnak megfelelő atomot. Páros számok esetén az *even* atomot adja vissza, míg páratlanok esetén az *odd* atomot.

```
1 -module(par).
2 -export([parity/1]).
3
4 parity(N) ->
5     case N rem 2 of
6         0 -> even;
7         1 -> odd
8     end.
```

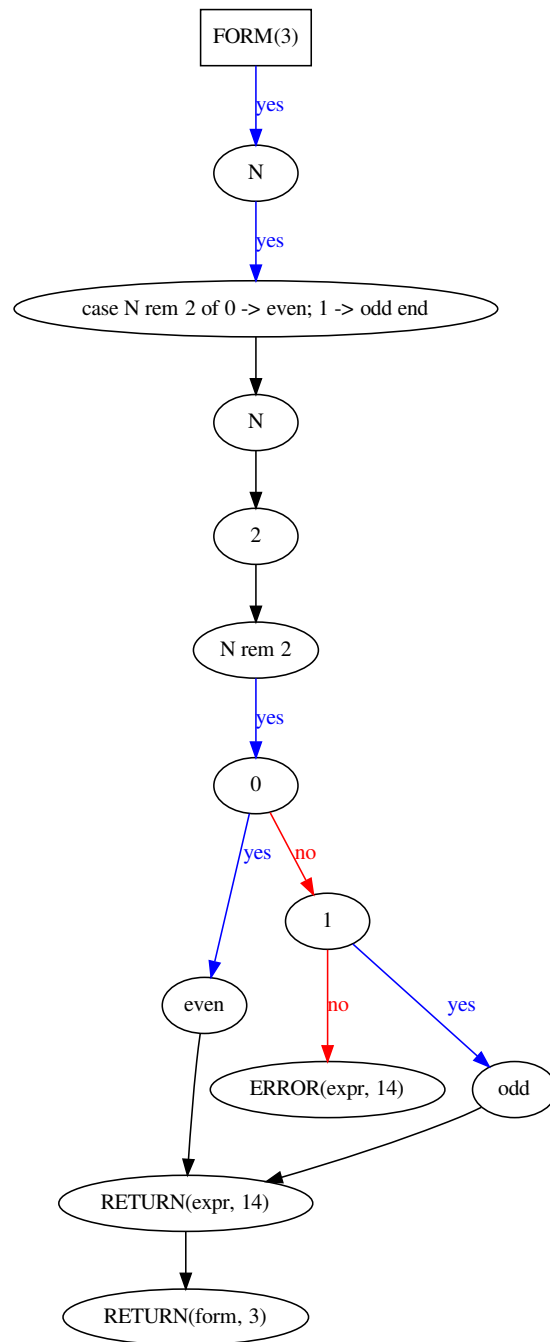
3.1. ábra. Példa egy kódrészletre a vezérlésfolyam-gráf bemutatásához.

Minden függvény esetén a hozzá tartozó vezérlésfolyam-gráfot a függvény szemantikus gráfbeli *form*jának megadásával kérjük le, ez a CFG gyökere. Ebben a példában a *parity* függvényhez az SPG $\{ '\$gn', \text{form}, 3 \}$ azonosítójú csúcsa tartozott, így az irányított gráfnak ez lesz a kezdőpontja.

A **FORM(3)** csúcsból *yes* él vezet az őt követő, **N** csúcsba. Ez egy egyszerű mintaillesztés, ami mindig teljesül. *no* él akkor lenne, ha például az argumentum értékére vonatkozó mintaillesztéssel egy *parity(0)* függvényág is bevezetésre került volna, vagy valamilyen őrfeltétel is meg lenne adva.

A mechanizmus leginkább érdekes pontja a *case* kifejezést reprezentáló részgráf. Kódszinten, *case* törzsében azok az esetek vannak lekezelve, amikor a bemeneti érték 2-vel vett osztási maradéka 0 vagy 1. Viszont nincs olyan ág, amely minden további esetet lefedne. Például, ha a függvény -1-gyel hívnánk meg, aminek a 2-vel osztási maradéka -1, akkor az futásidejű hibát eredményezne. Ezekre az esetekre, amikor fennáll annak a lehetősége, hogy a fejléc kiértékelése egyik meglévő esetre sem illeszkedik, egy új, **ERROR** csúcs lett bevezetve a vezérlésfolyam gráf-reprezentációjában.

Az *even* és az *odd* csúcsokból a **RETURN(expr, 14)** csúcsba vannak irányított élek, ez a teljes *case* visszatérési értékére vonatkozik. Ebből a csúcsból pedig irányított él indul a függvény kilépési pontjába. A vezérlésfolyam-gráf „utolsó” csúcsa a függvény kilépési pontja. Erre azért van szükség, hogy a függvény belépési pontjától a kilépési pontig követni tudjuk az összes lehetséges végrehajtási utat.



3.2. ábra. A 3.1 ábrán található *parity* függvényhez tartozó vezérlésfolyam-gráf.

3.2.2. Posztdominátorok fája

A következő lépés a CFG alapján a posztdominátorok és közvetlen posztdominátorok fájának építése.

A posztdominálás a következőképpen értelmezett: egy CFG-beli n_1 csúcs posztdominálja a CFG-beli n_2 csúcsot, ha bármely n_2 -ből a függvény kilépési pontja felé vezető végrehajtási út tartalmazza az n_1 csúcsot. Ennek egy finomítása a közvetlen posztdominálás (immediate postdominator) meghatározása, amely azt mondja ki, hogy n_1 közvetlenül posztdominálja n_2 -t, ha nem létezik olyan n_3 közbülső csúcs, amire az n_1 posztdominálja n_3 -at és n_3 posztdominálja n_2 -t.

A fenti definíciók alapján előbb megépítjük a felépített CFG-re vonatkozó posztdominátorok fáját (Postdominator Tree, röviden PDT). A PDT segítségével pedig felépítjük a CFG-re vonatkozó közvetlen posztdominátorok fáját (Immediate Postdominator Tree).

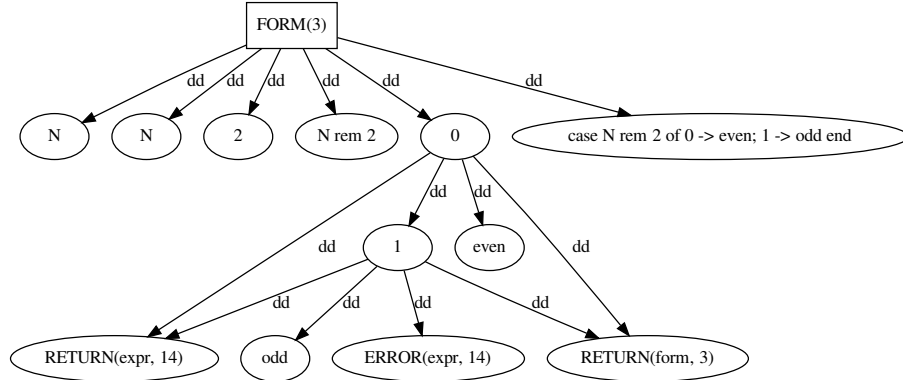
A közvetlen posztdominátor-fa építésének abban van szerepe, hogy a szekvenciák utasításai közötti függőségeket kiiktassuk a vezérlésfüggőségi gráf létrehozásakor. Ha előretekintünk a végső célra, ami a programszeletelés: attól még, hogy két utasítást egymás után hajtunk végre, ez nem feltétlenül jelenti azt, hogy az egyiknek hatása van a másikra. Az ilyen típusú kapcsolatokat szeretnénk itt megszüntetni.

3.2.3. Egyszerű és összetett vezérlésfüggőségi gráf

Az egyszerű vezérlésfüggőségi gráf (Basic Control Dependency Graph, röviden BCDG) a CFG-ből és az közvetlen posztdominátorok fájából állítható elő. Első lépésben kiemeljük a CFG-ből azokat az éleket, amelyek nem szerepelnek a közvetlen posztdominátorok között. Ezt követően, az olyan CFG-élek esetén, amelyek szerepelnek a közvetlen posztdominátorok fájában úgy járunk el, hogy megkeressük a közös őseit a CFG-ben, és testvérekként kötjük be őket a közös ős alá.

Az összetett vezérlésfüggőségi gráf az egyszerű vezérlésfüggőségi gráfból épül fel úgy, hogy a függvényhívásokat feloldjuk. Az egyszerű vezérlésfüggőségi gráfot kiegészítjük még két, függvényalkalmazásokkal kapcsolatos éltípussal. Ezek akkor relevánsak, ha a függvényben szerepel rekurzív hívás vagy egy olyan függvény, amely nem szerepel a betöltött fájlok között (például egy beépített *length* függvény). Az így kapott gráfot nevezzük összetett vezérlésfüggőségi gráfnak (Compound Control Dependency Graph, röviden CCDG). A CCDG-ből számunkra azoknak a csúcspároknek a listája érdekes, amelyek között létezik él.

A 3.1 ábrán található egyszerű függvény esetén az egyszerű és az összetett vezérlésfüggőségi gráfok megegyeznek, mivel nem szerepel benne további függvényhívás. A 3.1 ábrán található *parity* függvényhez a vezérlésfüggőségi gráf a 3.3 ábrán látható.



3.3. ábra. A 3.1 ábrán található *parity* függvényhez tartozó vezérlésfüggőségi gráf.

3.3. Adatfüggőségek felderítése

A vezérlésfüggőségi gráfok segítségével megkapjuk azokat a vezérlési függőségeket, amelyek értékes információt hordoznak a programszeletezés szempontjából. Viszont a vezérlésfüggőségi gráf nem hordoz az adatok áramlására vonatkozó információt, például a szekvenciák közötti függőségeket szándékosan eltüntettük egy korábbi lépésben (függetlenül attól, hogy az adott pontok között létezik-e adatfolyam általi kapcsolat), de nem tartalmazza az olyan típusú adatfüggőségeket sem, amikor egy adat függvényeken utazik keresztül.

Az adatfolyam-elemző algoritmus bemeneteként meg kell adnunk egy kifejezést, és a folyam irányát. Az alapfeltevésünk az, hogy ha minden, vezérlés szempontjából releváns csúcsra lekérjük az adatfolyamot, és ebből függőségeket hozunk létre, akkor elégséges adatunk származik a programszelet előállításához. Az implementációban a CCDG összes csúcsára lekérjük az adatfüggőségeit.

Az adatfüggőségek meghatározása tulajdonképpen a szemantikus programgráf egy bejárása, amelyet egy meghatározott csúcsból indítunk. A bejárás során a korábban ismertetett adatfolyamot reprezentáló éleken (*flow*, *sel*, *cons*, *dep*) lépkedünk. Hosszú adatfolyamok esetén nagyon sok lépést jelent, így az implementáció szintjén felső korlátot szabunk egy kifejezéshez tartozó adatfolyam kiszámítására vonatkozó időre.

Az adatfolyam-elemzést a RefactorErl *referl_core* könyvtárban található *reach* függvény végzi, a már korábban említett *Reaching Definition Analysis* módszert követve. Ennek az elemzésnek két fő megközelítési iránya létezik az implementáció szintjén, amely az irányokra vonatkozik. Az első esetben a függvényt egy kifejezéssel és a **[back, false]** argumentummal meghívva a kapott eredmény olyan kifejezések halmaza, amelyeket a belőle kiinduló irányított adatfüggőségi éleken keresztül érünk el. A másik esetben pedig, a **[back, true]** argumentummal azokat a kifejezéseket a halmazát kapjuk meg, amelyekből létezik irányított út a megadott kifejezésbe az adatfolyaméleken keresztül. Az algoritmus a **[back, false]** verziót használja alapértelmezetten.

3.4. Programszelet kiszámítása

A programszelet kiszámításához meg kell adnunk egy fájlt, és a kifejezést, amire a szeletet elő szeretnénk állítani a fájlban elfoglalt pozíciójának megfelelően, sorával és oszlopával. A pozíció alapján a RefactorErl hozzárendeli azt a szemantikus programgráfbeli csúcsot, amely a megadott kifejezésnek felel meg. Az SPG-csúcs alapján szemantikus lekérdezéssel lekérjük a függvényt, amelyhez a csúcs tartozik. Szintén szemantikus lekérdezések segítségével lekérjük a függvény által érintett hívási láncok lezártjait tartalmazó függvények SPG-beli azonosítóit.

A kapott függvények mindegyikére kiszámoljuk a hozzájuk tartozó összetett vezérlésfüggőségi gráfokat. A vezérlésfüggőségi gráfokban a csúcsok közötti kapcsolatokat rendezett n -esekben megjegyezzük. Lekérjük az összes megjegyzett csúcsra vonatkozó adatfüggőségeket az adafolyam-elemző algoritmus segítségével. Az adatfolyam-elemző algoritmus egy csúcshoz csúcsok listáját rendeli. A kiinduló csúcsból és a csúcsok listájának elemeiből újabb rendezett n -eseket hozunk létre, ezeket szintén megjegyezzük. Utolsó lépésként az eddig akkumulált függőségek segítségével pedig meghatározzuk a programszeletet.

A programszelet előállításánál a kezdőkifejezésünkből indulunk ki. Megvizsgáljuk, hogy az elmentett eredményeink közül melyek azok a csúcsok, amelyekkel valamilyen kapcsolatban (vezérlés- vagy adatfüggőség) áll. Az így kapott eredményt megjegyezzük, és az új csúcshalmazra szintén vizsgáljuk, hogy melyek azok a kifejezések, amelyekkel a halmazelemek függőségben állnak. Ezt a lépést egészen addig ismételjük, amíg már nem lehetséges, hogy a halmazunkhoz új csúcsot adjunk hozzá, mert elfogytak az elérhető függőségek.

Azok a kifejezések, amelyek szerepelnek ebben a halmazban, megadják a programszeletet a kifejezések szintjén. A kifejezésszintű eredményből lekérdezésekkel elő tudjuk állítani a függvény szintű eredményt úgy, hogy meghatározzuk, hogy melyik kifejezés mely függvénynek a része.

4. fejezet

Meglévő algoritmus elemzése

A RefactorErl szeletelő algoritmusának javításának szükségessége elsők között egy ipari szoftver-környezetben történt integráció során merült fel [7]. A tanulmányban levont tanulságokat szem előtt tartva az algoritmust több nyílt forráskódú szoftveren és kisebb példán is elemeztem.

4.1. Teljesítmény ipari környezetben

A Klarna [7] rendszerén végzett kísérletek jól mutatják a szeletelés alkalmazhatóságát az iparban. Bár a szeletelő algoritmust egy konkrét szoftver esetén alkalmazták, a probléma általános, és fennáll a legtöbb CI/CD (continuous integration/continuous delivery, azaz folyamatos integráció és folyamatos szállítás) keretrendszerben.

4.1.1. Integráció célja

A szoftverfejlesztés során fontos, hogy a kódbázison történt (általában a teljes szoftverhez képest relatív kisméretű) módosításokat és hatásukat a fejlesztés fő ágára (development branch) ellenőrizni tudjuk még azelőtt, hogy a fő ággal összefésülnénk (merge) őket a pull requestjeinken keresztül. Ekkor kerülnek képbe CI/CD keretrendszerek (például a Jenkins), amelyek biztosítani tudják különböző rendszerteszt végrehajtásával, hogy a változtatások nem rontanak el semmit. Ha a pull requestek száma megnövekedik, ez nagyon hosszú időbe telhet. Ha a rendszer nem bírja a terhelést, legrosszabb esetben összeomlik. Drasztikus esetben előfordulhat, hogy a pull requestek merge-ölése napokig szünetel.

4.1.2. Eredmények értékelése

A Klarna belső rendszerén, a *KRED*-en, az egyes pull requestek változtatásainak hatáselemzését a RefactorErl szeletelő algoritmusán keresztül végezték el. A cél az volt, hogy megkapják a változtatások által érintett függvényeket, így elégséges legyen kevesebb rendszerteszt végrehajtása, ez pedig jelentős mértékben gyorsítsa a pull request jobokat.

A kísérletet 5 kisebb pull request esetén végezték el, ezek a változtatások átfogóak voltak, azaz érintették a rendszer különböző rétegeit. Az elemzés az egyes pull requestek esetén azonban több óráig is eltartott. Így azt a tanulságot vonták le, hogy jelenlegi formájában nem alkalmazható a RefactorErl teszt szelekcióhoz.

A kiválasztott tesztesetek szintén ellenőrzésre kerültek. A RefactorErl mindegyik esetben az összes tesztesetek számának kevesebb, mint 1%-át választotta ki. Nem került kiválasztásra a változtatások szempontjából irreleváns teszteset, viszont nem sikerült megtalálnia az összes releváns tesztesetet. A releváns tesztesetek hiánya akkor merült fel, amikor a változtatások hatása átfért a szoftverarchitektúra különböző rétegein: például szerveroldal és felhasználói felület közötti vezérlésfolyamot nem sikerült lekövetnie.

4.2. Önálló elemzések

Az szeletelés kiértékelésének tekintetében megkülönböztetnek függványszintű és kifejezésszintű teljesítményeket. Erre azért van szükség, mert a szeletelés különféle alkalmazásainak megfelelően egyik vagy másik információ lehet értékes számunkra.

A szeletelés minden esetben a kifejezések szintjén történik. Függványszintű szeletelésnek nevezem azt az esetet, amikor a szeletben szereplő kifejezéseket leképezzük azokra a függvényekre, amelyek az egyes kifejezéseket tartalmazzák. Kifejezésszintű szeletelésnek nevezem azt az esetet, amikor a szeletben szereplő kifejezések nem kerülnek leképezésre, önmagukban érdekesek számunkra.

Ha a teszteset-szelekció a végső cél, akkor arra van szükségünk, hogy melyek azok a függvények, amelyekhez tartozó teszteket végre szeretnénk hajtani, így nem feltétlenül érdekes, hogy mik azok a konkrét kifejezések, amiken keresztül eljutottunk a függvényekhez. Előfordulhat, hogy nem mindegyik kifejezés szerepel a programszeletben, de mindegyik szükséges függvényhez szerepel legalább egy kifejezés a programszeletben, így elérjük magát a függvényt. Más a helyzet, ha párhuzamosítás vagy debugging szempontjából szeretnénk szeletelni, ekkor a programszeletnek tartalmaznia kell az összes szükséges kifejezést.

4.2.1. Elemzési bemenet

Az elemzések bemenetének kiválasztása már önmagában is kihívást jelentett. Fontos volt, hogy olyan inputfájlokat használjak az algoritmus elemzéséhez, amelyekből képes vagyok levonni helytálló következtetéseket, de ugyanakkor a módszer ne térjen vissza emberi szemszögből feldolgozhatatlan mennyiségű és minőségű információval. Kisebb kód bázisok esetén jellemzően követhető

a találatok listája, nagyobb kódbázisok esetén azonban akár ezres nagyságrendű találatszámról is beszélhetünk, ezeknek ellenőrzése bonyolult a hosszú adatfolyamok és a dinamikus függvényhívások lehetőségei miatt.

Elsőként általam készített kis példákön végeztem függvényszintű hatáselemzést. Ezekben az esetekben előnyös volt, hogy az algoritmus néhány másodpercen belül visszatért az eredménnyel, de egyben hátrányt is jelentett, ugyanis nem sikerült kiderítenem, hogy melyek azok konkrét pontok, amik kevésbé hatékonyak futásidő szempontjából. Az igazán nagy hátránya ennek a módszernek viszont, hogy az algoritmus belső működésének ismeretétől teljes mértékben nem lehet elvonatoztatni egy példa létrehozásakor. Ez különösen igaz függvényszintű hatáselemzés esetén, mivel ilyenkor szándékosan hoztam létre a függvényeket és építettem be függvényhívásokat, ezeknek egy részéről pedig tervezésből adódóan tudtam, hogy az elvárt találatok között szerepelnek-e vagy sem. Az effajta elfogultság fenyegetést jelenthet a következtetések érvényességére vonatkozóan.

A szeletelő algoritmust nem sikerült igazi kihívások elé állítanom egyéb nyílt forráskódú, interneten elérhető, alacsony-közepes komplexitású kódbázisokkal sem. Próbálkoztam - többek között - elsőbbségi sor implementáció [22] tesztelésével, időzóna-átalakítóval [23] és a RefactorErllel kapcsolatosan, egy szakdolgozat keretein belül született *Design Rule Client*tel is (ez utóbbinak a forráskódja nyilvánosan még nem érhető el). Általánosan elmondható, hogy ezekben a projekteknem jellemzőek a hosszabb adatfolyamok, függvényhívási láncok. Így az algoritmus viszonylag kevés találattal tért vissza rövid idő alatt, ezek megfeleleltek az elvárt eredménynek.

Összetettebb kódbázisok esetén „véletlenszerűen” választottam egy kifejezést és kértem le a hozzá tartozó programszeletben érintett függvényeket. Ezekben az esetekben nem vizsgáltam meg előzetesen a lehetséges függvényhívási láncokat és adatfolyamokat, csak arra figyeltem, hogy életszerű legyen a választás, tehát elképzelhető legyen, hogy a kifejezés értéke változik a szoftverrevolúció során. Az algoritmus futása általában órákat jelentett, cserébe azonosíthatóvá váltak az időigényes komponensek. A találatok viszont - természetesen a választott kifejezés és a kódbázis nagyságának függvényében - többszáz vagy akár ezer és kétezer közötti függvényt is tartalmazhattak. Ebben a kategóriában az OTP *mnesia* (4.20.1-es release) könyvtárát [24], illetve egy WhatsApp által közzétett kódtranszformáló eszközt, az *erlfmt*-t [25] választottam ki a szeletelés bemenetét képező kódbázisokként.

