View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ELTE Digital Institutional Repository (EDIT)

Eötvös Loránd Tudományegyetem
Informatikai Kar
Programozási Nyelvek és
Fordítóprogramok Tanszék

Szimbolikus végrehajtás állapotautomaták segítségével

Pataki Norbert
Adjunktus

Fülöp Endre
Programtervező Informatikus MSc

Budapest, 2019

Tartalomjegyzék

1. Bevezetés **4**

2. Hibakeresés **7**

2.1. Hibakeresés absztrakt reprezentációkban 7

2.2. Forráselemzés 8

2.2.1. Szintaktikai elemzés 9

2.2.2. Egyszerű szemantikai elemzés 9

2.2.3. Összetett szemantikai elemzés 9

2.2.4. Futásidejű hibakeresés 13

3. Statikus elemzések Clang-gel **16**

3.1. LLVM 16

3.2. Clang 18

3.3. Statikus analízis 25

3.3.1. Szöveges reprezentáció 26

3.3.2. Absztrakt Szintaxisfa 27

3.3.3. Program flow 28

3.3.4. Program path 29

4. Elemzés automatákkal **31**

4.1. Erőforrás-jellegű problémák 31

4.1.1. Fájlkezelés 32

4.1.2. Dinamikus memóriakezelés 32

4.2. Erőforrások általános modellje 34

4.2.1. Kiterjesztések 35

4.3. Detektálási problémák állapotgép-nézete 37

4.3.1. Checkerek megvalósítása hagyományosan 37

4.3.2. Checkerek megvalósítása állapot-orientált módon 39

4.4. Statikus analízis állapotgép-leíró nyelve 39

4.4.1. Szintaxis elemzése 40

4.4.2. Szemantika 44
4.4.3. Parsolás PEG-ként 45

5. Összefoglalás 48

Köszönetnyilvánítás

Köszönettel tartozom a témavezetőmnek, Pataki Norbertnek, aki lelkiismeretes munkájával felügyelte a dolgozat tartalmi és formai alakulását. Köszönöm a lehetőséget az Ericsson csapatának, többek között Krupp Dánielnek, Horváth Gábornak és Gera Zoltánnak, a velük töltött munka inspiráló volt számomra. Nem utolsó sorban köszönöm a családomnak, hogy türelemmel viseltettek irányomban a tanulmányaim és a dolgozat írása alatt.

1. fejezet

Bevezetés

A szoftvertechnológia területén a forráskód nemkívánatos tulajdonságainak felfedezésére és ezek vizsgálatára egyik lehetséges eszköz a statikus analízis. Néhány esetben ezek a hibák közvetlenül a programozási nyelvi megoldásokhoz köthetőek, és ekkor lehetőség van a konkrét domain ismerete nélkül lehet ezekre megoldásokat adni. Más hibák nem ennyire általános jellegűek, tehát ezen hibák felfedezése olyan megoldást igényel, amely megalkotása feltételezi a domain ismeretét. Annak érdekében, hogy a fejlesztő egy ilyen megoldást megalkosson, ismernie kell a domain-t, de emellett szerteágazó ismeretekkel kell, hogy rendelkezzen a használt statikus analízis keretrendszer architektúrájával, programozási felületével (API), és realisztikusan az implementációs részleteivel is. Ezen követelmények gátló akadályai lehetnek azon fejlesztők számára, akik saját szakterületükön szeretnének statikus analízis módszerekkel élni. Ezen helyzet megoldásaként adok a dolgozatban egy javaslatot, mely alacsonyabbá teheti a statikus analízis eszközök fejlesztésébe való belépés küszöbét. Ez a javaslat a statikus analízis probléma egy új megközelítése, mely egyben egy domain-specifikus-nyelvvel (DSL) — mint leíró eszközzel — kiegészített komplex megoldás. Ezen megoldás segítségével el lehet vonatkoztatni a hibaesetek detektálásakor a statikus analízis keretrendszer konkrét részleteitől. Állapotautomaták koncepcionális modelljével lehet leírni a szoftveres hibák felderítését, és az imperatív vagy funkcionális paradigma helyett állapotok és átmenetek leírásával. Ez a leírás az erőforrás-jellegű hibákhoz illeszkedik legjobban, de a nyelv tervezése során a kiterjeszthetőség és rugalmasság is fontos szempont, így lehetővé téve más jellegű problémák tömör, lényegretörő leírását is. Emellett a már meglévő megoldások hatékony integrációja is hangsúlyt kap, olyan formában, hogy a szintaktikai, illetve szemantikai ellenőrzések külön-külön már létező, vagy későbbiekben kifejlesztett megoldásai illeszthetők lehetnek.

A szoftverkörnyezetek nagyon változatosak, és mindnek megvannak a megoldandó problémái, illetve az ezek megoldásához természetesen illeszkedő programozási nyelvek. Ebből fakadóan a fejlesztési és futási követelmények, illetve az ezeket támogató módszerek tekintetében is nagy eltérések tapasztalhatóak.

A szoftverfejlesztés során egy, a specifikációban körvonalazott problémára keressük a megoldást valamilyen információs rendszer megfelelő megtervezésével és megvalósításával. A problémához illeszkedő elvi megoldás megtalálása csak egy korai lépés a teljes folyamatot tekintve. Az ezt követő megvalósítás során pedig jellemzően a kiválasztott programozási nyelven implementálásra kerülnek a megoldás algoritmusai. Ez a lépés már magában nagy többletkomplexitást adhat sok projekthez. Ennek okai többek között lehetnek:

- modularizálás az újrafelhasználhatóság növelése érdekében
- a megfelelő külső könyvtárak elérhetővé tétele és verziókezelése
- egyéb elő- és utófeldolgozást végző rendszerek
  - fordítás és forrástranszformáció
  - csomagolás, függőségek feloldása
  - kihelyezés, eljuttatás a futási környezetbe (deployment)

Ilyen körülmények között érdemes lehet automatizált eszközöket bevetni annak biztosítására, hogy a szoftver helyes működését ellenőrizni tudjuk.

Diplomamunkámban egy speciális, potenciálisan hasznos eredményeket produkáló szeletét vizsgálom az említett eszköztárnak, a tesztelési fázis során futó, szoftverhelyességet vizsgáló hibakereső eszközöket. Ezen belül bemutatom a fordításidőben történő hibakeresés jelenleg egyik legfejlettebb eszközét, a statikus analízist, és a Clang eszközzel végzett hibakeresési módszerek kiterjeszthető megvalósítását. Ezen megoldás képessé teszi a felhasználót arra, hogy saját szoftveres megoldásaiban egyedi detektáló logikát valósítson meg egy DSL segítségével, így paraméterezhetővé válik a detektálást végző szoftveres megoldás.

A diplomamunka a C nyelvcsaládbeli példák alapján mutatja be a statikus analízist, a megvalósított ellenőrző program is C, illetve C++ nyelveken végez ellenőrzést, illetve a *Clang* is egy C++-ban megvalósított projekt. Emellett azonban sok más, akár teljesen más paradigmákat képviselő programozás nyelv forráskódelemzése is aktívan vizsgált téma, és a bemutatott elvek közül sok érvényes ezekre is. Általánosan elmondható, hogy minél több a közös pont a kérdéses programnyelv és az erősen típusos, fordított, és hardverközeli C nyelv, valamint az arra épülő, magasszintű,

jelentősen bonyolultabb C++ nyelv között, a diplomamunka annál nagyobb része szolgálhat releváns információval.

2. fejezet

Hibakeresés

A szoftverfejlesztés implementációs fázisát sok esetben egy tesztelési eljárás követi. Ennek célja, hogy megbizonyosodjunk arról, hogy a specifikációban leírt működést produkálja a szoftver, illetve esetleg más esztétikai és kényelmi feltételeket teljesít-e. A tesztelés a program élettartamának számos pontján hatékonynak bizonyulhat. A hibakeresés lehetséges:

- tervezési fázisban, egy még absztrakt programleíró modellen
- megvalósítás közben, az implementációs programnyelven
- implementáció után, a fordítást követő reprezentáción, ez lehet:
  - bytekód
  - tárgykód
  - köztes reprezentáció
- futtatás előtt (pl. unit tesztek), futásidejű tesztekkel (pl. integrációs és performancia tesztek)
- rendeltetésszerű használat során a futási környezetben [4]

2.1. Hibakeresés absztrakt reprezentációkban

A szoftvertervezés korai szakaszában gyakran elvont ábrázolási módszereket vetnek be, mint például UML diagramok. Sokféle szerepet elláthatnak, például osztálydiagramok segítségével írják le a program strukturális szerkezetét, felhasználói-eset diagram segítségével a felhasználói interakciókat, szekvenciadiagramok segítségével pedig a programfolyamatok egymáshoz képesti ok-okozati, időbeli viszonyát. Már egy ilyen korai fázisban is érdemes lehet vizsgálni a szoftvert, a szekvenciadiagramok

validáció és verifikáció módszerét használó elemzése [1] már fejlesztői környezeteken belül is elérhető, a fejlesztési folyamat szerves részévé válhat.

2.2. Forráselemzés

Sok programozási nyelv esetében a forráskódot nem közvetlenül értelmezi a futtató környezet, hanem egy transzformációs lépésen át kell haladnia. A fordítás során jellemzően emberi feldolgozásra alkalmas formából valamilyen gépi szinten könnyebben értelmezhető alakba jutunk. Hogy ilyen többlépéses feldolgozást használ sok nyelv, annak többek között a program teljesítményéhez van köze. A C++ esetében például kifejezetten sok eszköz áll rendelkezésre, hogy azokat az információkat, amelyek a kód írása során rendelkezésre állnak, hatékonyan fel lehessen használni. A template metaprogramming technika például a C++ esetében egy teljes értékű programozási nyelvet alkot, melynek segítségével sok problémát már fordításidőben meg lehet oldani, illetve csökkenteni lehet a program futásidejét a fordításra szánt idő rovására. Jellemzően a fordítást megelőzően egy ellenőrző lépés keretében történik a vizsgálat, vannak azonban csak értelmezett (interpretált) nyelvek is. Ezek esetében is meg lehet vizsgálni a forráskódot, és az alapján is lehet érvelni a majd futó program tulajdonságait illetően, de általában kevesebb információ áll rendelkezésre. A dinamikus és gyenge típusozás egyaránt bővítheti az egyes változók lehetséges értékkészletét, illetve az esetleges gyengébb láthatósági és élettartam megkötések azt eredményezhetik, hogy kevésbé körülhatárolható, hogy a program mely pontján van lehetőség a változó értékének megváltozására.

Diplomamunkámban a forráselemzés kitüntetett szerepet kap, a statikus analízis a forráselemzés jelenleg legnagyobb komplexitással bíró eleme, mind a detektálásra szánt idő, mind a felismerhető problémák körét tekintve. Azonban a forráselemzés más technikákat is magába foglal. Forráselemzéshez tartozik a program forráskódjának minden lehetséges szinten történő elemzése:

- lexikális, vagyis, hogy a program megengedett tokenek sorozatából áll
- szintaktikai, mely során megvizsgáljuk, hogy a tokensorozat megengedett nyelvi elemeket ír le
- statikus szemantikai, vagyis a nyelvi elemek konstrukciói értelmesek-e
- specifikációhoz köthető szemantikai, a program jelentésével foglalkozó elemzés, vagyis hogy a program által leírt működés valóban megoldja-e a kitűzött problémát

A forráselemzés jobb kifejezés, mint a fordításidejű analízis, mert interpretált nyelvek esetében is van értelme a forrást vizsgálni. A módszerek, melyek a forráskódot mint olyan reprezentáció vizsgálják, amelyen automatizált elemzéseket és transzformációkat [2] lehet végrehajtani, egyre elterjedtebbek. A transzformációk célja lehet lehet többek között optimalizáció, vagy a futtatási környezethez való alkalmazkodás.

Erre példa lehet a JavaScript programozási nyelv, mely egy alapvetően interpretált, böngészőszoftverekben elterjedt nyelv. A nyelvi sztenderd fejlődése gyors a böngészők támogatásának elterjedéséhez képest, illetve régebbi böngészőverziók nem támogatnak bizonyos újabb nyelvi elemeket. Ennek köszönhetően az új sztenderd szerint helyes kódbázis nem kompatibilis a régi böngészőkkel. Ezt a problémát egy forrástranszformációs technikával [3] oldják meg, mely során az új nyelvi elemeket forráskód szintjén lecserélik egy olyan implementációra, mely viselkedését tekintve megegyezik az újjal, és kompatibilis a régi sztenderddel is.

2.2.1. Szintaktikai elemzés

Fordított programnyelvek esetében egyértelmű követelmény, hogy a forráskód minden részének helyesnek kell lennie a nyelv szabályai szerint, különben már a fordító hibaüzenettel jelez. Jó lehetőséget ad azonban ez a módszer nem fordított nyelveknél is, ahol ha ezeket az elemzéseket előre el tudjuk végezni az egész kódbázison, akkor sok hibát kiszűrhetünk.

