

Gráfok generálása nyelvfüggetlen szintaxisfák alapján

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

PROGRAMOZÁSI NYELVEK ÉS
FORDÍTÓPROGRAMOK TANSZÉK



Szerző:

Brunner Tibor

Témavezető:

Szűgyi Zalán
egyetemi tanársegéd

2012

Tartalomjegyzék

1. Bevezetés	2
2. Irodalmi áttekintés	4
2.1. Hasonló eszközök	4
2.1.1. Dr. Garbage Tools	4
2.1.2. Architexa	5
2.1.3. NDepend	6
2.2. Absztrakt szintaxisfa (AST)	8
3. Általános gráfépítés	9
3.1. Bevezető példa	9
3.2. Szabályok	10
3.3. A gráfépítés algoritmusai	13
3.4. A nyelvfüggetlenség biztosítása	18
3.4.1. Szekvencia	19
3.4.2. <code>while</code> és <code>do-while</code> ciklusok	19
3.4.3. <code>for</code> ciklus	20
3.4.4. Elágazás	20
3.4.5. Függvény	21
4. Felhasználási lehetőségek	22
4.1. Folyamatábra diagram	22
4.1.1. Fordítási egység	23
4.1.2. Függvény	23
4.1.3. <code>while</code> ciklus	24
4.1.4. <code>for</code> ciklus	25
4.1.5. Elágazás	27
4.1.6. Egyéb	28
4.2. Függőségi diagram	29
5. Összefoglalás	31
Irodalomjegyzék	32

1. fejezet

Bevezetés

A szoftverek gyors fejlődése következtében a szoftverfejlesztés technológiává nőtt ki magát. A szoftver egy termék, amit határidőre, adott költségekkel, jó minőségben kell elkészíteni. A szoftverfejlesztési folyamat lezárását követően azonban a terméket karban kell tartani, ami a kódsorok százezres nagyságrendű száma miatt egyáltalán nem könnyű feladat.

Az ipari méretű kódok kezelésének megkönnyítésére különböző szoftverek léteznek. Az olyan eszközöket, amelyek a program futtatása nélkül, pusztán a forráskód elemzésével nyernek ki információt az alkalmazás felépítéséről, statikus elemzőknek nevezzük [1]. A statikus elemzők célja sokféle lehet: a program helyességének vizsgálata [2, 3], hatáselemzés [4, 5, 6], elérhetetlen kódrészek felderítése [7] stb. A felsorolt funkciókkal több eszköz is rendelkezik, amelyek rendszerint a kód fejlesztésében és a szoftver megértésének megkönnyítésében játszanak szerepet.

Amikor karbantartási munkálatokról van szó, akkor olyan szoftverekre van szükség, amelyek a programjaink átláthatóságát és könnyen kezelhetőségét támogatják. Az összetett kódhalmazok megértését nagyban segítik a vizuális megjelenítések különféle módjai [8]. Egy ilyen mód az, ha a programjaink struktúráját különböző gráfokon, például vezérlési szerkezet diagramon, vagy függőségi gráfon keresztül szemlélítjük. A megértés szempontjából a fentiekén túl a rendszerek más aspektusai is hasznosak lehetnek, tehát érdemes ezek előállításának költségeit minimálisra csökkenteni.

Magából a forráskódból nehézkes az ilyen gráfok, nézetek előállítása, ezért célszerű elkészíteni a forráskód egy közbülső reprezentációját, az absztrakt szintaxisfát (AST), ami bemenete lehet e gráfkészítő algoritmusoknak. Azonban az egyes programozási nyelvek szintaxisfái más-más struktúrával rendelkeznek, sőt, ugyanazon nyelv elemzői is különböző AST-ket állíthatnak elő [9]. Ha egy olyan eszközt szeretnénk

készíteni, amely több programozási nyelvet is támogat, akkor szükséges egy olyan módszert definiálni, amely segítségével egységes módon tudjuk kezelni a szintaxis-fákat.

A forgalomban lévő statikus kódelemző szoftverek általában két ponton kötik meg a programozók kezét. Az egyik, hogy csak néhány előre meghatározott diagramtípust biztosítanak, amik esetleg nem fedik le azokat a feladatokat, amire egy ilyen eszközt használni szeretnénk. A másik pedig az, hogy rendszerint ezek az eszközök csak egy konkrét programozási nyelvet támogatnak.

Dolgozatomban egy olyan könyvtárcsomag működési elveit mutatom be, amely nyelvfüggetlen módon képes a felhasználó által megadott szabályok szerint felépíteni a kívánt gráfot a program AST-je alapján. Ehhez elkészítettem egy olyan algoritmust, amely előállítja a programnak azt a nézetét, amelyet a megfogalmazott szabályokkal definiálunk. A nyelvfüggetlenséget egy interfészen keresztül biztosítjuk. Ez az interfész csomagoló osztályok gyűjteménye, ami az adott programozási nyelvhez igazítva teljes mértékben lecserélhető.

A gráf a memóriában épül fel, melynek eltárolása az utolsó lépés. A kimenetnek is több alternatívája képzelhető el: Tárolhatjuk szövegfájlként XML [10] formátumban, elmenthetjük a széles körben használt GraphViz [11] alkalmazás bemeneteként – amely eszköz segítségével tetszőleges képfájlt generálhatunk –, vagy az egész könyvtárcsomagot egy statikus kódelemzőbe beépítve is hasznosíthatjuk.

A módszerem implementálásra került és az integrációja egy, az egyetemen futó kódmegértést támogató eszközbe folyamatban van.

2. fejezet

Irodalmi áttekintés

Ebben a fejezetben bemutatunk néhány meglévő eszközt, amely kapcsolódik a dolgozatunk témájához, majd a dolgozat megértéséhez szükséges fogalmakat részletezzük.

2.1. Hasonló eszközök

Több eszköz található, amelyek más fejlesztő környezetbe beépülve, vagy önállóan működve statikus kódelemzést végeznek, és az eredményeket grafikusán ábrázolják. A következőkben néhány ilyen eszközt vizsgálunk meg.

2.1.1. Dr. Garbage Tools

Ez az Eclipse-be épülő eszközgyűjtemény [12] három részből áll: *Bytecode visualizer*, *Sourcecode visualizer* és *Control Flow Graph Factory*. Az első két eszköz közös vonása, hogy a forráskód melletti ablakban, a kódsorokkal egy vonalban képesek megjeleníteni a megfelelő Java kódból generált folyamatábra diagramokat. Ez a megjelenítési mód hasznos lehet például hibakeresés közben, amikor egy algoritmus hibáját próbáljuk megtalálni, hiszen láthatóvá válik, hogy hová fog ugrani adott esetben a vezérlés egy *breakpoint* átlépése után.

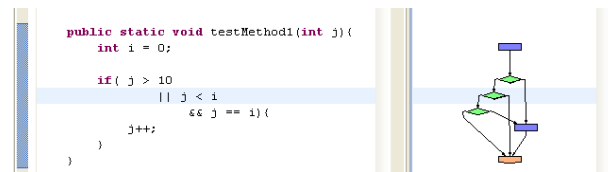
A *Bytecode visualizer* segítségével azonban nem csak a forráskód jeleníthető meg diagramként, hanem a *.class* fájlba fordult bájtkód is, hiszen az Eclipse ezeket is meg tudja nyitni, és ki tudja olvasni a tartalmukat. Ez rendkívül praktikus lehet, ha egy olyan programot szeretnénk elemezni, amelyhez nem biztosították a forráskódot, csak a bájtkódot.

Az eszköz alkalmas még az egyes függvények komplexitásának azonnali felméré-

sére, vagy egy összetettebb elágazás feltételrendszerének áttekintésére. Például, ha arra vagyunk kíváncsiak, hogy mely feltételek teljesülése szükséges a `j` változó megnöveléséhez a 2.1. ábrán, akkor elegendő csak azt megnézni, hogy mely élek vezetnek be a `j++` utasításhoz tartozó csúcsba, és az ezen élek mentén visszakövetni azt az utat, ami a feltétel csúcsokon keresztül vezet.

A *Control Flow Graph Factory* is vezérlési diagramokat állít elő, azonban ez már nem a kódsorokkal egy sorban jeleníti meg a kimenetet, hanem önállóan létező csúcs- és élcímkezett gráfokat állít elő mind a bájtódkból és a forráskódból. A csúcsok címkéi azon kódsorok száma, amely a csúcsnak megfelelő utasítást tartalmazza, ehhez a bájtódk nézet esetén a megfelelő bájtódk is konkatenálódik. Az élek többek közt az elágazás típusú utasításoknál címkéződnek. Egy `switch` szerkezetnél például a `case` ágakból adódik az élek felirata.