4.2.2. Időigényes komponensek azonosítása

A futásidő elemzéséhez szükségem volt egy eszközre, amellyel a szeletelés által felhasznált idő mérése lehetséges. Egy konkrét szeletelés alatt eltelt időt az Erlang *timer* könyvtárában levő *tc* függvénnyel mértem, amelynek meg kell adnunk egy függvényhívást, és az argumentumokat, amelyekkel a függvényt futtatni szeretnénk. A függvény jelen esetben a RefactorErlben programszeletelést végző függvény, az argumentumai pedig a fájl, és a kifejezés pozíciója, amelyre a szeletelést el szeretnénk végezni. A futásidő vizsgálatához *mnesia* különböző moduljait választottam alanyként.

A Klarnán végzett kísérletek [7] alapján az volt a sejtésem, hogy a gráfok számítása a szeletelés leginkább időigényes része. Ezt a feltevést alátámasztotta az is, hogy kis példákön a másodperc töredéke alatt visszatért a szeletelés az elvárt eredménnyel. Kezdetben arra gondoltam, hogy mindhárom, gráfokkal kapcsolatos, főbb rész, tehát a vezérlésfolyam, vezérlésfüggőség és adat-

függőség számítása nagyságrendileg ugyanolyan hosszú időbe telik, és ahhoz, hogy a futásidőt jelentősen csökkentsük, mindhárom esetben heurisztikákra lesz szükség.

Legelőször a három főbb művelet előtti és utáni timestampek (aktuális idő abban a pillanatban, amikor a vezérlés elér az utasításig) kiíratásával próbálkoztam külön-külön. Ebből az látszott, hogy a vezérlési gráf és a vezérlésfüggőségi gráf megépítése nem jelentőségteljes része a teljes futásidőnek. Az igazán időigényes rész pedig az adatfüggőségek felderítése.

Miután kezdett körvonalazódni, hogy az adatfüggőségi kapcsolatok keresése tart a leghosszabb ideig, úgy futtattam az algoritmust, hogy teljesen kiiktattam az adatfolyam-elemző réteget. Ez megerősítést adott arról, hogy futásidő szempontjából valóban ez a kritikus rész. Az időbeli összehasonlítás az eredeti és az adatfüggőségek elhagyása utáni állapotokra nézve a 4.1 táblázatban található. μs -sel jelöltem a mikroszekundumokban mért futásidőt. A könnyebb áttekinthetőség érdekében átszámoltam ezeket percbe is.

4.1. táblázat. A *mnesia* moduljain végzett futásidőre vonatkozó szeletelési kísérletek.

		Eredeti futásidő		Adatfolyam nélküli futásidő	
Modul	Pozíció (sor-oszlopok)	μs	Perc	μs	Perc
<code>mnesia_text</code>	(65:20-26)	4163173317	69.38	53737729	0.89
<code>mnesia_bup</code>	(553:27-30)	3292131151	54.86	771194	0.12
<code>mnesia_controller</code>	(509:9-27)	3946429987	65.77	52936536	0.88
<code>mnesia_locker</code>	(246:9-17)	224958335	3.74	571066	0.009
<code>mnesia_controller</code>	(539:24-28)	4121635328	68.69	48051713	0.80
<code>mnesia_index</code>	(191:5-8)	219327723	3.65	95705	0.001

4.2.3. Találatok ellenőrzése

Ellenőrzés függvény szinten

A `RefactorErl`-ben a függvény szintű programszeletet és eredményt a szeletben szereplő kifejezések eredményéből kapjuk meg, de nem szükséges az összes érintett kifejezés precíz ismerete ahhoz, hogy a függvény szintű eredmény teljes legyen.

A következő ellenőrzést egy *mnesia*s modulra futtatott szelet eredményeire tettem meg. Az eredmény függvények halmazát tartalmazza. A találatokat egyesével, kézzel ellenőriztem. Azt vizsgáltam, hogy az algoritmus által eredményként jelölt függvények közül melyek hamis pozitívak (false positive, röviden FP) és valódi pozitívak (true positive, röviden TP). Az algoritmus 78 találatot adott, ebből mindössze 16 találat volt true positive, a maradék 62 pedig false positive. A könyvtár összesen 2128 darab függvényt tartalmaz. Ha abból a szemszögből vizsgáljuk, hogy 2128 helyett csak 78 függvény tesztéseit kell futtatnunk, akkor természetesen ez egy elfogadható eredmény, viszont azt nézzük, hogy a false positive esetek száma majdnem négyszer annyi, mint a true positive-aké, akkor az algoritmus nem tekinthető megbízhatónak.

A találatok elemzése során felfedeztem egy mintát a false positive találatok esetén. A jelenséget a 4.1 ábrán található példán keresztül szeretném bemutatni.

```

1 -module(math_utils).
2 -export([geometric_mean/1, is_first_positive/1]).
3
4 geometric_mean({N1, N2}) ->
5     AbsN1 = abs_val(N1),
6     AbsN2 = abs_val(N2),
7     calc_mean(AbsN1, AbsN2).
8
9 calc_mean(N1, N2) ->
10     math:sqrt(N1 * N2).
11
12 abs_val(N) ->
13     case is_positive(N) of
14         true -> N;
15         false -> -N
16     end.
17
18 is_positive(N) -> N >= 0.
19
20 is_first_positive([H|_]) -> is_positive(H).

```

4.1. ábra. Példa a hamis pozitív találatok bemutatásához.

A példában néhány egyszerű, matematikai függvény szerepel, melyeknek felhasználásával ki szeretnénk számolni két szám mértani közepét, amennyiben a számok negatívak, akkor az abszolút értékeik mértani közepét. Tegyük fel, hogy az ötödik sorban található *abs_val* argumentumára, tehát az *N1*-re szeletelünk. Kicsit gyakorlatiasabban: meg szeretnénk találni az összes olyan függvényt, amelyre hatással van, amelyet esetlegesen adaptálnunk kellene, ha a függvényhívás argumentumán változtatunk valamilyen formában.

A kifejezésnek hatása lehet az *abs_val* függvényre első lépésben, mivel ezt a kifejezést adjuk át a függvényhívás paraméterében. Szintén hatása van az ötödik sorban található *AbsN1* kifejezésre, mivel ezt az előbbi függvény visszatérési értéke adja meg. Az *AbsN1* a *calc_mean* függvény bemenete, így ezek között úgyszintén fennáll az adatfüggőség. Legvégül, pedig magára a *geometric_mean* függvényre is hatása van, mert ennek a visszatérési értékét pedig a *calc_mean* függvény adja meg. Tehát a helyes eredmény a következő függvényeket tartalmazó halmaz: *{geometric_mean, abs_val, is_positive, calc_mean}*.

A találatok közé azonban mégis bekerül az *is_first_positive* függvény is. Ez annak az eredménye, hogy a vezérlésfüggőségi gráfba bekerül a 18. sorban található *N >= 0* kifejezés. Ez pedig adatfüggőségi kapcsolatban áll a szintén 18. sorban található *is_positive(N)* kifejezéssel, ami pedig elvezet minket a 20. sorban levő *is_positive(H)* kifejezésig. Ennek az az oka, hogy a vezérlésfüggőséggel meghatározott kifejezésekre adatfolyam alapú függőségi elemzést végzünk. A RefactorErl adatfolyam-elemzése elsőrendű és figyelembe veszi a függvényhívások kontextusát, de ezen a ponton sajnos nem a hívási pontból indítjuk az elemzést, hanem a vezérlésfüggőségi elemzéssel megtalált kifejezésből, így nem ismeri az eredeti kontextust és az összes hívási helyre eljuthat az adat statikus elemzési szempontból. Mivel a függvény szintű eredmény a kifejezés szintű eredmények összegzése, így tévesen, ezt a függvényt is találatként jelzi az algoritmus. A probléma az, hogy menetközben elveszíti az eredeti kifejezés kontextusát.

Ugyanez a jelenség áll fenn a *mnesián* végzett kísérletemnél is. A téves találatok mind arra vezethetők vissza, hogy eljutunk a komplexebb függvényektől haladva az elemibb, általános célú,

gyakran használt függvényekig. Ezeken a függvényeken keresztül pedig minden hívási helyüket tartalmazó függvényt a találatok részévé teszünk.

Ellenőrzés a kifejezések szintjén

A teljesítményt a kifejezések szintjén kisméretű forrásfájlokon ellenőriztem. A függvényszintű eredmény tulajdonképpen a kifejezésszintű eredményeken elvégzett lekérdezés, így a RefactorErl-ben elviekben nincs szükség algoritmikus módosításra, csak arra, hogy az utolsó lépést, amikor függvényeket rendelünk a kifejezésekhez, elhagyjuk.

A kifejezésszintű eredményről elmondható, hogy meglehetősen pontatlan: nem tartalmazza az összes szeletbe tartozó kifejezést, és bizonyos esetekben (amelyek az Erlang valós használata során gyakran előfordulnak, tehát nem feltétlenül tekinthetők speciálisnak) pedig hamis pozitív eredményeket tartalmazott.

A talált hibákat egy egyszerű példán szeretném bemutatni, amelynek forráskódja a 4.2 ábrán található. A példát részben Li és Thompson munkája [26] ihlette. A `calc_area_combined` függvénynek két fájl útvonalát adjuk meg, amelyekben téglalapokhoz tartozó adatokat tárolunk. Ezeket az adatokat egy mapek listája tárolja (Erlangban helyes szintaxissal megadva), minden mapben két kulcs található, az egyik a szélességre (`width`), a másik a hosszúságra (`length`) vonatkozik. A függvény feladata, hogy felolvassa az inputfájlban található adatszerkezeteket a `consult` függvénnyel, majd összegezze fájlanként a bennük tárolt téglalapok területeit (**AreaFst** az első fájl esetén, **AreaSnd** a második fájl esetén), és megadja, hogy az egyes fájlok esetén hány téglalapot talált (**NumFst** az első fájl esetén, **NumSnd** a második fájl esetén).

```
1 -module(proc).
2 -export([calc_area_combined/2]).
3
4 calc_area_combined(F1, F2) ->
5     {ok, [S1]} = file:consult(F1),
6     {ok, [S2]} = file:consult(F2),
7     AreaFst = lists:foldl(fun(Map, Sum) ->
8         L = maps:get("length", Map),
9         W = maps:get("width", Map),
10        Sum + L * W end,
11        0, S1),
12    NumFst = length(S1),
13    AreaSnd = lists:foldl(fun(Map, Sum) ->
14        L = maps:get("length", Map),
15        W = maps:get("width", Map),
16        Sum + L * W end,
17        0, S2),
18    NumSnd = length(S2),
19    {NumFst, AreaFst, NumSnd, AreaSnd}.
```

4.2. ábra. Példa a kifejezésszintű eredmény bemutatásához.

Tegyük fel, hogy a programszeletet a 19. sorban található **AreaFst** kifejezésre szeretnénk meghatározni. Ebben az esetben egyértelmű, hogy a hozzá tartozó szelet az 5. sor (mivel onnan kapjuk meg hozzátartozó fájlt), és a 7-11. sorok kifejezései (mivel ezek adják meg a tényleges értéket).

A függvényszintű eredmények esetén az előre felé menő szeletelést vizsgáltam meg. Ebben a példában viszont **visszafelé menő** szeletelést kell alkalmaznunk. A két különböző irány az adatfolyam-elemző algoritmus paramétereinek módosításával követhető le.

A visszafelé menő szeletelés esetén az adatfolyam-elemző algoritmus szintjén az olyan SPG-csúcsokat keressük, amelyekből adatfüggőségi éleken eljuthatunk az **AreaFst**-nek megfelelő SPG-csúcsba. A szelet a szükségesnél sokkal több kifejezést tartalmaz, nagyon sok hamis pozitív találattal. Belekerül például a 6. sorban található **S2** és a 13-18. sorok kifejezései, amelyekkel semmilyen függőségi kapcsolatban nem áll az eredeti kifejezésünk.

A problémát az okozza, hogy a vezérlésfüggőségi gráfot építő algoritmus nem képes tökéletesen feloldani az olyan függvények hívásait, amelyek nem szerepelnek a betöltött fájlok semelyikében sem. Így, mondhatni elővigyázatosságból, beköti a következő szekvencia összes kifejezését az ismeretlen függvényhívást szimbolizáló CDG-csúcs alá - ez adja ki az összetett vezérlésfüggőségi gráfot (compound CDG).

Csupán az eredeti csúcs adatfüggőségeire (vagy függőségeinek a további függőségeire) nem tudunk hagyatkozni. A jelenlegi példában az **AreaFst**-nek önmagában két adatfüggősége van: a 7. sortól kezdődő értékadás bal és jobb oldalai, a baloldal adatfüggősége a jobb oldalhoz vezet, a jobboldalnak mindössze önmaga az adatfüggősége a reprezentációban, ha a visszafelé menő adatfolyam-elemzést választjuk az előre felé menő helyett. Így nem tudunk eljutni az 5. sorig, pedig természetesen ez is a szelet része.

5. fejezet

Szeletelő algoritmus javítása

Ebben a fejezetben az előzőleg feltárt problémák kiküszöbölésének lehetőségeit vizsgálom, és bemutatom, hogy milyen javításokat, megközelítési módszereket alkalmaztam az egyes esetekben.

5.1. Futásidő racionalizálása

Figyelembe véve, hogy az adatfolyam-elemző réteg megszüntetése után a szeletelések semelyik esetben sem teltek több, mint egy percbe a *mnesián* végzett kísérleteim esetén (ez a 4.1 táblázatban látható), így a vezérlésfüggőségi gráfok építésének optimalizációjával nem foglalkoztam. Emellett szólt az az érv is, hogy a vezérlésfolyam- és vezérlésfüggőségi gráfok építése az implementációban megvalósítja a *gen_server* [27] viselkedést, az egyes műveletekre különböző időkorlátok vannak szabva, és egyéb hatékonyságot célzó intézkedések is szerepelnek a *RefactorErl*-ben. A futásidő javításának lehetőségeit a függvény szintű szeletelés esetén vizsgáltam.

Fontosnak tartom megjegyezni, hogy a már meglévő implementációban is felső időkorlát vonatkozott az egy kifejezéshez tartozó adatfolyam-elemzésre. Erlangban lehetőségünk van *receive-after* blokkokat írni, ami azt jelenti, hogy ha egy folyamat eredményét nem kapjuk meg egy milliszekundumban megadott idő alatt, akkor nem várunk tovább. Az adatfolyam kiszámítására mindegyik kifejezés esetén 10 másodpercet szánunk: ha ennyi idő alatt sikerül visszatérnie az adatfolyam-elemzésnek, akkor az eredményt lementjük, egyébként pedig semmisnek tekintjük az elemzést, azaz nem adunk hozzá semmilyen függőséget.

5.1.1. Adatfolyam-elemzés különböző időkorlátokkal

Legelső lépésben azt vizsgáltam, hogy, ha az egyetlen kifejezés adatfolyam-elemzésére vonatkozó 10000 milliszekundumos meglévő időkeretet csökkentem, az hogyan befolyásolja a találatok számának alakulását, illetve a teljes futásidőt különböző *mnesiás* modulok kifejezéseinek szeletelése esetén.

Az 5.1 táblázat első oszlopában a modul neve található, második oszlopában a kifejezés pozíciója (sorának és oszlopainak megadásával). Az eredeti 10000 milliszekundumos futásidő eredményei a *df_10000* oszlopaiban találhatóak, ezt rendre követik a 1000, 100 és 10 milliszekundumos adatfolyam-elemzés eredményei. (A *df* jelölést a *dataflow* kifejezés miatt választottam.) Az *Idő* oszlopokban található értékek a szeletelés teljes futásidejét jelentik, amelyet az áttekinthetőség érdekében percekben adok meg. A *Tal.* oszlopok megadják az adott teljes futásidő alatt keletkezett találatok számát.

5.1. táblázat. A mnesia moduljain különböző időkorlátokkal végzett kísérletek.

Modul	Poz.	<i>df</i> ₁₀₀₀₀		<i>df</i> ₁₀₀₀		<i>df</i> ₁₀₀		<i>df</i> ₁₀	
		Idő	Tal.	Idő	Tal.	Idő	Tal.	Idő	Tal.
mnesia_text	(65:20-26)	69.38	1430	22.27	1409	10.11	1264	4.65	1220
mnesia_bup	(553:27-30)	54.86	3	8.73	3	2.24	3	0.34	3
mnesia_controller	(509:9-27)	65.77	2	18.31	2	7.49	2	2.96	2
mnesia_locker	(246:9-17)	3.74	3	1.20	3	0.37	3	0.14	3
mnesia_controller	(539:24-28)	68.69	1358	21.44	1344	9.60	1286	5.16	1192
mnesia_index	(191:5-8)	3.65	78	1.24	78	0.31	78	0.06	49
mnesia_schema	(1262:5-8)	74.94	1474	27.75	1469	13.88	1286	8.41	1408

A táblázat eredményei azt mutatják, hogy a futásidő jelentősen csökken, hogyha az adatfolyam-elemzésre szánt időkorlátot csökkentjük különböző nagyságrendekkel. Ezzel együtt természetesen a találatok száma is csökken, de semelyik esetben sem veszítünk az eredeti találatok számának 13%-ánál többet.

A *df*₁₀₀₀₀ és a *df*₁₀₀₀ között a 7 különböző esetben összesen 40 találatot veszítettünk el, és 240 percet spóroltunk a futásidő tekintetében. A *df*₁₀₀ és *df*₁₀ esetén már sokkal nagyobb az ugrás a találatok számában.

Ha csak ezeket az eredményeket vesszük figyelembe, kézenfekvő kompromisszum lenne az 1000 milliszekundumos korlátot használni, de ez nem feltétlenül jó megközelítés, amire az *erlfmt* kód-bázisán végzett kísérletem világított rá.

A *mnesia* forrásfájljaira nem kifejezetten jellemző, hogy függvények definíciójánál sokszoros mintaillesztést használjanak, de ez a gyakorlatban egyáltalán nem ritka Erlang-fájlok esetén. Az *erlfmt*-ben ez egy előszeretettel alkalmazott konstrukció, az 5.1 ábrán található egy példa erre. Az adatfolyam-elemző algoritmusról tudjuk, hogy az ilyen és ehhez hasonló esetekben megnövekedett futásidőre számíthatunk. Ha egy olyan kifejezésre szeretnénk a programszeletet lekérni, amely relatív rövid úton egy ilyen mintaillesztéshez vezet, akkor előfordulhat, hogy érdemi eredmény nélkül térünk vissza egy alacsony időkorlát esetén, tehát a végeredmény csak azt a függvényt tartalmazza, amelyben a kezdőkifejezésünk szerepel. Az *erlfmt* esetén ez történt az 1000 milliszekundumos időkorlát mellett. A legalacsonyabb időkorlát, amely elfogadható eredményt produkált, 1550 milliszekundum volt. (A probléma több különböző bemenet esetén is fennállt.)

A fenti eredmények alapján az egyes kifejezések adatfolyam-elemzésére szánt időkorlátot 2000 milliszekundumban állapítottam meg.

```

1 do_expr_to_algebra({string, Meta, _Value}) ->
2   string_to_algebra(erlfmt_scan:get_anno(text, Meta));
3 do_expr_to_algebra({char, #{text := "$\u"}, $_s}) ->
4   <<"$\u\s">>;
5 do_expr_to_algebra({Atomic, Meta, _Value}) when ?IS_ATOMIC(Atomic) ->
6   string(erlfmt_scan:get_anno(text, Meta));
7 do_expr_to_algebra({pid, _Meta, Dotted}) ->
8   concat(<<"<">>, dotted_to_algebra(Dotted), <<">">>);
9 do_expr_to_algebra({concat, _Meta, Values}) ->
10  concat_to_algebra(Values);
11 do_expr_to_algebra({op, _Meta, Op, Expr}) ->
12  unary_op_to_algebra(Op, Expr); ...

```

5.1. ábra. Példa a mintaillesztésre függvénydefiníció esetén. A kód forrása az `erlfmt-n` [25] belüli `erlfmt_format.erl` fájl.

5.1.2. Taskfarm implementáció bevezetése

Az eredeti implementációban kiválasztottuk a CDG-ben szereplő összes kifejezést, és ezekhez egymás után, szekvenciálisan kértük le hozzájuk tartozó adatfüggőségeiket, amiket beszűrtünk ugyanabba a *dets bagbe* [28]. Egy kifejezéshez tartozó adatfüggőségek lekérdezése nincs kihasználással más kifejezésekhez tartozó adatfüggőségek lekérdezéseire. Ez természetesen veti fel a párhuzamosítás ötletét. A *dets* szintén támogatja a párhuzamos folyamatok általi beszűrást.