2.2.2. Egyszerű szemantikai elemzés

Egyszerű szemantikai elemzés esetében a forráskód alapján döntéseket hozunk a program jelentésének, várható viselkedésének helyességéről. Tipikusan egyszerűbb, nyilvánvalóan helytelen, illetve teljesen felesleges utasítások szűrhetőek ki ilyen módon. Jellemző példa lehet egy változó önértékadása, amely legtöbb esetben egy üresművelet és a fejlesztő figyelmetlenségéből ered. Példa C++ esetében, tegyük fel, hogy egy közönséges függvény törzsében vagyunk:

```
int a = 0;
a = a;
```

2.2.3. Összetett szemantikai elemzés

Látható, hogy ha a fenti példában nem szerepelne a változó nullára inicializálása sem, akkor még egy problémával találkoznánk: inicializálatlan változó értékének használata. Ennek a problémának a felderítése azonban általános esetben sokkal

nehezebb, mint egy közönséges önértékadás felfedezése. Szükség van arra, hogy a programozási nyelv szabályait követve meghatározzuk azt a környezetet, ahol a használatot megelőzően potenciálisan értéket kaphatott a változó, majd fel kell deríteni, hogy ez tényleg megtörtént-e.

```
void simple_read(char *filename, char *dst, size_t size) {
  FILE* file = fopen(filename, "r");
  fread(dst, sizeof(*dst), size, file);
  // nincs fclose hívás
}
```

Kódrészlet 2.1. Szemantikailag hibás függvény

A fenti 2.1 példa a Unix rendszerek alapvető I/O műveleteivel olvas ki egy fájlból valamennyi bájtnyi adatot egy céltárolóba. A fenti függvény szintaktikailag helyes, és még ha hiányzik is belőle az `fopen` és a `fread` függvények visszatérési értékének ellenőrzése, a legtöbb esetben az elvárt módon viselkedik. Mindenesetre addig, ameddig a folyamat el nem éri a számára megnyitható fájlleírók számát. Az `fopen` függvény ugyanis egy rendszerhívás köré írt vékony csomagoló, amely erőforrást foglal le a rendszerben. Az elvárt használata ennek az API-nak, hogy amikor végeztünk az erőforrás használatával, egy `fclose` hívással értesítsük erről a rendszert is. A rendszer felszabadítja az erőforrást, és a folyamat leírók lefoglalására szánt kerete visszaáll. Az ilyen jellegű hibák felismerésére több információval kell rendelkeznünk a nyelv szintaktikai szabályainál, és a detektálás során is szükséges, hogy egy kódot alaposabban megvizsgáljunk. A fenti példában a hiba felismeréséhez szükség van a következő információkra:

- volt egy `fopen` hívás
- az `fopen` hívás visszatérési értékével, mint paraméterrel nem volt hívás a `fclose` függvényre

Az, hogy egy változóban tároljuk el az `fopen` hívás értékét, technikai részlet. Mégis fontos, mert nem lenne sok értelme egy megnyitott leírót rögtön bezárni, és a hibakeresés lehetőségétől is elesnénk. Valójában azt is mondhatjuk, hogy minden szemantikailag helyes használata ennek az API-nak feltételezi, hogy a fájlleíró értékét a memóriában őrizzük, ameddig ténylegesen használjuk az erőforrást. Mindezt azért, mert így később fel tudjuk szabadítani. Az eddigi észrevételeinket az alábbi módon egészíthetjük ki:

- volt egy `fopen` hívás

- az `fopen` hívás visszatérési értékét egy változó formájában eltároljuk a memóriában, jelenleg ez a *file*
- a *file* változó élettartamszabályai által meghatározott kódrészben sehol sem történt a *file* változóval paraméterezett `fclose` hívás

Ha ilyen formában fogalmazzuk meg a problémát, akkor elég ránézni a kódrészlethez tartozó szintaxisfára. Ezt és még sok, az analízist, és fejlesztést megkönnyítő funkciót a *Clang* eszköz a rendelkezésünkre bocsát. A következő parancs segítségével szöveges formába ki tudjuk kérni a szöveges szintaxisfa-reprezentációt.

```
clang --analyze -Xanalyzer -ast-dump file_read.c
```

Kódrészlet 2.2. Az ast-dump használata

```
`-FunctionDecl 0x9ee8b60 <file_read.c:4:1, line:7:1> line:4:6 simple_read 'void (char *, char *, size_t)'
  |-ParmVarDecl 0x9ee8940 <col:18, col:24> col:24 used file 'char *'
  |-ParmVarDecl 0x9ee89b8 <col:30, col:36> col:36 used dst 'char *'
  |-ParmVarDecl 0x9ee8a28 <col:41, col:48> col:48 used size 'size_t':'unsigned long'
  `-CompoundStmt 0x9ee8f70 <col:54, line:7:1>
    |-DeclStmt 0x9ee8da8 <line:5:3, col:25>
    | `-VarDecl 0x9ee8c28 <col:3, col:24> col:7 used fd 'int' cinit
    |   `-CallExpr 0x9ee8d40 <col:12, col:24> 'int'
    |     |-ImplicitCastExpr 0x9ee8d28 <col:12> 'int (*)(const char *, int, ...)' <FunctionToPointerDecay>
    |     | `-DeclRefExpr 0x9ee8c88 <col:12> 'int (const char *, int, ...)' Function 0x9e93368 'open' 'int (const char *, int, ...)'
    |     |-ImplicitCastExpr 0x9ee8d90 <col:17> 'const char *' <BitCast>
    |     | `-ImplicitCastExpr 0x9ee8d78 <col:17> 'char *' <LValueToRValue>
    |     |   `-DeclRefExpr 0x9ee8cb0 <col:17> 'char *' lvalue ParmVar 0x9ee8940 'file' 'char *'
    |     `-IntegerLiteral 0x9ee8cd8 <col:23> 'int' 0
    `-CallExpr 0x9ee8ed0 <line:6:3, col:21> 'ssize_t':'long'
      |-ImplicitCastExpr 0x9ee8eb8 <col:3> 'ssize_t (*)(int, void *, size_t)' <FunctionToPointerDecay>
      | `-DeclRefExpr 0x9ee8dc0 <col:3> 'ssize_t (int, void *, size_t)' Function 0x9eab850 'read' 'ssize_t (int, void *, size_t)'
      |-ImplicitCastExpr 0x9ee8f10 <col:8> 'int' <LValueToRValue>
      | `-DeclRefExpr 0x9ee8de8 <col:8> 'int' lvalue Var 0x9ee8c28 'fd' 'int'
      |-ImplicitCastExpr 0x9ee8f40 <col:12> 'void *' <BitCast>
      | `-ImplicitCastExpr 0x9ee8f28 <col:12> 'char *' <LValueToRValue>
      |   `-DeclRefExpr 0x9ee8e10 <col:12> 'char *' lvalue ParmVar 0x9ee89b8 'dst' 'char *'
      `-ImplicitCastExpr 0x9ee8f58 <col:17> 'size_t':'unsigned long' <LValueToRValue>
        `-DeclRefExpr 0x9ee8e38 <col:17> 'size_t':'unsigned long' lvalue ParmVar 0x9ee8a28 'size' 'size_t':'unsigned long'
```

2.1. ábra. Szintaxisfa szöveges megjelenítése.

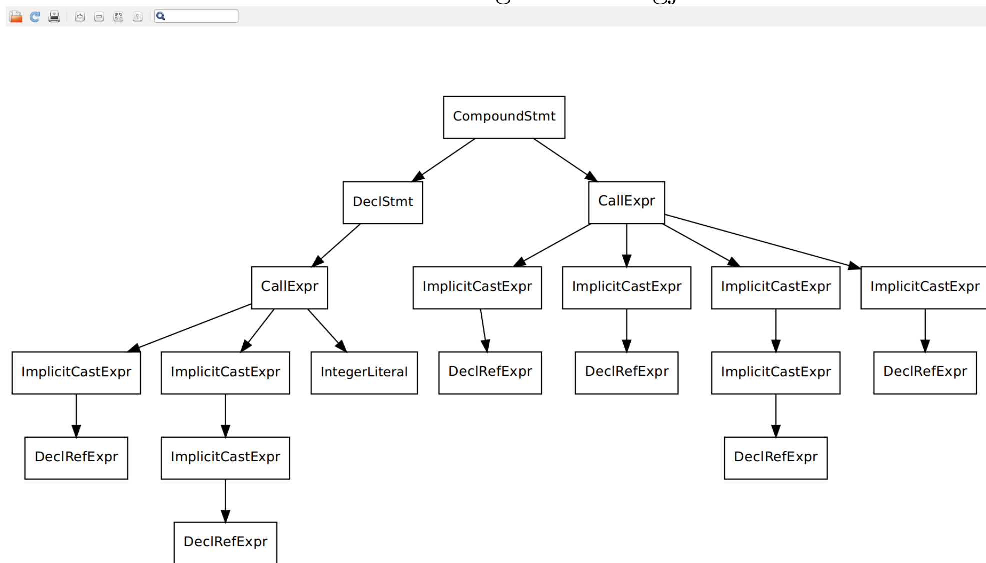
Lehetőség van arra is, hogy a szintaxisfát grafikusan megjelenítsük, ezt a *Clang* úgy támogatja, hogy egy *dot* formátumú fájlt generál, amit platform-specifikusan meg lehet tekinteni valamilyen eszközzel. Én az Ubuntu rendszeren elérhető xdot programot használtam.

```
clang --analyze -Xanalyzer -ast-view file_read.c
```

Kódrészlet 2.3. Az ast-view használata

Látható, hogy a hiba feltárásához nem elég a szintaxisfa áttekintése, mégis az ilyen jellegű hibák felismeréséhez szükséges információ szerepel benne. Szükség van a folyamat futásának modellezésére is, amelyet egy Control Flow Graph, vagy CFG segítségével tudunk szemléltetni (lásd ábra 2.3). Ilyen esetben fordulhatunk szimbolikus végrehajtás technikájához. A szimbolikus végrehajtás során szimuláljuk a program futását, azonban nem konkrét értékekkel, hanem az ezeket reprezentáló

2.2. ábra. Szintaxisfa grafikus megjelenítése.

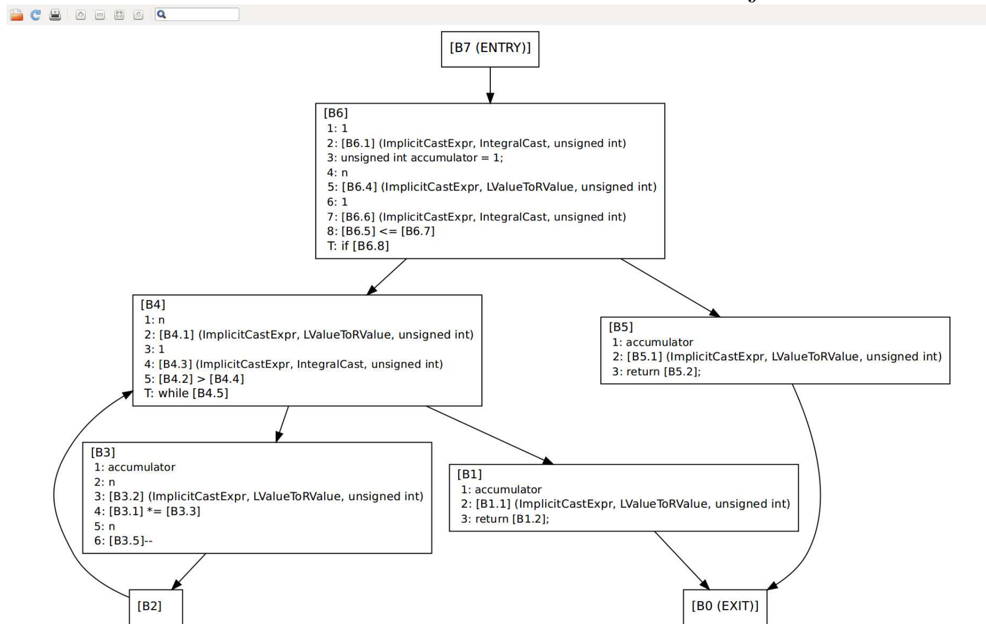


szimbolikus értékekkel. Az analízis során pedig az ezekre megfogalmazott kényszerek, és feltételek alapján tudunk következtetéseket levonni a kód helyességét illetően. A mai eszközök jellemzően a megszorításokról való érvelést visszavezetik egy SAT kielégíthetőségi problémára, amely során az aktuális programállapotban a változók lehetséges értékkészletét próbálják meg leszűkíteni. Ilyen módon ha kielégíthetetlen állapothoz értünk, akkor már a nyelv szabályai segítségével tudunk érvelni a futási utak bejárhatóságáról is.

```
unsigned factiter(unsigned n)
{
    unsigned accumulator = 1;
    if (n <= 1) return accumulator;
    while (n > 1)
    {
        accumulator *= n;
        n--;
    }
    return accumulator;
}
```

Kódrészlet 2.4. A faktoriális iteratív algoritmusa

2.3. ábra. Az iteratív faktoriális CFG-ja.



2.2.4. Futásidejű hibakeresés

Kézenfekvő lehet a program működését menet közben megfigyelni. Ilyen esetben valamilyen eszköz segítségével az elkészült program futási tulajdonságait vizsgáljuk. Ezen eszközök is széles skálán mozoghatnak interaktivitás, hatókör és az éppen vizsgált program formáját tekintve.

A futásidejű hibakeresés más hibák felfedezésére képes, mint az előbbi módszerek. Bizonyos szempontból több hibát fel tud ismerni, hiszen futás közben több információ áll már rendelkezésünkre, hiszen program használata közben fogadott inputok ilyenkor kapnak konkrét értéket. Ez a többletinformáció biztossá teheti például egy, a szoftvervizsgálat korábbi szakaszaiban csak valószínűsíthető hiba tényleges fennállását, vagy teljesen új hibás működésekre is felhívhatja a figyelmet. Mindemellett hátrányokkal is rendelkezik az eddig megismert módszerekkel szemben. A futásidejű hibakeresés csak olyan hibákat tud felfedezni a programban, amelyhez tartozó kódrészletek ténylegesen használatra kerülnek futás során.

Tegyük fel például, hogy van egy szoftver, melyet programkönyvtárként akarunk felhasználni. Ekkor ha futásidejű hibakeresést akarunk végezni, akkor szükségünk van olyan kódokra, amely ezt a könyvtárt felhasználja. Az is szempont lehet, hogy minél nagyobb mértékben legyen egy könyvtár ilyen módon letesztelve, hiszen sok egymással csak érintőlegesen kapcsolódó részből is állhat egy programkönyvtár. Ennek mérésére bevezetésre került pár metrika is, melyek kódminőség szempontjából jellemzik a szoftvert, például a code coverage, azt mondja meg, hogy a kódbázis mekkora része van valamilyen szempontból ellenőrzés alatt. A futásidejű tesztelés

esetében ezen metrikák magasan tartása úgy lehetséges, hogy a könyvtár lehető legtöbb részét felhasználó tesztelő szoftvert kell írni, ebben tesztesetek és azokon belüli assert-ek felelősek az egyes felhasználói esetek modellezéséért. Ezzel szemben statikus analízis során nem szükséges külön meghajtó kódot írni, hogy lehetséges legyen a tesztelés, elégséges a forráskód ismerete.

Vannak elméleti korlátai is annak, hogy a futásidejű hibakereséssel milyen problémákat lehet felfedezni. Bármely Turing-teljes nyelv esetében ugyanis eldönthetetlen a megállási probléma, vagy az erre visszavezethető problémák halmaza. Erre példa, hogy ha egy egyszerű szerkezetű végtelen ciklust szeretnénk detektálni, akkor arra a statikus módszerek megoldást adnak, míg a futásidejű elemzések legfeljebb valamilyen heurisztika alapján tudják valószínűsíteni, hogy hibás a program.

