Eötvös Loránd Tudományegyetem
Informatikai Kar
Programozási Nyelvek és Fordítóprogramok Tanszék

Interaktív gráfmegjelenítés Gview

Témavezetők:
Tóth Melinda, Bozó István
Egyetemi adjunktus

Szerző:
Komáromi Mátyás
Programtervező informatikus, Bsc

Budapest, 2019

Témabejelentő

Tartalomjegyzék

1. fejezet

Bevezetés

Napjainkban új informatikai eszközök fejlesztése mellett egyre nagyobb mennyiségű szoftvert kell karbantartani. Meglévő projektekbe csatlakoznak be új fejlesztők és jelenleg is használt programok hibáinak ezreit javítjuk ki nap nap után. Ezen három és ezekhez hasonló helyzetekben kiemelkedően hasznos lehet olyan eszközök bevetése, melyek különböző módszerekkel segítik a fejlesztőt az adott kód megértésében. Egy kódbázis számos olyan tulajdonsággal rendelkezhet, melyet kézzel összegyűjteni fáradságos, akár nagyon nehéz is lehet, mint például modulok egymástól való függése, vagy egy függvény paraméterének lehetséges értékei. Egy kódmegértést támogató eszköz ilyen és ezekhez hasonló információk kinyerésével, strukturálásával és prezentációjával segíti a felhasználóját.

Számos eszköz létezik már, ami támogat kódmegértési funkciókat. Szinte mindegyik modern IDE (Eclipse, Visual Studio, stb.) megjelenít egyszerűbb részleteket, metrikákat a szerkesztett kóddal kapcsolatban, mint a sorok száma, a függvény és osztály nevét, amiben a kurzor jelenleg elhelyezkedik.

A karunkhoz kötődően is található több olyan projekt, ami más funkciók mellett, a kódmegértést is támogatja. Például az Ericsson és az ELTE által fejlesztett nyílt forráskódú CodeCompass [1], ami, akár nagyobb méretű, C/C++ és Java kódok megértését célozza meg támogatni, vagy a RefactorErl [2], melyet a karunkon fejlesztenek és szintén nyílt, egy statikus forráskódelemzős és -refaktoráló eszköz az

Erlang programozási nyelvhez és az Erlang fejlesztők mindennapos kódmegértési feladatait támogatja.

1.1. Adatvizualizáció

Adatokat sokféleképpen közölhetünk a felhasználóval, talán a legegyszerűbb a szöveges, hiszen a meghatározott összefüggések egyszerűen átalakíthatók valamely olyan leírássá, amit egy ember megért. Azonban az emberek képi megjelenítés útján nagyobb és összetettebb információt képesek befogadni.

Erre a tényre épít az adatvizualizáció, melynek lényege, hogy a strukturált adatot (információt), amit szeretnénk a felhasználónak átadni, vizuális úton tesszük meg. A vizuális megjelenítés módszerén belül számos kategória húzódik meg, például különböző grafikonok alkalmazása.

1.2. RefactorErl

A RefactorErl egy statikus kódelemző, refaktoráló és kódmegértést támogató eszköz. Az eszköz az elemzett forráskódot egy Szemantikus Programgráfban (SPG) reprezentálja. A gráf rendelkezik egy kitüntetett csúccsal, a gyökércsúccsal, melyből az elemzések által a forráskódból kinyert összes információ elérhető. Az SPG tartalmazza a lexikális, szintaktikus és szemantikus információkat a forráskódról.

Az elemzések által előállított információk legegyszerűbben a szemantikus lekérdező nyelv, illetve az előre definiált különböző gráf nézetek segítségével nyerhető ki. A RefactorErl által megvalósított gráf nézetek statikusak, így ipari méretű alkalmazások esetén nehezen átlátható, bonyolult gráfokat eredményezhetnek. Így egy fontos feladat, hogy ezen gráfokat a felhasználónak könnyen értelmezhető és interaktív formában tegye elérhetővé.

1.3. Az alkalmazás célja

A programom (Gview) célja, hogy az adatvizualizáció eszközével nyújtson segítséget a projekten dolgozó vagy éppen az újonnan csatlakozó fejlesztőknek a szoftverek megértésében, interaktív módon, gráfmegjelenítés által. Az eszközöm újítása, hogy a programkódhoz kapcsolódó gráfok széles palettáját képes megjeleníteni és köztük interaktívan váltani.