A **taskfarm** megközelítés azokra az esetekre kínál hatékony megoldást, amikor több, egymástól független folyamatot szeretnénk elvégezni, végeredményben pedig a független folyamatok összegzett eredményére vagyunk kíváncsiak. Az általános algoritmus lényege, hogy van egy diszpécser (dispatcher), aki a feldolgozást menedzseli. Létrehozunk egy adott számú dolgozót (workers), amelynek kiosztunk valamilyen számítási feladatot. Miután a dolgozók elvégezték a rájuk bízott feladatokat, begyűjtjük az eredményt tőlük. A diszpécser ezután a felszabadult dolgozóknak kiosztja a következő számítási feladatot. A dolgozók egymással párhuzamosan futnak, és a dolgozók számát annak alapján határozzuk meg, hogy hány elérhető logikai processzorunk van, ezt Erlangban a `erlang:system_info(logical_processors_available)` paranccsal tudjuk lekérdezni. A számítási folyamat ebben az esetben a kifejezések adatfolyamának lekérésének, és a felderített adatfüggőségeknek beszűrásának felel meg a kapott kifejezéslista egy elemére.

A „taskfarmosított” implementáció megfelelt az előzetes várakozásoknak, az összehasonlítás az 5.2 táblázatban található. Az időt percben fejeztem ki. A futásidő a legjobb esetben az eredeti futásidő kevesebb, mint ötödére csökkent, a vizsgált esetekben pedig átlagosan 3.6-szor gyorsabban ért véget a szeleltetés.

Bár mindkét verziót ugyanazzal az adatfolyam-számításra vonatkozó időkorláttal adtam meg minimális találatvesztés itt is megfigyelhető. Ez nem feltétlenül jelent problémát. A kísérleteim során észrevettem, hogy az összes adatfüggőségből adódó találatszám eltérhet két, egymás után, ugyanazzal a verzióval futtatott szeleltetés esetén is. Ez azt jelenti a program szintjén, hogy az adatfolyam-elemzés futásideje alkalmanként eltérhet ugyanarra a kifejezésre futtatva. A RefactorErlt egy Intel Core i5-8250U processzorral rendelkező számítógépen használtam, a processzor minimum órajele 400, maximum 3400 MHz. A processzor 4 fizikai és 8 logikai maggal (szállal) rendelkezik.

5.2. táblázat. Az eredeti algoritmus futásideje és találat száma összehasonlítva a taskfarm bevezetése utáni verzióval.

Modul	Pozíció	Eredeti df_{10000}		Taskfarm df_{10000}	
		Idő	Találatok száma	Idő	Találatok száma
mnesia_text	(65:20-26)	69.38	1430	22.89	1411
mnesia_bup	(553:27-30)	54.86	3	9.99	3
mnesia_controller	(509:9-27)	65.77	2	20.98	2
mnesia_locker	(246:9-17)	3.74	3	1.26	3
mnesia_controller	(539:24-28)	68.69	1358	21.56	1346
mnesia_index	(191:5-8)	3.65	78	1.38	78
mnesia_schema	(1262:5-8)	74.94	1474	31.22	1465

5.1.3. Nulladrendű és elsőrendű megoldások

Az adatfolyam-elemzésnek létezik egy nulladrendű és elsőrendű verziója is [12]. A szeletelő algoritmus alapvetően az elsőrendű adatfolyam-elemzést használja, ezt a *reach_1st* nevű függvény valósítja meg. A két megközelítés közötti legnagyobb különbség a függvényeken keresztülívelő adatfolyamokra vonatkozik.

Az elsőrendű adatfolyam-elemzés esetén, amikor az elérés függvényeken megy keresztül, akkor megjegyezzük az adott függvényhívás kontextusát a szemantikus programgráf *call/sorszám* és *ret/sorszám* címkéjű élein keresztül. Ez például azt jelenti, hogy ha egy adatfolyamon belüli irányított útba egy függvény *call/1* élén keresztül jutottunk, akkor a folyam további részén a neki megfelelő *ret/1* élén fogunk tovább menni, még akkor is, ha létezik például egy szintén a függvényünkhöz tartozó *ret/2* él ugyanabból a csúcsból indítva. A nulladrendű adatfolyam-elemzés nem tesz különbséget, figyelmen kívül hagyja ezt a kontextust. Ha az előző példát nézzük nulladrendben: ugyanúgy figyelembe veszi a *ret/2* élt is, hiába jutottunk el erre a bizonyos pontra egy *call/1* élén keresztül *call/2* helyett. A RefactorErlben a nulladrendű elérést a *reach* függvény valósítja meg.

Az adatfolyam-elemzés a szeletelés alkalmazásánál a következőképpen néz ki: amennyiben az adott időkeret alatt sikeresen befejeződik az elemzés, akkor elmentjük az eredményt, különben megszakítjuk és semmisnek tekintjük a folyamatot, mivel az eredeti *reach* és *reach_1st* függvények nem képesek parciális eredményt visszaadni.

A nulladrendű adatfolyam-elemzőnek létezik azonban egy újonnan implementált verziója [29], amely képes arra, hogy a megadott időkereten belül előállt parciális adatfolyam-eredményt visszaadja. Ez a szeletelés szintjén az jelenthetné, hogy rövidebb idő alatt több találat előállítására vagyunk képesek, mivel azokat a függőségeket, amelyekre már rászántuk az időt, fel tudjuk használni, és nem tekintjük semmisnek, még akkor sem, hogyha a visszaadott eredmény nem teljes.

Megvizsgáltam a parciális eredményt visszaadó *reach* alkalmazásának lehetőségét is. Az összehasonlítás az 5.3 táblázatban található. Mivel a parciális adatfolyam-elemző csak a **nulladrendű** adatfolyamok esetén van implementálva jelenleg, így az eredeti nulladrendű *reach* szeletelési alkalmazásával vettem össze. Az összehasonlítást az előzetesen ismertetett *mnesia*-fájlokra alkalmaztam. A teljesítményt vizsgáltam különböző időkeretek megadásával. Az egyes kísérletekre

vonatkozó teljes futásidőt a táblázatban percekben adom meg, a találatok számát pedig a *Tal.* oszlopokban.

5.3. táblázat. Az eredeti és a parciális eredményt visszaadó *reach* összehasonlítása szeletelés esetén.

		<i>Eredeti</i> df_{10000}		<i>Eredeti</i> df_{1000}		<i>Parc.</i> df_{10000}		<i>Parc.</i> df_{1000}	
Modul	Poz.	Idő	Tal.	Idő	Tal.	Idő	Tal.	Idő	Tal.
mnesia_text	(65:20-26)	57.03	1288	14.14	1195	75.66	1475	21.65	1444
mnesia_bup	(553:27-30)	21.16	3	3.03	3	25.39	3	4.35	3
mnesia_controller	(509:9-27)	50.65	2	12.01	2	66.83	2	18.93	2
mnesia_locker	(246:9-17)	3.16	3	0.49	3	4.63	3	0.98	3
mnesia_controller	(539:24-28)	55.75	1218	13.30	1160	74.18	1431	22.47	1379
mnesia_index	(191:5-8)	3.23	78	0.39	78	4.13	78	0.70	78
mnesia_schema	(1262:5-8)	81.55	1319	21.19	1292	110.93	1481	28.29	1466

Az táblázatban levő kísérletek azt mutatják, hogy a találatszám szempontjából minden esetben jobbnak bizonyult a parciális eredményt visszaadó *reach*, viszont idő szempontjából rosszabbul teljesített az azonos időkeretet használó verziónál ugyanazon bemeneti kifejezések esetén. Azt, hogy idő szempontjából kevésbé volt hatékony, több tényező is befolyásolhatja. Ennek lehetséges okai között lehet az, hogy több függőség esetén több időbe telik a programszelet meghatározása, de az is elképzelhető, hogy a parciális eredmény kezelése miatt jár többletköltséggel a futásidő tekintetében.

Ha az eredeti, 10000 milliszekundumos verziót hasonlítjuk össze azzal a parciális eredményt visszaadó algoritmussal, amely 1000 milliszekundumos időkeretet használt, akkor viszont kirajzolódik, hogy minden esetben az eredeti futásidő kevesebb, mint fele alatt sikerült ugyanannyi, vagy több eredményt szolgáltatnia. A parciális függőségek segíthetnének a korábban ismeretett *erlfmt* kapcsán felmerült probléma megoldásában is.

A parciális nulladrendű *reach* függvényt végül nem vezettem be a szeletelésbe. Ennek az oka, hogy az algoritmus korábbi fejezetben kifejtett elemzése során arra a következtetésre jutottam, hogy az eredmények jelentős százalékát a hamis pozitív találatok teszik ki. A nulladrendű elérés, mivel a függvényhívási kontextusokat figyelmen kívül hagyja, természetéből adódóan nagy valószínűséggel hozna be újabb hamis pozitív találatokat. Sőt, mivel egy ilyen összetettebb kód alapján nehéz a találatok validálása, az sem zárható ki, hogy az összes újabb találat hamis pozitív. Tehát, nem előnyös csupán a találatszámra és a futásidőre hagyatkozni ebben az esetben.

Az adatfolyamot elemző *reach* algoritmusok implementálása magas komplexitású és időigényes feladat (nulladrendben és elsőrendben egyaránt), és a parciális eredményt visszaadni képes függvény megvalósítása sem különb ezektől. Ahhoz, hogy az adatfolyam-elemzést ténylegesen hatékonyabbá és jobba tehessük a szeletelés szempontjából, szükségünk lenne a parciális eredmény visszaadására képes elsőrendű adatfolyam-elemző algoritmus implementációjára. Ez mind futásidőben, mind a találathalmaz helyességének szempontjából segíthetne, és előfordulhat, hogy az 1000 milliszekundumos időkorlátnál többre nem is lenne szükség.

A RefactorErl projekt szempontjából az itteni eredmények is csak megerősítették a parciális eredményeket adó időkorlátos adatfolyam-elemzés létjogosultságát, így ennek implementálása már elkezdődött.

5.2. Hamis pozitív találatok szűrése

A programszelet ellenőrzése során arra a következtetésre jutottam, hogy a szeletelés függvény-szintű eredményének esetében a találatok jelentős százaléka lehet potenciálisan hamis pozitív. A hamis pozitív eredmények kiszűrése szükséges lépés ahhoz, hogy az algoritmus megbízhatóbbá váljon. Ugyanakkor szem előtt kellett tartanom a Klarna szoftverén végzett kísérletet [7], melynek során az volt a következtetés, hogy a RefactorErlben meglévő algoritmus számos esetben elmulasztotta detektálni az elvárt találatokat. Így egy olyan megoldást kellett találnom ami kiszűri a hamis pozitív eseteket, és biztosan csak hamis pozitív eseteket szűr ki, azaz a valódi pozitív találatokra nincs hatással.

5.2.1. Adatfolyam-elemzés és hamis pozitív találatok kapcsolata

A hamis pozitív találatok a már korábban bemutatott adatfolyam-elemző algoritmusra (vagy annak felhasználási módjára) vezethetőek vissza. A probléma az, hogy amikor adatfolyam-elemző algoritmust egy kifejezésre meghívjuk, akkor nem ismerjük az eredeti kifejezésünk kontextusát, amely a szeletelési kritériumot képezi. Nem annak függvényében adunk vissza adatfüggőségi kapcsolatokat, hogy ezek relevánsak-e erre az eredeti kifejezésre nézve vagy sem, hanem önállóan, az argumentumban átadott kifejezésre vonatkozó adatfolyamot határozzuk meg, és ezt hajtjuk végre az összes, vezérlésfüggőségi gráfban szereplő csúcsra. Tehát az adatfolyam-elemzés önmagában érvényes, csak az összes adatfolyam-elemzés eredménye együttesen nem helyes a programszelet előállítására nézve. Az elérési algoritmus egy adott kifejezés kontextusával való kiegészítése összetett feladat és nem biztos, hogy megvalósítható lenne úgy, hogy nagyméretű kódbázisok esetén is elfogadható legyen a futásideje. Gyakorlatilag egy újabb és komplexebb adatfolyam-elemző algoritmus implementációjára lenne szükség.

Megvizsgáltam azt az irányt, hogy egy CDG-kifejezésre vonatkozó adatfolyam eredményéből milyen kifejezéseket tudnánk esetlegesen kiszűrni a szeletelési kritériumban szereplő kifejezés figyelembevételével, még azelőtt, hogy az algoritmus beszúrná ezeket a függőségek közé, amelyekből a programszelet készül. Ezt kisméretű példákon próbáltam ki, ahol pontosan nyomon tudtam követni a forráskód leképezését a szemantikus programgráfbeli csúcsokra. Nem sikerült olyan megoldást találnom, amely kizárólag azokat az adatfüggőségeket szűrné ki, amelyek a programszeletben a hamis pozitív találatokat eredményezik. Így egy másik irányban indultam el, arra koncentrálva, hogy a szeletbe már bekerült találatokat felülvizsgáljam, és kiszűrjem, amennyiben meg lehet győződni arról, hogy biztosan hamis pozitívak.

5.2.2. Függvények utóelemzése adatfolyambeli szűrésekkel

Az adatfolyam-elemzés módosítására vonatkozó ötletek nem bizonyultak belátható időn belül megvalósíthatónak, így arra a kérdésre kerestem a választ, hogy milyen feltételeknek kell teljesülniük ahhoz, hogy egy, a programszeletben szereplő függvényt biztosan ki tudjunk szűrni a találatok közül, a szeletelés utolsó lépéseként beépített utóelemzéssel (posztemlemzéssel). Elképzelhető, hogy az adatfolyam-elemző algoritmusnak a jövőben születik olyan módosított verziója, amely törekszik szem előtt tartani a szeletelés kapcsán felmerült problémákat, de ennek ellenére nem tudja teljes bizonyossággal kiküszöbölni a problémás eseteket. Így a posztemlemző modul

a jövőben is használható az eredményekre vonatkozó megbízhatóság növelésének érdekében, ha nem sikerül tökéletes algoritmust írunk a szeleteléshez szükséges adafolyamok vizsgálatához.

Emlékeztető. A 4.2.3 fejezetben bemutatott példa (amelyet az olvashatóság kedvéért ismét beszúrtam, ez az 5.2 ábrán található) esetén egyértelmű volt, hogy az *is_first_positive* függvény hamis pozitív találatnak minősült. A szeletelési kritériumban szereplő, 5. sorban levő *N1* kifejezéshez tartozó *geometric_mean* nevű függvény tartozik.

```

1 -module(math_utils).
2 -export([geometric_mean/1, is_first_positive/1]).
3
4 geometric_mean({N1, N2}) ->
5     AbsN1 = abs_val(N1),
6     AbsN2 = abs_val(N2),
7     calc_mean(AbsN1, AbsN2).
8
9 calc_mean(N1, N2) ->
10     math:sqrt(N1 * N2).
11
12 abs_val(N) ->
13     case is_positive(N) of
14         true -> N;
15         false -> -N
16     end.
17
18 is_positive(N) -> N >= 0.
19
20 is_first_positive([H|_]) -> is_positive(H).

```

5.2. ábra. Példa a hamis pozitív találatok bemutatásához.

A programban előforduló, legalább 2 hosszúságú, lehetséges függvényhívási láncok a következők, ha csak a programból betöltött függvényeket vesszük figyelembe:

$$is_geometric_mean \rightarrow abs_val \rightarrow is_positive \rightarrow calc_mean$$

$$abs_val \rightarrow is_positive$$

$$is_first_positive \rightarrow is_positive$$

Látható, hogy nem létezik olyan függvényhívási lánc, amely érintené az *is_geometric_mean* és az *is_first_positive* függvényt is. Így az *is_first_positive* függvény **biztosan kiszűrhető** a végső találatok listájából.

Felvetődik a kérdés: milyen további esetekben szűrhetünk ki egy függvényt a találatok listájából, ha csupán függvényhívási láncokra vonatkozó információkra tudunk hagyatkozni, és ismerjük a függvényt, amely tartalmazza a szeletelési kritériumban szereplő kifejezést? Biztosan kiszűrhetjük például az olyan függvényeket, amelyekre teljesülnek a következő feltételek:

- o Nincsenek alkalmazva, meghívva egyetlen függvényben sem (leszámítva a saját maguk általi rekurzív hívásokat), és nem tartalmaznak olyan függvényhívásokat, amelyek elvezetnek a kifejezésünket tartalmazó függvénybe.

- A belőle induló függvényhívási láncok (az olyan függvényhívási láncok, amelyeknek ez a függvény a legelső eleme) nem hívják meg közvetlenül vagy közvetetten (más függvényeken keresztül) az eredeti kifejezést tartalmazó függvényt.
- Az olyan függvényhívási láncok, amelyek nem a függvényből indulnak, de érintik a függvényt, nem tartalmazzák a függvényt megelőzően az eredeti kifejezésünkhöz tartozó függvényt.

5.2.3. Utóelemző modul implementációja

Az utóelemzéshez először szükségünk van egy adott függvényhez tartozó különböző irányú függvényhívási láncok lekérdezésére. A RefactorErl programszeletelő algoritmus a szeletelés első érdemi mozzanataként lekérdezi a kifejezés által érintett függvényhez tartozó különböző függvényhívási láncokat, így csak a függvényhívási láncok iránya alapján kellett szigorúbb lekérdezéseket definiálnom. A függvényhívásokkal kapcsolatos lekérdezéseket a lekérdező nyelv *reflib_function:funcalls* és *reflib_function:called* függvényei támogatják. A függvényeket természetesen az SPG-beli reprezentációjukon keresztül érjük el.

Egy adott függvényhez tartozó függvényhívási láncok lekérdezésénél három fő esetet különböztettünk meg. Ezeket az 5.3 ábrán találató, kizárólag e célból létrehozott példán keresztül szeretném bemutatni.

```

1  -module(callchains).
2
3  a_fun() -> b_fun().
4
5  b_fun() ->
6      x_fun(),
7      c_fun(),
8      d_fun().
9
10 x_fun() -> ok.
11
12 c_fun() -> e_fun().
13 d_fun() -> f_fun().
14
15 e_fun() -> g_fun().
16 f_fun() -> ok.
17
18 g_fun() ->
19     h_fun(),
20     i_fun().
21
22 h_fun() -> ok.
23 i_fun() -> ok.
```

5.3. ábra. Elemi példa a különböző függvényhívási láncok bemutatásához.

Tegyük fel, hogy mindegyik esetben az *e_fun* nevű függvényhez tartozó egyes láncokat szeretnénk lekérdezni. A függvényhívási lánc minden esetben tartalmazza magát a kiinduló függvényt. A három eset a példán a következőképpen alakul:

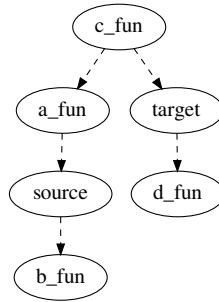
- **előremenő függvényhívási lánc** (forward callchain, röviden FW): tartalmazza a függvényből induló összes lehetséges függvényhívási láncon szereplő függvényt. Az *e_fun* esetén a FW callchain a következő: *e_fun*, *g_fun*, *h_fun*, *i_fun*.
- **visszafelé menő függvényhívási lánc** (backward callchain, röviden BW): tartalmazza azokat a függvényeket, amelyeken keresztüli hívásokon eljuthatunk a függvényhez. Az *e_fun* esetén a BW callchain a következő: *a_fun*, *b_fun*, *c_fun*, *e_fun*.
- **visszafelé-előrefelé menő függvényhívási lánc** (backward-forward callchain, röviden BW-FW): tartalmazza a backward callchain függvényeit, és a backward callchain függvényeihez tartozó forward callchainek elemeit is, kivéve a kiinduló függvény esetén. Az *e_fun* esetén a BW-FW callchain a következő: *a_fun*, *b_fun*, *x_fun*, *c_fun*, *e_fun*, *d_fun*, *f_fun*.

Jelölje **source** a programszeletelési kritériumban szereplő kifejezést tartalmazó függvényt, és **target** a programszeletben szereplő egy találatot. A source-hoz képest a target kiszűrhető, ha a source backward callchainjének nincs közös eleme sem a target forward callchainjével, sem a target backward-forward callchainjével, azaz:

$$BW_{source} \cap FW_{target} = \emptyset \text{ és } BW_{source} \cap BW-FW_{target} = \emptyset$$

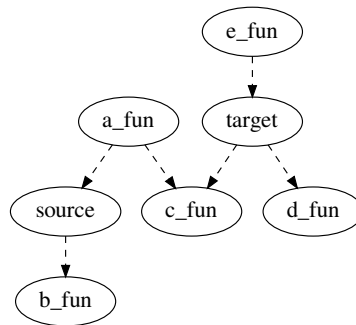
Az utóelemzés célja, hogy biztosan csak a hamis pozitív eseteket szűrjük ki. Így a *target* függvény esetén az összes olyan függvényt figyelembe vesszük, amelyre teljesül, hogy a *target*-ből elérhető (ez az előremenő függvényhívási lánc) vagy a *target* elérhető belőle, esetleg befolyásolhatja a target értékét közvetetten (ez a visszafelé-előrefelé menő függvényhívási lánc). Ez a szigorú feltétel további megoldásokkal valamelyest enyhíthető lenne, de ehhez elengedhetetlen, hogy ismerjük és használni tudjuk a függvények közötti tényleges adatfüggőségeket, viszont most csak a függvényalkalmazásokból származó szemantikus információra szeretnénk támaszkodni.

Hasonló megfontolást követek a *source*-ra vonatkozó érintett függvények meghatározásánál is. Ha csupán a *source*-ot vetjük össze a *target* által érintett függvényekkel, az nem elégséges, mivel előfordulhat, hogy a *target* a *source*-szal ténylegesen függőségben áll, csak ez *source*-ot meghívó függvényen vagy függvényeken keresztül történik. Erre egy példa az 5.4 ábrán található, a szaggatott vonalak a függvényhívásokat jelképezik.