Manuális futásidejű hibakeresés, debug

Az egyik legelterjedtebb hibakeresési módszer. A fejlesztőnek lehetősége van a futtató környezet állapotának folyamatos nyomonkövetésére, miközben a program működését előre meghatározott helyeken (jellemzően kódsorszámozás alapján) megállíthatja. Ezen helyek között léptethet is, és képes lehet a futtató környezet megváltoztatására, mely során például változók értékének felülírásával új végrehajtási utakat kényszeríthet ki.

módszer	automatizálható	reprezentáció
debug	nem	gépi kód[1]
unit tesztek	igen, könnyen	gépi kód
integrációs tesztek	igen, nehezebben	gépi kód
szintaktikai elemzés	igen	forráskód
egyszerű szemantikai elemzés	igen	forráskód
összetett szemantikai elemzés	igen, nehezebben	forráskód

2.1. táblázat. Hibakeresési módszerek összehasonlítása

Automatizált futásidejű hibakeresés, tesztek

A futásidejű hibakeresés során használt manuális módszerekkel szemben, tesztekkel vezérelt hibakeresés már jóval könnyebben automatizálható. A fejlesztő a program hibáinak realizálására megvalósít egy futtatható szoftvert. Ezt sok esetben az aktuálisan tesztelt program saját nyelvén, valamilyen, a programozási nyelvhez kidolgozott keretrendszer segítségével teszi (ilyen például a C és C++ nyelv esetén a Google Test [5]). A tesztek növelik a kódminőséget, mert a már említett coverage

értékét növelik, dokumentációként szolgálnak, és feltérképezhetőbbé teszik a szoftvert, emellett biztonsági hálóként szolgálnak a kódbázis módosításakor esetlegesen fellépő regressziós hibák ellen.

Természetesen a szoftverfejlesztési feladatok közben több módszerre is szükségünk lehet, a szoftver minőségi tulajdonságait mindegyik pozitívan befolyásolhatja. A módszerek közötti különbségeket a 2.1 táblázatban foglaltam össze.

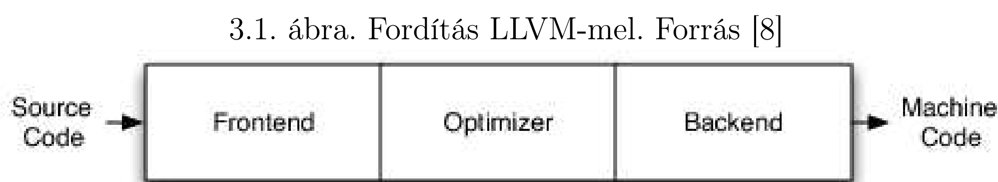
[1] Debug információkkal ellátott gépi kódról van szó, nem a release-ben használt optimalizált kódról

3. fejezet

Statikus elemzések Clang-gel

A *Clang* elsődlegesen egy a C nyelvi családhoz (C, C++, Objective C/C++) készült fordítóprogram. A *Clang* az *LLVM* projekt része, vagyis a teljes fordítási folyamat csak egy részét végzi. Az *LLVM* projekt [6] végzi a szintaktikailag ellenőrzött, szintaxisfa reprezentációba átírt kód IR, vagyis köztes reprezentációba hozását, az optimalizációs transzformációkat, és végül a kódgenerálást a megfelelő architektúrára. [6].

3.1. ábra. Fordítás LLVM-mel. Forrás [8]



3.1. LLVM

Az *LLVM* projekt maga is moduláris felépítéssel rendelkezik, sokféle architektúrát és programozási nyelvet képes összekötni. Alkotója Chris Lattner [9] 2000-ben kezdte meg fejlesztését, azzal a céllal, hogy létrehozzon egy olyan eszköztárat, amely általános, programnyelvtől független értelmezési, fordítási és optimalizációs feladatokat lát el. C++ nyelven íródott, eredetileg C és C++ nyelveket támogatott, 2019-re már több, mint 20 nyelvhez léteznek benne megfelelő modulok. Néhány programozási nyelv, melyre az *LLVM* rendelkezik támogatással:

- C#
- Common Lisp
- Delphi
- Fortran
- OpenGL
- Haskell

- Java
- Julia
- Python
- R
- Ruby
- Rust
- Scala
- Swift

IR

Az *LLVM* projekt, hogy a különböző programozási nyelveket le tudja fordítani a különböző architektúrákra, bevezetett egy köztes reprezentációt. A köztes reprezentáció maga is egy programozási nyelv, hasonlít az assembly-re, erősen típusos. Erre példa a 3.1 kódrészlet. Ennek köszönhetően ha egy új programozási nyelvet szeretnénk lefordítani az eddigi architektúrákra, elég csak egy úgynevezett frontend-et létrehozni hozzá, mely arról gondoskodik, hogy az új programozási nyelvet a köztes reprezentációra alakítja át. Hasonló a helyzet, ha egy új hardvertípust kell támogatni, itt a köztes reprezentációból a gépi kódra átalakító rész neve a backend. A köztes reprezentáció nagy előnye, hogy feloldja azt a skálázási problémát, amit egy új végpont (legyen az egy új nyelv, vagy új architektúra) bevezetése okoz.

```
@.str = private unnamed_addr constant [13 x i8] c"hello world\0A\00"
declare i32 @puts(i8* nocapture) nounwind
define i32 @main() { ; i32()*
  %cast210 = getelementptr [13 x i8], [13 x i8]* @.str, i64 0, i64 0
  call i32 @puts(i8* %cast210)
  ret i32 0
}

!0 = !{i32 42, null, !"string"}
!foo = !{!0}
```

Kódrészlet 3.1. Hello World LLVM IR

SSA

Az *LLVM* projekt optimalizálási technikák széles körét beveti, hogy hatékonyan tudja a lehető legoptimálisabb kódot generálni. Ezen optimalizációs technikák közül sok épít a köztes reprezentáció egy speciális tulajdonságára, a Static Single Assign-

ment alakra. Az *LLVM*-ben van eljárás az SSA alakra hozásra, és ezt érdemes is megtenni, mert a következő optimalizációs lépések hatékonyabbak:

- constant propagation
- dead code elimination
- global value numbering
- register allocation

3.2. Clang

A *Clang* egy frontend, amely a C nyelvi család fordítását végzi. Ennek eredménye az LLVM IR forma. A *Clang* fejlesztése közben emellett az is szempont volt, hogy megfelelő betekintést adjon a fordítás folyamatába, és jó minőségű diagnosztikai információval szolgáljon. Így a *Clang* fordító nem csak a végzetes hibákat jelző üzenetekkel kommunikál a fejlesztő felé, hanem figyelmeztető üzenetekkel is. Emellett a *Clang* forráselemző és forrástranszformációs feladatot is ellát. Ezek a tevékenységek a fordítás során végzett szintaktikus elemzés eredményeit használják fel, így kézenfekvő, hogy a *Clang* keretében kerültek megvalósításra.

Fordító

A *Clang* elsődleges célja, hogy a C nyelvi család általános fordítójaként használhassák. Ezt a feladatot a kiadásakor legszélesebb körben a GNU Compiler Collection (GCC) látta el, és még ma is meghatározó a szerepe. A *Clang*-et a fejlesztők úgy alkották meg, hogy a lehető leginkább egy, a GCC fordítóval kompatibilis, annak alternatív fordítójaként használható eszköz legyen. Emellett, hogy sok platformon el tudjon terjedni, szüksége van arra, hogy a megfelelő nyelvi forrásfájlokat, illetve programkönyvtárakat fel tudja használni. A legtöbb UNIX alapú rendszer mellett a *Clang* már Windows rendszeren képes akár a natív Visual Studio által kialakított környezettel együttműködni [10].

A GCC-vel való kompatibilitás és az a funkciója, hogy képes más fordítási- és rendszerkörnyezetekkel együttműködni jelentős többletkomplexitást ad az „egyszerű" fordítási feladatához. Emiatt a megvalósítása során többrétegű alkalmazásmodellt választottak a fejlesztők. A *Clang* eszköz a végfelhasználó számára elsődlegesen egy, a platformnak megfelelően lefordított és csomagolt bináris állományként áll rendelkezésre. Ez az eszköz alapértelmezetten a *Clang* driver részét használja, amely meghajtja a fordítási folyamatot. Ez felelős a parancssori opciók feldolgozásáért, a programkönyvtárak és a rendszerhez tartozó forrásfájlok megtalálásáért. Ez a rész az, ami a beérkező parancs és argumentumok együtteséből egy konkrét művelet tervét elkészíti. Lásd 3.2 ábra.

A fordítás folyamata 5 fő lépésre tagolható:

- Parse
- Pipeline
- Bind
- Translate
- Execute

Parse

Ebben a fázisban a parancssori argumentumok feldolgozása történik. Az argumentumok tárolása közben megjelenik az a hozzáállás, amely az *LLVM* projektre jellemző, a hatékonyságra törekvés, miszerint minden opció csak egyszer legyen beolvasva, és ne legyen több példányban eltárolva. Ezt a *Clang* driver kódja az opcióknak megfelelő Arg osztály példányainak egy hatékony, központosított tárolóban történő elhelyezésével segíti elő. Ez az ArgList. Ez kihasználja a C nyelvre jellemző string-ábrázolás sajátosságait és összhangban van C++ nyelv költség nélküli absztrakcióival.

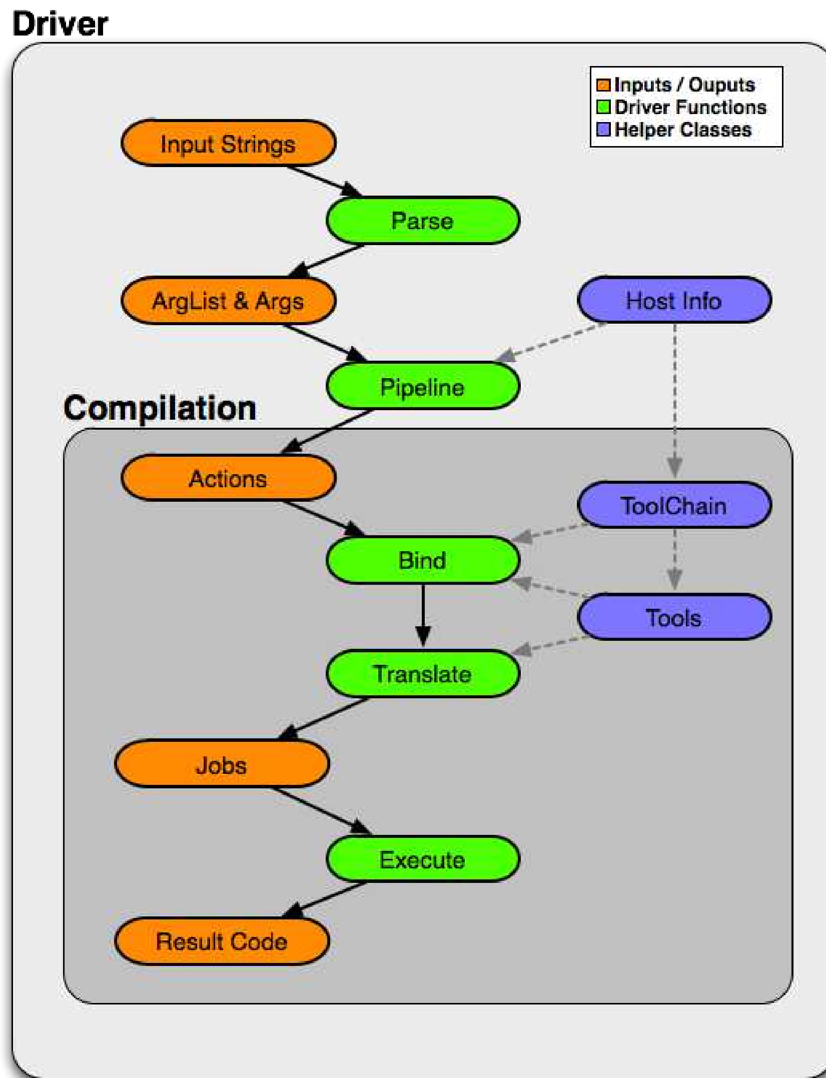
Pipeline

Miután az argumentumok feldolgozása megtörtént, a fordító megvizsgálja, hogy milyen input és output fájlokkal fog dolgozni a működése során, majd az ezeket összekötő lépéseket nagyvonalakban definiálja. Ezeket a lépéseket az Action osztály szimbolizálja a *Clang* driver-en belül. Action példányok egy sorozata ad egy feladatot, Task-ot. A programfordítás során előkerülő eszközök beszédes neveivel jelöli a *Clang* eszköz ezeket az Action lépéseket, például preprocessor, compiler, assembler. A fordításhoz kialakított Action lépések listáját a felhasználó meg is tekintheti.

```
clang -ccc-print-phases simple_read.c
0: input, "simple_read.c", c
1: preprocessor, {0}, cpp-output
2: compiler, {1}, ir
3: backend, {2}, assembler
4: assembler, {3}, object
5: linker, {4}, image
```

Kódrészlet 3.2. A fordítási pipeline lekérdezése

3.2. ábra. A Clang Driver szerkezete. Forrás [13]



Bind

Ebben a lépésben a fordító a kialakított Action sorozatokra próbál meg konkrét megvalósításokat találni. Ezt a ToolChain osztály segíti, amely eszközöket, Tool-okat tartalmaz. A Tool-ok végzik el a konkrét fordítási fázis egy-egy folyamatát. A fordító először megkeresi melyik ToolChain melyik Tool-ja szükséges egy Action végrehajtásához, majd a későbbiekben a fordító a Tool-lal kommunikálva újabb folyamatokat szúrhat be a végrehajtási listába.

```
clang -ccc-print-bindings -arch i368 simple_read.c
# "x86_64-unknown-linux-gnu" - "clang", inputs: ["simple_read.c"], output:
    "/tmp/simple_read-3c1f07.o"
# "x86_64-unknown-linux-gnu" - "GNU::Linker", inputs: ["/tmp/simple_read-3
    c1f07.o"], output: "a.out"
```

Kódrészlet 3.3. A fordítási tool-ok lekérdezése

Translate

Miután a fordító kiválasztott egy eszközt, az eszköz feladata, hogy Command típusú objektumokat állítson elő. Ezek a ténylegesen futtatandó alfolyamatokat, és argumentumaikat szimbolizálják. A legtöbb munka azzal van, hogy a GCC-vel kompatibilis opciókat a Tool lefordítsa a futtatandó eszköz számára értelmezhető beállításokká. Sok helyen ez nem bonyolult, az assembler például tipikusan kevés külső opcióval dolgozó program, de a linkerek ezzel szemben argumentumok széles skáláját használják aktívan [11].