Egy csúcshoz kapcsolódóan több másik nézetet is szeretnénk elérhetővé tenni, ennek eszközei a választógombok. Ezek a gombok kisebb csúcsokként jelennek meg az aktív csúcs körül és hasonlóan viselkednek a csúcsokhoz; kattinthatók és kiválaszthatók. Azonban nem vesznek részt az elrendezés kialakításában, csupán interakciós szerepük van.

2. fejezet

Felhasználói dokumentáció

A felhasználói dokumentációban ismertetem az eszköz használatához szükséges rendszerkövetelményeket, az eszköz és függőségeinek telepítési lépéseit és az eszköz használatát.

2.1. Rendszerkövetelmények

A program futtatásához szükséges minimális hardver- és szoftverkörnyezet:

- Windows 10 operációs rendszer
- Legalább 128 MB memória
- Ötödik generációs Intel i5-tel ekvivalens vagy újabb processzor
- Az OpenGL 3.2 2009-es standardot támogató videókártya (akár integrált)
- Legalább 32 MB szabad lemezterület

A fejlesztés során az eszköz aktívan teszteltve volt Ubuntu operációs rendszeren is. Az eszköz működőképes bármely Xlib-et [3] támogató Linux rendszeren, amely az Flib könyvtár egyik függősége.

A minimum követelményen túl, a GPU alapú algoritmus hatékonyabb működéshez ajánlott hardver- és szoftverkörnyezet:

- Legalább 512 MB memória
- Legalább 256 MB VRAM
- Az OpenGL 4.3 2012-es standardot támogató videókártya (akár integrált, például Intel HD 6000)

A telepítéshez szükséges eszközök:

- GCC 8.1.0 vagy annál frissebb fordítóprogram
- GNU Make 3.81 vagy újabb
- OpenGL és Windows API fejlesztői könyvtárak
  (általában a fordító telepítésével elérhetővé válnak)

2.2. Telepítési útmutató

A program telepítése több lépésből áll és különböző függéségekkel rendelkezik. A fejezetben, a szükséges környezet és az eszköz telepítése kerül bemutatásra.

2.2.1. A GCC fordító telepítése

A GCC fordító a Gview és az általa használt Flib fordításához szükséges. Telepítéséhez, mivel a Windows 10 operációs rendszer nem rendelkezik beépített csomagkezelővel, a GCC weboldaláról [4] érdemes letölteni a programot és az ott feltüntetett lépések alapján telepíteni. A telepített GCC verzióját az következő paranccsal ellenőrizhetjük:

```
gcc --version
```

2.2.2. A RefactorErl telepítése

A dolgozatban megjelenik a gráfmegjelenítő integrációja a RefactorErl [2] statikus kódelemző és refaktoráló eszközzel. A RefactorErllel való használathoz természetesen szükséges a RefactorErl eszköz, a dolgozathoz mellékelt CD lemezen megtalálható a legújabb verziója. Az elemző legújabb verziója megtalálható a RefactorErl hivatalos weboldalán [5].

A RefactorErl, tekintve, hogy Erlang nyelven írt projektek elemzésére és refaktorálására készült, igényli az Erlang telepítését. Ehhez Windows rendszerekre az Erlang telepítési oldalán [6] kiváló útmutató található.

A telepítés első lépése a letöltött állományok kicsomagolása. A kicsomagolt mappában (a projekt gyökerében) nyissunk egy konzolt! Az eszköz lefordításához a GCC fordítóra van szükség. Kiadva az alábbi parancsot az eszköz forrása lefordul:

```
bin\referl -build tool
```

A sikeres fordítást követően megbizonyosodhatunk az eszköz működéséről, az alábbi parancsot kiadva egy Erlang shell indul el.

```
bin\referl
```

Részeletes útmutatás a telepítéshez a RefactorErl wiki oldalán [7] található.

2.2.3. Az Flib telepítése

A Gview telepítéséhez elengedhetetlen az Flib, hiszen az eszköz erre a könyvtárra épül. Az Flib könyvtár v1.1 verziója, amelyet a dolgozatban is használok, megtalálható a dolgozathoz mellékelt CD lemezen. A telepítéshez letölthetjük a könyvtár forrását annak GitHub oldaláról [8], akár tömörített állományként, akár az alábbi paranccsal, amennyiben elérhető a `git` parancs:

```
git clone https://github.com/Frontier789/Flib.git
```

Amennyiben tömörített állományként töltöttük le, tömörítsük ki a forrást!