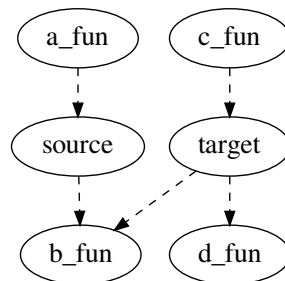
5.4. ábra. Példa egy olyan esetre, amikor a *target* nem szűrhető ki.

Ha pedig a *source*-ra vonatkozó visszafelé-előrefelé menő függvényhívások lennének figyelembe véve, az egy túl szigorú feltételhez vezetne. Az 5.5 ábrán egy olyan eset található, amikor *a_fun* nevű függvény meghívja a *source*-ot és a *c_fun* függvényt is. A *target* szintén meghívja a *c_fun*-t. A *c_fun* szerepel a *source* visszafelé-előrefelé menő függvényhívási láncában, de nem hoz létre függőséget a *target* és a *source* között. A rendelkezésre álló információk alapján a *target* kiszűrhető a *source*-hoz képest.



5.5. ábra. Példa egy olyan esetre, amikor a *target* kiszűrhető.

A *source* előremenő függvényhívási láncain levő függvényeket nem szükséges figyelembe venni. Bármelyik ilyen függvénybe hív bele a *target*, vagy a *target* által érintettek köre, ez nem hoz létre függőséget a *source* és a *target* között. Erre egy példa az 5.6 ábrán található.



5.6. ábra. Szemléltetése annak, hogy a *source* előremenő függvényhívási láncra figyelmen kívül hagyható.

Az előzőekben bemutatott megoldás hátránya, hogy a függvényhívási láncokat lekérdező függvény meglévő implementációja csak a „szokványos” függvényhívásokat veszi figyelembe. Erlangban viszont lehetőségünk van úgynevezett dinamikus függvényhívások megadására. Az 5.7 ábra 7. sorában található egy példa, amikor a *case* fejlécében levő kifejezés kiértékelésének függvényében választjuk ki a modult, amelyből a *select/5* függvényt meghívjuk. Ahhoz, hogy ne szűrjünk ki valódi pozitív találatokat, szükségünk van ezen típusú függvényhívási láncok feldolgozására is.

```

1 select(Tab, Pat, LockKind)
2   when is_atom(Tab), Tab /= schema, is_list(Pat) ->
3     case get(mnesia_activity_state) of
4       {?DEFAULT_ACCESS, Tid, Ts} ->
5         select(Tid, Ts, Tab, Pat, LockKind);
6       {Mod, Tid, Ts} ->
7         Mod:select(Tid, Ts, Tab, Pat, LockKind);
8     - ->
9       abort(no_transaction)
10  end.

```

5.7. ábra. Példa dinamikus függvényhívásra. Forrás: *mnesia*.

A RefactorErlben a szemantikus programgráf megépítésénél alapvetően nincsenek figyelembe véve a dinamikus függvényhívások. Azonban rendelkezésünkre áll a *refanal_dynfun:analyse/0* függvény, amelyet lefuttatva hozzáadásra kerülnek a szemantikus programgráfhoz a dinamikus függvényhívásokat reprezentáló élek. A dinamikus függvényhívásokra vonatkozó kapcsolatokat még a találatok függvényhívási láncainak elemzése előtt ellenőrzöm, és feljegyzem a hozzájuk tartozó függvényhívási láncokkal együtt. Amikor egy találatról eldöntöm, hogy szükséges-e a végső programszeletbe, akkor kiegészítem a függvényhívási láncot ezekkel a dinamikusan előállt függőségekkel. A dinamikus függvények hozzáadása nagy kódbázisok esetén sikertelen lehet, például timeout miatt, így fennáll annak a veszélye, hogy nem végső soron mégsem sikerül az effajta függőségeket detektálnunk.

A posztelemzés egyértelműen negatív irányban befolyásolja a szeletelés teljes futásidőjét. A függvényhívási láncok lekérése is időigényes művelet lehet nagy kódbázison. Ha a kezdetleges programszelet sok függvényt tartalmaz, a futásidő jelentősen megnövekedhet. Annak érdekében, hogy moderálni tudjam a megnövekedett futásidőt, ennek a modulnak az esetében is bevezettem a korábban már bemutatott taskfarm implementációt. Egy számítási feladatnak ebben az esetben az felel meg, hogy egy függvényről eldöntsük, hogy kiszűrhető-e a kezdeti függvényünk tekintetében, a begyűjtött végső eredmény pedig az összes programszeletből szűrhető függvény.

5.2.4. Utóelemzés eredményeinek kiértékelése

Az utóelemző modul bevezetése utáni eredményeket futásidő és találatok szempontjából a korábban ismertetett *mnesiás* fájlokon vizsgáltam meg. Az eredmények az 5.4 táblázatban találhatóak. A futásidőt percekben adtam meg. Mindegyik esetben az adatfolyam-elemzésre szánt időt 2000 milliszekundumban határoztam meg.

5.4. táblázat. Az eredeti algoritmus összehasonlítása a posztelemzéssel kiegészített algoritmussal.

Modul	Pozíció	Eredeti algoritmus		Posztelemzéssel kiegészített algoritmus	
		Futásidő	Találatszám	Futásidő	Találatszám
mnesia_index	(191:5-8)	0.66	78	0.71	16
mnesia_text	(65:20-26)	14.38	1310	15.44	744
mnesia_controller	(539:24-28)	13.81	1246	14.69	713
mnesia_schema	(1262:5-8)	19.02	1382	22.71	921

A korábban használt *mnesia*-modulok közül csak azokat választottam ki, amelyeknél magas volt

a találatok száma, alacsony találatszám mellett nem releváns az utóelemzés, például akkor, ha csupán 2-3 találat van összesen. Annak érdekében, hogy a posztelemzés hatása áttekinthetőbb legyen, meghatároztam a kiszűrt találatok és az eredeti találatok arányát, új futásidő és eredeti futásidő arányát, mindkét esetben százalékban, ez az 5.5 táblázatban található.

5.5. táblázat. Posztelemzés hatásai a találatok számára és a futásidőre.

Modul	Pozíció	Kiszűrt találatok	Futásidő növekedése
mnesia_index	(191:5-8)	79.49%	107.57%
mnesia_text	(65:20-26)	43.20%	114.54%
mnesia_controller	(539:24-28)	42.77%	106.37%
mnesia_schema	(1262:5-8)	33.35%	119.40%

Érvek és ellenérvek az utóelemző algoritmus használatáról

A szeletelés posztelemző algoritmussal való kiegészítése nagyban hozzájárulhat a hamis pozitív találatok kiszűréséhez. Az algoritmus mellett szól, hogy a *mnesia* moduljain végzett kísérletek esetén mindegyik esetben legalább az eredeti találatok 33%-át kiszűrte.

Az algoritmus tervezésből adódóan óvatosan kezeli, hogy egy függvény kiszűrhető-e a találatok közül, így nagy valószínűséggel nem szűri ki a valódi pozitív találatokat. A korábban említett *mnesia_index*-en meghatározott hamis pozitív találatokat az algoritmus maradéktalanul kiszűrte, csak és kizárólag hamis pozitív találatokat szűrt.

A legrosszabb esetben a futásidőt majdnem 20%-kal növelte. Ebből az a következtetés vonható le, hogy az utóelemzés alkalmazása előtt mérlegelnünk kell, hogy fel tudunk-e áldozni futásidőből az eredmények pontosságának növelésének érdekében.

Fennáll az a veszély is, hogy valamilyen speciális, a klasszikus függvényhívásoktól különböző függvényhívási konstrukciót figyelmen kívül hagy az algoritmus, amelyet még a dinamikus függvényhívási élek a szemantikus programgráfhoz adásával sem sikerül észrevennie. Ez azt jelenti, hogy nem sikerül felépítenie a teljes függvényhívási láncokat a szűrésnél vizsgált függvényekre. Így végeredményben mégiscsak kiszűrésre kerülhet néhány valódi pozitív találat is.

Alternatíva az utóelemzésre

A posztelemzés ötlete annak kapcsán merül fel, amikor a vezérlésfüggőségi gráfban megjelentek a gyakran használt, a teljes kódbázis szintjén elemi műveleteket megvalósító függvények, és ezen függvényekhez tartozó adatfüggőségek vezették téves útra a programszeletelést. A posztelemzés jelenlegi formájában természetesen más típusú függvények kiszűrésére is képes, amelyek teljesítik a függvényhívási láncokra vonatkozó feltételeket.

Ha sikerülne definiálnunk valamilyen heurisztikát ezeknek az elemi függvényeknek a detektálására, akkor a hozzájuk tartozó szemantikus programgráfbeli csúcsokat kiszűrhetnénk az adafolyam-elemzésnek átadott csúcsok listájából. Ezzel a módszerrel pedig elkerülhetnénk a szükségtelen adafolyamok elemzését is, ez a futásidő csökkentésében is segítené, illetve megközelíthetnénk a posztelemzés eredményeit is. A módszer még az utóelemzésél is biztonságosabb lehetne.

5.3. Kifejezésszintű eredmények pontosítása

A szeletelés kifejezések szintjén értelmezett eredménye több szempontból is kifogásolható: a helyességet fenyegetik a hamis pozitív találatok, a teljességet pedig a hamis negatív esetek, azaz amikor nem sikerül a szelethez tartozó összes kifejezést felfedeznünk. Ebben a fejezetben az elemzés során észlelt kihívásokra vizsgált megoldási lehetőségeket mutatom be.

5.3.1. Kifejezésszint hatásköre

Egy kifejezéshez tartozó, kifejezések szintjén értelmezett programszelet hatáskörét értelmezhetjük intraprocedurális vagy interprocedurális nézőpontból. Intraprocedurális szeletelésről beszélünk akkor, ha csak a kifejezéshez tartozó függvényen belüli szeletet keressük, míg interprocedurális szeletelés esetén érdekeltek vagyunk azokban az érintett kifejezésekben is, amelyek a függvényen kívül, más függvényekben találhatók.

A kifejezésszintű eredményeket egyszerű, pársoros példákon vizsgáltam és ellenőriztem, mivel komplex kódázisok esetén könnyen követhetlenné válik az eredmény manuális ellenőrzés esetén, kimondottan azokban az esetekben, amikor a kifejezéshez tartozó szeletet interprocedurálisan szeretnénk meghatározni.

A megoldási lehetőségeket elsősorban **intraprocedurálisan, visszafelé menő szeletelés esetén** vizsgáltam, de kitekintettem az interprocedurális, valamint előre felé menő esetekre is.

5.3.2. Adatfolyam-elemzés eredményének kiegészítése

Az adatfolyam-elemzés során egy konkrét kifejezés értékének a lehetséges másolatait vizsgáljuk, azt nézzük meg, hogy hogyan utazik végig ennek a kifejezésnek az értéke a program különböző függvényein, melyik konkrét pontokat érinti. Ha hatáselemzésről vagy programszeletelésről beszélünk, ennél jóval többre van szükségünk. Egy szeletelési kritériumban szereplő kifejezés programszeletébe nem csak azok a kifejezések vagy változók tartoznak, amelyek másolatnak tekinthetők a kezdőkifejezés értékére nézve.

Azok az esetek, melyek során a szeletelő algoritmus elmulasztotta felfedezni az teljes találathalmazt a kifejezések szintén, arra vezethetőek vissza, hogy az adatfolyam-elemzés csak egy konkrét kifejezés másolatait nézi. Ehhez még hozzá vannak adva a *dep* típusú éleken keresztül létrejövő kapcsolatok a csúcsok között, de önmagában ez szintén nem elégséges. Annak érdekében, hogy a hamis negatív esetek bekerüljenek a programszeletbe, néhány a további éltípust is figyelembe kell vennünk *refsc_data_dep* modul függőségeket vizsgáló komponensénél. Ezeknek a szükségességét példákon keresztül szeretném bemutatni.

Esub élek

Az 5.8 ábrán egy egyszerű $f(x) = (\frac{x}{2} - 1)^2$ függvény implementációja található, ahol az $\frac{x}{2}$ értékét az A , $\frac{x}{2} - 1$ értékét pedig a B változóba mentjük el. Tegyük fel, hogy az **visszafelé menő** programszeletet a 6. sorban található B változóhoz szeretnénk lekérni. A programszeletnek az 5. és 6. sorok kifejezéseit kell tartalmaznia, ugyanis ezekben találhatóak azok a kifejezések, amelyek befolyásolják a B kifejezés értékét. Az algoritmus viszont csak a 6. sorban található értékadást sorolja a programszeletbe, mivel az adatfolyam-elemzés eredménye az, hogy a B változó értéke *egy az egyben* csak ebben az értékadásban jelenik meg.

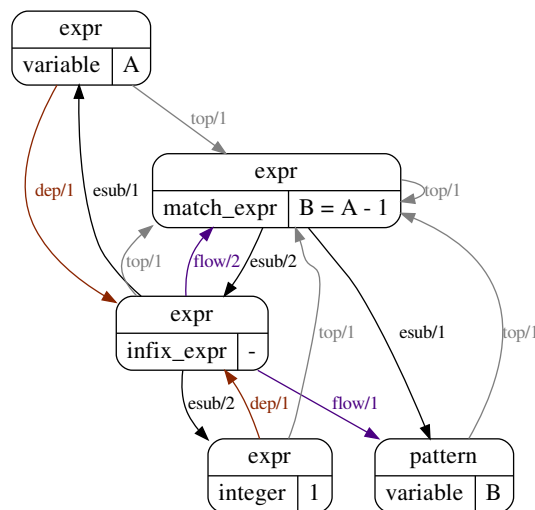
```

1  -module(sub).
2  -export([f/1]).
3
4  f(X) ->
5      A = X / 2,
6      B = A - 1,
7      B * B.

```

5.8. ábra. Példa forráskódra, az *esub* élek indoklásához.

Az 5.9 ábrán található a szemantikus programgráf a forrásfájlban 6. sorban található kifejezések esetén releváns részlete. Az eredeti SPG-rajzból töröltem a belső reprezentációra vonatkozó azonosítókat, és beillesztettem a megfelelő kifejezéseket szövegesen. Látható, hogy az A és $B = A - 1$ kifejezések között csak úgy tudnánk kapcsolatot teremteni, ha vagy az A csúcsból kiinduló *top* élet jegyezzük meg és használjuk fel, vagy a $B = A - 1$ kifejezésből, az *esub/2*, majd onnan az mínuszt reprezentáló csúcsból tovább haladva, az *esub/1* élen keresztül. Ilyen esetekben az *esub* általi kapcsolatokat választottam, amelyet egy kifejezés és a kifejezés részkifejezése közötti kapcsolat reprezentációjára hoz létre az SPG.



5.9. ábra. A szemantikus programgráf egy részlete, az *esub* élek indoklásához.

Az *esub* általi kapcsolatokat, mint minden más adatfüggőséget, párokban jegyzem fel. Mivel visszafelé menő szeletelésről van szó, a pár első eleme az elem, amelyből az él indul, a második eleme pedig az a csúcs, amelyikbe az *esub* él visz az első elemből.

Az adatfolyam-elemzés utolsó lépéseként vizsgáljuk a *dep* típusú éleken keresztül létrejövő függőségi kapcsolatokat, ilyen típusú élék ebben a példában is előfordulnak. A problémát valójában ennél a bemutatott esetnél az jelenti, hogy a programszelet szempontjából vizsgálnunk kellene a *dep* éleken elért csúcsok további adatfüggőségeit is, tehát újra le kellene futtatnunk az adatfolyamelemzést, hogy kiszámoljuk mely csúcsok értéke ugyanaz, mint a függősége. Majd ezt az iterációt újra folytathatnánk, azaz az adatfolyam-elemzés és „*dep* éleken elért csúcsok feljegyzése” műveleteketet tranzitív lezártját kellene kiszámolni. Ez nagyon költséges lenne, ehelyett vezettem be a fenti heurisztikát.

Sel_e, sel és cons_e élek

A *sel_e*, *sel* és *cons_e* élek által jelölt kapcsolatok figyelembe vételének szükségességét ugyanazon a példán szeretném bemutatni. A példa Li és Thompson munkájában [26] használt egyik példa egyszerűsített verziója. A forrásfájlban szereplő kód az 5.10 ábrán található.

Tegyük fel, hogy a 7. sorban található *R1* kifejezéshez tartozó visszafelé menő programszeletet szeretnénk lekérni. A programszeletben szerepelnie kell a 4., 5. és 6 sorok megfelelő kifejezéseinek, a fontosabbak ezek közül: *R1*, *Cols*, *PM1*, *FileName*.

```

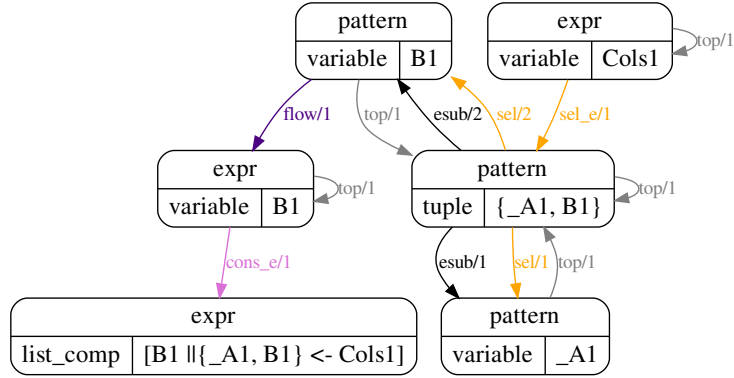
1  -module(image_proc).
2
3  readImage(FileName) ->
4      {ok, #erl_image{format=F1, pixmaps=[PM1]}} = erl_img:load(FileName),
5      Cols1 = PM1#erl_pixmap.pixels,
6      R1 = [B1 || {_A1, B1} <- Cols1],
7      {R1, F1}.

```

5.10. ábra. Példa forráskódra, az *sel_e*, *sel*, *cons_e* élek indoklásához.

Pusztán az adatfolyam-elemzésre hagyatkozva csak a $[B1 \parallel \{_A1, B1\} <- Cols1]$ kifejezésig tudunk eljutni, tehát a programszelet egyedül ezt a kifejezést fogja a tartalmazni, hiszen ebben jelenik meg a kiindulási alapként szolgáló *R1* ugyanabban a formájában. A $[B1 \parallel \{_A1, B1\} <- Cols1]$ kifejezésből pedig szintén nem lehetséges a részkifejezéseinek az elérése az adatfolyam-elemző algoritmussal. A releváns SPG-beli csúcsok és a közöttük levő élek az 5.11 ábrán találhatóak.

A feladat a következő: a $[B1 \parallel \{_A1, B1\} <- Cols1]$ csúcsból el kell jutnunk valahogyan a *Cols1* csúcsba. Más kifejezésünk nincs, amelyből kiindulva ez lehetséges lenne. A *cons_e/1* élen kívül pedig nincs más élünk, amelyen el tudnánk indulni a *Cols1* felé. A *cons_e* élekkel a konstruktor éleket jelöljük, például az ilyen esetekben, amikor a egy listát valamilyen egyéb létező kifejezés alapján konstruálunk meg. Ebből adódóan, helyes a $[B1 \parallel _A1, B1 <- Cols1]$ és *B1-expr* csúcsokat egymással adatfüggőségben állónak jelölnünk. A *B1-expr* és *B1-pattern* csúcsok között van *flow* típusú él, így ezt az adatfüggőséget biztosan detektálni fogjuk.



5.11. ábra. A szemantikus programgráf egy részlete, az `sel_e`, `sel`, `cons_e` élek élek indoklásához.

A `B1-pattern` csúcsba el tudunk jutni a `{_A1, B1}` csúcsból, ha figyelembe vesszük a `sel/2` élt. Itt fordított az irányítás, de ez a programszeletet előállító függvény szempontjából nem jelent majd problémát. A két csúcs között szintén helyes adatfüggőséget megállapítanunk, mivel a `B1-pattern` csúcs komponense az `{_A1, B1}`-nek.

Az `{_A1, B1}` csúcsba pedig a `sel_e/1` élen keresztül tudunk eljutni. Az `{_A1, B1}` pár az értékét a `Cols1` értéke alapján veszi fel, így ebben az esetben is tényleges adatfüggőségről van szó.

A `sel_e`, `sel`, `cons_e` élekkel az adatfolyam-elemzés eredményét egészítjük ki a következőképpen: amikor lekerünk egy kifejezésre vonatkozó adatfolyamot, akkor az összes adatfolyambeli kifejezéshez megvizsgáljuk, hogy létezik-e a kifejezésnek megfelelő SPG-csúcsból `sel_e`, `sel` vagy `cons_e` él. Ezeket a kifejezéseket párokban tárolom, melyeknek első eleme az adatfolyambeli elem, második eleme pedig a belőle `sel_e`, `sel` vagy `cons_e` élel vezető csúcs.

A `sel`, `sel_e` és `cons_e` élek egyfajta `dep` élként viselkednek, azaz a `dep` éltípus speciális eseteinek tekinthetők. Ha a korábban (`esub` élekkel kapcsolatosan) említett adatfolyam-elemzés és „`dep` típusú éleken létrejött kapcsolatok vizsgálata” műveletek tranzitív alkalmazását vetnénk be, szintén nem lenne szükséges ezeknek az éleknek ezen leválasztott figyelembevétele. Természetesen, ez is költséges művelet, ezért inkább külön tekintetem ezeket az éltípusokat.