Execute

A fordítás végrehajtása a driver számára a legegyszerűbb lépés, kevés opcióval, van azonban lehetőség a programkimenetek átirányítására, az egyes eszközök visszatérési értékeinek továbbadására, illetve a folyamatok teljesítménymérésére is.

Elemző

A *Clang* eszköz alapértelmezetten megpróbál segítséget nyújtani a fordítás során akkor is, ha külön nem kérjük rá. A fordításkor felmerülő hibaüzenetek mellett figyelmeztetőüzenetek széles skáláját használja ahhoz, hogy az esetleges hibák, nem várt működések kiküszöbölésében segítse a programozót.

A fordítás közben kiadott hibaüzenetek és figyelmeztetések mellett lehetőség van más vizsgálatok lefuttatására is, amelyek a tüzetesebb vizsgálatnak vetik alá a programot. Ezek általában valamilyen futás- vagy fordításidejű költséggel rendelkeznek, ezért a közönséges fordítás alatt nem futnak le alapértelmezetten. A *Clang* és *LLVM* infrastruktúra több ilyen módszerrel is bír. A különböző Sanitizer névre hallgató megvalósítások, például AddressSanitizer, MemorySanitizer, UndefinedBehaviorSanitizer lényege, hogy a program lefordításakor a keletkező bináris állományban a program futása mellett a program működését vizsgáló logika is fut, és hiba érzékelése esetén fájlban vagy sztenderd kimeneten értesítést erről. Ezek futásidejű hibakeresé-

si módszerek. Dolgozatomban a forráselemzés által felfedhető hibák detektálásával foglalkozom, így a *Clang* eszköznek ezen részeit ismertetem a továbbiakban.

Hogy a *Clang* fordító és a *Clang* driver kapcsolatát jobban megérthessük nézzük meg az absztrakt szintaxisfa (AST, abstract syntax tree) kinyomtatásához felhasznált parancsot (lásd kódrészlet 2). A következő két parancs ugyanazt az Action-t adja, és ugyanazt az analízismodulhoz tartozó műveletet indítja el. Az egyetlen különbség, hogy a második nem megy végig a driver előfeldolgozó részén, és így az esetleges rendszerhez tartozó header fájlok nem láthatóak a fordítónak, azokat a GCC esetében már megismert -I és -isystem kapcsolókkal megadhatjuk (valójában ezt teszi a driver is).

```
//driver
clang --analyze -Xanalyzer -ast-dump file_read.c
//clang-frontenden
clang -cc1 -analyze -ast-dump file_read.c
```

Kódrészlet 3.4. A driver és a compiler közvetlen kapcsolata

Integrációs lehetőségek

A *Clang* könyvtárcsomag számos diagnosztikai információt tud szolgáltatni a C nyelvcsalád programjaival kapcsolatban. Amikor forráskódelemzést végzünk, akkor a *Clang* fordítónak általában csak az előfeldolgozó (a driver), és a szintaxisfa felépítéséért felelős részét használjuk. A köztes reprezentációra alakítás, illetve az ebből történő tárgykód-előállítás elmarad. Ennek fő oka, hogy a hibák detektálásához a legtöbb statikus analízist bevető megoldás a vizsgált nyelv sajátosságait használja fel, míg az általános IR reprezentációban szereplő esetleges nem várt, vagy nem hatékony működéseket az optimalizációs lépés küszöböli ki. Így nincs tehát szükségünk statikus analízis vizsgálatok lefuttatásához az IR-re, megállhatunk a kész szintaxisfa szintjén.

Ezt a szintet engedi vizsgálni a *Clang* könyvtár három fő interfészén keresztül, amelyet a *Clang*et felhasználó fejlesztők rendelkezésére bocsát. A főbb felületek:

- LibClang
- LibTooling
- Clang Plugin

LibClang

A *Clang* fejlesztői fontosabbnak tartják a hatékonyságot szem előtt tartó folyamatos átalakításokat, illetve az új funkciók minél hamarabbi bevezetését, mint a szigorúan betartott visszafele-kompatibilitást, amikor a kódbázisról van szó. Ennek köszönhetően hatékony szoftvert kapunk, amikor a *Clang*-et használjuk, de ez a fejlesztőktől aktívabb közreműködést vár el, amikor egy *Clang* belső reprezentációval rendelkező szoftvert kell fenntartaniuk. Ez főleg a szintaxisfát érinti, amikor egy új *Clang* verzióban a szintaxisfa reprezentációja megváltozik, vagy esetleg új nyelvi elemeket emelne be, akkor egy régebbi *Clang* verzióval kompatibilis könyvtár elromolhat az új verzióban. Akik egy stabilabb fejlesztési felületre vágynak, azoknak megfelelő a LibClang interfész. A LibClang egy C nyelven írt könyvtár, amely bár nem biztosítja az AST-hez való közvetlen hozzáférést, ezt áldozza fel a stabilitás kedvéért.

A LibClang könyvtár emellett a megszorítás mellett is sok területen kedvelt és használt. Ennek a segítségével megvalósítható a *Clang* által kezelt nyelvek forráskód szintű vizsgálata. Lehetőséget a szintaxisfa bejárására a látogató (visitor) programozási minta segítségével, így relatíve magas absztrakciós szinten ad lehetőséget a forráskód statisztikai elemzésére (pl. használt nyelvi elemek száma, előfordulási sűrűsége egyes modulokon belül, más elemekkel együtt való használat gyakorisága stb.), intelligens kontextusvizsgálatra, illetve nyelvi konstrukciók forráskódbeli helyükkel való pontos összekötésére [17]. Ezen funkciók kihasználása teszi lehetővé, hogy olyan eszközöket készítsünk a LibClang felhasználásával, mint automatikus forráskódformázó eszközök, amelyek egy kódstílust leíró konfiguráció segítségével tetszőleges forrásállományt át tudnak formázni úgy, hogy csak külalakra változik, struktúrája azonban ugyanaz marad. Emellett képesek lehetünk automatikus kódkiegészítők alkotására, melyek integrált fejlesztői környezetekben (IDE) kontextusfüggően képesek javaslatokat tenni a szerkesztés alatt levő forráskódokra. Emellett a LibClang könyvtár nagy előnye a másik két lehetőséggel szemben, hogy mivel C nyelvi interfész, ezt lehet a legkönnyebben más, akár dinamikusan típusos script-nyelvek programjaiba is integrálni. Python nyelvre van a hivatalosan nyilvántartott dokumentációban is hivatkozás, mint ahogy arra is, hogy az XCode IDE az OSX rendszeren használja ezt a lehetőséget [18].

LibTooling

A LibTooling interfész C++ nyelvet használ, emiatt sokkal természetesebb módon, nagyobb manipulációs lehetőséget biztosít az AST-t tekintve, mint a LibC-

lang. Elsődlegesen különálló alkalmazások fejlesztésére tervezték. Megvan a LibClang könyvtár ismertetésekor felhozott hátránya, miszerint a *Clang* szintaxisfa-reprezentációinak változásakor könnyen visszafele-kompatibilitási problémák léphetnek fel. Emellett a közvetlen szintaxisfa közvetlen elérése lehetőséget ad arra, hogy bonyolultabb átalakításokat végezzünk forráskódokon. Nem csak kozmetikai változások lehetségesek, hanem akár ekvivalens, vagy valamilyen tulajdonságot megtartó strukturális átalakítások is (pl. refaktorálás vagy obfuszkálás). Emellett természetesen rendelkezésünkre áll minden, ami LibClang használata esetében is, így nagyon pontos a nyelvi elemek és forráskódbeli pozíciójuk közötti leképezés, tehát forrásdiagnosztika szoftverek alapjául is szolgálhat.

Clang Plugin

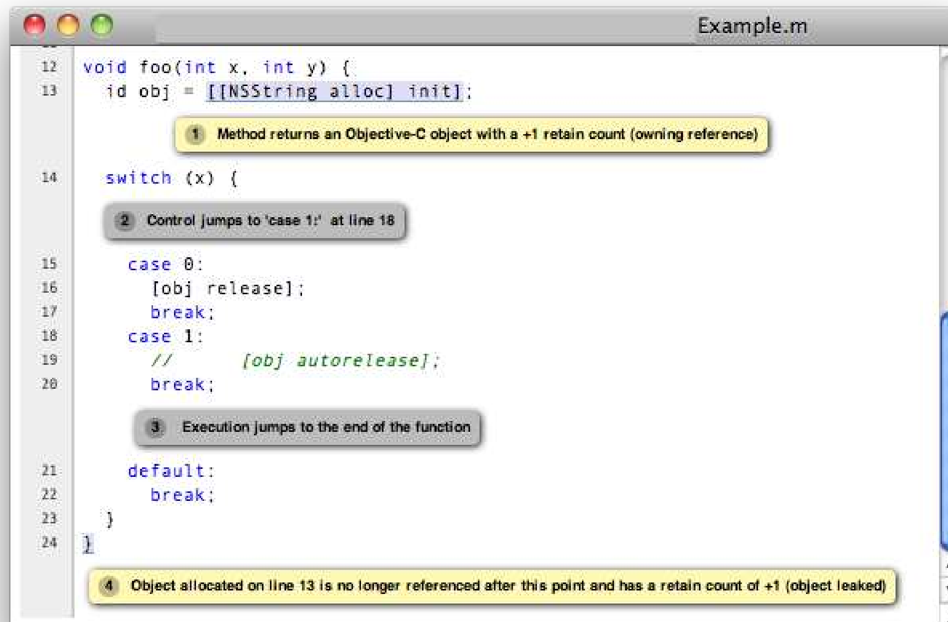
A *Clang* modulárisan felépített szoftver, lehetőséget ad a kiterjesztésre. A Clang Plugin rendszer segítségével a fordítás közben különböző műveleteket végezhetünk a szintaxisfán. Ezek a műveletek többek között analitikai műveletek is lehetnek, a Clang Static Analyzer (Clang SA) projekt is ezt a megoldást választotta a különböző problémákat felismerő ellenőrző modulok (checker-ek) futtatásához. Lehetőség van statikus linkelésű checker-ek, illetve dinamikus linkelésű beépülő modulok létrehozására is. A Clang SA alapvetően statikusan linkelt checker-eket használ. A Plugin rendszer hátránya a LibTooling-hoz képest, hogy alapvetően a fordítási folyamat részeként fut az ellenőrzés, emiatt a build-rendszer vezérli, és így egyetlen különálló forrásfájl magában nem igazán támogatott. Sokkal idiomatikusabb, és egyszerűbb ennél az eszköznél egy egész projekt elemzése. Tipikusan Plugin-t használnak még a fordítási figyelmeztetéseket adó, illetve a build-folyamat során járulékos állományokat létrehozó modulok.

Clang Static Analyzer

A Clang Static Analyzer egy forráselemző, amellyel C, C++, Objective-C programok hibáit lehet megtalálni. Alapvetően parancssori eszköz, azonban hivatalos integrációja létezik az Apple XCode fejlesztői környezettel, ahol alapértelmezetten be van építve. Használata inkább kötődik a kód fordítását végző build-folyamathoz, ekkor nem egy-egy forrásfájlról ad hibainformációkat, hanem egy egész projektről, de emellett lehetőség van egyszeri fordítás ellenőrzött elvégzésére is.

Alapvetően a használatához szükség van a *Clang* forráskódjával csomagolt scan-build és scan-view eszközökre, de elérhető bináris formában OSX rendszerre is [14]. A scan-build alapvetően Perl nyelven íródott, de egy modernebb megvalósítása elérhető Python nyelven is, ennek használatát javasolják a fejlesztők új projektek esetében.

3.3. ábra. Clang Static Analyzer hibainformációk. Forrás [16]



A parancssori futás eredményét egy webes felületen megtekinthetjük, a scan-build-py eszköz html fájlok formájában is eltárolja egy-egy futás eredményét. A scan-view alkalmazás megtalálja az elemzés eredményét és megmutatja.

Az eszköz már magában is képes teljes értékű elemző, azonban a megvalósított program az Ericsson által fejlesztett *CodeChecker* [19] alkalmazást használja, amely kifinomultabb, jobb kezelhetőséget, és felhasználói élményt biztosít az ellenőrzések futtatására.

3.3. Statikus analízis

A statikus analízis során definíció szerint eltekintünk a vizsgált szoftver futásideji viselkedésétől, tehát úgy érvelünk a szoftver-tulajdonságokról, illetve keresünk hibákat, hogy csak olyan eszközöket használunk, amelyek valamilyen forrás-reprezentáción dolgoznak. Ez a forrás azonban nem feltétlenül homogén jellegű, akár a használt programozási nyelvek, akár a reprezentáció absztrakciós szintje szempontjából. A különböző statikus analízis eszközök jelentős eltérést mutatnak, főként abban, hogy milyen formájában kezelik a forráskódot. Mivel a statikus analízis közeli kapcsolatban van a fordítási folyamattal, legtöbb esetben a fordítás során használt reprezentáció használatosak itt is.