A könyvtár fordítása az alábbi parancsokkal végezhető el:

```
cd src
make -j4
```

A fordításhoz az Flib gyökérkönyvtárában kell lennünk, illetve a GCC elérhető kell, hogy legyen a `gcc` utasítással. A fordítás sikerességéről egyszerűen meggyőződhetünk, hiszen a hibák rögtön megjelennek a konzolon. Amennyiben egy utasítással szeretnénk ellenőrizni a sikert, a parancs kilépési kódjára hagyatkozhatunk:

```
echo %errorlevel%
```

További részleteket az Flib GitHub oldalán található `README.md` nyújt.

2.2.4. A Gview telepítése

Az eszköz a RefactorErl jövőbeli kiadásának részét fogja képezni, ezért az alapértelmezett telepítési helye a RefactorErl projekt gyökérmappájából a `lib\referl_ui\gview\` úton érhető el. Amennyiben ettől különböző helyre tesszük a futtatható állományt, a RefactorErlből való használat során meg kell adnunk a módosított elérési helyet, ennek lehetséges módjáról később olvashatunk.

A mellékelt CD lemezen a Gview forrása a RefactorErl gyökérkönyvtárához viszonyítva a `lib\referl_ui\gview\` mappában helyezkedik el (ez az alapértelmezett mappa). A Gview forrása elérhető [9] a GitHub-on is.

A forrás mappájában egy új parancssort nyitva, a következő utasítással fordíthatjuk le a Gview-t, feltéve hogy az Flib is ebben a mappában található (a lemezen így van):

```
make
```

Amennyiben a gépünk CPU-ja több fizikai maggal rendelkezik, érdemes lehet párhu-

zamosan futtatni a fordítást:

```
make -j4
```

A parancsban a 4 szám a használható szálak maximális számát jelenti, ez tetszőleges pozitív egész szám lehet.

Amennyiben az Flib nem azonos mappában található a Gview-vel, a parancs paramétereként meg kell adnunk annak elérési útvonalát is:

```
make FPATH=../flib -j4
```

A példában a Gview mappájához viszonyított relatív utat adtunk meg, a `../flib` mappát, azonban ez lehet abszolút elérési útvonal is.

A parancs kiadásának hatására lefordulnak a Gview fájljai a GCC fordítóval, majd elkészül a `gview.exe` fájl.

A telepítés sikerességét ellenőrizhetjük az *exe* fájl elindításával, amennyiben minden helyénvalóan futott le, egy $5 * 5$ rács jelenik meg az ablakban. Az ablakot bezárhatjuk.

2.3. A Gview használata

Az alkalmazás kezelhető közvetlenül felhasználói interakció által a grafikus ablakban megjelenített *UI* elemek segítségével, illetve a hosztalkalmazással való integrációja által nyújtott alkalmazásprogramozási interfészen (API-n) keresztül. Az alábbiakban ezen két mód kerül tárgyalásra.

2.3.1. A RefactorErl interfész használata

A RefactorErl sikeres telepítését követően az elemző elindítható a telepítés gyökérkönyvtárából az alábbi paranccsal:

```
bin\referl
```

Az indítás után az Erlang shellhez hasonló felületet kapunk, melybe Erlang nyelven tudunk kifejezéseket írni és a RefactorErl funkcionalitásai is elérhetők.

A Gview indítása. Amennyiben a Gview az alapértelmezett helyén található a RefactorErl könyvtárszerkezetén belül, úgy az alábbi függvényhívással indíthatunk egy új példányt:

```
X = gview:start().
```

Ellenkező esetben, a `gview:start\2` függvény segítségével tetszőleges elérési utat és futtatható állománynevet is megadhatunk. Javasolt a paraméter nélküli változat használata. Példa:

```
X = gview:start("gview.exe -i erlang","C:\projects\gview").
```

Az említett két függvény visszatérési értéke a megnyitott megjelenítőhez való kapcsolatot tároló érték, tehát a példákban az `X` változó eredetileg szabad változónak kellett lennie, hogy a hívás sikeresen le tudjon futni. A visszaadott azonosítók szerkezete implementációs részlet, azaz csakis a `gview` modul függvényei építhetnek pontos kialakítására.