Az eszkögyűjtemény használati eseteinek sora bővebb a fent említetteknél, de ezek közös tulajdonsága, hogy csak Java nyelvű (bájt)kódon lehet futtatni, valamint csak a program vezérlési folyamatát áttekintő diagramon keresztül lehet szemlélni a kódot.

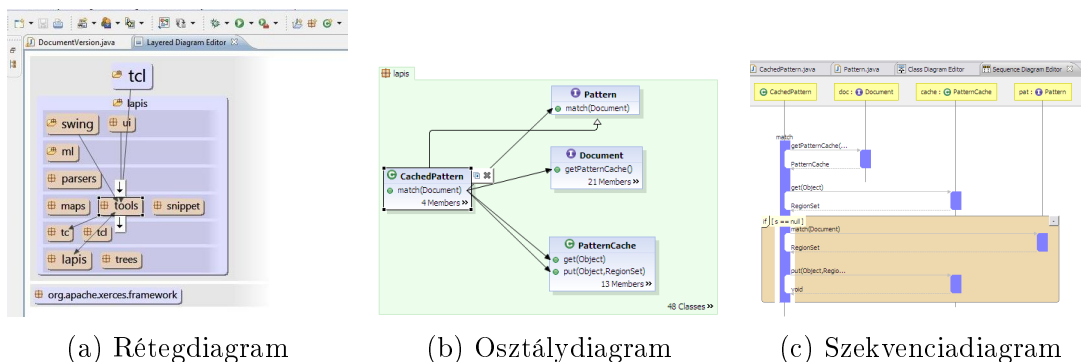


2.1. ábra. Sourecode Visualizer

2.1.2. Architexa

Az *Architexa* [13, 14] is a statikus kódelemzők közé tartozik. Szintén egy Eclipse pluginról van szó, amelyet nagy méretű kódok architektúrájának feltérképezésére terveztek. Ez az eszköz is többféle diagramon keresztül próbál segíteni a szoftver struktúrájában való eligazodásban.

Az egyik ilyen diagram a *réteg diagram*. Lényegében egy fa szerkezetről van szó, amelyet egymásba ágyazott dobozok halmazaként kell elképzelni. Egy doboz egy modulhoz vagy osztályhoz tartozik, és tartalmazza azokat az elemeket, amelyekről az adott modul függ. Természetesen minden doboz interaktívan működik, tehát érzékelik az egér eseményeit. Például dupla kattintás hatására a doboz kinyílik, és láthatóvá válnak annak gyerek elemei. Vagy egy doboz fölé húzva az egeret, megjelennek azok a dobozokat összekötő élek, amelyek azt mutatják, hogy az adott egység



(a) Rétegdíagram

(b) Osztálydíagram

(c) Szekvenciadiagram

2.2. ábra. Architexa diagramjai

módosítása mely más elemekre lesz hatással. Ciklikus függőség esetén színkódokkal jelöli a program az egymástól függő elemeket.

A megjelenítések egy másik típusa az osztálydiagram. Ez az UML-ből megszokott diagram típus, amely az osztályok közötti kapcsolatokra reflektál. Megjeleníti az osztályok adattagjait, valamint az öröklődési és interfész megvalósítási kapcsolatokat, és mindezt kultúrált formában teszi. Alapértelmezetten csak az osztály nevét látjuk, és egy sort, amely a tartalmazott függvények és mezők számára utal. Ugyanígy az osztályok és interfészek közti kapcsolatokat is csak akkor látjuk, ha kattintással hívjuk elő őket.

Érdekes, hogy statikus kódelemző létre az *Architexa* egy dinamikus diagram megjelenítésére is képes, ez pedig a szekvenciadiagram. Ilyenkor az osztályok objektumai az ablak tetején egy vízszintes sávban helyezkednek el. Az objektumokból egy-egy függőleges életvonal indul, amelyeken a függvényhívásokat jelző nyilak mutatják, hogy melyik objektum mely műveletére került a vezérlés.

2.1.3. NDepend

Az *NDepend* egy sok funkcionalitással bíró statikus kódelemző eszköz [15]. Alapvetően C# nyelvű kódok vizsgálatára szolgál, de a megfelelő *JArchitect* és *CppDepend* nevű programok a Java és C++ nyelvű forrásokon is működnek. A rengeteg szolgáltatás közül ezúttal is a gráf alapú megjelenítésekkel foglalkozunk.

Az eszköz által megjelenített gráfok első nagy csoportja a függőségi gráf. Ennek több szintjét is értelmezi az *NDepend*. A legmagasabb szint a .NET egyes elemei között fennálló függőségekről szól. Ezen túl a projektfa-nézetéből a jobb egérgomb megnyomására előugró menüből hívható elő az a gráf, amely az adott projekt névterei között fennálló függőségeket reprezentálja. Ennél alacsonyabb szintű ábrázolás az, amikor a felhasználói (pl. osztály) és egyéb típusok közti kapcsolatokat vizsgáljuk.

Végül lehetőség van egy osztályon belüli adattagok és függvények közti összefüggéseket is tanulmányozni.

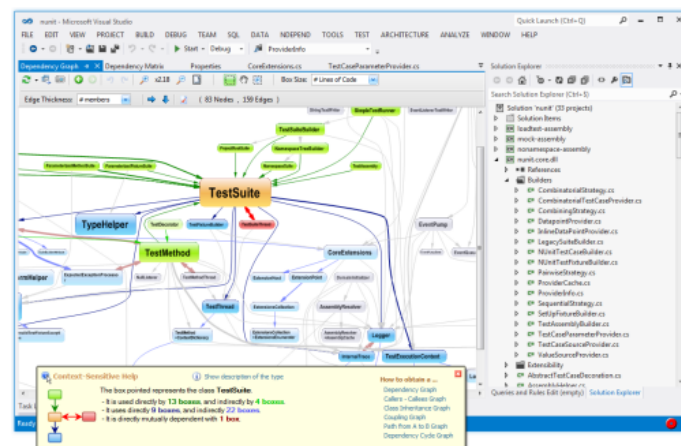
Egy másik típusú gráf a hívási gráf. A hívó és hívott feleket a felhasználónak áll módjában összegyűjteni a számos előre biztosított metrika alapján. Ezt az összegyűjtést egy az SQL-hez hasonló lekérdező nyelven keresztül lehet megfogalmazni.

Ugyanígy megadható egy osztályok közötti öröklődési kapcsolatot reprezentáló gráf is. Az alábbi példában olvasható lekérdezés egy ilyen gráfot állít elő:

```
from t in Types
let depth0 = t.DepthOfDeriveFrom("Microsoft.Scripting.
Interpreter.LocalAccessInstruction")
where depth0 >= 0 orderby depth0
select new { t, depth0 }
```

Látható, hogy könnyen lehet szabályozni, illetve finomítani egy ilyen lekérdezést minden metrika és előre bevezetett fogalom felhasználásával. Ha nem vagyunk kíváncsiak a közvetlen, hanem csak a közvetett öröklődési kapcsolatokra, akkor elegendő a feltételben szereplő relációs jelet „> 1”-re cserélni.

A különféle függőségek mátrixban is szemléltethetők. Rendeljük egy négyzetes mátrix minden sorához és oszlopához rendre ugyanazokat az elemeket, amelyek között a függőséget vizsgálni szeretnénk. A mátrix celláiba írjuk azt a számot két elem metszéspontjához, amely annak a legrövidebb útnak a hossza, ami a függések mentén vezet a két elem között. Ezt a lineáris útvonal gráfot az *NDepend* szintén képes megjeleníteni. Speciális esetben a függőségi körök is könnyen feltárhatók ezzel a reprezentációval.