3.4. ábra. A scan-build-py eszköz parancssori futása.

```
[17:42] - Using analyzer:
/home/gamesh411/llvm_build/bin/clang-5.0
clang version 5.0.0 (http://llvm.org/git/clang.git 050aef390288e8cd4a8d66336ade7a5cba0c4493) (http://
Target: x86_64-unknown-linux-gnu
Thread model: posix
InstalledDir: /home/gamesh411/llvm_build/bin
[17:42] - Using analyzer:
clang-tidy
LLVM (http://llvm.org/):
  LLVM version 5.0.0svn
  DEBUG build with assertions.
  Default target: x86_64-unknown-linux-gnu
  Host CPU: broadwell
[17:42] - Static analysis is starting ...
simple_read.c:17:7: Value stored to 'd' during its initialization is never read [deadcode.DeadStores]
  int d = c / b;
      ^

simple_read.c:17:13: The right operand of '/' is a garbage value [core.UndefinedBinaryOperatorResult]
  int d = c / b;
            ^

clangsa found 2 defect(s) while analyzing simple_read.c
```

3.5. ábra. A scan-build-py eszköz eredményeinek webes megjelenítése.

```
 8
 9 void uninitialized(int a) {
10   int b;
      'b' declared without an initial value
11   int c = 23;
12
13   if (a < 3) {
          Assuming 'a' is >= 3
14     b = 42;
15   }
16
17   int d = c / b;
              The right operand of '/' is a garbage value
18 }
19
```

3.3.1. Szöveges reprezentáció

Egyes analízis eszközök a nyers forrásszövegen dolgoznak. Ez a forma természetesen olvasható a fejlesztő számára, ezen kerül megfogalmazásra a szoftveres megoldás. Emiatt a más fejlesztők általi review folyamat is ezen a reprezentáción történik, illetve a kód stílusát és a megoldás módjának a megoldandó feladathoz való illeszkedését is itt lehet ellenőrizni (például egyes nem-funkcionális követelményeket befolyásoló implementációs döntések, mint a választott adatszerkezet vagy algoritmus idő- és tárkomplexitása). A legfőbb hiányossága ennek a reprezentációnak, hogy az automatizált ellenőrzést végző módszerek lényegében egy token-sorozaton dolgoznak, ami nem ad igazi rálátást a program szerkezetére. A fordítási folyamatban ez az információ később, a parser által kerül hozzáadásra. Emellett az ezen szinten megfogalmazott megoldások igen érzékenyek a kód újraformázására, refaktorálására. Mindezek mi-

att nehezen elképzelhető a szoftver komplex tulajdonságaira rákérdező ellenőrzések stabil és újrafelhasználható megvalósítása ezen absztrakciós szinten.

```
#define Z 0
#define DIV(a, b) ((a)/(b))

int main() {
  int a = 1 / 0; // triviális hiba
  int b = 1 / Z; // preprocesszor-információ szükséges
  int c = DIV(1, 0); // szintén

  // egyszerű, naív implementációk által nem felfedezett hibák
  int d = 1 / (1 - 1);
  int e = DIV(1, 1) / (DIV(1, 1) - DIV(1, 1));
}
```

Kódrészlet 3.5. A szöveges reprezentáció lehetőségei és korlátai

3.3.2. Absztrakt Szintaxisfa

A szöveges reprezentációhoz képest, mélyebb vizsgálatra ad lehetőséget a parser által — a programozási nyelv formális nyelvtana alapján — a szöveges forráskódból előállított absztrakt szintaxisfa (AST). A szintaxisfa sok esetben nem csak szintaktikai információt tartalmaz, a csúcsok szemantikus információt is hordoznak. Amellett hogy ez a reprezentáció rávilágít a program szerkezetére, típusinformációkat is tartalmazhat, mely főként az erősen és statikusan típusos programozási nyelvek esetén jelent nagy előnyt az analízis szempontjából. Az AST előállítása szükséges a fordítás kódgenerálási fázisában is, és a nyelv szabályainak komplexitásától függően költséges is lehet. Az AST-n dolgozó módszerek másik megkötése, hogy ebből a reprezentációból még nehezen visszanyerhető a program dinamikus viselkedése, a lehetséges végrehajtási útvonalakat nem lehet pusztán ennek segítségével könnyen megadni, tehát azok az ellenőrzések, amelyek ezekre hivatkoznának nem írhatók le ezen a szinten. A Clang infrastruktúrában létezik egy DSL, amely az AST-re való illesztések megadását támogatja (ASTMatchers). Ebben deklaratív módon lehet megadni az AST-ben egyes csúcsokat kijelölő, és ezeket bejáró kifejezéseket, és lehetőség az egyes csúcsokat névvel ellátva később meghivatkozni, így azok más illesztő kifejezések bemenetei is lehetnek. A 3.7 kódrészleten látható egy ASTMatcher kifejezés, amelyik a globális névtérben lévő `MyClass` osztály deklarációjára illeszkedik. Ha egy fordítási egység AST reprezentációjára, vagy annak egy részfájára ezzel illesz-

tést végzünk, akkor hozzájutunk az említett osztály deklarációjához tartozó AST csúcshoz, és később hivatkozhatunk rá `myClassDecl` néven, a bind részkifejezésnek köszönhetően.

```
// Clang Tidy check: cppcoreguidelines-slicing
struct B { int x; virtual int f(); virtual ~B() noexcept; };
struct D : B { int y; int f() override; };

void use(B b) { // Referencia helyett érték szerinti paraméter-átadás
  b.f(); // B::f kerül meghívásra.
}

D d;
use(d); // Vágás (Slicing) történik.
```

Kódrészlet 3.6. Egy AST-n alapuló ellenőrzés

```
// Clang ASTMatchers példa
// A MyClass nevű CXXRecordDecl típusú csúcsot elnevezzük
// myClassDecl-nek.

recordDecl(hasName("::MyClass")).bind("myClassDecl")
```

Kódrészlet 3.7. Illesztés az AST-re deklaratív módon

3.3.3. Program flow

Általánosan elmondató, hogy az AST önmagában nem biztosít robusztus, könnyen megvalósítható megoldási lehetőséget a 3.8 példán látható, és ehhez hasonló problémák detektálására. Ha egy módszer olyan találatot jelez, amely valójában nem létező problémára (hamis pozitív) utal, akkor ezek között a valódi találatok nehezen azonosíthatóak, és mivel a találatokat a fejlesztők validálják, hosszú távon nagyban csökkentik a módszer hatékonyságát. Hogy a program dinamikus tulajdonságairól érvelni tudjunk, számos data-flow elemzési algoritmus segítségünkre lehet, és ezeket jellemzően a fordító is felhasználja. Kihívást jelenthet viszont ezen algoritmusok eredményeit analízis céljából felhasználni, hiszen a fordítás során más szempontok alapján kerülnek implementálásra a fordítási folyamat jellemzően későbbi, optimalizálási fázisában.

```
enum Choice { YES, NO };
```

```
int f(Choice c) {
  if (c == YES) { return 1; }
  else { return 2; }
}

int main() {
  int a = f(getInputChoice());

  // a értéke nem lehet nagyobb, mint 2
  // függetlenül f paraméterétől
  if (a > 2) {
    // elérhetetlen végrehajtási pont
    return 1 / 0;
  }
}
```

Kódrészlet 3.8. Kevés információra építő elemzés hamis pozitív találatokhoz vezethet

3.3.4. Program path

Más fejlett statikus analízis megoldáshoz hasonlóan a Clang Static Analyzer is egy, a flow információnál is részletesebb reprezentációs szint, a program végrehajtási út (program path) kiértékelésével segíti a detektálási feladatok megoldását. A program path analízis — ideális esetben — minden lehetséges végrehajtási útvonalat feltérképez. A Clang Static Analyzer a végrehajtási utak bejárása során szimbolikus végrehajtást használ, hogy érvelni tudjon a változók lehetséges értékeiről, majd az ebből az egyes változókra vonatkozó megszorításokat gyűjti és kezeli. A program path analízis hátránya, hogy problématere — a bejárandó utak száma — általánosan exponenciális függvénye a programban található elágazások számának. Ez azt eredményezi, hogy ennek a módszernek minden praktikus felhasználásra valamilyen heurisztikát is használ arra, hogy eldöntse mely utakat zárja ki, és mely utakat járja be ténylegesen.

```
int main() {
  // a 'getInputChoice()' kifejezés visszatérési
  // vagy az enum egyik értéke vagy ismeretlen
  int a = f(getInputChoice());
```

```
}
```

Kódrészlet 3.9. Végrehajtási utak mentén modellezett szimbolikus értékek

A Clang SA keretrendszer különböző stratégiákat implementál a bejárás sorrendiségének meghatározására. Az első implementációk egy mélységi-jellegű megoldást választottak, ami egy adott lehetséges végrehajtás mentén próbált a lehető legtovább menni. Ennek alternatívájaként jelent meg később egy szélességi bejárás. Ez inkább azon programpontok elemzését részesítette előnyben, amelyek még ez eddigi analízis során még nem érintettünk. A kettő különbségeire példa lehet, hogy egy ciklus modellezése esetén amíg az első stratégia sorozatosan azt próbálja modellezni, hogy újra és újra belépünk a ciklusmagba, addig a második a ciklusmagba először belépés kiértékelése után megvizsgálja az alternatív lehetőséget, vagyis hogy mi a helyzet, ha rögtön a ciklus elérésekor sem teljesül a ciklusfeltétel.

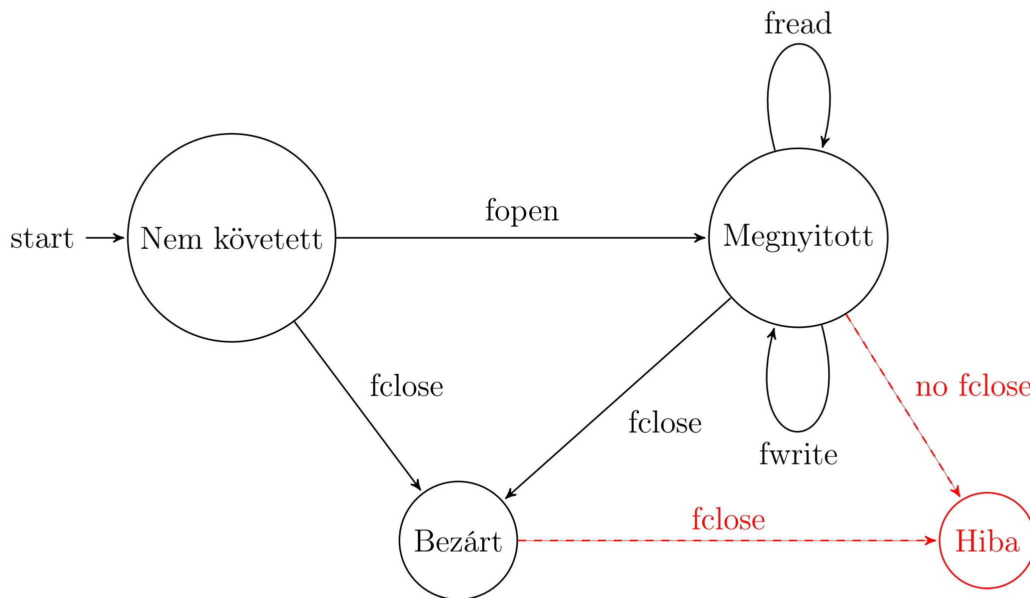
4. fejezet

Elemzés automatákkal

A statikus analízis eszközök absztrakciós szintjeinek tárgyalásánál láthattuk, hogy az egyre több információval rendelkező szintek egyre bonyolultabb detektálási problémák megfogalmazását teszik lehetővé, úgy hogy a hamis pozitív találatokat is minimális szinten tartják. Ezen kifejezőerő számlájára írható azonban az is, hogy megnő az ellenőrző logika implementációjának komplexitása, így a statikus analízis módszerek implementálásához a fejlesztőnek birtokában kell lenni az elemző keretrendszer működési modelljének, amely a megvalósító nyelv függvényében lehet imperatív, funkcionális, objektum-orientált. Általános esetben elmondható, hogy a rendszer összes lehetőségének kiaknázásához ez megkerülhetetlen, azonban speciális esetekben élhetünk olyan absztrakciókkal — megfigyelve a detektálási feladatok közös jellemzőit — amelyek nagyban megkönnyíthetik a leírást.

4.1. Erőforrás-jellegű problémák

Számos olyan hiba detektálására alkalmaznak statikus analízis módszereket, amelyek oka a futtatási környezet szolgáltatásainak nem megfelelő használata. Jellemzően a program egy jól meghatározott szabályok szerint felépített mintát sért meg azáltal, hogy az alkalmazásprogramozási felület (API) által biztosított lehetőségeket rosszul használja. Erőforrás-kezelő API-k esetén jellemzően megszorítások adódnak a használt szolgáltatás-objektumok számára, az ezeken bekövetkező események sorrendjére, illetve az egész szolgáltatás használatot követő állapotára. Abban az esetben, ha ezeknek nem felel meg a konkrét programozási megoldás, az valamilyen nemkívánatos következménnyel jár, mint erőforrás-túlhasználat (resource overuse), vagy versenyhelyzet (race condition). A nyelvi konstrukciók használata, a domain logikája, illetve az erőforrások kezelésének együttes komplexitása nehezen detektálható és reprodukálható hibákhoz vezethet. Az API által megkövetelt szerződések



4.1. ábra. Unix file API

minden végrehajtási úton való betartását praktikusan lehet kezelni a szimbolikus végrehajtás segítségével. Megalkottam két modellt az erőforrás-jellegű problémák vizsgálata során, az egyik a Unix filekezelő API, a másik a C memóriakezelő API.

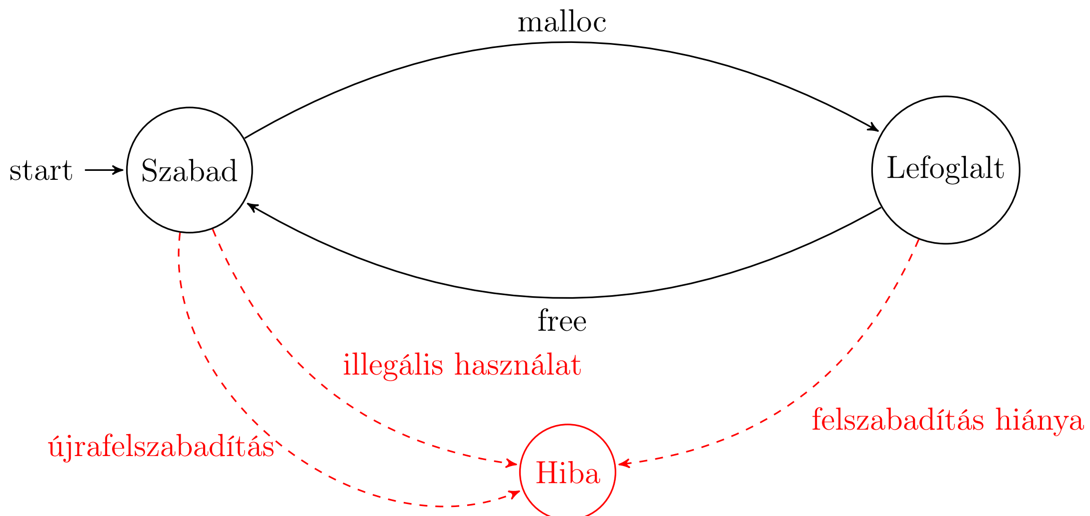
4.1.1. Fájlkezelés

Az erőforrás-jellegű problémák egyik példája a Unix rendszerek fájlkezelő API-ja. A fájlrendszer egyes elemeinek olvasásához, illetve írásához először szükséges az állomány megnyitása, csak ezután lehetséges a tartalomhoz való hozzáférés. Emellett az operációs rendszer előre defininált, véges számú állományhoz való hozzáférést támogat egyidejűleg. Ebből adódóan célszerű a használat után a megnyitott állományok bezárásáról gondoskodni, hogy a későbbiekben ne következzen be kiéheztetés a fájlrendszer használatában.