A `gview:start()` hívás hatására megnyílt egy üres ablak, melyben a megjelenítendő nézetek kerülnek majd kirajzolásra.

Modulnézet betöltése. A modulnézet a legmagasabb szintű elérhető nézet, tartalmazza a RefactorErl által betöltött forrásokban definiált modulokat (vagy azok egy részét, kérés szerint) és az azokban definiált függvényeket.

A használathoz először a RefactorErlbe be kell töltenie az elemezni kívánt fájlokat. A fájlok betöltése az alábbi módon történhet:

```
ri:add("test\")
```

A bemutatott példában a `test` könyvtár két fájlt tartalmaz: a *calculator.erl*-t és a *user.erl*-t, melyek tartalma a 2.1. és 2.2. ábrákon látható.

```
1  -module(user).
2
3  -export([calc/3,fib/2]).
4
5  calc(add,A,B) -> calculator:add(A,B);
6  calc(sub,A,B) -> calculator:sub(A,B);
7  calc(mul,A,B) -> calculator:mul(A,B);
8  calc(_,_,_) -> erlang:error(badarith).
9
10 fib(slow,N) -> calculate:fib_slow(N);
11 fib(fast,N) -> calculate:fib_slow(N);
12 fib(_,_) -> erlang:error(badspeed).
```

2.1. Ábra. `user.erl` modul

```
1  -module(calculator).
2
3  -export([add/2,sub/2,mul/2,fib_slow/1,fib/1]).
4
5  % some basic operations
6  add(A,B) -> A+B.
7  sub(A,B) -> A-B.
8  mul(A,B) -> A*B.
9
10 % exponential fibonacci
11 fib_slow(0) -> 1;
12 fib_slow(1) -> 1;
13 fib_slow(N) -> fib_slow(N-1) + fib_slow(N-2).
14
15 % linear fibonacci
16 fib_helper(0) -> [1];
17 fib_helper(1) -> [1,1];
18 fib_helper(N) ->
19     [A,B|T] = fib_helper(N-1),
20     [A+B,A,B|T].
21
22 fib(N) -> hd(fib_helper(N)).
```

2.2. Ábra. `user.erl` modul

A fájlok elemzésének sikerességéről a RefactorErl a konzolban ad tájékoztatást, illetve az `ri:ls()` függvény kiértékelésével ellenőrizhető, mely modulok kerültek elemzésre. Sikeres betöltés esetén a 2.3. ábrán látható kimenetet kapunk.

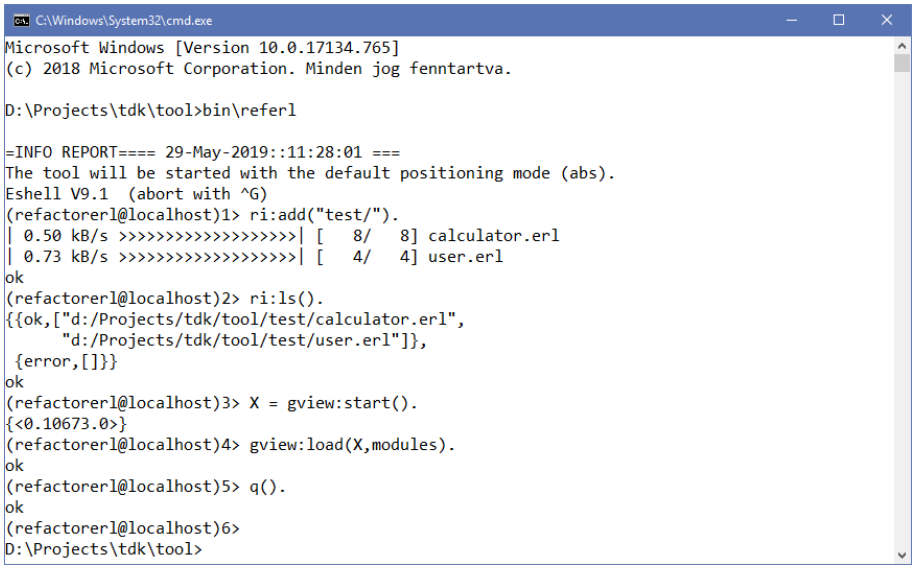
```
1 ri:ls().
2
3 {{ok,["d:/Projects/tool/test/user.erl",
4       "d:/Projects/tool/test/calculator.erl"]},
5  {error,[]}}
6 ok
```