5.3.3. Adatfolyam-elemzés iránya

Az elemzés során elsőként észlelt probléma az adatfolyam-elemzés irányára vonatkozott. Az adatfolyam-elemző algoritmus alapértelmezetten az argumentumban átadott csúcsból **kiinduló éleken** keresztül térképezi fel az adatfüggőségeket. A szeletelési kritériumban megadott kifejezés és a szemantikus programgráf reprezentáció miatt előfordulhat, hogy azokra a csúcsokra van szükség, amelyek **elvezetnek** a csúcsba az adatfüggőségi éleken keresztül. Az adatfolyamot elemző algoritmusnak tehát követnie kell a programszeletelés által meghatározott irányt.

A szeletelő algoritmust az adatfolyam-elemzés irányának függvényében egy példán keresztül szeretném bemutatni. Az 5.12 ábrán egy olyan függvény található, amely kiszámolja, hogy egy kétéves befektetés évente átlagosan mennyi hozammal jár, ha az első évben 10%-ot, a második évben pedig 20%-ot hoz az eredeti összeghez képest. Tegyük fel, hogy a 7. sorban található *YearFst* változóhoz tartozó programszeletet szeretnénk lekérni. A továbbiakban a programszelet megfelelő kifejezéseire a sorszám megadásával fogok hivatkozni.

```
1 -module(invest).  
2 -export([avg/1]).  
3  
4 avg(Base) ->  
5     Fst = Base + Base * 1/10,  
6     Snd = Base + Base * 2/10,  
7     YearFst = Fst / 2,  
8     YearSnd = Snd / 2,  
9     Avg = YearFst + YearSnd,  
10    Avg - Base.
```

5.12. ábra. Példa forráskódra, a szeletelés és adatfolyam-elemzés irányainak bemutatásához.

Visszafelé menő szelet

A 7. sorban levő *YearFst* esetén a visszafelé menő szelet az 5. és 7. sorokat tartalmazza. A kiegészített adatfüggőségeknek köszönhetően a 7. sorbeli értékadás jobb oldalának elérjük az *Fst* nevű kifejezését. A 7. sor *Fst* kifejezésre pedig visszafelé menő adatfolyam-elemzést kell alkalmaznunk, mivel azokat a kifejezéseket keressük, amelyek őt megelőzik, és az értékük az *Fst* másolatának tekinthető. Így jutunk el az 5. sorban található *Fst* kifejezéshez.

Előrefelé menő szelet

A 7. sorban levő *YearFst* esetén az előrefelé menő szeletet a 7., 9. és 10. sorok alkotják. Ebben az esetben arra van szükségünk, hogy melyek az a *YearFst* után következő kifejezések, amelyekben ez megjelenik. Ezért előremenő adatfolyam-elemzésre van szükségünk. Ennek segítségével el tudunk jutni a 9. sorig. A 9. sor értékadása a szemantikus programgráf szintjén egy *match_expr* típusú kifejezéssel van ábrázolva, amelyből *esub* élek vezetnek az értékadás bal és jobb oldalába is. A kiegészítésnek köszönhetően elérjük a 9. sorban található *Avg* kifejezést, amelyre az előrefelé menő adatfolyam-elemzés detektálja a 10. sorban levő előfordulását. Így teljessé válik a programszelet.

5.3.4. Vezérlési függőségek korlátozása

A kifejezésszintű eredmények vizsgálatánál felfedeztem azt a problémát, hogy, ha szerepelnek a kifejezéshez tartozó függvényben olyan további függvényhívások, amelyeknek a törzsét nem töltjük be a szemantikus programgráfba (mert például beépített Erlang-függvények), akkor ez a szeletelés szempontjából negatívan befolyásolja a vezérlésfüggőségi gráf felépítését. Ezekben

az esetekben felesleges vezérlési függőségek keletkeznek, amelyek téves útra terelik a programszeletet előállító algoritmust, és hamis pozitív találatokat eredményeznek. A hamis függőségek kiküszöbölésének lehetőségeit két módszerrel is megvizsgáltam.

Vezérlési függőségek elhagyása

A kiegészített adatfolyam-elemzés megfelel arra, hogy az adatfüggőségeket felderítsük a kifejezések között, előfordulhat, hogy nincs is szükségünk a konkrét vezérlési függőségekre, amelyeket a CDG biztosít számunkra.

Ezt a gondolatmenetet követve kipróbáltam azt a megoldást, hogy a vezérlésfüggőségi gráfokból csak a csúcsokat jegyzem meg (amelyekre elvégzem a kiegészített adatfolyam-elemzést), de a biztosított vezérlési függőségeket pedig nem veszem figyelembe a programszelet előállításánál. Ezzel a megoldással visszafelé menő szeletelés esetén sikerült intra- és interprocedurális kifejezésszintű szeletelésnél is megadni a megfelelő, hamis pozitív találatok nélküli programszeletet.

Egyszerű vezérlésfüggőségi gráfok használata

A hamis pozitív találatokat eredményező vezérlési függőségeket a szeletelés tevékenységei közül az okozza, amikor az összetett vezérlésfüggőségi gráfot előállítjuk az egyszerű vezérlésfüggőségi gráfból, a függvényhívásokra vonatkozó információk hozzáadásával.

Ha az összetett vezérlésfüggőségi gráf helyett csak az egyszerű vezérlésfüggőségi gráfból adódó kapcsolatokat vesszük figyelembe a programszeletelésnél, akkor ezek a téves vezérlési függőségek eltűnnek.

A helyes programszelet az egyszerű vezérlésfüggőségi gráfból adódó információk segítségével előállt. Ezt visszafelé menő szeletelés esetén, intraprocedurális és interprocedurális esetben is ellenőriztem. Ez egy jobb megközelítésnek tekinthető, mint a vezérlési függőségek teljes figyelmen kívül hagyása. A kipróbált példák esetén teljesült, hogy nem volt szükségünk ezen típusú függőségekre, de létezhet olyan példa, amikor viszont a vezérlési függőségek értékes és releváns információt hordoznak a programszeletelésre nézve.

5.4. Egyéb megoldások vizsgálata

A meglévő szeletelő algoritmus többek között azért lassú, mert teljes mértékben statikus elemzéseket vesz alapul. Ez természetesen felvetette azt, hogy megvizsgáljam, milyen lehetőségek, kapcsolódó eredmények léteznek a dinamikus szeletelés terén. Az algoritmus elemzésénél észrevettem, hogy az adatfolyam-elemzés nem mindegyik esetben járul hozzá érdemben a programszelet eredményeihez, ezért az elemzés hatókörének szűkítésével is kísérleteztem. Illetve, mivel a programszeletet függvényhívási láncokra vonatkozó információk mentén állítjuk elő, kipróbáltam, hogy csak ezekre az információkra hagyatkozva milyen eredmények érhetőek el.

5.4.1. Dinamikus információk beépítése

A dinamikus információk beépítésének lehetőségeit Gyimóthy Tibor és szerzőtársai [20] munkája, valamint Gergely Tamás disszertációja [30] mentén vizsgáltam meg. Az általuk kutatott, úgynevezett $d: U$ (ami a definition-use, azaz definíció-használat rövidítése) algoritmus lényege, hogy futásidőben keletkező adatok elemeihez hozzárendeli a változókra vonatkozó statikus információkat. Ezáltal megközelíti a kifejezésszintű statikus szeletelés eredményeit. Az algoritmusnak több verziója is létezik, visszafelé és előre felé menő szeletelésként is alkalmazható.

A $d: U$ szeletelés a programban előforduló összes változó definíciójának a helyét, illetve a felhasználási helyeit tartja számon, tehát elkészíti a program összes utasításához a definíció-használat reprezentációt. A reprezentáció elemeit a végrehajtási történetben szereplő utasításokhoz rendeli. Az így keletkező adatok segítségével felderíti a végrehajtási nyom egy eleméhez az utasításokat, amelyek változókön keresztül kapcsolatba kerülnek vele a végrehajtás folyamán. Így áll elő az utasítást érintő szeletelési kritériumhoz tartozó programszelet.

A végrehajtási történetet (execution history) a következőképpen definiálták: minden programbeli utasításhoz egy sorszámot rendelnek, és ehhez pedig megadják felső indexként, hogy a végrehajtás során hányadik lépésben került végrehajtásra. A módszer mindenekelőtt **imperatív** programozási nyelvekhez lett kidolgozva, és feltételezi, hogy egy utasításban csak egyetlen változót definiálhatunk, és azt, hogy valamilyen bemenetre rendelkezésünkre áll a végrehajtási történet.

A dinamikus szeletelési módszerekről általánosan, így a $d: U$ -ról is elmondható, hogy szükségünk van valamilyen input esetén előálló végrehajtási történetre, például tesztesetekből származó információra. A végrehajtási történetet a forráskód instrumentálásával állítják elő. A kód instrumentálása röviden annyit jelent, hogy képesek vagyunk nyomon követni (track) egy program végrehajtott utasításait a végrehajtás sorrendjében, anélkül, hogy a program utasításainak a viselkedését befolyásolnánk. Imperatív nyelvekhez léteznek keretrendszerek, amelyek a kód instrumentálását lehetővé teszik, és ezáltal a végrehajtási történet segítségükkel kinyerhető. Erlanghoz nem létezik ilyen publikus eszköz (a végrehajtási történetre vonatkozóan). A *dbg* modullal lehetséges előállítani a függvényyszintű hívási sorozatot, de ez kevés az utasításszintű nyomhoz képest.

A program utasításaihoz, kódrészleteihez a RefactorErlben *megoldható lehet* az előállítása a $d: U$ reprezentációnak a szemantikus programgráf által nyújtott funkcionalitások segítségével. Ezt egy primitív példán keresztül szeretném demonstrálni, melynek forráskódja az 5.13 ábrán található.

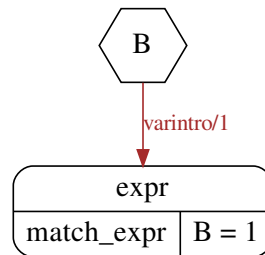
```

1  -module(du_ex).
2  -export([foo/1]).
3
4  foo(A) ->
5      B = 1,
6      C = 2,
7      D = case A > 0 of
8          true  -> B + 1;
9          false -> C - 1
10         end,
11      X = D,
12      Y = D - 1,
13      {X, Y}.

```

5.13. ábra. Példa forráskódra, amelyre lehetséges d : U reprezentáció megadása.

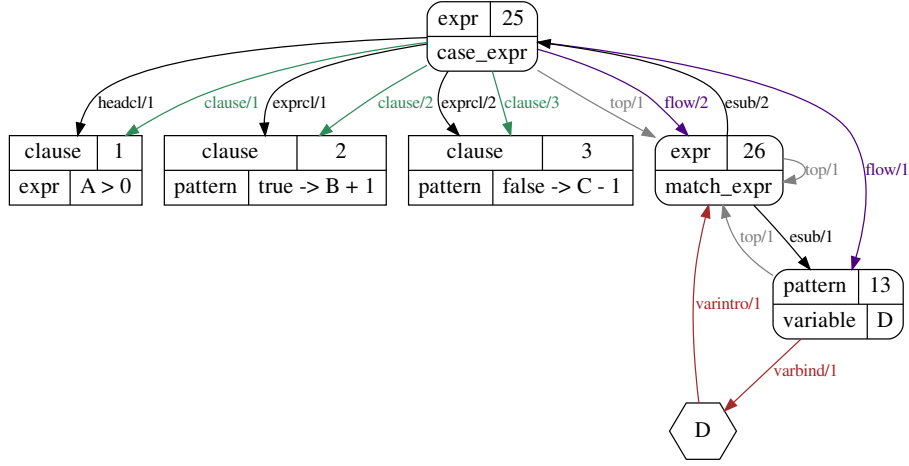
Tegyük fel, hogy ismerjük a forráskódban előforduló végrehajtható utasításokhoz tartozó szemantikus programgráfbeli csúcsokat. A szemantikus programgráfban minden változó esetén létezik egy külön csúcs, amelyből *varintro* címkéjű él mutat arra az utasításra, amelyben a változó létrehozásra került. Például, az 5. sorban található utasításnak megfelelő csúcsba *varintro* él mutat a B változónak megfelelő csúcsból, az SPG releváns részlete ehhez az 5.14 ábrán található. Ugyanez a kapcsolat érvényes a C változóra és a 6. sorban található utasítás SPG-csúcsára.



5.14. ábra. Az SPG egy részlete, a *varintro* él bemutatásához.

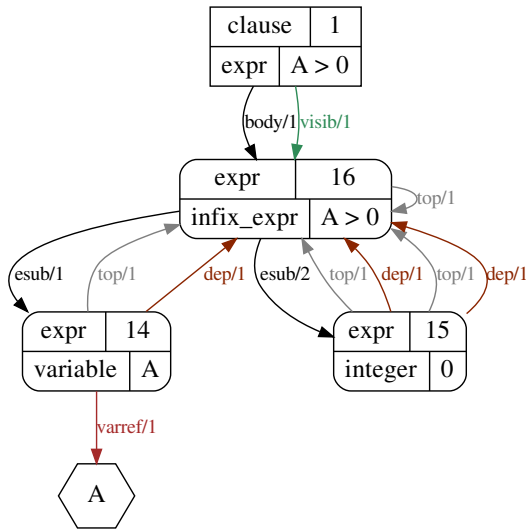
A 7-10. sorokban található *case* kifejezést tartalmazó értékadást egyetlen utasításként reprezentálja az SPG, de ebből csúcsból *esub/1* él indul az értékadás bal oldalába, a D változóba, illetve *esub/2* él a *case ... end* kifejezésbe. A *case* kifejezések esetén a szemantikus programgráf ki-tüntetett módon kezeli a fejlécben található kifejezést. Így a teljes *case...end* kifejezésből *headcl* éllel fogunk az $A > 0$ kifejezésbe jutni. Az erre vonatkozó SPG-részlet az 5.15 ábrán található.

A *clause*-okból lehetséges eljutni azokhoz a változókhöz, amelyeket használnak. Az 5.16 ábrán az $A > 0$ *clause*-ból először a *body/1* élen keresztül elérjük az $A > 0$ kifejezést. Innen elérhető az A változó, aminek a típusa *variable*, de mutat *varref/1* él az A változóba is (ha inkább annak a használata lenne a kézenfekvő az implementáció szempontjából). A *true -> B + 1 clause*-ból a



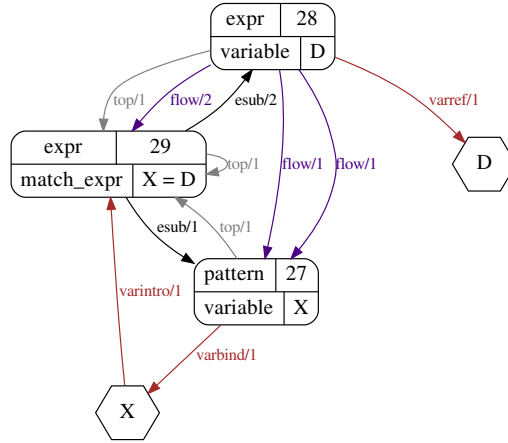
5.15. ábra. Az SPG egy részlete, az 5.13 ábrán található *case*-t tartalmazó értékadáshoz tartozó csúcsok bemutatásához.

B változót, valamint a $false \rightarrow C - 1$ *clause*-ből a C változót hasonlóan tudjuk elérni. Tekintve, hogy a *case* kifejezés fejlécében használjuk az A változót, ezért a *pattern* ágaknak megfelelő *clause*-okban is használjuk az A változó értékét a $d: U$ reprezentáció létrehozási szabályainak megfelelően.



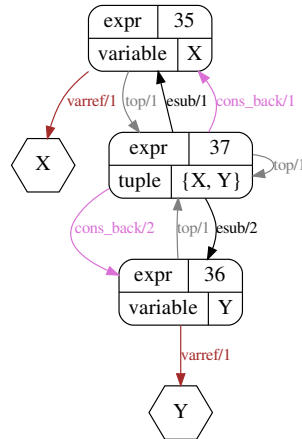
5.16. ábra. Az SPG egy részlete, A változó elérése az $A > 0$ *clause*-ból.

A 11. sorban található értékadásból az X változót *varintro* élen keresztül, a D változót pedig egy köztes lépés után *varref* élen keresztül tudjuk elérni, az erre vonatkozó SPG-részlet az 5.17 ábrán található. Ennek mintájára derítjük fel a 12. sor értékadása által érintett változókat is.



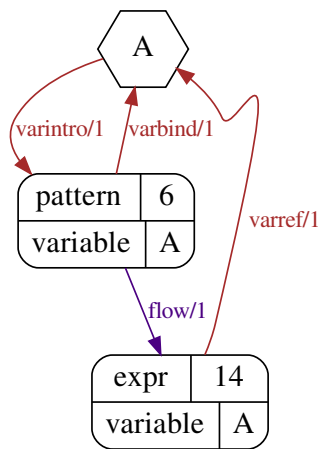
5.17. ábra. Az X és D változók elérése az $X = D$ utasításból.

A 13. sorban levő *tuple* a függvény visszatérési értéke. Ahhoz, hogy ebből el tudjuk érni a változókat, előbb a *cons_back/1* és *cons_back/2* éleken keresztül kell egyet-egyed lépni. Ezután elérjük azokat a kifejezéseket, amelyeknek változó a típusa, de ha az implementációban indokolt magát a változót szimbolizáló szemantikus programgráfbeli csúcsot megtalálni, akkor a *varref* élen keresztül ezt egy további lépéssel megtehetjük. Ez az 5.18 ábrán található.



5.18. ábra. Az SPG egy részlete, X és Y változók elérése az $\{X, Y\}$ tuple-ből.

A program összes részletének változóit sikerült azonosítani, kivéve az A változót. Az A változót a függvény argumentumaként kapjuk meg, egyszer használjuk, a *case* fejlécében. Figyelembe véve, hogy a függvény argumentuma, ezért nem lehetséges a definíciójának egyszerű megadása, mint ahogyan például a B és C változók esetén. Szerencsére, a szemantikus programgráf oly módon ábrázolja ezeket az eseteket is, hogy a definícióját mégis viszonylag könnyen meg lehet adni. Az ilyen esetekben, amikor a változó egy függvényparaméterben érkezik, akkor a szemantikus programgráf úgy jár el, hogy a függvény argumentumát *pattern*ként kezeli. Ez a *pattern* pedig bevezeti a függvényargumentumot, ami jelen esetben az A változó. Így szintén lehetséges, hogy elérjük vagy a kifejezést, amelynek *variable* a típusa egy *flow* élen keresztül, vagy magát a változót reprezentáló SPG-csúcsot, a *varintro* él segítségével. Ennek a szemléltetése az 5.19 ábrán található.



5.19. ábra. Az SPG egy részlete, függvény argumentumaként megkapott A változó kezelése.

Mostmár minden kódrészlethez (legyen az végrehajtható programrészlet vagy *clause*) sikerült beazonosítanunk a vele kapcsolatban álló változókat. A munkákban, melyekből a d : U alapú szeletelés ötletét nyertem, nem tettek említést az ilyen esetekkel kapcsolatban, amikor egy változó értékét egy komplexebb utasításblokk adja meg, jelen példában a D változó értékére gondolok. Ha például a visszafelé menő szeletelést néznénk az X változóra, akkor az általa használt változók mentén szeretnénk meghatározni a programszeletet. Ennek a halmaznak természetesen a D változót kell tartalmaznia. A következő lépésben a D változó utolsó felhasználási helyét keressük. Ezért muszáj felvenni a *case* ágaira vonatkozó használati halmazba a D -t, mivel enélkül a keresés során „átugranánk” ezeket a kódrészeket. Az X értékét a D értéke adja meg, a D értékét pedig (az A -t leszámítva) a *case* ágainak megfelelő műveletek.

Az említett szempontoknak megfelelően az 5.6 táblázatban található program soraihoz a d : U reprezentáció. Az első sorban található A változót, ami a függvény argumentuma, *FunArg*-gal, a függvény visszatérési értékét, pedig *FunRet* jelöléssel egészítettem ki.

5.6. táblázat. Lehetséges $d: U$ reprezentáció az 5.13 ábrán található forrásfájlhoz.

Sorszám	Kódrészlet	$d: U$
1	A (FunArg)	$A : \emptyset$
2	B = 1	$B : \emptyset$
3	C = 2	$C : \emptyset$
4	D = case A > 0 of	$D : \{A\}$
5	true -> B + 1	$P1 : \{A, B, D\}$
6	false -> C - 1	$P2 : \{A, C, D\}$
7	X = D	$X : \{D\}$
8	Y = D - 1	$Y : \{D\}$
9	{X, Y}	$\text{FunRet} : \{X, Y\}$

Ha a kiemelt kódrészletekre vonatkozóan rendelkezésünkre állna egy végrehajtási nyom (például tesztvégrehajtásból), akkor a kódrészletek $d: U$ reprezentációját a végrehajtási nyomban szereplő elemekhez rendelnénk. Így a megadott algoritmusok [31] segítségével előállíthatnánk egy szeletelési kritérium változójára vonatkozó programszeletet, csupán a változók definíciója és használata közötti összefüggések mentén, így nem lenne szükségünk futásidő költséges tekintetében adatfolyam-elemzésre.