Bár, hogy a megnyitott állományokat reprezentáló file-leírók bezárásáról gondoskodni érdemes, az egyszer már bezárt leíró bármilyen használata nem definiált viselkedéshez vezet (undefined behaviour, UB), tehát a kétszeres lezárás is. Ebből a leírásból jól látható, hogy a probléma jól modellezhető egy olyan állapotgéppel, amelynek állapotait koncepcionálisan megfeleltetjük a kezelt fájlt azonosító leíró állapotaival, és átmeneteit a megfelelő API hívásoknak.

4.1.2. Dinamikus memóriakezelés

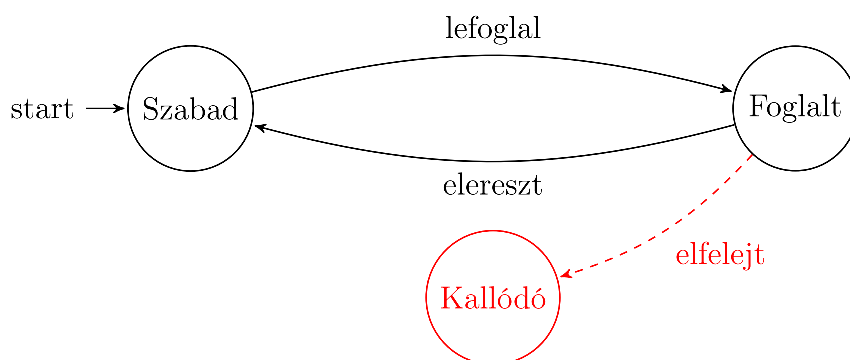
A program által használható, de a végrehajtási verem (runtime stack) limitációit megkerülő heap memória is erőforrásként viselkedik a legtöbb rendszer esetében. A



4.2. ábra. C malloc API

véges fizikai korlát mellett más, architekturális és szoftveres megszorítások is hozzájárulnak, hogy a memória mennyisége, illetve foglaltságának mintázata (vagyis hogy mennyire homogén a lefoglalt és szabad memória eloszlásának aránya) komoly tényezője a program helyességének. Sok programozási nyelv automatikus szemétgyűjtéssel veszi le a terhet a programozó válláról, amikor a dinamikus memóriakezelés problémája felmerül, ám teljesítménybeli megszorítások ellehetetleníthetik ezt a megközelítést. A dinamikus memória foglalása, és később a már nem használt memória állapotának explicit jelzése az operációs rendszer felé megfelelő implementáció esetén jobb futásideji, illetve tárkapacitásbeli hatékonyságot eredményezhet, viszont a használat után ezen explicit jelzés elmulasztása memória-szivárgáshoz (memory-leak) vezet.

A fájlrendszert bemutató példa nyomán, a második példa is a C nyelv egyik erőforrás-kezelő API-ja, a `malloc`, illetve `free` függvények. A memóriához a szabványos C API pointerváltozókon keresztül enged hozzáférni, olyan formában, hogy a `malloc` függvény paramétereként meg kell adni az igényelt memóriaméretet, és az operációs rendszer ezt a kérést vagy ki tudja szolgálni és egy nem nulla értékű memóriabeli címmel tér vissza, vagy nem tudja, és ekkor egy nullpointerrel. Ennek az esetnek több oka lehet, például kevesebb szabad memória van az igényeltnél, vagy nincsen egy egybefüggő memóriablokk, amely elérné a kívánt méretet. Ez jelenség a 4.2 ábrán ismertetett modellben azért nem jelenik meg, mert legtöbb esetben a memória allokálásának sikerességéről a szimbolikus végrehajtás nem tud érvelni, az tisztán futásideji feltételektől függő viselkedés. C++ nyelv esetén lehetőségünk



4.3. ábra. Erőforrások általános modellje

van saját allokátort használni. Ebben az esetben ezek a működési elvéről lehet egy hasonló modellt alkotni.

4.2. Erőforrások általános modellje

A fájlrendszer és a memória kezelése hasonlóságot mutat, és emellett még számos más programozási feladat is illeszkedő mintát követ. Ilyen lehet még a konkurens programozás területén a kölcsönös kizárás megvalósítása, mely minden szálkezelő könyvtárban megjelenik, vagy az operációs rendszerben a folyamatok ütemezésének komplex követelményei. A fenti két kifejtett, illetve egyéb említett példák alapján általánosítottam, és megalkottam az erőforrások helyes kezelésének programmodellét (4.3 ábra), legtöbb esetben adott egy olyan művelet, amely explicit jelzésként szolgál az erőforrás lefoglalására, és amely azt eredményezi, hogy konkrét erőforráspéldányt valamilyen azonosítható módon a programozó kezelésbe veszi — a legtöbb programozási nyelvben ez hozzáférési- (handle) objektum segítségével történik —, illetve adott az a művelet is, amely explicit módon jelzi az erőforrás használatának végét. Fontos, általános elv továbbá, hogy az explicit lezáró művelet elmulasztása hibajelenségnek tekinthető.

Az általános modell természetesen csak minimális, elvonatkoztat minden domain-specifikus használati esettől. Ennek ellenére eléggé jól megfeleltethető mindkét kifejtett példa az általános esetnek, olyan módon, hogy az általános modell minden állapotához és átmenetéhez találunk egy-egy illeszkedő állapotot, illetve átmenetet a speciális modellben. A filerendszer modelljében (4.1 ábra) például a `Szabad` állapotnak megfelel a `Nem követett`-nek, a `Foglalt` a `Megnyitott`-nak, a `Kallódó` pedig a `Hiba`-nak. Az átmenetek közül pedig a `lefoglal` feleltethető meg az `fopen`-nek, az `elereszt` pedig a `fclose`-nak. A memóriakezelés esetében értelemszerűen meg lehet feleltetni az állapotokat és az átmeneteket.

Az is látható a megfeleltetések vizsgálatakor, hogy a filekezelés esetében részletesebb a használati modell, vagyis az állapotok és átmenetek számossága nagyobb az általános modellhez képest. Azonban ezek a további elemek nem járulnak hozzá érdemben az erőforrás kezelésének jellegéhez, csak kiegészítő szabályokként jelennek meg. Ez nem egyértelmű megfeleltetés, és a domainek sajátos tulajdonságai azt eredményezik, hogy nem lehet egy konkrét, teljesen általános megoldást adni minden erőforrás-jellegű probléma detektálására, valamilyen mértékben mindig meg kell adni a lehetőséget a specifikus szabályok leírására.

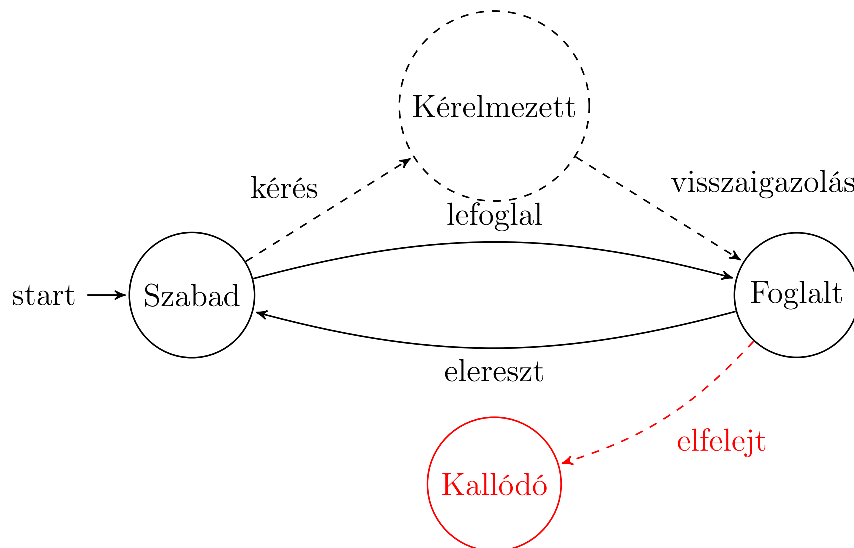
4.2.1. Kiterjesztések

Amikor a speciális esetek általános modellnek való megfeleltetését vizsgáltuk, a nem illeszkedő átmenetek, illetve állapotok az erőforrás-tulajdonság szempontjából nem voltak relevánsak. A filekezelés esetében, az a tény a modell megenged írási (`fread`) és olvasási (`fwrite`) műveleteket a `megnyitott` állapotban, amelyek nem befolyásolják az erőforrás állapotát (az átmenet kiinduló- és célállapota is ugyanaz), azt eredményezi, hogy az ilyen jellegű események modellezése nem kötődik szorosan az erőforrás-jelleghez, annak ellenőrzésével párhuzamosan, ortogonális módon történhet. Ugyanilyen jellegű tulajdonsága a konkrét modellnek az is, hogy lezárt file-leírót kétszer lezárni is hiba, ezt az erőforrás életciklusától függetlenül is lehetőségünk van ellenőrzni. Megintcsak ilyen példával állunk szemben akkor, amikor a memória-kezelés esetében azt az eseményt tekintjük hibának, amikor nem lefoglalt dinamikus memória tartalmára hivatkozik.

Ezekre úgy is tekinthetünk, mint az erőforrás-jellegű probléma domain-specifikus, vagy nem-strukturális kiterjesztéseire. Az eddigi esetekben nem volt példa olyan kiterjesztésre, ami ne ilyen jellegű lett volna. Egy más természetű módosítás látható a 4.4 ábrán.

Ebben a modellben az eredeti `lefoglal` átmenet mellett, vagy ennek alternatívájaképpen egy többlépéses lefoglalási folyamat szerepel. Ez lehet a 4.4 ábrán látható kétlépéses, vagy akár ennek komplexebb változata is, a lényeg, hogy nem feltétlenül kapja meg az igénylő az erőforrást rögtön azt követően, hogy azt igényelte. Az egyik lehetséges felhasználási területe ennek a modellnek a tranzakcionális memóriakezelés lehet, amely főleg párhuzamos programozás esetén nyújthat teljesítménybeli előnyöket a konvencionális kölcsönös kizáráson alapuló, szinkronizációs mechanizmusok alternatívájaként.

Párhuzamos programozás esetén azt, hogy a párhuzamosan futó szálak ne férjenek hozzá ugyanazokhoz az erőforrásokhoz egyidejűleg, az implementált vezérlési logika biztosítja. Hagyományosan több végrehajtási szál által is elért adatok elérésének



4.4. ábra. Tranzakcionális erőforrás-modell

megszervezése zárakon keresztül történik, például a kölcsönös kizárást megvalósító mutex. Az adatot éppen módosítani kívánó szál előbb megszerzi a zárat, majd írás után a zár birtoklásáról lemond. Minden esetben csak az írhatja az adatot, akinél a zár van. A zároláson alapuló szinkronizálás bizonyos problémákat okozhat, mivel a szálaknak várniuk kell a zárolt adatok frissítéséhez.

A tranzakciós memória a zár alapú szinkronizálást kiváltó megoldás lehet. Megpróbál egyszerűsíteni a párhuzamos programozást az olvasási és írási műveletek csoportosításával és egy művelet végrehajtásával. A tranzakciós memória hasonlít az adatbázis-tranzakciókra, ahol az összes megosztott memória elérés és azok hatásai együttesen kerülnek végrehajtásra (commit) vagy csoportként eldobhatók (rollback). Minden szál egyszerre léphet be a kritikus régióba, és ha konfliktusok vannak a megosztott memóriaadatok elérésekor, a szálak ismét megpróbálják elérni a megosztott memória adatait, vagy megállnak a megosztott memóriaadatok frissítése nélkül. Ezért a tranzakciós memóriát zármentes szinkronizálásnak is nevezik. A tranzakciós memória versenyképes alternatívát jelenthet a zár alapú szinkronizáláshoz.

Látható, hogy mind a domain-specifikus, mind a strukturális esetben az általános model kiegészítésre szorul. Ez a többlet-információ a statikus analízis keretrendszerek esetében modulárisan illeszthető szoftverkomponesek formájában kerül hozzáadásra. A Clang SA esetében erre lehetőséget az ellenőrző modulok (checker-ek) biztosítanak, és bár az eszköztámogatás adott, hogy akár az ellenőrző eszközbe fordításidőben, akár futás közben dinamikus betölthető módon hozzátegyük a domain-specifikus logikát, a megvalósítás módja minden esetben az imperatív paradigma szerinti implementáció. A már említett ASTMatchers könyvtár segít a szintaktikai információk alapján történő szűrési feltételek megfogalmazásában, de a szimbolikus

végrehajtás eredményeihez nincs olyan kifejező, deklaratív eszköz, mint az ASTMatchers. Ez a kiegészítés jelenleg is fejlesztés alatt áll a Clang fejlesztői közösség által, és ha elkészül, akkor integrálható lesz az állapot-automaták nézetén alapuló leírásba.

4.3. Detektálási problémák állapotgép-nézete

Tekintsük ezek után az állapotgép-leírás alkalmazhatóságát a konkrét statikus analízis problémák esetében. Említettem, hogy elsősorban az erőforrás-jellegű problémák esetében jól illeszkedő ez a fajta leírás, de mivel maguk az állapotgépek általános eszközként használhatóak, ezért más jellegű problémák esetében is hasznosak. Emellett, mivel az általános modell — a legtriviálisabb esetektől eltekintve — mindig kiterjesztésre szorul, szükséges hogy ezen domain-specifikus, vagy strukturális eltérések kifejtésére adjunk egy eszközt, amely megkönnyíti a fejlesztési folyamatot.

4.3.1. Checkerek megvalósítása hagyományosan

A statikus analízis problémák, melyek feltételezik mind a szintaktikai, mind a szemantikai információkat, hogy a hibajelenséget detektálják a Clang infrastruktúrában — azon belül a Clang SA keretrendszerben — egy checker formájában kerülnek implementációra. Itt a szimbolikus végrehajtás során a feltérképezett program-útvonalak mentén bekövetkező eseményekre írhat a detekció fejlesztője eseménykezelő rutinokat. Két valós problémára valódi találatokkal rendelkező megoldást is adtam BSc-s szakdolgozatomban [22]. A lehetőségek tárháza meglehetősen nagy, ahogy ezt a 4.1 kódrészlet is mutatja, illetve még ennél is nagyobb, mivel négy esetben láthatunk típusfüggő ősosztályt az öröklődési hierarchiában. Ezen esetekben úgy van megoldva az ellenőrzés, hogy az elfogadott típusparaméterek halmaza valamilyen szervezőelv mentén egy hierarchiát alkot, és az ellenőrzés csak a konkrét típusparaméternek megfelelő eseményeket, vagy ennek speciálisabb eseteit kezeli.