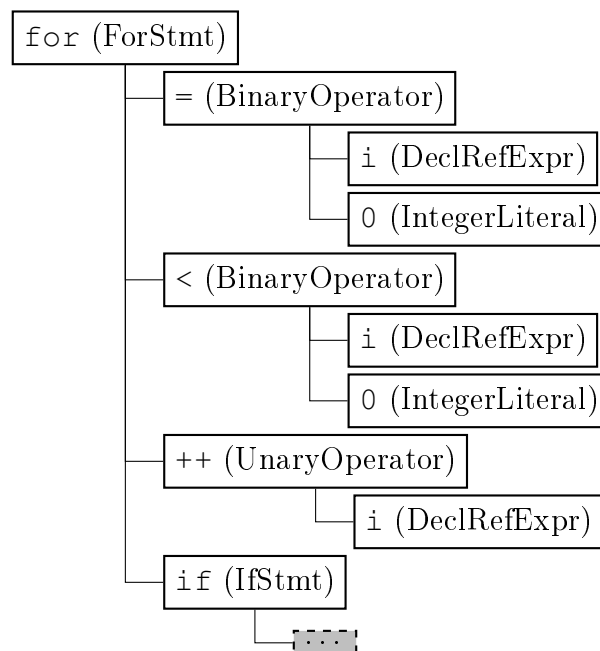


2.3. ábra. NDepend

2.2. Absztrakt szintaxisfa (AST)

Egy program fordítása alapvetően négy lépésből áll: lexikai elemzés, szintaxis elemzés, szemantikai elemzés, valamint kódgenerálás. A lexikai és a szintaxis elemzés során a forráskód szövege a további elemzők számára könnyebben értelmezhető formára alakul át. Ez a forma a szintaxisfa [16], melynek két változata a konkrét és az absztrakt szintaxisfa. A konkrét szintaxisfa a forrásszöveg minden elemét tartalmazza, beleértve az utasítás végi pontosvesszőket, vagy a paramétereket elhatároló vessző karaktereket is. Az absztrakt fa ezzel szemben csak a fordítás szempontjából fontos elemeket tartalmazza. A kétféle fa között megfeleltetés létesíthető, hiszen a fa struktúrája az AST-ben is egyértelművé teszi, hogy melyik nyelvi elem hová tartozik. Mivel a statikus kódelemzés legtöbb funkciója csak a forráskód szerkezetét vizsgálja, ezért elegendő az absztrakt fát elemeznie.

Az AST egy olyan fa, melynek gyökér csúcsa a fordítási egységet szimbolizálja, leszámazottjai pedig a forráskódban leírt nyelvi elemeket tartalmazzák. Például, ha a fa egy csúcsa egy C++ nyelvű `for` ciklushoz tartozik, akkor annak négy gyereke rendre az inicializáló kifejezéshez, a ciklusfeltételhez, az iterációs kifejezéshez és a ciklustörzshöz tartozik. Ha a ciklustörzs üres, akkor a negyedik gyerek a `null`, vagy több utasítás esetén a blokk utasítás (compound statement).



2.4. ábra. `for` ciklus AST-beli részfája

3. fejezet

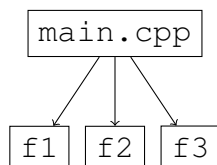
Általános gráfépítés

Ebben a fejezetben pszeudó kód segítségével mutatjuk be azokat az eszközöket, amelyeket a könyvtárcsomag megvalósít. Ismertetjük a gráfépítés algoritmusát, és az ehhez szükséges fogalmakat.

3.1. Bevezető példa

Mielőtt rátérnénk a gráfépítő algoritmusunk ismertetésére, egy nagyon rövid példán keresztül bemutatjuk annak használatát. Állítsunk elő egy olyan gráfot, amelynek gyökere egy fordítási egység, gyerekei pedig a fordítási egységben szereplő függvények. Ehhez elegendő megmondanunk azokat a feltételeket, amelyek a gráf csúcsait és éleit határozzák meg. Az ilyen feltételek megadásának nyelvi eszköze pedig a függvény. A továbbiakban C++ szintaxisú példakódokat használunk.

```
1 bool isTranslationUnit (SemanticNode node)
2 {
3     return node.getASTType() == "TranslationUnit";
4 }
5
6 std::vector<SemanticNode> children (SemanticNode node)
7 {
8     std::vector<SemanticNode> result;
9
10    for (SemanticNode n : node.getChildren())
11        if (Function(n).isValid())
12            result.push_back(n);
13
14    return result;
15 }
```



3.1. ábra. TranslationUnit és függvényei

Látható, hogy a `SemanticNode` és `Function` fogalmak bevezetésével milyen egyszerű megvalósításhoz jutottunk. Minél erősebb és magasabb szintű fogalmakat definiál a nyelvi interfész, annál rövidebb és olvashatóbb függvények megadásával tudjuk a megfelelő gráfot felépíteni. A felépítés mechanizmusát ezek a függvények vezérlik oly módon, hogy az építő algoritmus adott kezdőcsúcsra végigpróbálja az összes betáplált szabályt. Ha valamelyik szabály – a későbbiekben tárgyalt – `isNode` attribútuma (jelen esetben az `isTranslationUnit()` függvény) illeszkedik a csúcsra, akkor azt a csúcsot felveszi a kimeneti gráfba, majd meghatározza a gyerekeit a `children`, `childSequence` és `childSequences` attribútumokban megfogalmazott függvények segítségével. A következő iterációban ezek a gyerekek lesznek a soron következő aktuális csúcsok, míg az összes így elérhető csúcs feldolgozásra nem kerül.

Tegyük fel, hogy adott egy `main.cpp` nevű fájl, amiben `v1`, `v2` globális változók és `f1`, `f2`, `f3` függvények találhatók. Az fenti szabályokkal a 3.1. ábrán látható eredmény áll elő. A `children` függvény 10. sorában található `node.getChildren()` a `main.cpp`-beli fordítási egység közvetlen gyerekeit határozza meg, beleértve a globális változókat. Ezeket azonban kiszűrjük, hiszen csak a függvény típusú csúcsokat válogatjuk be a kimeneti gráfban a `main.cpp` gyerekeinek listájába. A 4.1. részben ezt a példát továbbgondolva, a program teljes folyamatábráját előállító példán keresztül illusztráljuk a könyvtárcsomag lehetőségeit.

3.2. Szabályok

A gráf a felhasználók által megadott szabályok szerint épül fel az AST alapján. Egy szabály függvények rekordja, amelyek értelmezési tartományában és értékkészletében szereplő típusokat az alábbiak alapján kell értelmezni:

- $\mathbb{L} = \{igaz, hamis\}$
- T : Az AST csúcsainak típusa

- N : A gráf csúcsainak típusa
- A : A gráf éleinek típusa
- X^* : X típusú elemek sorozata

A gráf csúcsait és éleit meghatározó függvények azért kerülnek egy rekordba, mert az egy csúcshoz tartozó szabályokat ezek a függvények közösen írják le.

$$\begin{aligned}
 Rule = \{ \quad & isNode : \quad T \quad \rightarrow \quad \mathbb{L} \\
 & children : \quad T \quad \rightarrow \quad T^* \\
 & childSequence : \quad T \quad \rightarrow \quad T^* \\
 & childSequences : \quad T \quad \rightarrow \quad (T^*)^* \\
 & furtherArcs : \quad T \quad \rightarrow \quad (T, T)^* \\
 & inNodes : \quad T \quad \rightarrow \quad T^* \\
 & outNodes : \quad T \quad \rightarrow \quad T^* \\
 & nodeAction : \quad (T, N) \\
 & arcAction : \quad (T, T, A) \\
 & \}
 \end{aligned}$$

A fenti függvények a következő jelentésekkel bírnak:

isNode: Az *isNode* egy logikai függvény, amely a paraméterül kapott AST-beli csúcsról megmondja, hogy lesz-e annak megfelelője az eredmény gráfban. Ha igen, akkor erre a csúcsra vonatkozik az összes többi függvény is ugyanazon a szabályon belül. Ez nem azt jelenti, hogy a többi függvény értelmezési tartományának eleme a csúcs, vagyis a *Rule* rekord függvényei parciálisak. Később látni fogjuk, hogy ha a csúcshoz – amelyre az *isNode* igaz értéket ad – például *inNodes* nem rendel semmit, akkor ezt az esetet is le tudjuk kezelni.

children: Ez a függvény megadja egy olyan AST-beli csúcsnak a gyerekeit, amire az *isNode* igazat ad. Felhívjuk a figyelmet, hogy itt két gráfot tárgyalunk egyszerre: az AST-t és a létrehozandó gráfot. Amikor azt határozzuk meg, hogy melyek az adott csúcs gyerekei, akkor azt úgy értjük, hogy az AST egy csúcsának van egy megfelelője az új gráfban. A *children* függvény meghatározza azokat a csúcsokat az AST-ből, amelyek gyerekei lesznek ennek az új csúcsnak, majd összeköti őket a kimeneti gráfban. Például a folyamatábra diagramjában a fordítási egységek gyerekei a függvény típusú csúcsok.

childSequence: Ez a függvény csak abban különbözik az előzőtől, hogy segítségével egy csúcshoz megadhatunk egy szekvenciát, ami abból a csúcsból indul ki. A

blokk típusú szerkezetek gyerekeit (vezérlési szerkezetek, függvények törzsei) rendszerint ezzel állítjuk elő.