Következtetések

A RefactorErlben a szemantikus programgráf nyújtotta információknak köszönhetően nagy valószínűséggel megadható bármely program $d: U$ alapú reprezentációja, annak ellenére, hogy ezt csak néhány nagyon egyszerű példán ellenőriztem. Elképzelhetőnek tartom, hogy a vezérlésfolyamatot leíró gráfok építési szabályainak mintájára az ilyen feldolgozási szempontból bonyolultabb programkonstrukciókra, mint például a példában a *case*, lehetséges olyan szabályokat megadnunk, amelyekkel előállítható a reprezentáció.

A definíció-használat alapú szeletelést a végrehajtási történet nélkül az eredeti formájában nem lehetséges alkalmazni. Erlangban jelenleg nem létezik olyan megoldás, amely legalább egy kezdetleges megoldást nyújtana erre a problémára. További bonyodalmakat okoz, hogy a végrehajtási történetet nem lenne elégséges végrehajtható programrészletenként megadni, a példában a *case* kifejezés esetén kizárólag az szükséges, hogy melyik konkrét ágára futunk rá a végrehajtás során, ha teljes $D = \text{case} \dots \text{end}$ utasítás kerülne be a végrehajtási történetbe, akkor nem tudunk helyes programszeletet előállítani. Egyáltalán nem elhanyagolható a $d: U$ reprezentáció előállításának költsége sem.

Előfordulhat, hogy a $d: U$ reprezentáció egy variációját és a RefactorErl meglévő adatfolyam-elemző algoritmusát valamilyen heurisztika mentén akár kombinálni lehetne, és ezek együttesen gyorsabban és pontosabban teljesítenének a kifejezések szintjén, mint a kiegészített adatfolyam-elemző algoritmus önmagában. Ezt a feltételezést arra alapozom, hogy a $d: U$ reprezentáció előállítása teljesen statikus módon történik és tulajdonképpen egy függvényen belül megadja egy változóra, hogy mely másik változókkal áll vagy állhat kapcsolatban. Továbbá, az előző fejezetekben látható volt, hogy az adatfolyam-elemzés eredményeit új élek általi kapcsolatokkal volt szükséges kiegészíteni annak érdekében, hogy a kifejezésszint pontos lehessen, így lehetséges, hogy a $d: U$ reprezentáció használata egy megfelelő alternatívája lehetne ennek.

A definíció-használat alapú szeletelést csak elméletben vizsgáltam meg, azzal kapcsolatosan, hogy a RefactorErl szemantikus programgráf reprezentációjából képesek vagyunk-e előállítani a módszer bemenetéhez szükséges adatokat. Implementáció ehhez nem készült.

5.4.2. Elemzés hatókörének szűkítése

Az elemzés hatókörének szűkítése egy kombinált megoldásnak tekinthető, elsődleges célja a futásidő csökkentése, de magával vonhatja a hamis találatok számának csökkenését is. Bár az 5.1 fejezetben adott megoldások jelentősen redukálják a futásidőt, amelynek a legnagyobb jelentősége a függvényszintű elemzés esetén van, a javított verzió a teszt szelekció esetén továbbra is hosszú órákon keresztül futott.

A javításokat főként a *mnesián* végzett kísérletekre alapoztam, illetve az *erlfmt* fő komponenseire. A közös hiányosság ezekben az volt, hogy a RefactorErlbe egyik kódbázis (repository) esetén sem voltak betöltve teszteseteket tartalmazó fájlok.

A szeletelő algoritmus minden, az eredeti kifejezés függvénye által érintett függvénybe „belefolylík”, így azokba a függvényekbe is, amelyek a teszteket valósítják meg. Ezekre a függvényekre nem feltétlenül szükséges ugyanazokat a számításigényes műveleteket elvégeznünk, mivel csupán annyit szeretnénk róluk tudni, hogy végre kell-e őket hajtaniuk vagy sem. Ha valamilyen egyéb függvény is meghívásra kerül bennük, például a tényleges teszt adatait inicializáló függvények, az feleslegesen hamis pozitív találatokat hozhat magával.

Az *erlfmt* repository követi az Erlang-keretrendszer konvencionális struktúráját: a „lényegi” forrásfájlok, tehát az alkalmazás működését és logikáját megvalósító fájlok a repository-n belül egy *src* nevű mappában, a tesztfájlok pedig a *test* mappában találhatóak. Más mappában nem található Erlang-fájl, így nem áll fenn az a fenyegetés, hogy valamilyen egyéb fájlok bevonása okozza a megemelkedett futásidőt.

Az *erlfmt* kódbázis *erlfmt_algebra* modulján futtatott kísérleteim jól mutatják, a különbség mennyire drasztikus a futásidő tekintetében. A kísérletek futásidőre vonatkozó teljesítménye az 5.7 táblázatban található. A futásidőt mikroszekundumban (μs), és a könnyebb olvashatóság érdekében percben is megadtam.

5.7. táblázat. Összehasonlítás a szeletelés futásidejében a teljes repository-n (src és test mappák) és csak az src mappában található fájlokon.

Függvény	Futásidő a teljes repository-n	Futásidő az src-fájlokon
next_break_fits/2	26274881544 μs / 437.91 perc	723911764 μs / 12.06 perc
concat/1	26210175836 μs / 436.83 perc	604635369 μs / 10.07 perc
nest/3	24526503083 μs / 408.77 perc	636349415 μs / 10.60 perc

Elemzett függvények szűrése

A futásidőt először az elemzésre szánt függvények előszűrésével próbáltam csökkenteni.

A szeletelő algoritmus első lépésben lekéri az eredeti függvényhez tartozó visszafelé és előre felé menő függvényhívási láncok lezártjait, az elemzést (különböző függőségi gráfok építése, adatfolyamok lekérése) pedig ezeken a függvényeken végzi el. Ebből kiindulva, kezdetben azzal próbálkoztam, hogy a tesztmodulokban levő függvényeket kizártam az elemzés hatóköréből, mivel ha ismerjük az érintett függvényeket az *src* mappán belüli fájlok esetén, akkor onnan a *reflib_function:applications* függvényével le tudjuk kérni a függvény alkalmazási helyeit, ebbe pedig a tesztfüggvényekben történő alkalmazási helyek is beletartoznak. Ez a függvény kifejezések listáját adja vissza, a kifejezésekből pedig néhány köztes lépéssel el tudjuk érni hozzájuk tartozó függvényeket. Így utolsó lépésben csak arra lenne szükség, hogy kiválogassuk a tesztfájlokban szereplő függvényeket. Erlangban a tesztfájlok nevei, ezért a bennük szereplő modulok nevei is, kötelezően tartalmazzák a *SUITE* utótagot (amennyiben a rendszerteszttekhez elterjedt Common Test [32] keretrendszert használják), így a függvények könnyen kiválogathatóak.

Ez a megközelítés nem hozott érdemi javulást. A *concat/1* függvény esetén a tesztfüggvények kizárásával a futásidő 23089345541 μ s, azaz 385.82 perc volt. Természetesen jobb, mint az eredeti, nagyjából 437 perces futásidő, de ha a tesztfájlok és függvényeik teljes figyelmen kívül hagyásával vett, körülbelül 10 perces futásidőt vesszük alapul, akkor ez továbbra is hatalmas különbség.

Tesztszelekció két lépésben

A függvények szűrésére vonatkozó kísérlet azért nem hozta meg az elvárt eredményt, mert attól függetlenül, hogy a tesztfüggvényekre külön nem készülnek elemzések, az adatfolyam-elemző algoritmus továbbra is eléri a tesztesetek kifejezéseit a megfelelő adatfüggőségi éleken keresztül. Az adatfolyam-elemzést pedig nem lehetséges a betöltött fájlok csak egy meghatározott rétegére elvégezni, például kihagyni a teszteseteket tartalmazó mappát.

Így egy olyan megoldásra van szükség, amely első lépésben csak az *src*-fájlokban levő függvényeken adja meg a kezdőfüggvényhez tartozó hatáselemzést, a második fő lépésében pedig ezekhez meghatározza a tesztfüggvényeket. Ez a következő lépésekből áll:

1. A RefactorErlbe betöltöm csak az *src* mappában található fájlokat. (Ha más mappákban találhatóak egyéb, működési logikát tartalmazó fájlok, akkor természetesen azokat a mappákat is szükséges betölteni.)
2. Elvégzem a megfelelő kifejezésre vonatkozó függvény szintű hatáselemzést, ennek a visszatérési eredménye a szemantikus programgráfban függvényeket reprezentáló csúcsok listája.
3. A RefactorErlbe betöltöm a *test* mappában található fájlokat is.
4. Az immár teljes kódbázisra vonatkozóan lekérem a 2. lépésben meghatározott függvényekre, hogy milyen függvények hívják meg őket.
5. A hívó függvények közül kiválogatom azokat, amelyek olyan modulokban találhatóak, amelyek tesztekre utaló elnevezéssel rendelkeznek, tehát a *SUITE* utótaggal.

Ennek a megoldásnak az előnye, hogy a 4-5. lépésben megadott műveletek a másodperc töredéke alatt visszatérnek a végeredménnyel, ez pedig az eredeti *src*-s futásidőt nem növeli érdemben (a három különböző eset egyikében sem telt 0.2 másodpercnél hosszabb időbe). A futásidők és találatsszámok összehasonlítása az 5.8 táblázatban található. A futásidőt mindkét esetben mikroszekundumban és percben is megadtam, a *Tal.* oszlopokban a találatok számát tartalmazzák.

A futásidők jelen esetben átlagosan 8-9 perc között alakultak, az 5.7 táblázatban levő 10-12 perc helyett, annak ellenére, hogy a szeletelő algoritmuson nem változtattam, csak a visszatérési értékét adtam meg másképp. A kísérleteim során sokszor fordult elő ilyen viszonylag alacsony (10-15% körüli) időbeli eltérés két, ugyanazzal a paraméterezéssel megadott szeletelés esetén, amikor újratöltöttem a feldolgozni kívánt kódot, vagy esetleg újraindítottam a RefactorErlt a két futás között. Ebben az esetben is ez történt, így ennek nem tulajdonítok jelentőséget.

5.8. táblázat. Az eredeti teszt szelekció összehasonlítása a kétlépcsős teszt szelekcióval.

Függvény	Eredeti teszt szelekció		Kétlépcsős teszt szelekció	
	Futásidő	Tal.	$\sum$ Futásidő	Tal.
<code>next_break_fits/2</code>	26274881544 μ s / 437.91 perc	91	519317997 μ s / 8.65 perc	45
<code>concat/1</code>	26210175836 μ s / 436.83 perc	105	495676125 μ s / 8.26 perc	46
<code>nest/3</code>	24526503083 μ s / 408.77 perc	96	505850370 μ s / 8.43 perc	45

Az új találatsszám mindegyik esetben kevesebb, mint fele az eredeti találatsszámnak. A hiányzó találatokat ellenőriztem a *next_break_fits* esetén. Kizárólag hamis pozitív találatok kerültek kiszűrésre. Ezt viszonylag egyszerűen el tudtam dönteni, mivel javarészt nem az *erlfmt_algebra* modulhoz tartozó *erlfmt_algebra_SUITE.erl* fájlban található tesztek kerültek kiszűrésre, illetve a kiszűrt tesztek egyéb modulok elemibb függvényeit tesztelik, amelyek nem befolyásolják az eredeti függvényt.

A módszer mind találatok, mind futásidő szempontjából ígéretesnek tűnik. Azonban nem elhanyagolható az a fajta kellemetlenség, hogy valós integráció esetén folyamatosan felügyelnünk kell, hogy milyen fájlok vannak éppen betöltve az adatbázisba.

5.4.3. Kifejezések előszűrése

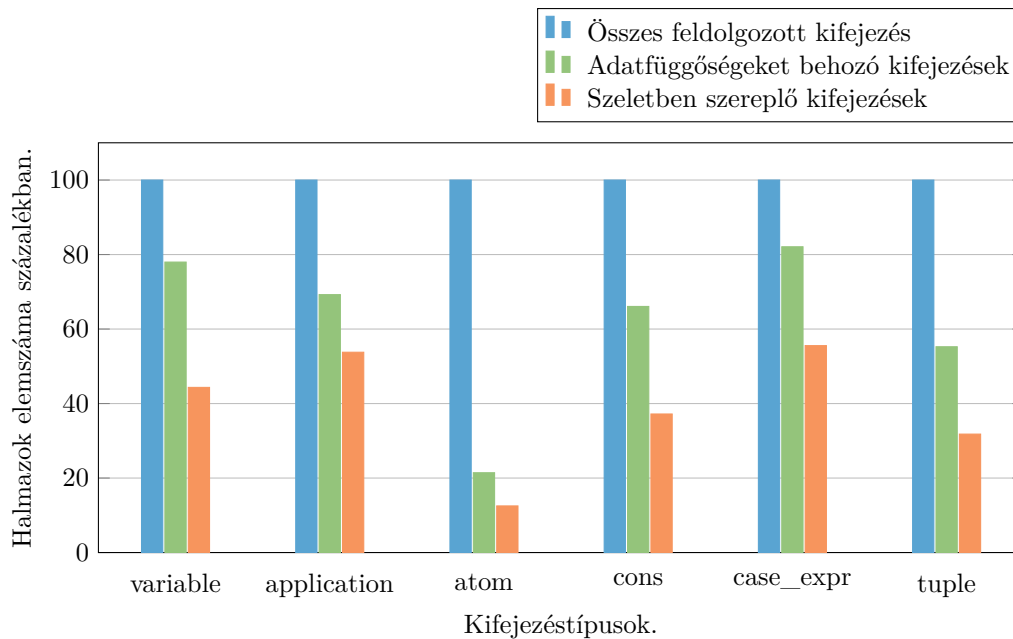
Az összetett vezérlésfüggőségi gráf mindegyik kifejezésére elvégezzük az adatfolyam-elemzést, ez egy biztonságos megoldásnak tekinthető. A kifejezésekről viszont rendelkezésünkre állnak értékes szemantikus információk, a szemantikus programgráf belső reprezentációjának köszönhetően. A kifejezések típusait meg tudjuk különböztetni, és hatékonyan le tudjuk kérni a *reflib_expression:type/1* függvénnyel. Megkülönböztetünk például változókat, atomokat, függvényhívásokat. Ez vetette fel annak az ötletét, hogy megvizsgáljam, hogy a kifejezés típusa és a kifejezésre vonatkozó adatfolyam-elemzés szükségessége között van-e valamilyen összefüggés, ha függvény szintű eredményt várunk el a szeleteléstől. Egy kifejezést kevésbé költséges még az adatfolyam-elemzés előtti lépésben kiszűrni valamilyen egyszerű feltétel alapján, mint felderíteni a hozzá tartozó adatfüggőségeket.

A kísérleteimből levont következtetéseket a korábban már ismertetett, *mnesia_schemán* végzett szeletelésen keresztül szeretném bemutatni. A többi „szokásos” *mnesia*-modulra is megvizsgáltam a kifejezések típusait, mindegyik esetben ugyanaz a jelenség áll fenn.

A kifejezések vizsgálatához a következő lépéseket alkalmaztam:

1. Az összes CDG-beli kifejezésre, tehát azokra a kifejezésekre, amelyekre az adatfolyam-elemzést elkészítjük, lekértem a hozzá tartozó típust.
2. Kiválogattam az összes kifejezés közül azokat, amelyekhez az adatfolyam-elemzés nem üres eredménnyel tért vissza. Ezek azok a kifejezések, amelyeknek van felfedezhető adatfüggőségük (önmagukon kívül) a szemantikus programgráfban, vagy a hozzájuk tartozó adatfolyam-elemzés nem végezhető el a megadott időkereten belül.
3. Az előző lépésben meghatározott kifejezések közül kiválogattam azokat, amelyek belekerültek a programszeletbe, tehát a szeletet meghatározó algoritmus szempontjából relevánsnak bizonyultak.
4. A kifejezéseket csoportosítottam a típusuk (*variable*, *application* stb.) szerint.
5. Csoportonként ellenőriztem, hogy az egy csoportba eső kifejezések közül hány darab járul hozzá az adatfüggőségekhez, valamint a végső programszelethez.

Az 5.20 ábrán található néhány fontosabbnak bizonyult kifejezéstípus, és a hozzájuk tartozó relatív információk az adatfüggőségekre és a programszeletre nézve. A kék oszlopok egy típusra vonatkozó összes CDG-beli kifejezést jelölik. A zöld oszlopok az összes kifejezéshez képest az adatfüggőségeket behozó kifejezések halmazának elemszámát, a narancssárga oszlopok pedig az összes kifejezéshez képest a programszeletben szereplő kifejezések halmazának elemszámát jelentik. Ezt a viszonyítást mindkét esetben (zöld és narancssárga oszlopok) százalékban adtam meg az eredeti halmaz elemszámához képest.



5.20. ábra. A *mnesia_schema* CDG-kifejezéseinek típusaira vonatkozó információk. Az ábrán csak a jelentősebb típusok szerepelnek.

Észrevehető, hogy nagyon sok kifejezésre vonatkozó adatfolyam-elemzés nem szolgál adatfüggőséggel, és az adatfüggőségeket behozó kifejezések közül sem mindegyik releváns a végső programszelet számára. A legtöbb esetben az eredeti CDG-kifejezések kevesebb, mint 50%-a szerepel a programszeletben.

Az 5.9 táblázatban találhatóak az egyes halmazok elemszámai. Az 5.20 ábrán található kifejezéseket a *field_list*, *joker*, *arglist* típusú kifejezésekkel egészítettem ki. (Ezekon felül még sok egyéb típusú kifejezés is van, például *integer*, *string* stb.)

A három leggyakrabban előforduló kifejezéstípus a *variable*, *application* és *atom*. Az összes vizsgált esetben, így a *mnesia_schema* esetén is, fennállt, hogy a *field_list*, *joker*, *arglist* típusú kifejezésekre végzett adatfolyam-elemzések nem hoznak eredményt. (Természetesen ezek a kifejezések a programszeletben sem szerepelnek.)

Kiindulva ezekből a statisztikákból, két megközelítés mentén szűrtem a kifejezéseket: az első esetben kiszűrtem az adatfolyam-elemzésre szánt kifejezések közül a *field_list*, *joker*, *arglist* kifejezéseket, a második esetben pedig csak a három leggyakoribb kifejezéstípust vettem figyelembe.

5.9. táblázat. A *mnesia_schema* összes, adatfüggőségeket behozó és a programszeletben felhasznált kifejezéseinek száma, típus szerint csoportosítva.

Kifejezés típusa	Összes	Adatfüggőséget behozó	Szelethez felhasznált
variable	23989	18709	10638
application	5729	3967	3064
atom	17777	3810	2226
cons	1922	1270	715
case_expr	810	665	450
tuple	3968	1994	1262
field_list	239	0	0
joker	565	0	0
arglist	7311	0	0

Az 5.10 táblázatban található a két szűrés eredményeinek összehasonlítása az eredeti algoritmus-sal. Az eredeti algoritmust a posztelemző modul nélkül vizsgáltam, annak érdekében, hogy a posztelemzés többletideje ne zavarja meg esetlegesen a végső eredményeket. Az olvashatóság érdekében Szűrés₁-gyel jelöltem az a verziót, amikor a kifejezések közül kiszűrtem a *field_list*, *joker*, *arglist* kifejezéseket, és ezen a szűrt halmazon végeztem el az adatfolyam-elemzést. Azt a verziót, amikor csak a *variable*, *application* és *atom* típusú kifejezéseket vettem figyelembe, Szűrés₂-vel jelöltem. A futásidőt percekben adtam meg, a *Tal.* oszlopokban pedig a végső függvényyszintű találatok számát adtam meg.

5.10. táblázat. Kifejezések szűrésének összehasonlítása.

Teszteset	Eredeti algoritmus		Szűrés ₁		Szűrés ₂	
	Futásidő	Tal.	Futásidő	Tal.	Futásidő	Tal.
mnesia_index	0.66	78	0.66	78	0.39	12
mnesia_text	13.05	1321	13.19	1321	9.23	1213
mnesia_controller	15.95	1241	12.94	1241	9.64	1134
mnesia_schema	17.59	1392	17.40	1392	13.18	1291

A táblázat eredményeiből észrevehető, hogy a Szűrés₁ esetén a találatszám teljes mértékben megegyezett az eredeti találatsszámmal, a futásidő pedig vagy nagyjából megegyezett az eredetivel, vagy kevesebb lett. A *mnesia_controller* esetén 3 percet spórolt futásidő tekintetében. Ebből az a következtetés vonható le, hogy bizonyos esetekben jelentős mértékben csökkenthető lehet a futásidő, úgy, hogy a találatszám nem változik.

A Szűrés₂ esetén a *mnesia_index* esetén drasztikus volt a találatcsökkenés, azonban a többi esetben az eredeti találatok minimum 90%-át sikerült felderítenünk, és a futásidő pedig minimum 29%-kal csökkent. Így, ha a függvényszintű hatáselemzés alkalmazásánál nem jelent nagy problémát a találatvesztés, ez a megoldás egy jó kompromisszum lehet a futásidőre nézve.