A megvalósított eseménykezelő függvényben van lehetőség további ellenőrzéseket végezni, illetve ha olyan pontra ér a detektálás, akkor a hibára figyelmeztető diagnosztikai üzenetet kibocsátani. Fontos jellemzője még a hibakezelésnek, hogy a hiba forráskódbeli helyének információtartalma magas, tehát szinte minden esetben kíséri a konkrét hibaüzenetet fájlnév-, sor- és oszlopinformáció is.

Az architektúrát tekintve, a checker bizonyos típusokkal felparaméterezett `Checker` osztálysablonból leszármazó osztályok. Ezen osztályok példányait a CheckerManager példányosítja, minden analízis futás esetében egy-egy példányban. Ebből következik, hogy mivel az analízis során a szimbolikus végrehajtás a lehetséges végrehajtási utak gráfjának (Control Flow Graph, CFG) teljesen független, akár

egymást kizáró pontjain is járhat, ezért a checkerek is állapotmentesek kell hogy legyenek, így elkerülve, hogy inkonzisztens legyen az ellenőrzött modell. Erre példa, hogy ha egy csak futásidőben meghatározható értéktől függ, hogy egy file-leírót bezárunk-e vagy sem, akkor a szimbolikus végrehajtás elágazik, és lesz egy olyan feltérképezett út, ahol megtörtént a lezárás, és egy, ahol nem. Ha ezen két út pontjain kiváltott eseményekre felváltva válaszolna a checker a saját fenntartott belső állapotával, akkor nem juthatnánk konzisztens eredményekre.

```
class CheckerDocumentation :
  public Checker< check::PreStmt<ReturnStmt>,
                  check::PostStmt<DeclStmt>,
                  check::PreObjCMessage,
                  check::PostObjCMessage,
                  check::ObjCMessageNil,
                  check::PreCall,
                  check::PostCall,
                  check::BranchCondition,
                  check::Location,
                  check::Bind,
                  check::DeadSymbols,
                  check::BeginFunction,
                  check::EndFunction,
                  check::EndAnalysis,
                  check::EndOfTranslationUnit,
                  eval::Call,
                  eval::Assume,
                  check::LiveSymbols,
                  check::RegionChanges,
                  check::PointerEscape,
                  check::ConstPointerEscape,
                  check::Event<ImplicitNullDerefEvent>,
                  check::ASTDecl<FunctionDecl> > {...}
```

Kódrészlet 4.1. Minden típusú eseményre reagálni képes checker öröklődési hierarchiája

Mivel Clang SA esetében nem kapunk arra garanciát, hogy egy programvégrehajtási utat végigjárunk, mielőtt egy alternatív út egyes elemeit érintenénk, és mivel nem készül a checker példányokból másolat minden elágazási ponton, ezért a checkerek saját belső állapotot nem tartanak fenn. Ehelyett egy olyan adatszerke-

zetet használnak, amelynek karbantartását a keretrendszer végzi, és garantálja, hogy minden írás és olvasás ezen olyan megfigyelhető eredménnyel jár, mintha minden bejárási út külön adatszerkezetet tartana fenn. Lehetőség van halmazok és asszociatív konténerek kifejezésére ezzel az eszköztárral, és valójában a tárolt értékek egy helyen kerülnek eltárolásra az éppen aktuális pontot számontartott szimbolikus értékek információival. Ennek az adatszerkezetnek a megnevezése: Generic Data Map (`GDM`).

4.3.2. Checkerek megvalósítása állapot-orientált módon

A hagyományos megvalósításból kitűnik, hogy miért van szükség a `GDM`-re. Ha egy checker nem rendelkezik belső állapottal és a programállapotoknak a bejárási utak mentén szervezett gráfját (Exploded Graph) sem képes címkézni, akkor az ellenőrző logika nem építhet arra, hogy a végrehajtási út egy előbbi pontján milyen szimbolikus értékekkel találkozott, tehát a csak lokális jellegű jelenségek detektálása lehetséges. Ez azonban pont az erőforrás jellegű problémák esetében okoz gondot. A `GDM` pont ennek a problémának a megoldására készült eszköz. Lehetőséget biztosít a checkerek megvalósítóinak, hogy bizonyos pontokon megcímkézzék az Exploded Graph-ot, majd később ellenőrizzék, hogy milyen értékeket használtak a címkékben.

Látható tehát, hogy a checkerek koncepcionálisan állapottal rendelkeznek, de a megvalósítás szintjén ez implicit módon jelenik csak meg, a `GDM` használata által. Ezáltal a programozási feladat komplexitása megnő az állapot ábrázolásából adódó indirekció miatt. Különösen nagy nehézséget jelent ez azoknak, akik a domain ismeretében, de szimbolikus végrehajtás részletes ismeretének hiányában szeretnének megfogalmazni ellenőrző logikát. Az erőforrások állapot-orientált, explicit leírásának tehát az lehet a haszna, hogy közvetlenebb módon lehet kifejezni a hibaeseteket.

4.4. Statikus analízis állapotgép-leíró nyelve

A statikus analízis állapotgép nézetének leírását egy domain-specifikus nyelv definiálásával segítem. A nyelv nem általános állapotgépek leírására használatos, arra számos programozási paradigmában, nyelven és segédkönyvtárban kész megoldások adottak. Az általam adott nyelv (checkerlang) speciálisan Clang SA checkerek állapot-orientált leírására alkalmas, és fő motivációja, hogy technikai eszközökkel is megkönnyítse a statikus analízis probléma leírását, akár a Clang SA belső működésében kevésbé otthonosan mozgók számára is. A jelenlegi definíció lehetőségei nem a végleges állapotot tükrözik, az egyes nyelvi elemek bemutatása során kitérek a tervezett jövőbeli kiterjesztésekre. A nyelv egyszerűsített ABNF (Augmented BNF) [24] leírása a 4.2 kódrészleten látható.

```
root = checkerName *transitionSpecification
checkerName = "checker" id
transitionSpecification =
  transitionHead transitionBody transitionTail
transitionHead =
  "{" (constructorHead / stateChangeHead) "identified" "by" identityExpr"}"
constructorHead = "Construct" id
stateChangeHead = "Transition" id "from" id
identityExpr =
    "RESULT" "(" headExprSpecifier ")" / "PARAM" "(" headExprSpecifier ","
        num ")"
headExprSpecifier = id
transitionBody = *symbolicExtractorStmt
symbolicExtractorStmt = symbolicAssignment / symbolicAssertion /
    diagExpression
symbolicAssignment = id "=" bodyExpr
bodyExpr = "RESULT" "(" bodyExprSpecifier ")" / "PARAM" "("
    bodyExprSpecifier "," num ")"
bodyE
symbolicAssertion = id ("==" / "!=") id
transitionTail = "{" id ["," diagnosticExpression] "}"
diagnosticExpression = "diag(" diagMessage ")"
diagMessage = 1*( str / num / id )

id = ALPHA 1*(ALPHA / DIGIT)
str = 1*(ALPHA)
num = 1*(DIGIT)
```

Kódrészlet 4.2. A checkerlang nyelv egyszerűsített ABNF leírása

4.4.1. Szintaxis elemzése

A nyelv segítségével meg lehet adni a problémát leíró állapotgépet. Ez a leírás az átmenetekből indul ki, ezek explicit módon jelennek meg, míg az állapotot csak implicit módon kerülnek felvételre, azáltal, hogy az átmenet-specifikációk megfelelő helyein említésre kerülnek. A statikus analízis problémák, és speciálisan az erőforrás-jellegű problémák esetében ez tűnik célravezetőnek, hiszen az imperatív kódok ilyen jellegű hibái is jellemzően speciális utasítások bizonyos sorrendiségéből adódnak. A leírásból látszik a nyelv egyetlen forrásfájlja megfeleltethető egy checkernek. A nyelv

első kötelező eleme a checker nevének deklarációja, majd ezt követi a checker által számontartott állapotgépek leírásai, olyan módon, hogy átmenet-specifikációkat adunk meg. Fontos sajátosság, csak olyan szimbolikus értékekre értelmezett állapotautomatákat szeretnénk modellezni, amelyek relevánsak a probléma szempontjából. Ezt úgy érhetjük el, hogy kétféle állapotátmenet-kategóriát különböztetünk meg, az új állapotgép megkonstruálását, illetve a már létező gépek állapotváltozásait. Ennek a különbségnek a nyelvtanban is nyoma van, a `transitionHead` nem-terminális szimbólum produkciós szabályai között a `constructorHead` és `stateChangeHead` szabályok alternatívákként szerepelnek. A konstruáláskor és az állapotváltozáskor is egy adott gép egy adott példányával foglalkozunk. Tehát egy checker akár több különböző névvel ellátott gépnek több különböző szimbolikus értékkel azonosított példányát tartja számon. Ez lehetőséget biztosít arra, hogy egy checkeren belül akár más és más szempontok alapján is modellezzünk, és arra is van lehetőség, hogy egy adott nevű gépnek több példányát is kövessük. A különböző nevű gépek különböző viselkedési aspektusokat tudnak modellezni, míg az egyes példányok biztosítják az esetszintű multiplicitást.

Az egyes konkrét gépeket így a checker neve, a gép azonosítója, illetve modellezett szimbolikus érték együttesen határozza meg. A konkrét szimbolikus érték azonosítására szolgál a `transitionHead` szabály részét alkotó `identityExpr`, mely szimbolikus érték típusú kifejezésnek tekinthető. A Clang SA esetén a szimbolikus értékek egyszer létrejönnek, és soha nem változnak. Ennek eredményeképpen a szimbolikus értéket végig lehet követni élete folyamán, és ha egy változó értékadás keretében új értéket kap, akkor új szimbolikus érték lesz hozzárendelve. Ez a működés garantálja, hogy egy erőforrás-lefoglaló API hívás által megadott szimbolikus értéket akkor is követni tudunk, hogy ha már nem az eredeti változóhoz köthetően következnek be rajta események. A nyelv ismertetett formájában támogatja a függvényhívás kifejezések visszatérési értékének, illetve valamelyik paraméterének szimbolikus kinyerését, illetve összehasonlítását más szimbolikus értékekkel. Az egyenlőség, illetve egyenlőtlenség-vizsgálat támogatott. A keretrendszer támogatja az egész reprezentációval rendelkező nyelvbeli értékek modellezését, és ezeken jónéhány bináris predikátum kiértékelését. Erre a nyelv is képes a `symbolicAssertion` szabály további elemekkel való analóg kiterjesztése után, amire most az egyszerűség kedvéért nem tértem ki.

```
checker StreamChecker

{ Construct machine identified by RESULT(fopen) }
{ Acquired, diag("File left open") }
```

```
{ Transition machine from Acquired identified by PARAM(fclose, 1) }
{ Released }

{ Transition machine from Released indentified by PARAM(fclose, 1) }
  diag("File closed twice")
{ DoubleClosed }
```

Kódrészlet 4.3. Fájlkezelési példa checkerlang nyelven

```
checker StreamChecker

{
  --> Acquired;
  ID: RESULT(fopen)
}

{
  Acquired -->;
  diag("File left open")
}

{
  Acquired --> Released;
  ID: PARAM(fclose, 1)
}

{
  Released --> DoubleClosed;
  ID: PARAM(fclose, 1);
  diag("File closed twice")
}
```

Kódrészlet 4.4. Fájlkezelési példa explicit formája

A szimbolikus értékek kinyerése során a `RESULT` egy függvényhívás kifejezést, a `PARAM` szabály pedig egy függvényhívás kifejezés részkifejezését — a megfelelő sorszámú aktuális paramétert — azonosítja. A szimbolikus végrehajtás során lehetőségünk van tetszőleges kifejezésről elkérni hozzá tartozó, számontartott szimbolikus értéket, tehát a felsorolt két szabály nem tér el lényegesen, mert mindkettő egy kifejezést je-

löl ki. Ennek megfelelően természetes általánosítási lehetőség a nyelv szintjén, hogy tetszőleges kifejezés kijelölhető legyen a szimbolikus végrehajtás által aktuálisan vizsgált kifejezés mellett. Egy ilyen eszköz lehet az ASTMatchers könyvtár. Hogy ez használható legyen, a checkerek korábban említett lehetőségei közül leginkább éppen a típusparaméterezett lehetőségek között találunk megfelelő eseménykezelőket.

A szimbolikus végrehajtás során a keretrendszer egy utasítás kezelésére két alkalmat is kínál. Egyszer a modellezett végrehajtás előtt közvetlen, illetve egyszer azt követően. Ennek azért van jelentősége, mert bizonyos kifejezések kiértékelésének szemantikája mellékhatásos, és nem garantált, hogy például egy függvényhívás szimbolikus végrehajtásakor a paramétereknek megfelelő részkifejezések értéke egyáltalán hivatkozható még akkor is, amikor visszatértünk a függvényhívásból. A modellezett nyelv szemantikája is közrejátszik abban, hogy van-e értelme különbséget tenni az utasítás eleji, illetve végi ellenőrzés között. Erre példa a C++ nyelv 11-es szabványában megjelenő move szemantika, mely lehetőséget ad arra, hogy egy értéket egyedi módon, másolás nélkül mozgassunk a végrehajtás során egyik változóból a másikba. Ebben az esetben, ha egy érték bemozog a függvény lokális változójába, már a függvényhívás pillanatában sem garantált semmi annak a változónak a tartalmára nézve, amiből az érték ki lett mozgatva. A `PreStmt<T>`, illetve `PostStmt<T>` osztálysablonokból leszármazó checkerek képesek, hogy a `T` típusparaméter által meghatározott utasításokon (`Stmt` osztály és leszármazottai) keresztülhaladó szimbolikus végrehajtás során ezeket kezeljék. `T` típus egy `Stmt` altípus. A checker által megvalósított eseménykezelő függvény pedig olyan állításokat fog kezelni, melyek ennek a `T` típusnak altípusai. Az események kezelése közben lehetőség adódik az állítás szimbolikus értékének kinyerésére. Ahhoz, hogy tetszőleges kifejezés értékét meg tudjuk kapni, elég az említett két eseménykezelő (callback) esetén T típust `Expr`-nek választani, mert ennek az osztálynak őse az utasításokat szimbolizáló `Stmt` osztály (közvetett módon, a `ValueStmt` osztályon keresztül). A checkerlang nyelv könnyen kiterjeszthető ezen megfontolásoknak köszönhetően egy olyan résznyelvvel, amely egy az egyben követi az ASTMatchers szintaxisát, és szemantikailag is illeszkedő módon viselkedik.