childSequences: Az előző függvényre tekinthetünk úgy, mint egy olyan speciális esetre, ahol egy csúcsnak csak egy szekvencia gyereke van. Ezzel a függvénnyel több szekvenciát is a csúcs alá rendelhetünk. Csupán kényelmi okai vannak annak, hogy mind a két esetet bevezettük, de ezek persze nem zárják ki egymást. Ha egyszerre mindkét függvénnyel gyerek szekvenciákat fűzünk a csúcshoz, akkor mindegyik meg fog jelenni a kimeneti gráfban.

furtherArcs: Csúcspárok listájaként meg lehet adni további éleket is mindentől függetlenül, az *isNode* predikátum által meghatározott csúcs felhasználásával.

inNodes: Elképzelhető olyan szituáció, amikor egy csúcsba bemenő éleket egy másik AST-beli csúcshoz szeretnénk átirányítani. Ez történik akkor is, amikor például egy `S; while (felt) { ... }` szekvencia folyamatábráját szeretnénk felrajzolni. Ilyenkor ugyanis az `S` és `while` AST csúcsok testvérek a szintaxisfában (hiszen szekvenciális kapcsolatban állnak), de az `S` és `felt` csúcsokat szeretnénk a kimeneti ábránkban szülő-gyerek relációban feltüntetni.

outNodes: E függvény segítségével a csúcsból kimenő éleket lehet manipulálni. Az elágazás folyamatábrájában például az `if (felt) S1; else S2; S;` szekvencia esetén nem az `if`-hez tartozó AST csúcsot szeretnénk az `S`-hez kötni, hanem az `S1` és `S2`-höz tartozókat.

nodeAction: Az eljárásnak két paramétere van: az adott AST-beli és az annak megfelelő gráfbeli csúcs. Az eljárás segítségével tetszőleges tevékenységet elvégezhetünk akkor, ha az algoritmus egy újabb csúcshoz érkezik. Például csúcscímkeket vehetünk fel a gráfhoz.

arcAction: Ennek az eljárásnak három bemenő paramétere a forrás- és a célcsúcs, valamint a köztük húzódó gráfbeli él. Ez az eljárás csak akkor hívódik meg, ha valóban egy újabb él kerül a gráfba. Például, ha egy csúcsnak szekvencia típusú gyereke van, akkor az adott csúcs elérésekor csak a közte és a szekvencia első eleme között lévő él kerül feldolgozásra, tehát az ehhez az élhez tartozó *arcAction* függvény hívódik meg.

3.3. A gráfépítés algoritmusa

Az algoritmus egy gráf adatszerkezeten működik. A gráfon, illetve annak csúcsain és élein ható műveletek a következők:

- `addNode()`: Csúcs hozzáadása
- `addArc(u, v)`: Él hozzáadása
- `inArcs(u)`: Az u csúcsba bemenő éleket adja meg
- `outArcs(u)`: Az u csúcsból kivezető éleket adja meg
- `setTarget(u)`: Egy éltre meghívva átállítja annak cél csúcsát u -ra
- `setSource(u)`: Egy éltre meghívva átállítja annak forrás csúcsát u -ra

Az építés során felhasználunk egy sor adatszerkezetet is, amit a *queue* objektumban tárolunk. Továbbá szükség van a *todoOut* és *todoIn* multimap típusú változókra is, amelyek egy csúcshoz más csúcsokat tárolnak. Ezeket a változókat és a gráfot globálisnak tekintjük.

Az algoritmus [17] rögtön egy függvényhívással indul. A `getNodeOrCreateNode` függvény első paramétere az AST egy csúcsa, visszatérési értéke pedig az ennek megfelelő gráfbeli csúcs. A függvény mellékkhatása, hogy ha az adott elem még nem volt azelőtt feldolgozva, azaz még nincs megfelelője a kimeneti gráfban, akkor nem csak a gráfba kerül be az új csúcs, hanem a *queue* sorba is. Erre azért van szükség, mert az algoritmus 3. sorában található fő ciklus addig fut, amíg a sor ki nem ürül. Ez a módszer a szélességi bejárásra emlékeztethet. Kezdetben tehát létrehozuk a gyöker csúcsnak megfelelő elemet, és elhelyezzük a sorba. A `getNodeOrCreateNode` függvény ily módon biztosítja azt is, hogy minden csúcsot csak egyszer dolgozzunk fel. Ezt később ki is használjuk.

Amíg a sor nem üres, addig vesszük az első elemét, és ezt értékül adjuk a *currentNode* változónak, majd kitöröljük a sorból. Továbbá eltároljuk az aktuális csúcs gráfbeli megfelelőjét is *b_currentNode* néven, hiszen az élek behúzásakor erre lesz szükség. A kétféle gráf csúcsai közötti leképezést a *nodeMap* végzi.

Ezután megnézzük, hogy mely szabályok illenek a *currentNode*-ban tárolt csúcsra. A szabályokon a 7. sorban kezdődő ciklussal iterálunk végig, azonban csak azokat a szabályokat vesszük figyelembe, amelyek *isNode* függvénye *igaz* helyettesítési értéket ad az aktuális csúcsra. Ennek a legegyszerűbb módja az, hogy *hamis* függvényérték esetén a következő iterációra lépünk. Erről a 8. sor feltétele gondoskodik.

Algorithm 1 A gráf felépítése

```
1: function BUILDGRAPH(rules, startNode)
2:   getOrCreateNode(startNode)
3:   while  $\neg$  isEmpty(queue) do
4:     currentNode := front(queue)
5:     b_currentNode := nodeMap[currentNode]
6:     pop(queue)
7:     for all rule  $\in$  rules do
8:       if  $\neg$  rule.isNode(currentNode) then
9:         continue
10:      end if
11:      rule.nodeAction(currentNode, b_currentNode)
12:      addChildren(currentNode, rule)
13:      manageTodoArcs(currentNode, rule)
14:      for all (from, to)  $\in$  rule.furtherArcs(currentNode) do
15:        arc = graph.addArc(getOrCreateNode(from),
getOrCreateNode(to))
16:        rule.arcAction(from, to, arc)
17:      end for
18:    end for
19:    for all item  $\in$  todoOut[currentNode] do
20:      addArc(currentNode, item)
21:      todoOut.erase(currentNode)
22:    end for
23:  end while
24:  return graph
25: end function
```

Algorithm 2 Csúcs beszúrása vagy visszaadása

```
1: function GETORCREATENODE(node)
2:   if nodeMap.contains(node) then
3:     b_node := nodeMap[node]
4:   else
5:     queue.push(node)
6:     b_node := graph.addNode()
7:     nodeMap[node] := b_node
8:   end if
9:   return b_node
10: end function
```

Algorithm 3 A csúcs gyerekeinek felvétele

```
1: procedure ADDCHILDREN(currentNode, rule)
2:   for all childNode  $\in$  rule.children(currentNode) do
3:     arc = addArc(b_currentNode, childNode)
4:     rule.arcAction(currentNode, childNode, arc)
5:   end for
6:   sequences := rule.childSequences(currentNode)
7:   sequences := sequences  $\cup$  rule.childSequence(currentNode)
8:   for all sequence  $\in$  rule.childSequences(currentNode) do
9:     if  $\neg$  isEmpty(sequence) then
10:      arc = addArc(b_currentNode, sequence[0])
11:      rule.arcAction(currentNode, sequence[0], arc)
12:      prevNode := sequence[0]
13:    end if
14:    for i := 1  $\rightarrow$  length(sequence) do
15:      getOrCreateNode(sequence[i])
16:      todoOut[prevNode].insert(sequence[i])
17:      prevNode := sequence[i]
18:    end for
19:  end for
20: end procedure
```

Algorithm 4 Későbbi élek kezelése

```
1: procedure MANAGETODOARCS(currentNode, rule)
2:   for all item  $\in$  todoOut[currentNode] do
3:     if currentNode  $\in$   $\mathcal{D}_{rule.outNodes}$  then
4:       for all outNode  $\in$  rule.outNodes do
5:         todoOut[outNode].insert(item)
6:       end for
7:     else
8:       addArc(b_currentNode, item)
9:     end if
10:    todoOut.erase(currentNode)
11:  end for
12:  for all inNode  $\in$  rule.inNodes(currentNode) do
13:    todoIn[currentNode].insert(inNode)
14:    for all arc  $\in$  graph.inArcs(b_currentNode) do
15:      arc.setTarget(getOrCreateNode(inNode))
16:    end for
17:  end for
18: end procedure
```

Algorithm 5 Él hozzáadása

```
1: function ADDARC(source, target)
2:   if target  $\in$  todoIn then
3:     for all item  $\in$  todoIn[target] do
4:       arc = graph.addArc(source, item)
5:     end for
6:   else
7:     arc = graph.addArc(source, getOrCreateNode(target))
8:   end if
9:   return arc
10: end function
```

A 11. sorban található a *nodeAction* függvény meghívása az aktuális csúcsra. Ez egy felhasználói függvény, aminek az algoritmus szempontjából nem sok szerepe van. Segítségével azonban tetszőleges műveletet végre lehet hajtani a gráf építése közben, például él- és csúcscímkeket tárolhatunk el, amelyeket később a gráf mentésekor felhasználhatunk.