5.4.4. Függvényhívási láncok vizsgálata

Egy függvényhez, amely hatókörébe tartozik a szeletelési kritériumban meghatározott kifejezés, a RefactorErl akár egy másodpercen belül képes előállítani az előre felé menő és visszafelé-előrefelé menő függvényhívási láncokat. A programszeletet pedig a függvényhívási láncok alapján állítjuk elő. Ez adta annak az ötletét, hogy megvizsgáljam, hogy, amennyiben a programszeletnek csak a függvényhívási láncokon szereplő függvényeket tekintjük és adatfolyam-elemzést sem végzünk, akkor a találatok hogyan viszonyulnak az eredeti úton előállított programszelet találataihoz képest.

Az összehasonlítás eredménye az 5.11 táblázatban található. Az *Eredeti találatok* oszlopban az eredeti algoritmus találatainak számát, a *Függvényhívási lánc* oszlopban pedig a függvényhívási láncokon található függvények számát adtam meg. Az utolsó két oszlopban pedig ezen a halmazok különbségeinek elemszámát.

5.11. táblázat. Az eredeti algoritmus függvényszintű találatainak összehasonlítása a függvényhívási láncok elemeivel.

Teszt eset	Eredeti találatok = $ E $	Függvényhívási lánc = $ F $	$ E \setminus F $	$ F \setminus E $
<i>mnesia_index</i>	78	80	66	68
<i>mnesia_text</i>	1321	939	624	241
<i>mnesia_controller</i>	1241	903	570	233
<i>mnesia_schema</i>	1392	1146	523	277

Az utolsó két oszlopban megadott különbségekből látszik igazán, hogy ezek a halmazok nagymértékben eltérő eredményt adnak. Ennek az okai a következők:

1. A függvényhívási láncok elemei között szerepel az összes importált Erlang-modulból származó függvény is, tehát az olyan függvények, amelyek nem kerülnek betöltésre a szemantikus programgráfba, mert nem a vizsgált kód bázis részei (például az *is_tuple* vagy az *is_atom*).
2. Az, hogy egy függvény szerepel a függvényhívási láncban, még nem jelenti azt, hogy a kiinduló függvénnyel tényleges adatfüggőségi vagy vezérlésfüggőségi kapcsolatban áll a vizsgált kifejezés kontextusában. A függvényhívási láncokra vonatkozó információkat nem a végrehajtási utak függvényében számoljuk ki, ezért olyan esetek is bekerülnek, amely a valós futás alatt egyáltalán nem fordulhatnának elő.

3. A függvényhívási láncok esetén nem vesszük figyelembe a dinamikus függvényhívásokat, emiatt értékes találatokat veszítünk. Az adatfolyam-elemzés viszont képes felderíteni ezeket a függőségeket is.

Ha csak a függvényhívási láncokat tekintenénk, az hamis pozitív találatokat, illetve a valódi pozitív találatok figyelmen kívül hagyását eredményezné. Így a módszer nem használható a programszelet előállítására.

6. fejezet

Javított algoritmus alkalmazásai

Az előző fejezetekben bemutatott, javított algoritmusok több területen is alkalmazhatóak. A kifejezésszintre konfigurált, visszafelé menő szeletelést megvalósító algoritmus felhasználható többek között refaktorálásnál, debuggingnál, míg a függvényszinten vizsgált, előre felé menő szeletelés regressziós tesztselekciónál esetén. A lehetséges alkalmazások közül két esetben vizsgáltam meg a szeletelés rendeltetésszerű alkalmazását: tesztselekciónál és párhuzamosításnál.

6.1. Regressziós tesztelés

A futásidő rövidítését célzó intézkedések maguk után vonják a találatyszámok csökkenését. Így felmerülhet a kérdés, hogy valóban helyes és értelmes javításoknak tekinthetőek-e? Egy valós integráció során az esetleges hibák a lerövidült futásidőnek köszönhetően hamarabb derülhetnek ki, mintha a teljes tesztbázist újrafuttattuk volna. Ilyenkor viszont szükséges, hogy időnként a teljes tesztbázist végrehajtsuk, hogy a korábban le nem futott tesztek is újbóli végrehajtásra kerüljenek a teljes biztonság érdekében, például éjszakánként vagy heti rendszerességgel (nightly, weekly jobs).

A korábbi 2.4, 4.1 és 5.4.2 fejezetekben már érintettem a regressziós tesztelés témakörét és a programszeletelés alkalmazhatóságát. A programszeletelés gyakorlati alkalmazása tesztselekciónál ideális esetben a következőképpen nézne ki:

1. Frissítjük a RefactorErlbe betöltött fájlokat.
2. Azonosítjuk a pull requestben megváltozott kifejezéseket.
3. Ezekhez a kifejezésekhez lekérjük a függvényszintű programszeletet.
4. A programszeletben levő függvényekhez azonosítjuk a megfelelő teszteseteket.
5. Lefuttatjuk a kiválasztott teszteseteket.

6. Amennyiben a tesztek sikeresek, a pull request merge-ölhetővé válik a tesztek szempontjából. (Természetesen review-kra, vagy akár más típusú jobokra továbbra is szükség van.)

A programszeletbe eső függvények szemantikus programgráfbeli csúcsaihoz le tudjuk kérni a függvény modulját, nevét és aritását, a *replib_function:mod_fun_arity* függvényével. Ha egy függvény modulneve tartalmazza az Erlangban teszteseteket tartalmazó modulok elnevezési konvenciójának megfelelő *SUITE* utótagot (suffix) vagy a *test* szavat valamilyen formában, akkor a függvényt kiválasztjuk újrafuttatásra.

A tesztszelekciót kipróbáltam a *dq* [33], *GeoCouch* [34], *PropEr* [35], *Teles* [36], *Datum* [37] nevű alkalmazásokon, ezeket egy Erlang-kódbázisokat gyűjtő GitHub oldalon találtam [38].

A kísérletek eredményei a 6.1 táblázatban találhatóak. A futásidőt ezúttal másodpercben adtam meg. A *Találatok* oszlopban az szerepel, hogy hány tesztet választott ki a megfelelő függvényben bekövetkezett változás esetén újrafuttatásra, a *Tesztek* oszlopban pedig a forrásfájlokhoz tartozó összes tesztfüggvények száma található.

A *Geocouch* esetén a legtöbb tesztet nem Erlangban írták (hanem .t kiterjesztésű fájlokban), így ezeket nem képes betölteni és kezelni a RefactorErl. A *PropEr* esetén létező tesztek struktúrája nem rendszerezett, például a tesztek segédfüggvényei is a *test* mappán belül vannak (összesen 37 fájl), némelyik esetben van magyarázat a fájlhoz, máshol pedig nincs. Így egy becslést készítettem arra, hogy nagyságrendileg hány tényleges tesztfüggvény létezhet.

6.1. táblázat. Szeletelő algoritmus tesztszelekciója különböző repository-kon.

Repository	Függvény	Futásidő	Találatok	Tesztek
dq	dq:is_empty/1	0.12 másodperc	1	4
geocouch	wkb_writer:geojson_to_wkb/1	0.08 másodperc	2	3
proper	proper_fsm:run_commands/3	5.17 másodperc	1	210
proper	proper_typeserver:translate_type/1	6.40 másodperc	2	210
proper	proper:quickcheck/2	167.83 másodperc	52	210
teles	teles_data_manager:add_object/3	3.67 másodperc	5	14
teles	teles_handler:process_cmd/2	5.13 másodperc	8	14
datum	generic:generic_to/3	0.67 másodperc	0	285
datum	lens:get/2	0.80 másodperc	30	285
datum	lens:isob/3	0.85 másodperc	37	285
datum	generic:labelled_to/3	0.67 másodperc	2	285

Az eredmények azt mutatják, hogy a RefactorErlnek minden esetben sikerült kiválasztania az összes tesztek egy részhalmazát. Azoknál a repository-knál, ahol a minden teszt Erlang van megírva, az összes teszt legfeljebb 25%-át választotta ki. Ezeknél az alacsony-közepes komplexitású kódbázisoknál a javított algoritmus néhány másodpercen belül visszatért az eredménnyel, ami egyértelműen a használat mellett szól.

6.2. Párhuzamosítás

Az Erlang nyelvben széles eszköztár biztosítja a párhuzamos folyamatok támogatását. A párhuzamos és elosztott rendszerek kezelésének lehetősége egyértelműen megkülönbözteti a többi funkcionális nyelvtől [39]. A programszeletelés jól alkalmazható azokban az esetekben, amikor a program bizonyos részeit ki szeretnénk emelni egymással párhuzamosan futó folyamatokká, ugyanis képes megadni egy kifejezéshez az összes további programrészt, amely szükséges az értékének előállításához.

A régebben megírt (legacy) Erlangban írt programok jelentős hányadára jellemző, hogy nem oly módon tervezték őket, hogy a különböző részek egymással párhuzamosan futhassanak. Tóth Melinda és szerzőtársai korábban rávilágítottak arra, hogy a RefactorErlben létező függőségi kapcsolatokon keresztül lehetséges a párhuzamosítható komponensek azonosítása [40].

A szeletelő algoritmus finomhangolásához példaötleteket, hogy párhuzamosítás esetén is működjön, Li és Thompson [26] Wrangler nevű keretrendszeréhez [41] készített szeleteléssel kapcsolatos munkájából nyertem.

6.2.1. Szeletelés tulajdonságai

Amikor egy programrészt valamilyen párhuzamosan futtatható komponensbe szeretnénk kiszervezni, gyakran egy kifejezés értékének kiszámításához szükséges, összetettebb számítási műveletláncról beszélünk. A forráskód szintjén ez azt jelenti, hogy egy kifejezéshez meg kell kapnunk az összes olyan programkód-részletet, amelyeknek segítségével az értéke kiszámolható, tehát azokat az programrészeket, amelyek a kifejezés értékadása előtt kerülnek végrehajtásra és relevánsak rá nézve.

Ha a fent leírt esetben a programszeletelést választjuk, akkor a kifejezésre vonatkozó, kifejezés-szintű visszafelé menő programszeletet kell előállítanunk. A RefactorErl javított programszeletelő algoritmusát e célból kipróbáltam intraprocedurálisan és interprocedurálisan is, az összes tesztalanyként használt kifejezésre sikerült előállítanom a megfelelő programszeletet.

6.2.2. Végrehajtható programkód-részlet létrehozása

A párhuzamosításhoz szükségünk van arra, hogy a programszeletben megkapott kifejezésekből végrehajtható programkód-részleteket (executable) készítsünk.

A szemantikus programgráf reprezentációjának köszönhetően a hierarchikus kapcsolatok jelölve vannak a *top* éleken keresztül: egy részkifejezést reprezentáló SPG-csúcsból *top* címkével jelzett él megy az őt tartalmazó legkisebb top-level kifejezéshez. A top-level kifejezés azt jelenti, hogy közvetlen klóz alatt található. Mivel klózái nem csak a függvényeknek vannak, hanem például olyan összetett kifejezéseknek, mint a *case* kifejezéseknek, vagy éppen a lista értelmezők is. Így a korábban már bemutatott *top* élek nem elegendőek ahhoz, hogy az executable szintet elérjük. A 6.1 ábrán található a már korábban ismertetett példa [26] egy kiragadott részlete. Ha a programszeletbe belekerül például a `{_A, B}` kifejezés, akkor természetesen az egész 5. sorban

található értékadást szeretnénk a végrehajtható programrészletek közé helyezni. Azonban az SPG-reprezentációban a $\{_A, B\}$ kifejezésre vonatkozó *top* címkéjű él egy hurokél azért, mert a lista értelmező kifejezés generátora egy új, beágyazott klózba kerül, mivel a változó láthatóság új láthatósági kört nyit.

```
1 -module(top).
2
3 foo(Cols) ->
4   ...
5   R = [B || {_A, B} <- Cols].
```

6.1. ábra. Példa a *top* címkéjű élék problémájának bemutatásához.

Ezért a végrehajtható programszelet meghatározásához meg kell vizsgálnunk, hogy a kifejezés *top* kifejezése milyen klózhoz tartozik. Ha ez egy beágyazott klóz, akkor megkeresi a klózt tartalmazó *top-level* kifejezést. A többszörös beágyazás miatt mindezt rekurzívan teszi addig, amíg egy függvény klózához nem ér.

6.2.3. Intraprocedurális eset

Az intraprocedurális eset bemutatásához a példát Li és Thompson munkájából [26] kölcsönöztem, a forráskód a 6.2 ábrán található.

```
1 -module(image_proc).
2
3 readImage(FileName, FileName2) ->
4   {ok, #erl_image{format=F1, pixmaps=[PM1]}} = erl_img:load(FileName),
5   Cols1 = PM1#erl_pixmap.pixels,
6   {ok, #erl_image{format=F2, pixmaps=[PM2]}} = erl_img:load(FileName2),
7   Cols2 = PM2#erl_pixmap.pixels,
8   R1 = [B1 || {_A1, B1} <- Cols1],
9   R2 = [B2 || {_A2, B2} <- Cols2],
10  {R1, F1, R2, F2}.
```

6.2. ábra. Példa az intraprocedurális szeletelés bemutatásához párhuzamosítási célból.

A tegyük fel, hogy a párhuzamosítást a 10. sorban található *R1* kifejezés értékére vonatkozóan szeretnénk elvégezni. A szeletelő algoritmus előállítja a programszeletet, amely a 4., 5., 8. és 10. sorok kifejezéseit tartalmazza. A végrehajtható programrészleteket generáljuk a programszelet összes kifejezéséhez. Ez az azonban tartalmazni fogja a 10. sort is, amelyre nincs szükségünk. Ezt úgy oldottam meg, hogy, mivel a szeletelő algoritmus legelején meghatározzuk a kifejezéshez tartozó szemantikus programgráfbeli csúcsot, erre külön lekérem a végrehajtható programrészletet, amelyet kivonok az összes végrehajtható programkód-részletet tartalmazó listából. Így megkapjuk a párhuzamosításhoz szükséges programrészeknek megfelelő szemantikus programgráf-csúcsokat, amelyek a 6.3 ábrán programrészleteknek felelnek meg. (A szemantikus programgráfok csúcsai által reprezentált kódot szöveges formában a *refcore_syntax:flat_text* függvényével kértem le.)

```
"      {ok, #erl_image{format=F1, pixmaps=[PM1]}} = erl_img:load(FileName) "  
"      Cols1 = PM1#erl_pixmap.pixels "  
"      Cols1 = PM1#erl_pixmap.pixels "
```

6.3. ábra. Példa intraprocedurális párhuzamosítható szeletre.

6.2.4. Interprocedurális eset

Az interprocedurális eset eredményei kísérleti jellegűek. A 6.4 ábrán található egy különböző statisztikák [42] kiszámolására képes modul, és a hozzá tartozó függvényeket tartalmazó forrásfájl.

Tegyük fel, hogy a 10. sorban található `LRPos` értékének számítását szeretnénk párhuzamosítani. A szeletelő algoritmus megtalálja a megfelelő kifejezéseket, amelyekből végrehajtható programrészleteket állít elő, az ezekhez tartozó konkrét kódsorok szöveges formátumban a 6.5 ábrán találhatóak.

Látható, hogy minden szükséges kifejezéshez tartozó végrehajtható programkód-részletet sikerült előállítanunk, de ennek az alkalmazhatósága további kérdéseket vet fel, például:

- o Milyen formátumban szeretnénk elvégezni a párhuzamosítást, hogyha a kifejezés értékét egy függvény visszatérési értéke adja meg?
- o Szükséges-e egy olyan változó értékadásának a kiszervezése egy különálló folyamatba, amelyet nem tudunk „tisztán” kiszervezni, mert más komponensek is használják? A példában ez az 5. sorban található *Rates* változó, amely nem csak a 10., hanem a 11. sor függvényhívásának is az argumentuma.
- o Hogyan kezeljük a párhuzamosítást az ilyen függvényeken átívelő esetekben? Ebben a példában elég lenne csak a függvényhívásokat kiszerveznünk, de mi történik akkor, ha a függvényeknek vannak olyan részei is, amelyek nem relevánsak a kezdőkifejezésünkre nézve?

```

1 -module(dor).
2 -export([init_rates/0]).
3
4 init_rates() ->
5     Rates = #{ "TP" => 25, "FP" => 10, "TN"=> 3, "FN" => 5},
6     diagnostic_odd_ratio(Rates).
7
8 diagnostic_odd_ratio(Rates) ->
9     LRpos = positive_likelihood_ratio(Rates),
10    LRneg = negative_likelihood_ratio(Rates),
11    LRpos / LRneg.
12
13 positive_likelihood_ratio(Rates) ->
14     TPR = true_positive_rate(Rates),
15     FPR = false_positive_rate(Rates),
16     TPR / FPR.
17
18 negative_likelihood_ratio(Rates) ->
19     FNR = false_negative_rate(Rates),
20     TNR = true_negative_rate(Rates),
21     FNR / TNR.
22
23 true_positive_rate(Rates) ->
24     TP = maps:get("TP", Rates),
25     FN = maps:get("FN", Rates),
26     TP / (TP + FN).
27
28 false_positive_rate(Rates) ->
29     FP = maps:get("FP", Rates),
30     TN = maps:get("TN", Rates),
31     FP / (FP + TN).
32
33 false_negative_rate(Rates) ->
34     FN = maps:get("FN", Rates),
35     TP = maps:get("TP", Rates),
36     FN / (FN + TP).
37
38 true_negative_rate(Rates) ->
39     TN = maps:get("TN", Rates),
40     FP = maps:get("FP", Rates),
41     TN / (TN + FP).

```

6.4. ábra. Példa az interprocedurális szeletelés bemutatásához párhuzamosítási célból.

```

"    Rates = #{ "TP" => 25, "FP" => 10, "TN"=> 3, "FN" => 5}"
"    diagnostic_odd_ratio(Rates)"
"    LRpos = positive_likelihood_ratio(Rates)"
"    TPR = true_positive_rate(Rates)"
"    FPR = false_positive_rate(Rates)"
"    TPR / FPR"
"    TP = maps:get("TP", Rates)"
"    FN = maps:get("FN", Rates)"
"    TP / (TP + FN)"
"    FP = maps:get("FP", Rates)"
"    TN = maps:get("TN", Rates)"
"    FP / (FP + TN)"

```

6.5. ábra. Példa intraprocedurális párhuzamosíthatás céljából *felhasználható* szeletre.

7. fejezet

Kapcsolódó munkák

A programszeletelés a fogalom bevezetése óta előszeretettel kutatott tématerület, széles szakirodalommal. A meglévő publikációk javarészt az adatfüggőségek feltérképezésének problémájára kínálnak elméleti algoritmusokat. Ebben a fejezetben olyan szeletelő keretrendszereket és eszközöket mutatok be, amelyek a gyakorlatban is megvalósításra kerültek, többségük imperatív programozási nyelvekhez készült.

7.1. Frama-C

A Frama-C [43] egy olyan nyílt forráskódú, grafikus felhasználói felülettel rendelkező keretrendszer, amely többféle statikus elemzésre biztosít lehetőséget C és C++ nyelven írt programok esetén. A keretrendszer tartalmaz többek között olyan funkcionálisokat, amelyek képesek a program egy változójának lehetséges értékeit kiszámolni (evolved value analysis). Egy további érdekes funkcionálitása, hogy képes ellenőrizni, hogy a *ANSI/ISO C Specification Language* [44] annotációkkal leírt viselkedési követelmények teljesülnek-e a program minden lehetséges végrehajtása esetén, például az előfeltételre, utófeltételre, invariánsra és variánsfüggvényre vonatkozó megkötések. A Frama-C tartalmaz egy *Impact* és egy *Slicing* nevű komponenst is.

A felhasználó felületen kijelölt kódrészlethez kérhetjük le a hozzá tartozó hatáselemzést, ennek a visszatérési értéke a program utasításainak részhalmaza. Az eredmény halmazban nem szereplő elemekre nézve garantált, hogy azok biztosan nem tartoznak a kijelölt kódsorok hatókörébe. A programszeletet egy felhasználó által megadott szeletelési kritériumhoz határozzák meg. A programszeletről biztosított, hogy önállóan fordítható (compilable) C-kód, és ugyanazt a viselkedést produkálja a szeletelési kritériumra nézve, mint az teljes eredeti program.

A hatáselemzés és a programszeletelés is program függőségi gráfját (Program Dependency Graph, röviden PDG) használja [45]. A PDG interprocedurális, tartalmazza a vezérlés- és adatfüggőségeket is. Egy s_2 programbeli utasítást adatfüggőnek tekintenek egy s_1 programutasítástól, ha az s_2 utasításban levő számításához szükséges egy az s_1 utasítás által írt érték. Hasonlóan, két utasítás között akkor van vezérlési függőség, ha az egyik végrehajtása függ a másik utasítás

végrehajtásától.

A cikk értelmezése szerint, a programszeletelés és a hatáselemzés között a különbség a PDG bejárásában van. A hatáselemzés esetén az eredeti utasításból kiindulva azokat az utasításokat keresi, amelyek kiértékeléséhez szükségesek a már megtalált utasítások. A programszeletelés esetén pedig visszafelé halad, és azokat az utasításokat keresi, amelyek befolyásolják a kezdőutasítást. (Az *impact* az előre felé menő szeletelés, a *slicing* pedig a visszafelé menő utasításszintű programszeletelésnek feleltethető meg a dolgozatomban használt fogalmak szintjén.)