A checker törzsében nem csak a szimbolikus értékek felett értelmezett predikátumok kiértékelésére van lehetőségünk, hanem az állapotgép-példányt azonosító szimbolikus érték mellett más szimbolikus értékek állapotgép-változóban történő tárolására is. Ennek szerepe, hogy ki tudjunk fejezni az állapotgép állapotán kívül más állapotot is, vagyis egy címkézett állapotgép kifejező erejével ruházzuk fel a modellezést. Ezek a szimbolikus értékek ezután más átmenetekben hivatkozhatóak, összehasonlíthatóak, vagy akár felül is írhatóak. A változók definiálására a `transitionBodyStmt` szabály `symbolicAssignment` alternatívája ad lehetőséget.

Ezek a változók, ugyanúgy mint a gép-példány azonosítására szolgáló érték, a már említett GDM-ben kell, hogy tárolásra kerüljenek a checker-osztályok állapotmentessége miatt. Az azonosításra használt szimbolikus érték egy állapotgép példány esetében minden esetben meghivatkozható ID néven, ám nem módosítható. Ezt a kitételt a szintaxis nem garantálja, szemantikus szinten kerül ellenőrzésre.

A modellezés és detektálás mellett a checker legfontosabb funkciója, hogy a detektált hibáról diagnosztikát tud kibocsátani. Ezt jelöli a nyelv diagExpr szabálya. Érdekesség, hogy ez a szabály két helyen is előfordulhat, a transitionBody-ban, illetve a transitionTail-ben is. Ennek oka, hogy két jellegű mulasztás esetén szeretnénk diagnosztikát kibocsátani, amikor aktív módon detektáljuk a hibát, illetve, amikor a passzív mulasztás ténye beigazolódik. Ennek megfelelően a törzsben szereplő diagnosztika fogja jelölni az aktív, míg a zárórészben a passzív okokkal rendelkező hibákat. Hogy ez a működés explicitté váljon létezik a nyelvnek egy köztes reprezentációja is, amely képes a diagnosztika szempontjából előálló helyzetet explicit formában ábrázolni, azonban ez csak köztes reprezentációként, és nem a nyelv fő formájaként szerepel, mert kevésbé kompakt leíráshoz vezet. A már ismertetett fájlkezelési probléma (4.1 ábra) checkerlang nyelven történő leírását a 4.3 kódrészlet mutatja, illetve ennek az említett alternatív formája a 4.4 kódrészleten látható.

4.4.2. Szemantika

A nyelv által leírt megoldás szemantikáját egy olyan checker működésével adhatjuk meg, amely tartalmaza a PostStmt<Expr>, PostStmt<Expr>, és a DeadSymbols események kezelését megvalósító callback-eket. A checkerlang leírásban minden előforduló gépre van egy a checker és a gép nevével együttesen azonosított bejegyzés, amely egy lista formájában az egyes gépek példányainak adatait tartalmazza. A definiált átmeneteknek megfeleltetett kezelő kódok rendezett sorrendben kerülnek legenerálásra, olyan módon, hogy előbb kerüljenek kezelésre a létező állapotok közötti átmenetek, és csak ezt követően az új állapotgépek felvételei. Fontos szempont, hogy egy szimbolikus értéke csak egyszer illeszkedhet egy callback kiértékelése során, ellenkező esetben duplikált eseményeket detektálna az implementáció helyes megvalósítás esetében is.

A nyelvben szereplő predikátum-kiértékelések hivatkozhatnak egy speciális, impliciten mindig elérhető szimbolikus értékre ID név alatt. Ha egy predikátum hamisra értékelődik ki, akkor az átmenet meghiúsultnak minősül. Ha bármikor ismeretlen lenne a predikátum kiértékelés eredménye és nem egyértelműen igaz, vagy hamis, akkor nem tekintjük illeszkedőnek az átmenetet, és nem történik meg sem a diagnosztika kibocsátása, sem az állapotátmenet. Az állapotátmenetek közül, a szimbo-

likus értékek életének végét kezelők a `DeadSymbols` callbackben kerülnek kezelésre. Itt az összes már nem kezelt szimbólum közül a modellezettek esetében történik egy diagnosztika kibocsátás abban az esetben, ha hibás állapotban vannak. Mind a konstruktor, mind az állapotváltozás átmenetek esetében vagy a `PreStmt<Expr>`, vagy a `PostStmt<Expr>` callback a megfelelő hely az ellenőrzésre, attól függően, hogy milyen típusú a kifejezésből a szimbolikus értéket kinyerő checkerlang-beli utasítás. Speciálisan a függvény visszatérési értékét jelölő `RESULT` esetében a szimbolikus végrehajtás akkor ad jó eredményt, ha `PostStmt<T>` callback-el ellenőrzünk, `PARAM` esetén pedig ha a `PreStmt<T>`-t használjuk. Az esetleges kiterjesztések implementációjakor erre a megkülönböztetésre figyelni kell. Ha egy átmenetben többféle callback-hez tartozó szimbolikus érték kinyerést végzünk, akkor ezek egymáshoz való viszonyát rögzíteni kell, és ekkor a szimbolikus végrehajtás során számon kell tartani az ilyen közös átmenethez tartozó ellenőrzés-párokat, amelyek különböző callback-ekbe kerültek. Ennek teljes tisztázása további vizsgálatot igényel.

4.4.3. Parsolás PEG-ként

A checkerlang nyelvhez implementálható egy parser, mely képes a szintaktikai és szemantikai szabályoknak megfelelő checker-kódot implementálni. Az ABNF leírás segítségével megadott nyelvtan egy generatív, környezetfüggetlen (Chomsky 2-es) formális nyelvtan, melynek felismerési problémájára több algoritmus is ismert. A Parsing Expression Grammar (PEG) forma előnye, hogy kiküszöböli a nyelvtanban lévő nem-egyértelmű produkciós szabályok problémáját azáltal, hogy ha kettő ilyen szabály is illeszkedne a parse-olás során, akkor sorrendi prioritás alapján, az először előfordulót részesíteni előnyben. Egy lehetséges implementációt elkészítettem PEGTL parser-kombinátor könyvtár [23] segítségével, mely az egyes produkciós szabályokat C++ sablonosztályok segítségével valósítja meg, és ezek kompozícióját pedig ezen osztályok közötti öröklődéssel fejezi ki. Az általam definiált nyelvet felismerő megvalósítás a 4.5 kódrészleten látható, melyen nem tüntettem fel a szóközök kezelését végző szabályokat az olvashatóság kedvéért.

```
struct checkerName;
struct transitionSpecification;
struct root : seq<checkerName, star<transitionSpecification>> {};
struct id;
struct checkerName : seq<ISTR("checker"), id> {};
struct transitionHead;
struct transitionBody;
struct transitionTail;
```

```
struct transitionSpecification
    : seq<transitionHead, transitionBody, transitionTail> {};

// Head-specific rules.

struct constructorHead;
struct stateChangeHead;
struct identityExpr;
struct transitionHead
    : seq<one<'{'>, sor<constructorHead, stateChangeHead>, ISTR("identified"),
        ISTR("by"), identityExpr, one<'}'>> {};
struct constructorHead : seq<ISTR("Construct"), id> {};
struct stateChangeHead : seq<ISTR("Transition"), id, ISTR("from"), id> {};
struct headExprSpecifier;
struct num;
struct identityExpr
    : sor<seq<ISTR("RESULT"), one<'('>, headExprSpecifier, one<')'>>,
        seq<ISTR("PARAM"), one<'('>, headExprSpecifier, one<','>, num,
            one<')'>>> {};
struct headExprSpecifier : id {};

// Body-specific rules.

struct symbolicExtractorStmt;
struct transitionBody : star<symbolicExtractorStmt> {};
struct symbolicAssignment;
struct symbolicAssertion;
struct diagnosticExpression;
struct symbolicExtractorStmt
    : sor<symbolicAssignment, symbolicAssertion, diagnosticExpression> {};
struct bodyExpr;
struct symbolicAssignment : seq<id, one<'='>, bodyExpr> {};
struct bodyExpr
    : sor<seq<ISTR("RESULT"), one<'('>, bodyExprSpecifier, one<')'>>,
        seq<ISTR("PARAM"), one<'('>, bodyExprSpecifier, one<','>, num,
            one<')'>>> {};
struct symbolicAssertion : seq<id, sor<STR("=="), STR("!=")>, id> {};
```

```
// Tail-specific rules.

struct transitionTail
    : seq<one<'{'>, id, opt<seq<one<','>, diagnosticExpression>>, one<'}'>> {};
struct diagMessage;
struct diagnosticExpression
    : seq<ISTR("diag(">, diagMessage, one<')'>> {};
struct str;
struct diagMessage : plus<sor<str, num, id>> {};

// Generic rules.

struct id : seq<ALPHA, plus<sor<ALPHA, DIGIT>>> {};
struct str : plus<ALPHA> {};
struct num : plus<DIGIT> {};
```

Kódrészlet 4.5. PEGTL parser

5. fejezet

Összefoglalás

Diplomamunkámban ismertettem a modern szoftverfejlesztés eszköztárának fontosabb elemeit. Összehasonlításra került a statikus és a dinamikus kódelemzési stratégia, külön hangsúlyt fektetve a statikus elemzés jellegzetességeire. Emellett több példán keresztül bemutatásra került az eszköztár, illetve a modellezési módszerek, amelyek ezt lehetővé teszik. Bemutattam az *LLVM* és a *Clang* eszköz lehetőségeit. Egy olyan leírását adom a statikus analízis problémának, amely az egyes domainek logikáját praktikus és tömör módon teszik kifejezhetővé, mindezt a szimbolikus végrehajtást végző keretrendszer részleteitől lehető leginkább elvonatkoztató módon. A Clang SA keretrendszer által kifejezhető megoldások egy kellően bő részhalmazát teszi elérhetővé az állapotgép-orientált leírás. Ezáltal csökkenhet statikus analízis megoldások belépési küszöbe, több fejlesztő implementálhat ellenőrzéseket az egyszerű, kifejező és rugalmasan kiterjeszthető checkerlang nyelv segítségével.

A nyelv számos továbbfejlesztési lehetőséggel rendelkezik. A konkrét parsolási megvalósítások területén érdemes megvizsgálni, hogy az egyes lehetséges alternatívák milyen sebesség-beli, kifejezőerő-beli és integrálhatósági karakterisztikákkal rendelkeznek. Speciálisan a PEG forma parsolása esetén vizsgálandó a visszalépések számának minimalizálása. A nyelv parsolása lehetséges lehet a checker futása közben is, így lehetséges lenne egy általános checker implementációja, amely az ellenőrzés alatt futás közben parsolhatná a működését definiáló nyelvi leírást, ezzel az interpretált nyelvek futtatásához hasonló viselkedést tudnánk elérni.

A statikus forráselemzés a modern szoftverfejlesztés szerves részévé vált, a kódminőség javítása érdekében érdemes automatizált futtatóeszközök által élni a lehetőségeivel. Sok hiba felfedezésre kerül, amelyeket lehetetlen, vagy igen körülményes lenne a futásidejű elemzések segítségével megtalálni. Kifejezőerejük mellett, illetve pont emiatt a statikus analízis eszközök komplexitása magas, az állapotgép-orientált nézet ezen a komplexitáson képes enyhíteni, ezzel segítve a kódminőséget javító meg-

oldások szélesebb körben történő elterjedését. Megalkottam egy DSL-t, amellyel az állapotgép-leírás tömören és rugalmasan megvalósítható, mégis nyitott a kiterjesztésre. Megvalósítottam ennek a nyelvnek egy parser-ét a PEGTL könyvtár segítségével, és könnyedén megvalósítható a kódgenerálás is a parse-olt forma birtokában.

Irodalomjegyzék

[1] Makan Pourzandi *Formal Verification and Validation of UML 2.0 Sequence Diagrams using Source and Destination of Messages.* 2009 https://doi.org/10.1016/j.entcs.2009.09.064 2019.05.11

[2] Bücker H.M., Petera M., Vehreschild A. (2008) *Code Optimization Techniques in Source Transformations for Interpreted Languages.* In: Bischof C.H., Bücker H.M., Hovland P., Naumann U., Utke J. (eds) Advances in Automatic Differentiation. Lecture Notes in Computational Science and Engineering, vol 64. Springer, Berlin, Heidelberg

[3] Javascript nyelvi transzformációk. https://thenewstack.io/javascript-transpilers-need-know/ 2019.05.11

[4] Chaos Monkey. https://insights.sei.cmu.edu/devops/2015/04/devops-case-study-netflix-and-the-chaos-monkey.html 2019.05.11

[5] Google Test https://en.wikipedia.org/wiki/Google_Test

[6] LLVM honlap. http://llvm.org 2019.05.11

[7] LLVM könyv http://www.aosabook.org/en/llvm.html 2019.05.11

[8] Egyszerűsített fordítómodell. http://www.aosabook.org/images/llvm/SimpleCompiler.png 2019.05.11

[9] Chris Lattner weboldala. http://nondot.org/~sabre/ 2019.05.11

[10] Clang hivatalos felhasználói dokumentáció. https://clang.llvm.org/docs/UsersManual.html 2019.05.11

[11] Clang driver leírás. https://clang.llvm.org/docs/DriverInternals.html 2019.05.11

[12] Clang fordítóeszközök. https://clang.llvm.org/docs/Toolchain.html 2019.05.11

[13] Clang driver szerkezet. https://clang.llvm.org/docs/_images/DriverArchitecture.png 2019.05.11

[14] Clang SA weboldal. http://clang-analyzer.llvm.org 2019.05.11

[15] Clang SA XCode integráció. http://clang-analyzer.llvm.org/images/analyzer_xcode.png 2019.05.11

[16] Clang SA HTML nézet. http://clang-analyzer.llvm.org/images/analyzer_html.png 2019.05.11

[17] LibClang dokumentáció. https://clang.llvm.org/doxygen/group__CINDEX.html 2019.05.11

[18] Clang könyvtárak dokumentáció. https://clang.llvm.org/docs/Tooling.html 2019.05.11

[19] CodeChecker kódbázis. https://github.com/Ericsson/codechecker 2019.05.11

[20] LLVM felhasználói dokumentáció. http://llvm.org/docs/GettingStarted.html 2019.05.11

[21] Clang SA checker dokumentáció. http://clang-analyzer.llvm.org/checker_dev_manual.html 2019.05.11

[22] Statikus elemzések tervezése Clang eszközzel. Fülöp Endre, BSc szakdolgozat 2017

[23] Parsing Expression Grammar Template Library. https://github.com/taocpp/PEGTL 2019.05.11

[24] ABNF RFC specifikáció https://tools.ietf.org/html/rfc5234 2019.05.11