Az *addChildren* függvényhívás már egy lényeges lépés, hiszen itt határozzuk meg az aktuális csúcs gyerekeit. Ez két lépésből áll. Elsőként a *children* függvény által visszaadott egyszerű gyerek csúcsokat kötjük össze az aktuális csúccsal, másodszor pedig azokat a szekvenciákat kapcsoljuk hozzá, amelyeket a *childSequences* és a *childSequence* függvények együtt adnak. Az utóbbi esetben annyival másabb a helyzet, hogy a szekvencia elemeit még nem kötjük össze a kimeneti gráfban, hanem csak a szekvencia első elemét kapcsoljuk az aktuális csúcshoz. Az *arcAction* függvény meghívására ezeken a pontokon kerül sor. A csúcs szekvencia gyerekei-nél csak az aktuális csúcs és a szekvencia első eleme közötti élre hívjuk meg az *arcAction* függvényt, a szekvencia többi eleme közti élekre majd azok feldolgozásokor kerül sor. Ehhez persze az kell, hogy ezeket az elemeket ne vesszük el, ezért a *getOrCreateNode* függvénnyel berakjuk ezeket a queue sorba, valamint a *todoOut* multimap adatszerkezetbe feljegyezzük, hogy később össze kell kötni a szekvencia egy elemének kimeneteit a rákövetkező elemmel. A *todoOut*-ra azért van szükség, mert egy szekvencia elemeit nem biztos, hogy mindenáron össze szeretnénk kötni. Tegyük fel, hogy a szekvencia egy köztes eleme egy elágazás. Ha most behúznánk a szekvencia elemei között futó éleket, akkor az *if*-hez tartozó AST-beli elemet kötnénk a következő utasításhoz, holott a *then* és *else* ágakat kellene. További bonyodalmakat jelentene, ha a két ág utolsó utasításaiban egy újabb elágazás szerepelne, de ezzel a módszerrel ez is megoldódik.

Az algoritmusban két különböző `addArc()` függvény is szerepel. Az egyik a gráf típus művelete, ami ténylegesen felveszi a kérdéses élt a gráfba, a másik pedig egy segédfüggvény, amelyet az 5. algoritmus taglal.

Az utólag behúzendó éleket kezelő `manageTodoArcs` függvény is két részre osztható. Az első rész az aktuális csúcsból kivezető élekkel, a második a bejövőekkel foglalkozik.

Vegyük sorra a `todoOut[currentNode]`-hoz tárolt *item* elemeket. Ezek azok az elemek, amelyeket korábban még nem állt módunkban összekötni az aktuális csúccsal. Ha az aktuális csúcs benne van a `rule.outNodes` függvény értelmezési tartományában, akkor nézzük meg, hogy itt milyen *outNode* értékeket vesz fel, ugyanis talán innen indulnak majd a kimenő élek. Ahelyett, hogy most a gráfban behúznánk az *outNode*-ok és *item*-ek között vezető éleket, inkább vegyük fel ezt a párt újra a `todoOut` multimapbe, hiszen még mindig nem lehetünk biztosak abban, hogy valóban a `rule.outNodes` által megjelölt csúcsokból fognak-e kivezetni az élek. Gondoljunk csak arra az esetre, amikor egy elágazás utolsó utasítása is elágazás. A másik eset, hogy az aktuális csúcs nincs a `rule.outNodes` értelmezési tartományában, azaz nincs olyan csúcs, ahová a kimenő élek forrásait át kellene irányítani. Ekkor behúzzuk a `currentNode` és az *item* között vezető élt. Végül a `todoOut`-ból törölhető az aktuális csúcsához felvett összes bejegyzés, mert ezekre a későbbiekben már nem lesz szükség.

A `manageTodoArcs` második fázisában az aktuális csúcsba bemenő élekkel foglalkozunk. Ha a `rule.inNodes` elemeket rendel az aktuális csúcsához, akkor ezeket feljegyezzük a `todoIn` multimapbe, és a csúcs bemenő éleit át is irányítjuk ezekre a helyekre.

Felmerülhet a kérdés, hogy mi történik abban az esetben, ha később egy olyan kört találunk a gráfban, amelynek végpontja ez az aktuális csúcs. Ilyenkor ugyanis a kör utolsó elemére vonatkozó szabály értelmében ebből az utolsó elemből kiinduló élt a kör első elemére kell állítani. Ekkor pedig már nem lesz lehetőség még egyszer megvizsgálni a kör első eleméhez tartozó *inNodes* függvényt, hiszen minden csúcsot csak egyszer dolgozunk fel. Mégis, hogyan irányítjuk át ezeket az éleket a kör első elemének *inNodes* függvénye által megadott csúcsokra? Ennek a problémának a megoldásaként vezettük be az `addArc` függvényt, ami a `manageTodoArcs` második fázisában feltöltött `todoIn` multimap alapján állítja be az újonnan behúzendó élek végpontjait.

Mostanra végeztünk az aktuális csúcs – akár egyszerű, akár szekvenciális – gyerekeinek felvételével, valamint az utólagosan módosítandó élek behúzásával. Már csak

a szabályként megadott extra élek bekötése van hátra, amit a *rule.furtherArcs*-on való végigiterálással könnyen megtehetünk. Vegyük észre, hogy ez a ciklus (16. sor) az eddigiektől eltérően, közvetlenül a gráfban veszi fel az éleket, ami azt jelenti, hogy nem vonatkoznak rá az *inNodes* és *outNodes* függvények által megfogalmazott szabályok. Ne felejtjük el erre az élre is elvégezni a megfelelő *arcAction* függvényt.

Ne feledkezzünk meg azokról a csúcsokról, amelyek ugyan bekerültek a *todoOut*-ba, de 7–18. sorban található ciklusban nem kerültek feldolgozásra. Ilyen csúcs egy elágazás valamely ágán lévő utasítás is, amire nem vonatkozik semmilyen szabály. Ezt az esetet a 19–22. sorban lévő ciklus kezeli le.

3.4. A nyelvfüggetlenség biztosítása

A 2.2. szakaszban szó volt arról, hogy az AST szerkezete miképpen határozza meg a program szintaxisát. Belátható, hogy az egyes nyelvi elemek részeinek az AST részfáinak indexeléssel történő elérése nagyban rontja a program hordozhatóságát, módosíthatóságát. Ezért érdemes magasabb szintű fogalmakat bevezetni, mint a *for*, *while*, *do-while* ciklus, *if* elágazás, függvény, stb. Így, ha más nyelvhez tartozó, vagy ugyanazon nyelvhez, de más elemzővel készített, más felépítésű AST-t szeretnénk feldolgozni, akkor elegendő ezt az interfészt implementálni az adott változathoz.

Ha az AST-t kiterjesztjük a gyökere fölött fájlokhoz, esetleg projektekhez tartozó csúcsokkal, akkor az ezekre a fogalmakra vonatkozó összefüggéseket is definiálhatunk később, mint a fájlok közötti függőségeket.

Vezessük be a *szemantikus node* fogalmát. A *node* az AST egy csúcsára utal, a *szemantikus* jelző pedig arra, hogy egy ilyen csúcsához különféle információk rendelkezhetők, amelyek a csúcs tulajdonságait tartalmazzák. Ilyen tulajdonságok a csúcs azonosítója, tartalmazó fájlja, szövege, típusa, forráskódbeli pozíciója, az esetleges definíciójához tartozó szemantikus node, vagy a csúcs gyerekeinek vektora. Ezek az attribútumok a megfelelő függvények segítségével lekérdezhetők, amiket a használati esetek tárgyalásakor fel is fogunk használni.

A dolgozat írásakor a Clang C++ fordító használatával építettük fel a forráskódból az AST-t, ezért a következő néhány példában a Clang specifikus AST-hez készített csomagoló osztályokkal dolgozunk.