2.3. Ábra. `user.erl` modul

Miután a fájlokat sikeresen betöltöttük, kérhetjük a modulok megjelenítését az alábbi hívással, feltéve, hogy az `X` változóban tároltuk el a kapcsolat azonosítóját:

```
gview:load(X,modules).
```

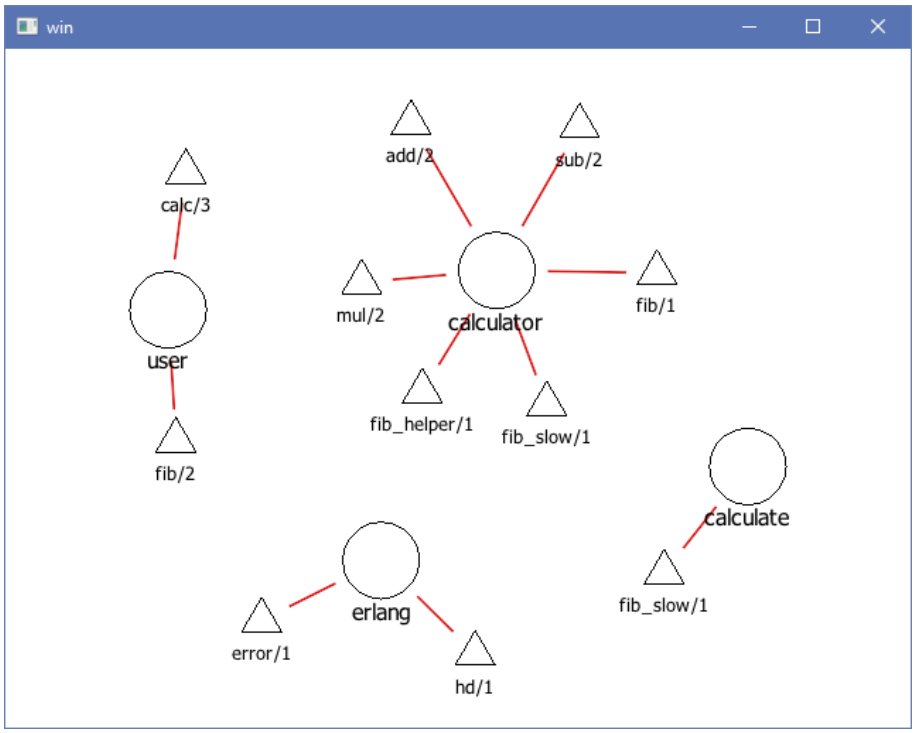
A 2.4. ábrán láthatók az eddigiekben leírt lépések és eredményük a konzolban, míg a 2.5. ábrán a megnyílt grafikus ablakban látható nézet tekinthető meg.



2.4. Ábra. A RefaktorErl indítása, egy új Gview munkamenet megnyitása, fájlok betöltése és a modul nézet.

Függvényhívási nézet. A függvényhívási nézet egy középpontjában egy függvény áll. A nézet csúcsai az e függvényt direkten vagy indirekten meghívó függvények, illetve az e függvény által direkten vagy indirekten hivatkozott függvények.

A függvényhívási nézetet az alábbi hívással lehet kérni:

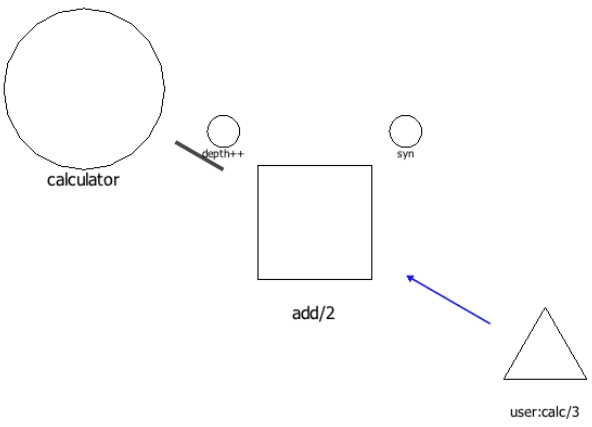


2.5. Ábra. Egy egyszerű, két fájlból álló projekt modul nézete.

```
gview:load(X,{funcall,{calculator,add,2,1}}).
```

Az `X` változó a megjelenítővel való kapcsolatot tárolja, a `calculator,add,2` a megjelenítendő függvény modulja, neve és aritása, az utolsó 1-es pedig az ábrázolni kívánt hívási mélységet jelenti.