7.2. Indus

Az Indus [46] egy olyan keretrendszer, amely Java-programokhoz írt különböző statikus elemző módszereket és kódtranszformációs eszközöket tartalmaz. A statikus elemzéseket tartalmazó modul olyan elemzéseket tartalmaz, mint például az objektumok folyamának elemzése (Object Flow Analysis), különböző függőségi elemzések (Dependence Analyses) és mellékhatás-elemzés (Side-Effect Analysis).

A programszeletelést végző modul a statikus elemzések közül a függőségi elemzések, illetve az objektumfolyam-elemzés eredményeire támaszkodik. Képes előállítani egy szeletelési kritériumhoz tartozó vissza- és előre felé menő szeletet is. Létezik támogatás különböző szeleteléssel kapcsolatos alkalmatosságokhoz, mint például a programszelet végrehajtható utasításokká való alakítása vagy a programszelet hatókörének az alkalmazás egy adott rétegére való szűkítése. Az Indus programszeletelő kapcsán létrehoztak egy *Kaveri* nevű Eclipse-bővítményt is, ami a programszeletelés felhasználói felületeként szolgál.

Az Indus programszeletelőjét eredetileg a konkurens Java programokon végzett modell-ellenőrzés (model checking) költségeinek redukálásáért hozták létre [47]. A model checking a program összes lehetséges végrehajtásának szimulációját, és ezek helyességének ellenőrzését jelenti valamilyen előre meghatározott kikötéssel szemben. Több különböző konkurens programon hajtottak végre kísérleteket, és ezek során a szeletelő algoritmusuk egyik speciális verziója képes volt arra, hogy a kódbázisokat utasításszinten minimum 77%-kal redukálja 3 perc alatti futásidő mellett.

7.3. Wrangler

A Wrangler [41] egy Erlang-forrásfájlokhoz készült, grafikus felhasználói felülettel rendelkező kódrefaktoráló keretrendszer. Az eszköz segítségével különböző, kisebb-nagyobb refaktorálásokat lehet végrehajtani, kezdve a változók átnevezésétől egészen a kód egyes részeire vonatkozó párhuzamosítás bevezetéséig. A Wrangler egy forrásfájl esetén a következő fő folyamatokat hajtja végre: létrehozza az absztrakt szintaxisfát (AST), majd szemantikus elemzéseket végez a fájlban, ezeknek eredményével kiegészíti az AST-t, ez az Annotated Abstract Syntax Tree (röviden AAST), végezetül pedig transzformációkat végez az AAST-n.

Li és Thompson [26] azt vizsgálták, hogy melyek azok a tipikus esetek, amelyeknél kézenfekvő az, hogy utólag párhuzamosítsuk a forráskód valamely komponenseit. Ez fontos szereppel bír az

régebben íródott (legacy) programok hatékonyabbá tételének vizsgálatánál. A párhuzamosítás utólagos bevezetésének három indokolt fajtáját azonosították:

- A *gen_server handle_call* függvénye esetén speciális körülmények között.
- Ha a vizsgált függvény egy részét szeretnénk kiemelni.
- Farok-rekurzív (tail-recursive) függvények esetében.

Mindegyik fenti esetre sikerült megadniuk a megfelelő refaktorálásokat. Azt, hogy melyek azok az utasítások, amelyeket szükséges kiemelni a párhuzamosan futó folyamatba, programszeletelés segítségével határozták meg.

Egy változó alapján meghatározott, intraprocedurális, visszafelé menő programszelet tartalmazza azokat az utasításokat, amelyek szükségesek a változó értékének kiszámításához, ha csak változót tartalmazó függvény hatáskörét vizsgáljuk. A Wrangler támogatja az effajta programszeletelést. A programszeletet az AAST mentén állították elő. Az AAST tartalmazta az összes olyan információt, amely szükséges volt a programszeleteléshez, és a párhuzamosítást megadó refaktoráló funkció implementációjában.

7.4. Static Execute After

A Static Execute After (röviden SEA) módszerét Beszédes Árpád és szerzőtársai [48] adták meg egy dinamikus hatáselemzéshez megalkotott hatáselemző módszer [49] statikus párjaként. A módszer a hatáselemzés egy olyan eszköze, amely egy hatékonyabb alternatíva a magas számításigényű statikus programszeletelésre. A programszeletelés függvény szintű eredményeit magas pontosság és szenzitivitás mellett képes reprodukálni.

A SEA függvények közötti relációk halmazát adja meg. Egy f és g függvényre $(f, g) \in SEA$, ha a g függvény bármely része végrehajtható, miután f valamelyik részét végrehajtottuk [30]. A módszer nagyszámú hamis pozitív találatot eredményez. Az alkalmazásához a programot egy úgynevezett *ICCFG* (Interprocedural Component Control Flow Graph) reprezentációban kell megadni, amely az intraprocedurális vezérlésfolyam-gráfok egy függvényhívási információkkal való kiterjesztésének tekinthető. Ennek az ábrázolásnak köszönhetően képesek vagyunk megadni a program *SEA* relációit. A hatáselemzést a *SEA* relációk segítségével adja meg.

Megjegyzés.

$$\text{pontosság (precision)} = \frac{\text{valódi pozitív}}{\text{valódi pozitív} + \text{hamis pozitív}} \text{ esetek száma}$$

$$\text{szenzitivitás (recall)} = \frac{\text{valódi pozitív}}{\text{valódi pozitív} + \text{hamis negatív}} \text{ esetek száma}$$

Az esettanulmány során *SEA*-t a statikus programszeleteléssel hasonlították össze Javában implementált programok esetén. A *SEA* a programszeleteléshez viszonyítva a három vizsgált program

esetén átlagosan 85%-os pontosságra, és több, mint 99%-os szenzitivitásra volt képes. Ebből azt a következtetést vonták le, hogy függvényyszintű hatáselemzés esetén *előnyben részesíthető* a SEA a programszeleteléssel szemben, abból adódóan, hogy a függvények közötti kapcsolatok leírása egy hatékonyabb módja az adatfolyamok felderítésének.

A szerzők szerint a SEA képes felfedezni az adatfüggőségeket számos fajtáját: például a mutatókat (pointer) és a dinamikus memóriakezelést is. Ugyanakkor felvetették a kérdést, hogy valóban felfedez-e *minden* lehetséges adatfüggőségi kapcsolatot, például az olyan esetekben, amelyekkel esetlegesen nem találkoztak a kísérleteik során.

7.5. Definíció-használat alapú szeletelés

A definíció-használat (definition-use, röviden $d: U$) alapú dinamikus szeletelést a kifejezés szintű statikus programszeletelés hatékonyabb alternatívájaként dolgozták ki Gyimóthy Tibor és szerzőtársai [20]. Az algoritmus az adatfüggőségi kapcsolatok és gráfok helyett minden programutasításra létrehozza a definíció-használat (definition-use) reprezentációt, majd pedig a végrehajtási történet elemeihez rendeli a megfelelő sorhoz tartozó, változókra vonatkozó definíció-használat információt. A futásidőben előállt végrehajtási történet segítségével lehetséges egy pontosabb programszelet előállítása a statikus módszerhez képest. A $d: U$ reprezentációt használó szeletelésekre később több különböző algoritmust adtak meg, például a különböző szeletelési irányok mentén [31].

A $d: U$ reprezentációt használó algoritmusok teljesítményét Gergely Tamás [30] értékelte ki, aki a kísérletei során arra a következtetésre jutott, hogy a statikus programszeleteléssel előálló szelet sokkal több utasítást tartalmaz, mint amennyi ténylegesen szükséges. Ezért a statikus szeletelést különböző bemeneti adatokra meghatározott dinamikusan előálló szeletek uniójával váltotta ki. A C és Java nyelven írt programok esetén a $d: U$ alapú algoritmus alternatívája lehet az adatfüggőségi gráfokon végzett statikus szeletelésnek. Futásidő szempontjából nem sikerült belátnia, hogy ez a megközelítés egyértelműen jobban teljesítene az adatfolyamra vonatkozó információkat használó statikus szeletelőknél.

8. fejezet

Összefoglaló

A hatáselemzés és a programszeletelés a szoftverfejlesztés számos tevékenysége során támogathatja a fejlesztők mindennapi munkáját, a gyakorlati hasznuk vitathatatlan. A fogalom bevezetése óta számos publikáció jelent meg a témakörben, azonban elenyésző azok olyan keretrendszereknek a száma, amely racionális futásidő és magas precizitás mellett képesek szeletelésre, sőt, a létező eszközök többségénél nem is áll rendelkezésre effajta információ.

A statikus programszeletelésről, hasonlóan egyéb, összetett statikus elemzésekhez, általánosságban elmondható, hogy közepes-magas komplexitású kódbázisok esetén lassú és pontatlan is egyben. A legbonyolultabb problémát az adatfüggőségek feltérképezése jelenti, de nem egyértelmű a vezérlési függőségek meghatározása sem. A kutatásom során több heurisztikát is megadtam, valamint kiértékeltem ezeknek az eredményeit több szempont mentén: futásidő, találatok száma és találatok jellege (valódi pozitív, hamis pozitív).

A dolgozatomban többdimenziós optimalizálást végeztem. Redukáltam a szeletelő algoritmus futási idejét úgy, hogy ne keletkezzen jelentős veszteség a találatok számában, csökkentettem a hamis pozitív találatok számát úgy, hogy ez ne menjen jelentősen a futási idő kárára, valamint növeltem a valódi pozitív találatok számát.

A függvény szintű eredmények esetén, megvizsgáltam az adatfolyam-elemzésre vonatkozó felső időkorlátokat, és egy olyan korlátot választottam ki, amelyet a kísérleteim igazoltak. Ellenőriztem a meglévő adatfolyam-elemző algoritmusok beépíthetőségét, és megfogalmaztam a szükséges fejlesztési igényt az adatfolyam-elemző algoritmusra nézve. A meglévő adatfolyam-elemzést párhuzamosítottam, és megadtam egy olyan posztemelzést végző modult, amely képes kiszűrni a hamis pozitív találatok többségét. Megvizsgáltam további, kombinált megoldásokat is, például az elemzés hatókörének szűkítését, valamint az adatfolyam-elemzéssel vizsgált kifejezések előszűrését - ezek egyaránt csökkenthetik a hamis pozitív találatokat és a futásidőt is. A javítás bizonyos lépései találatvesztést eredményezhetnek, azonban a szeletelő algoritmus valós integrációja esetén az esetleges negatív mellékhatások teljes mértékben kiküszöbölhetőek.

A szeletelő algoritmust konfiguráltam arra, hogy képes legyen előállítani egy növelt precizítású, programszeletet a kifejezések szintjén. A kifejezésszintű programszelet pontosságának érdekében

új adatfüggőségi kapcsolatokat vezettem be, illetve limitáltam a figyelembevett vezérlésfolyam-beli függőségeket. Sikertült megadnom egy algoritmust, amely a kifejezések alapján képes előállítani a hozzájuk tartozó programkód-részletet. A kifejezésszintű szeletelés esetén egyaránt tanulmányoztam az intraprocedurális és interprocedurális esetet is.

A megoldásaim jelenleg még csak a RefactorErl privát repository-jában érhetőek el, a közeljövőben belekerülnek a keretrendszer legújabb release-elts verziójába.

8.1. Továbbfejlesztési lehetőségek

A leginkább szükséges következő fejlesztési lépés egy olyan elsőrendű adatfolyam-elemző algoritmus megadása, amely képes egy időkeret alatt keletkezett eredmények listáját visszaadni. Ennek segítségével a jelenleginél sokkal rövidebb idő alatt képesek lennénk meghatározni a függvény-szintű programszeletet, amelyet regressziós tesztselekciónál alkalmazhatunk.

Szükséges megvizsgálni, hogy az adatfolyam-elemzésre adott felső időkorlátok esetén az elvesztett találatok milyen jellegűek, azaz hamis pozitív találatokat veszítünk-e el vagy valódi pozitívakat.

A függvény-szintű eredményeket szükséges tovább pontosítani. Az igyekezeteim ellenére, feltehetőleg továbbra is szerepelnek hamis pozitív találatok a szeletben. A dolgozatomban nem foglalkoztam a hamis negatív esetekkel. Fontos lenne kideríteni, hogy melyek azok a konstrukciók vagy speciális függvényhívási módszerek, amelyek esetén elvesztjük vagy nem érjük el az adatfolyam-elemzésen keresztül az olyan találatokat, amelyeknek szerepelniük kellene az függvény-szintű programszeletben eredményként.

Bár a dinamikus szeletelés jelenleg a végrehajtási történet hiányában nem építhető be, érdemes lehet a kifejezésszintű programszeletelés esetén megvizsgálni a $d: U$ reprezentáció alkalmazhatóságát, esetleg valamilyen kombinációját a meglévő adatfolyam-elemző algoritmussal.

A kifejezések szintjén, mivel sikerült megadni a megfelelő intraprocedurális szeletet, a következő lépés egy refaktorálás megadása lehetne, amellyel a függvény egy megfelelő része kiemelhető egy párhuzamosan futó folyamatba.

Irodalomjegyzék

- [1] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, 23(8):613–646, 2013.
- [2] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, page 439–449. IEEE Press, 1981.
- [3] I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Köszegi, Tejfel. M., and M Tóth. Refactorerl - source code analysis and refactoring in erlang. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools*, ISBN 978-9949-23-178-2, pages 138–148, Tallin, Estonia, October 2011.
- [4] Eötvös Loránd Tudományegyetem. RefactorErl. <http://pnyf.inf.elte.hu/trac/refactorerl>. [Hozzáférés dátuma: 2023.04.29].
- [5] Ericsson AB. Erlang Programming Language. <http://www.erlang.org>. [Hozzáférés dátuma: 2023.04.29].
- [6] M. Tóth and I. Bozó. Static analysis of complex software systems implemented in erlang. Central European Functional Programming Summer School – Fourth Summer School, CEFPS 2011, Revisited Selected Lectures, Lecture Notes in Computer Science (LNCS), Vol. 7241, pp. 451-514, Springer-Verlag, ISSN: 0302-9743, 2012.
- [7] Viktória Fördös, István Bozó, and Melinda Tóth. Towards change-driven testing. In *Proceedings of the 16th ACM SIGPLAN International Workshop on Erlang*, Erlang 2017, page 64–65, New York, NY, USA, 2017. Association for Computing Machinery.
- [8] ELTE IK-Programozási Nyelvek és Fordítóprogramok Tanszék. Az Erlang programozási nyelv. <http://nyelvek.inf.elte.hu/leirasok/Erlang>. [Hozzáférés dátuma: 2023.04.29].
- [9] F. Cesarini. Which companies are using Erlang, and why? <https://www.erlang-solutions.com/blog/which-companies-are-using-erlang-and-why-mytopdogstatus>. [Hozzáférés dátuma: 2023.04.29].
- [10] Ericsson AB. System Architecture. https://www.erlang.org/doc/system_architecture_intro/sys_arch_intro.html. [Hozzáférés dátuma: 2023.04.29].
- [11] Ericsson AB. OTP Design Principles. https://www.erlang.org/doc/design_principles/des Princ.html. [Hozzáférés dátuma: 2023.04.29].

- [12] Melinda Tóth and István Bozó. Static analysis of complex software systems implemented in erlang. In *Central European Functional Programming School: 4th Summer School, CEFP 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*, pages 440–498. Springer Berlin, Heidelberg, 2012.
- [13] M. Tóth, I. Bozó, J. Kőszegi, and Z. Horváth. Static analysis based support for program comprehension in erlang,. In *Acta Electrotechnica et Informatica*, Volume 11, Number 03, October 2011. Publisher: Versita, Warsaw, ISSN 1335-8243 (print), ISSN 1338-3957 (online), pages 3-10.
- [14] Geeks for Geeks. Data flow analysis in Compiler. <https://www.geeksforgeeks.org/data-flow-analysis-compiler>. [Hozzáférés dátuma: 2023.04.30].
- [15] Flemming Nielson, Hanne Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Berlin, Heidelberg, 1999.
- [16] N.F. Schneidewind. The state of software maintenance. *IEEE Transactions on Software Engineering*, SE-13(3):303–310, 1987.
- [17] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208, 2001.
- [18] Mark David Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, University of Michigan, USA, 1979. AAI8007856.
- [19] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, mar 2005.
- [20] Tibor Gyimóthy, Árpád Beszédes, and István Forgács. An efficient relevant slicing method for debugging. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-7*, page 303–321, Berlin, Heidelberg, 1999. Springer-Verlag.
- [21] István Bozó, Melinda Tóth, and Zoltán Horváth. Reduction of regression tests for Erlang based on impact analysis. In *Studia Universitatis Babeş-Bolyai Informatica*, Volume LIX, Special Issue 1, 2014. 10th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, May 21-25, 2014.
- [22] Michael Truog. Erlang Priority Queue Implementation. <https://github.com/okepqueueuday/pqueue>. [Hozzáférés dátuma: 2023.05.03].
- [23] Dmitry Melnikov. Erlang Localtime. https://github.com/dmitryme/erlang_localtime. [Hozzáférés dátuma: 2023.05.03].
- [24] Ericsson AB. Erlang – mnesia. <https://www.erlang.org/doc/man/mnesia.html>. [Hozzáférés dátuma: 2023.05. 03].
- [25] Facebook Inc. WhatsApp/erlfmt: An automated code formatter for Erlang. <https://github.com/WhatsApp/erlfmt>. [Hozzáférés dátuma: 2023.05. 03].
- [26] Huiqing Li and Simon Thompson. Safe concurrency introduction through slicing. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation, PEPM '15*, page 103–113, New York, NY, USA, 2015. Association for Computing Machinery.

- [27] Ericsson AB. Erlang – gen_server. https://www.erlang.org/doc/man/gen_server.html. [Hozzáférés dátuma: 2023.05.06].
- [28] Ericsson AB. Erlang – dets. <https://www.erlang.org/doc/man/dets.html>. [Hozzáférés dátuma: 2023.05.07].
- [29] Banka Tamás. RefactorErl adatfolyam elemző algoritmusának továbbfejlesztése. BSc Szakdolgozat, Eötvös Loránd Tudományegyetem, 2022.
- [30] Gergely Tamás. *Programok Statikus és Dinamikus Analízise*. PhD thesis, Szegedi Tudományegyetem, Informatika Doktori Iskola, Szeged, 2010.
- [31] Árpád Beszédes, Tamás Gergely, and Tibor Gyimóthy. Graph-less dynamic dependence-based dynamic slicing algorithms. In *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 21 – 30, 10 2006.
- [32] Ericsson AB. Erlang – Common Test Basics. https://www.erlang.org/doc/apps/common_test/basics_chapter.html. [Hozzáférés dátuma: 2023.05.15].
- [33] Darach Ennis. Distributed Fault Tolerant Queue Library. <https://github.com/darach/dq>. [Hozzáférés dátuma: 2023.05.15].
- [34] Cloudant. GeoCouch, a spatial index for CouchDB. <https://github.com/couchbase/geocouch>. [Hozzáférés dátuma: 2023.05.15].
- [35] Manolis Papadakis, Eirini Arvaniti, and Kostis Sagonas. PropEr: a QuickCheck-inspired property-based testing tool for Erlang. <https://github.com/proper-testing/proper>. [Hozzáférés dátuma: 2023.05.15].
- [36] Armon Dadgar. An Erlang network service for manipulating geographic data. <https://github.com/armon/teles>. [Hozzáférés dátuma: 2023.05.15].
- [37] Dmitry Kolesnikov. pure functional and generic programming for Erlang. <https://github.com/fogfish/datum>. [Hozzáférés dátuma: 2023.05.15].
- [38] David Robakowski. A curated list of awesome Erlang libraries, resources and shiny things. <https://github.com/drobakowski/awesome-erlang>. [Hozzáférés dátuma: 2023.05.15].
- [39] Ericsson AB. Concurrent Programming. https://www.erlang.org/doc/getting_started/conc_prog.html. [Hozzáférés dátuma: 2023.05.09].
- [40] M. Tóth, I. Bozó, Z. Horváth, and A. Erdödi. Static analysis and refactoring towards erlang multicore programming. In *Pre-proceedings of the Fourth Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES'11*, Saarbrücken, Germany, April 2011.
- [41] Huiqing Li and Simon Thompson. Wrangler - an Erlang Refactoring Tool. <http://refactoringtools.github.io/wrangler/overview-summary.html>. [Hozzáférés dátuma: 2023.05.10].
- [42] Wikimedia Foundation Inc. Sensitivity and specificity. https://en.wikipedia.org/wiki/Sensitivity_and_specificity#Confusion_matrix. [Hozzáférés dátuma: 2023.05.10].
- [43] FRAMA-C. Frama-C - Framework for Modular Analysis of C programs. <https://www.frama-c.com>. [Hozzáférés dátuma: 2023.05.14].

- [44] FRAMA-C. ACSL - Frama-C. <https://www.frama-c.com/html/acsl.html>. [Hozzáférés dátuma: 2023.05.14].
- [45] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, May 2015.
- [46] Kansas State University. About Indus. <https://indus.projects.cs.ksu.edu>. [Hozzáférés dátuma: 2023.05.14].
- [47] Venkatesh Prasad Ranganath and John Hatcliff. Slicing concurrent java programs using indus and kaveri. *International Journal on Software Tools for Technology Transfer*, 9(5):489–504, Oct 2007.
- [48] 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 *2007 IEEE International Conference on Software Maintenance*, pages 295 – 304, 11 2007.
- [49] T. Apiwattanapong, A. Orso, and M.J. Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 432–441, 2005.