3.4.1. Szekvencia

Az első magasabb szintű fogalmunk a szekvencia. Ez az elnevezés annyiban félrevezető lehet, hogy a szekvencia valójában egy programkonstrukció, de ebben az esetben tetszőleges AST-beli csúcsok egymásutánját jelentheti. Azért választottuk mégis ezt az elnevezést, mert a többi programozási fogalom erre a szerkezetre valódi szekvenciaként tekint.

A `Sequence` osztálynak így két konstruktora van. Az egyik szemantikus node-ok vektorát, másik pedig egy szemantikus node-ot vár paraméterül. Mindkét esetben a szekvencia lényegében node-ok vektora lesz, a különbség csak annyi, hogy az előbbi esetben a megadott csúcsokat tartalmazza, utóbbiban pedig a paraméter csúcs gyerekeit. Ennek az a haszna, hogy ha egy függvény, ciklus, elágazás, vagy valamilyen más szerkezet törzsét szekvenciába szeretnénk foglalni, akkor elegendő csak a szekvenciát magábafoglaló blokkhoz tartozó csúcsot megadni.

A szekvencia elemeinek lekérdezésére a `getItems()` függvény való, melynek visszatérési értéke szemantikus node-ok vektora. Továbbá az elemek elérését iteratív módon is elérhetjük a `setCurrent(size_t)`, `getCurrentItem()`, `getNextItem()` és `hasNextItem()` függvényekkel.

Sequence
<code>setCurrent(int)</code>
<code>getSize()</code>
<code>getItems()</code>
<code>getCurrentItem()</code>
<code>getLastItem()</code>
<code>getNextItem()</code>
<code>hasNextItem()</code>

3.2. ábra. A `Sequence` osztály műveletei

3.4.2. **while** és **do-while** ciklusok

Ezt a két ciklusfajtát együtt tárgyaljuk, hiszen az elemeiket ugyanazon az interfészen keresztül lehet elérni. Ilyen elemek a törzs, amely lehet egy tetszőleges utasításhoz, null- vagy blokkutasításhoz tartozó node. Bármelyik is legyen, a törzs utasításainak szekvenciáját a `getBodySequence()` függvény segítségével tudjuk lekérdezni, amely az utóbbi két esetben üres szekvenciát ad. A `getCondition()` értelemszerűen a feltétel kifejezéshez tartozó csúcsot adja vissza, végül az `isValid()` megmondja az adott entitásról, hogy tényleg a megfelelő típusú ciklushoz tartozó

csúcsról van-e szó. Erre azért van szükség, mert a konstruktorban tetszőleges AST-beli csúcsot megadhatunk, és mivel a ciklus egyes részeit lekérdező függvények az AST ciklushoz tartozó részfájának struktúrája alapján tudják megadni annak elemét, ezért egy nem az adott ciklus típusához tartozó node esetén könnyen szegmentálási hibába futhatunk.

Ez a további nyelvi elemekre is igaz, miszerint a konstruktorukon keresztül lehet megadni azt az AST csúcsot, ami az adott elemre hivatkozik. Az, hogy ez a csúcs valóban olyan típusú legyen, amelyet az adott csomagoló osztály elvár, a programozó felelőssége.

While
getBody() getBodySequence() getCondition() isValid()

3.3. ábra. A While és DoWhile osztályok műveletei

3.4.3. for ciklus

A while és do-while ciklusokhoz képest ez a csomagoló osztály csak két függvénnyel tartalmaz többet. Ezek a `getInit()` az inicializáló, valamint a `getIteration()`, az iteráló kifejezés lekérdezésére alkalmas függvények.

For
getBody() getBodySequence() getCondition() getInit() getIteration() isValid()

3.4. ábra. A For osztály műveletei

3.4.4. Elágazás

Az elágazásnak is természetesen van `getCondition()` függvénye a feltétel lekérdezésére, azonban ennek a programkonstrukciónak nem egyszerűen csak egy törzse van, hiszen `then` és `else` ágakról beszélünk. Ennek megfelelően működnek a

`getThen[Sequence]()` és `getElse[Sequence]()` variánsai. Egy elágazásnak nem feltétlenül van `else` ága, de erről a `hasElse()` segítségével megbizonyosodhatunk.

If
<code>getCondition()</code> <code>getThen()</code> <code>getThenSequence()</code> <code>getElse()</code> <code>getElseSequence()</code> <code>hasElse()</code> <code>isValid()</code>

3.5. ábra. Az `If` osztály műveletei

3.4.5. Függvény

Az előzőekhez hasonlóan lekérdezhető a függvény típusú csúcs törzse. Ezen túl a `getName()` visszaadja a függvény nevét, a `getParameters()` a paraméterek listáját, a `getReturnType()` pedig a visszatérési érték típusát. A `isDefinition()` segítségével lehetőség van annak eldöntésére is, hogy az adott függvény definíció, vagy csak deklaráció-e.

Function
<code>getBody()</code> <code>getBodySequence()</code> <code>getName()</code> <code>getParameters()</code> <code>getReturnType()</code> <code>isDefinition()</code> <code>isValid()</code>

3.6. ábra. A `Function` osztály műveletei

4. fejezet

Felhasználási lehetőségek

Ebben a fejezetben a könyvtári csomagunk két felhasználási lehetőségét részletezzük. Az első a program folyamatábrájának megfelelő gráfot állítja elő, a második pedig egyfajta függési viszonyt létesít fájlok között.

4.1. Folyamatábra diagram

Azért választottuk ezt a típusú diagramot, mert ezen keresztül a bevezetett függvények mindegyike bemutatatható. A folyamatábra egy jól strukturált program [18] esetén a három programkonstrukció (szekvencia, elágazás, ciklus) ábrázolását jelenti azzal a kiegészítéssel, hogy a folyamatábránk gráfjának gyökere a fordítási egység lesz. Ennek gyerekei az adott fordítási egységekben található függvényekhez tartozó csúcsok, és ezen függvények leszármazottai a tényleges folyamatábra elemei. A továbbiakban ezeket a fogalmakat vesszük sorra.

A könnyebbség kedvéért bevezetünk két típuszinonímát a szemantikus node-ok vektorára és a belőlük alkotott párokra:

```
1 typedef std::pair<SemanticNode, SemanticNode> SNPair;  
2 typedef std::vector<SemanticNode> SNVector;
```

Ezen túl deklarálunk egy `GraphBuilder` típusú változót, melynek feladata a gráf felépítése lesz a megadott szabályok alapján. Mivel egy osztálysablonról van szó, ezért azt a megfelelő típussal paraméterezni kell. Ebben az esetben szemantikus node-ok gráfját építjük fel. A szabályok megadására a `GraphBuilder<T>::Rule` struktúra használható.

```
1 typedef GraphBuilder<SemanticNode> GB;  
2 GB builder;
```

4.1.1. Fordítási egység

A fordítási egység szabálya könnyen megfogalmazható, hiszen csak azt kell megmondani, hogy egy csúcs mikor számít fordítási egységnek, és mik a gyerekei:

```
1 bool isTru(SemanticNode node)
2 {
3     return node.getASTType() == "TranslationUnit";
4 }
5
6 SNVector truChildren(SemanticNode node)
7 {
8     SNVector children = node.getChildren();
9     SNVector result;
10
11     for (size_t i = 0; i < children.size(); ++i)
12         if (Function(children[i]).isValid())
13             result.push_back(children[i]);
14
15     return result;
16 }
```

Az `isTru()` igazat ad, ha a paraméterül kapott csúcs egy fordítási egységhez tartozik. A `truChildren()` pedig szemantikus node-ok vektorában adja vissza a paraméterben lévő csúcs gyerekeit. Mivel ez a függvény egy szabályba kerül az `isTru()` függvénnyel, ezért biztosak lehetünk abban, hogy a `truChildren()` paraméterében szereplő node egy fordítási egységet reprezentál az AST-ben. Így a gyerekei között szerepelnek azok a forráskódbeli függvények, melyeket meg szeretnénk jeleníteni a folyamatábra gráfján. Ehhez egy újabb vektorban gyűjtsük össze a függvény típusú gyerek csúcsokat a már korábban látott `Function` csomagoló osztály segítségével.