A hívás eredményét a 2.6. ábrán láthatjuk. Az add függvényt a `user` modul `calc` függvénye hívja, míg ő nem hív más függvényeket.



2.6. Ábra. Az add függvény hívási gráfja: egyetlen függvény hívja.

A gráfleíró szerkezete. Tetszőleges gráf kirajzolásához szükséges elkészítenünk annak egy gráfleíróját. Ez egy egy Erlang `map` típusú érték.

A `map`-ek kulcs-érték párok tárolására és azok hatékony elérésére használt adatszerkezetek.

A gráfleíró `map` elemei opcionálisak, a lehetséges kulcsokat és jelentésüket a 2.1. táblázat tartalmazza.

kulcs	típus	jelentés	alapérték
nodes	[any]	Csúcsazonosítók listája	[]
node_palette	[npale]	Csúcstípusok	[]
edge_palette	[epale]	Éltípusok	[]
labels	[string]	Csúcscímkék listája	[]
node_wts	[int]	Csúcssúlyok listája	[]
node_tys	[int]	Csúcsok típusindexekei	[]
edges	[{any,any}]	Élek listája	[]
edge_wts	[int]	Élek súlya	[]
edge_tys	[int]	Élek típusindexei	[]
selectors	[[id]]	Csúcsonként a választógombok	[]
tooltips	[string]	Eszkezleírók szövege	[]

2.1. Táblázat. A gráfleíró `map` kulcsai és értelmezése

Az `epale` egy él kinézetét leíró típus, a `{float,atom,atom,[int,int,int]}`-el egyezik meg. A rendezett négyes első eleme az él vastagsága, második és harmadik eleme az él kezdetének és végének az alakja (lehetséges értékek: `circle`, `square`, `triangle`, `none`), az utolsó elem pedig és az él színének *RGB* kódja.

Az `npale` egy csúcs kinézetét leíró típus, `{int,atom}`-el egyezik meg. A rendezett pár első eleme a csúcs méretét, a második pedig a formáját adja meg. A csúcs formája `circle,square,triangle,none` értékeket veheti fel.

A választógombok `id` típusa kétféle lehet, `{string,any}` vagy `string`. A `string` mindkét esetben az eszközleíróban megjelenő szöveget tárolja.

Az élek párok listájaként kerülnek reprezentálásra, ezért a párok elemei csak létező csúcsok azonosítói lehetnek.

A 2.1. lista egy példa egy nyolc elemű kör lehetséges leírására. A példagráfban a

csúcsok címkéi felváltva `1`-ek és `0`-ák. Az élek vékony, szürke nyilak, az `a` csúcs a többitől nagyobb háromszög, a többi csúcs kör alakú. Az `a` csúcsnak van csak eszközleírója, ami a `"Hello"` szöveg, megjelenítéskor idézőjelek nélkül.

2.1. Felsorolás. Példa gráfleíróra Erlang nyelven, ami egy 8 hosszú kört ír le.

```
#{
    nodes => [a,b,c,d,e,f,g,h],
    labels => [1,0,1,0,1,0,1,0],
    edge_palette => [{3,none,triangle,[80,80,80]}],
    node_palette => [{7,circle},{10,triangle}],
    tooltips => ["Hello"],
    node_tys => [1,0,0, 0,0,0, 0,0,0],
    edges => [{a,b},{b,c},{c,d},
              {d,e},{e,f},{f,g},
              {g,h},{h,a}]
}
```

Tetszőleges gráf kirajzoltatása. A Gview lehetőséget biztosít tetszőleges gráf kirajzolására. Ehhez egy a fent leírt módon specifikált gráfleíróra van szükség.