Végül hozzunk létre egy szabály objektumot, amiben elhelyezhetjük a megfogalmazott szabályokat, majd adjuk hozzá a gráf építő szabálylistájához:

```
1 GB::Rule truRule;
2 truRule.isNode = isTru;
3 truRule.children = truChildren;
4 builder.addRule(truRule);
```

4.1.2. Függvény

Az alábbi függvények az AST függvény típusú csúcsait detektálják:

```

1 bool isFunction(SemanticNode node)
2 {
3     return Function(node).isValid();
4 }
5
6 SNVector functionChild(SemanticNode node)
7 {
8     return Function(node).getBodySequence().getItems();
9 }

```

Egy dolgot azért érdemes megjegyezni a szabály felvételekor. A függvény típusú gráfbeli csúcs alá egy szekvenciát szeretnénk befűzni. Azonban nem áll módunkban olyan szabályt megfogalmazni, ami tetszőleges csúcsra hatékonyan meg tudná mondani, hogy melyik az őt tartalmazó szekvenciában a rákövetkező elem. Szerencsére a szabály biztosít egy olyan lehetőséget, hogy egy csúcshoz megadhatunk szekvenciákat, amely szekvenciák a csúcs gyerekei lesznek. A szekvencia elemeit a gráfépítő algoritmus köti majd össze élekkel. Egy csúcsnak több szekvencia gyereke is lehet, ezért a szabály egy vektort tart fenn az ezeket a szekvenciákat előállító függvényeknek.

```

1 GB::Rule functionRule;
2 functionRule.isNode = isFunction;
3 functionRule.childSequence = functionChild;
4 builder.addRule(functionRule);

```

4.1.3. while ciklus

A while ciklushoz – ahogy a későbbi szerkezetekhez is – tartozik csomagoló osztály, amely segítségével hivatkozhatunk annak egyes részeire. Ezek segítségével könnyen megfogalmazhatóak a szabályok:

```

1 bool isWhile(SemanticNode node)
2 {
3     return While(node).isValid();
4 }
5
6 SNVector whileChild(SemanticNode node)
7 {
8     SNVector sequence = While(node).getBodySequence().getItems();
9     sequence.push_back(node);
10    return sequence;
11 }

```

Vegyük észre, hogy a ciklus gyerekének csak annak törzsét jelöltük meg, a szekvenciában a ciklus után következő utasítást nem. Ennek az az oka, hogy a ciklus csúcs biztosan egy szekvenciának lesz egy eleme, a szekvencia elemeit pedig az algoritmus automatikusan összeköti, ha arra kerül a sor. Megemlíjtük továbbá, hogy a szekvencia utolsó eleme maga a paraméterül megadott csúcs lesz, hiszen a ciklustörzs lefutása után újra a ciklusfeltétel ellenőrzése következik.

Végül a szabály az alábbi módon alakul:

```
1 GB::Rule whileRule;  
2 whileRule.isNode = isWhile;  
3 whileRule.childSequence = whileChild;  
4 builder.addRule(whileRule);
```

4.1.4. for ciklus

A for ciklus gráfjának felépítése már kicsit összetettebb művelet. Az `isFor()` és `forChild()` függvények megírása az előzőekhez hasonlóan alakul, értelemszerűen a `For` csomagoló osztály felhasználásával. Pontosabban a ciklustörzs utolsó utasítása után az iterációs kifejezésre kell ugrani, és ezúttal nem kötjük össze az iterációs kifejezést a ciklusfeltétellel.

```
1 bool isFor(SemanticNode node)  
2 {  
3     return For(node).isValid();  
4 }  
5  
6 SNVector forChild(SemanticNode node)  
7 {  
8     For forNode(node);  
9  
10    SNVector sequence = forNode.getBodySequence().getItems();  
11    sequence.push_back(forNode.getIteration());  
12  
13    return sequence;  
14 }
```

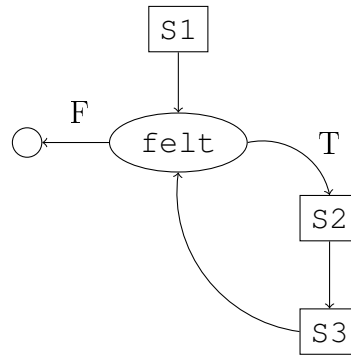
A for ciklus esetében elsőként mindig a ciklus inicializáló kifejezése értékelődik ki, ezért minden bemenő élt át kell irányítani az inicializáló kifejezéshez tartozó csúcshoz. Ezek után érthető, hogy miért nem kötöttük össze az iterációs kifejezést újra a for ciklussal. Ugyanis, ha minden bemenő élt átírányítunk, akkor a kimeneti diagramban az iterációs kifejezést az inicializálás követné. A bemenő élek átírányítá-

```

for (S1; felt; S3)
    S2;
-----
S1;
while (felt) {
    S2; S3;
}

```

(a) Forráskód



(b) Folyamatábra

4.1. ábra. Ciklus

sához csak azokat a csúcsokat kell megadni, amelyekre ezek az élek mutatni fognak:

```

1 | SNVector forInNodes(SemanticNode node)
2 | {
3 |     SNVector result;
4 |     result.push_back(For(node).getInit());
5 |     return result;
6 | }

```

Felmerül a kérdés, hogy az iterációs és az inicializáló kifejezés miképp kötődik a ciklusfeltételhez. Ennek megoldására vezettük be a `furtherArcs()` függvényt, amellyel további éleket vehetük fel a gráfba. Ezúttal nem kell attól tartanunk, hogy esetleg ezen élek végpontjai is átirányítódnak az `inNodes()` révén, mert a `furtherArcs()` éleire nincs hatással az átirányítás.

```

1 | std::vector<SNPair> forFurtherArcs(SemanticNode node)
2 | {
3 |     For forNode(node);
4 |
5 |     std::vector<SNPair> result;
6 |     result.push_back(SNPair(forNode.getInit(), node));
7 |     result.push_back(SNPair(forNode.getIteration(), node));
8 |
9 |     return result;
10 | }

1 | GB::Rule forRule;
2 | forRule.isNode = isFor;
3 | forRule.childSequence = forChild;
4 | forRule.furtherArcs = forFurtherArcs;
5 | forRule.inNodes = forInNodes;
6 | builder.addRule(forRule);

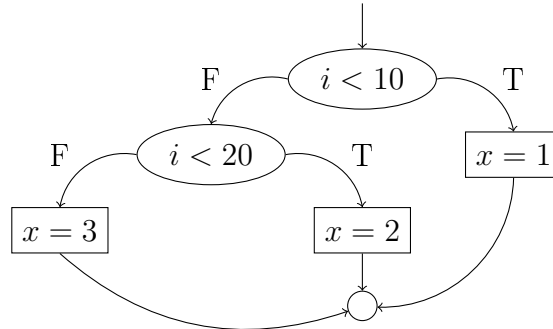
```

```

if (i < 10)
    x = 1;
else if (i < 20)
    x = 2;
else
    x = 3;

```

(a) Forráskód



(b) Folyamatábra

4.2. ábra. Elágazás

4.1.5. Elágazás

Egy `if` típusú elágazásnak egy vagy két gyereke lehet, mivel az `else` ág használata nem kötelező. Minthogy több szekvencia is lehet a csúcs leszármazottja, a *childSequences* függvényt kell használnunk:

```

1 bool isIf(SemanticNode node)
2 {
3     return If(node).isValid();
4 }
5
6 std::vector<SNVector> ifChildren(SemanticNode node)
7 {
8     std::vector<SNVector> result;
9
10    If ifNode(node);
11
12    result.push_back(ifNode.getThenSequence().getItems());
13    if (ifNode.hasElse())
14        result.push_back(ifNode.getElseSequence().getItems());
15
16    return result;
17 }

```

Tegyük fel, hogy az elágazás egy szekvencia közbülső eleme. Ekkor az algoritmus azon része, amelyik a szekvencia elemeit köti össze, egy élt húzna be az `if`-hez és a szekvenciában rákövetkező elemhez tartozó csúcsok közé. Mi nem ezt szeretnénk elérni, hanem azt, hogy az elágazás gyerekeinek végét kövessék a további csúcsok. Ehhez meg kell adni, hogy melyek legyenek az elágazásból kivezető élek forrásai:

```

1 SNVector ifOutNodes(SemanticNode node)
2 {

```

```

3   SNVector result;
4
5   SNVector thenNodes = ifThenChildren(node);
6   SNVector elseNodes = ifElseChildren(node);
7
8   result.push_back(thenNodes[thenNodes.size() - 1]);
9   if (!elseNodes.empty())
10      result.push_back(elseNodes[elseNodes.size() - 1]);
11
12   return result;
13 }
14
15 GB::Rule ifRule;
16 ifRule.isNode = isIf;
17 ifRule.childSequences = ifChildren;
18 ifRule.outNodes = ifOutNodes;
19 builder.addRule(ifRule);

```

4.1.6. Egyéb

Megfigyelhettük, hogy amikor a példákban egy ciklushoz vagy elágazáshoz értünk, akkor nem a feltételekhez tartozó AST-beli csúcsokra hivatkoztunk, hanem azok szülőjére, ami az adott vezérlési szerkezet részfájának gyökere. Ez persze nem okoz gondot, mert a létrejött gráf adatszerkezetben sehol nem jelöljük, hogy egy csúcs melyik szemantikus node-nak felel meg az AST-ben, és ez nem csak a vezérlési szerkezetekre, hanem minden egyéb csúcsra is igaz. Ez a megfeleltetés azonban még hátra van.

Hozzunk tehát létre egy olyan szabályt, ami minden csúcsra lefut, vagyis a hozzá tartozó *isNode* függvény azonosan igaz. Ezen kívül csak a *nodeAction*-t kell kitölteni, amelyben a megfelelő címkézést oldjuk meg.

```

1 bool isOther(SemanticNode node)
2 {
3     return true;
4 }
5
6 void getInfo(SemanticNode node, Graph::Node b_node)
7 {
8     textMap[b_node] = node.getText();
9
10    if (For(node).isValid())

```

```

11 |     textMap[b_node] = For(node).getCondition().getText();
12 |     else if (While(node).isValid())
13 |         textMap[b_node] = While(node).getCondition().getText();
14 |     else if (If(node).isValid())
15 |         textMap[b_node] = If(node).getCondition().getText();
16 |     else if (Function(node).isValid())
17 |         textMap[b_node] = Function(node).getSignature();
18 | }

```

4.2. Függőségi diagram

Ebben a szakaszban egy egyszerűbb, egy szabályból álló példán mutatjuk be, hogy hogyan lehet fájlok közötti kapcsolatokat modellezni az építő eszköz segítségével. A fájlok közti függést definiáljuk úgy, hogy az A fájl függ a B fájlról, ha A -ban olyan függvényt hívunk meg, amit B -ben definiáltunk.

Elsőként vezessük be azt a típuszinonímát, amelyben a gráfépítőt fájlokra példányosítjuk, majd hozzunk létre ebből egy objektumot:

```

1 | typedef GraphBuilder<File> GB;
2 | GB builder;

```

Potenciálisan bármelyik fájl szerepelhet csúcsként a gráfban, ezért a gráf csúcsait meghatározó logikai függvény mindenhol azonosan igazat ad.

```

1 | bool isFile(File file)
2 | {
3 |     return true;
4 | }

```

A csúcs gyerekeit rekurzívan keressük meg. Sorra vesszük a fájlban található függvénydefiníciókat, majd bejárjuk a törzsüket. Ha a törzsük valamely eleme egy függvényhívás, akkor megnézzük, hogy a függvény melyik fájlban lett definiálva, ugyanis ez a fájl gyereke lesz az aktuális csúcsnak. Ellenkező esetben pedig rekurzívan az AST-ben mélyebb szintre lépve keressük tovább az esetleges függvényhívásokat.

```

1 | std::vector<File> children(File file)
2 | {
3 |     std::vector<File> result;
4 |
5 |     for ( SemanticNode globalNode
6 |         : file.getSemanticGraph().getChildren())
7 |         if (Function(globalNode).isValid() &&
8 |             Function(globalNode).isDefinition())

```

```

9      for ( SemanticNode node
10          : Function(globalNode).getBodySequence().getItems() )
11          appendFiles(result, node);
12
13  return result;
14  }
15
16  void appendFiles(std::vector<File>& files,
17                  SemanticNode node)
18  {
19  if (node.getASTType() == "CallExpr")
20      files.push_back(node.gotoDef().getFile());
21  else
22      for (SemanticNode child : node.getChildren())
23          appendFiles(files, child);
24  }

```

Végül nincs más hátra, mint létrehozni a megfelelő szabály objektumot:

```

1  GB::Rule rule;
2  rule.isNode = isFile;
3  rule.children = children;
4  builder.addRule(rule);

```

A 4.3. ábrán egy példaprojekt és a belőle generált függőségi gráf látható.

```

main.cpp:
int main() {
    int a = 3 + f(x);
}

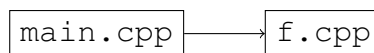
```

```

f.cpp:
int f() {
    return 5;
}

```

(a) Forráskód



(b) Diagram

4.3. ábra. Függőségi diagram

5. fejezet

Összefoglalás

Az irodalmi áttekintésben láthattuk, hogy az iparban igény van olyan eszközökre, amelyek a kód vizualizációjával foglalkoznak. Azonban ezek az eszközök elsősorban vagy csak egy konkrét programozási nyelvre koncentrálnak, vagy bizonyos számú előre meghatározott diagramot tudnak ábrázolni.

A dolgozatomban egy olyan módszert ismertettem, amely segítségével különböző programozási nyelvhez tartozó szintaxisfa alapján tudunk gráfokat generálni. A gráfot generáló algoritmus kvázi deklaratív módon, a felhasználói igényeknek megfelelően felparaméterezhető, ezáltal a program tetszőleges vetületét képes előállítani. Figyeljük meg, hogy az algoritmusban sehol nem tettünk megkötést arra vonatkozóan, hogy a szintaxisfának milyen paradigmájú nyelvhez kell tartoznia, így a módszer kellően általános tetszőleges nyelv elemzéséhez. Az előállított gráfokból többféle kimenetet is generálhatunk: elmenthetjük őket szövegesen XML formátumban, lehetnek bemenetei más gráfábrázoló programoknak, de az eszközüm akár beépíthető más statikus kódelemzőbe is.

Ezen statikus kódelemzőkbe épülve a bemutatott módszer a kód vizualizációjánál általánosabb feladatkörökben is sikeresen alkalmazható. Ugyanis a felépített gráfokat nem csak a kód megértésére lehet használni, hanem ezek alapjai lehetnek más, bonyolultabb elemzéseknek is, mint például a helyességbizonyítás, adathatás elemzés, nem használt, illetve párhuzamosítható kódrészek felderítése.

Irodalomjegyzék

- [1] F. Nielson, H. Nielson, and C. Hankin, *Principles of program analysis*. Springer, 2004.
- [2] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen, *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer Publishing Company, Incorporated, 1st ed., 2010.
- [3] Z. Szűgyi and Z. Porkoláb, „Necessary test cases for decision coverage and modified condition / decision coverage,” *Per. Pol. Elec. Eng.*, vol. 52, no. 3–4, pp. 187–195, 2008.
- [4] M. Weiser, „Program slicing,” in *Proceedings of the 5th international conference on Software engineering*, ICSE '81, (Piscataway, NJ, USA), pp. 439–449, IEEE Press, 1981.
- [5] I. Forgács and T. Gyimóthy, „An efficient interprocedural slicing method for large programs,” in *9th International Conference on Software Engineering and Knowledge Engineering*, pp. 279–287, 1996.
- [6] I. Forgács, „The precise determination of definition-use pairs in the interprocedural case,” in *Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture-Information Processing'92, Volume 1-Volume I*, pp. 200–209, North-Holland Publishing Co., 1992.
- [7] M. Takimoto and K. Harada, „Partial dead code elimination using extended value graph,” in *Static Analysis* (A. Cortesi and G. Filé, eds.), vol. 1694 of *Lecture Notes in Computer Science*, pp. 179–193, Springer Berlin Heidelberg, 1999.
- [8] S. Diehl, *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. NJ, USA: Springer-Verlag New York, Inc, 2007.

- [9] S. S. Muchnick, *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.
- [10] E. Ray, *learning XML*. O'Reilly Media, Incorporated, 2003.
- [11] J. Ellson, E. Gansner, L. Koutsofios, S. North, and G. Woodhull, „Graphviz—open source graph drawing tools,” in *Graph Drawing* (P. Mutzel, M. Jünger, and S. Leipert, eds.), vol. 2265 of *Lecture Notes in Computer Science*, pp. 483–484, Springer Berlin Heidelberg, 2002.
- [12] „Dr. Garbage Tools.” <http://www.drgarbage.com/>.
- [13] E. Murnane and V. Sinha, „Interactive exploration of compacted visualizations for understanding behavior in complex software,” in *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pp. 729–730, ACM, 2008.
- [14] „Architexa.” <http://www.architexa.com/>.
- [15] „NDepend.” <http://www.ndepend.com/>.
- [16] A. Salomaa, *Formal languages*. Academic Press Professional, Inc., 1987.
- [17] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2001.
- [18] L. Kozma and L. Varga, *A szoftvertechnológia elméleti kérdései*. ELTE Eötvös Kiadó, 2004.