A megjelenítéshez az alábbi függvényhívás használható:

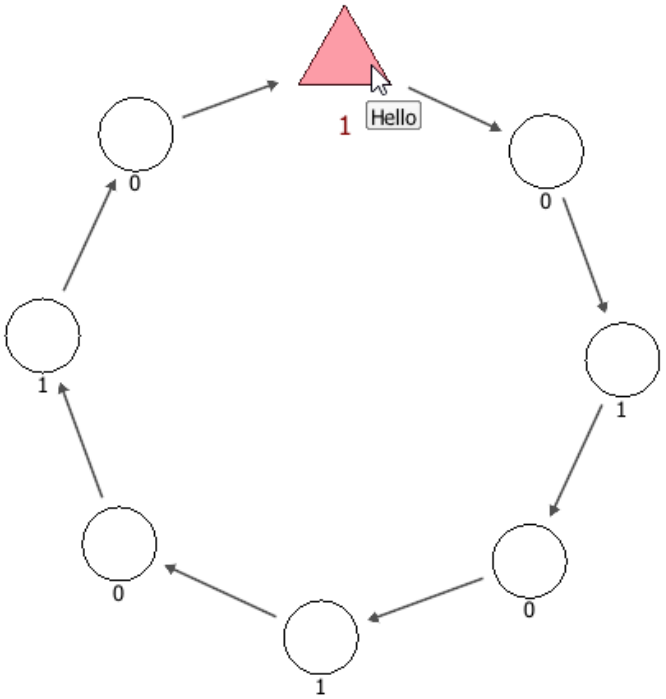
```
gview:plot(X,G).
```

Amennyiben a `G` változó a 2.1. listában szereplő értéket tárolja, úgy az előbbi hívás hatására a 2.7. ábrán látható kör rajzolódik ki. Az ábrán a `"Hello"` eszköleíró is megfigyelhető.

Ahol `X` a kapcsolat azonosítója `G` pedig a gráf specifikációját tárolja.

Kapcsolat ellenőrzése. A megjelenítővel való kapcsolat ellenőrzésére az alábbi hívás használható:

```
gview:status(X,1000).
```



2.7. Ábra. Egy egyszerű, 8 csúcsból álló kör megjelenítése az erő alapú elrendezéssel.

Ahol az `X` a kapcsolat azonosítója, az `1000` pedig a maximális várakozási idő, milliszekundumban, opcionális paraméter.

Lehetséges kimenetek:

- `ok` – a kapcsolat ép;
- `{exit, normal}` – a kapcsolat rendeltetésszerűen leállt;
- `{error, E}` – a kapcsolat hibával állt le, a hiba oka az `E` kifejezés értéke;
- `timeout` – a megadott várakozási idő lejárt, azaz a monitor nem kapott választ a megadott időn belül. Ez előfordulhat abban az esetben is, ha hibás kapcsolat azonosító került megadásra.

Visszavonás és ismétlés. A nézetek közötti váltások során felépül egy visszavonási verem, melynek segítségével visszaállhatunk előző nézetekre. Ehhez az alábbi függvény használható:

```
gview:undo(X).
```

A visszavonás során az aktuális nézet is az ismétlési verembe kerül, így a történetben előre is tudunk lépni. Ehhez az alábbi függvény használható:

```
gview:redo(X).
```

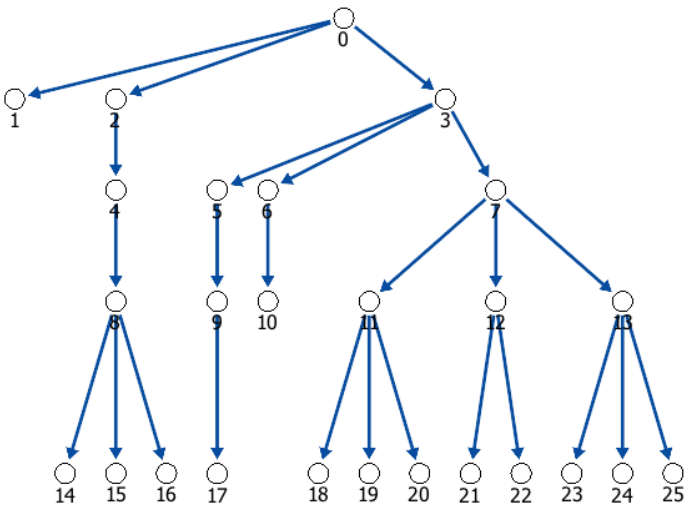
Mindkét függvény paramétere a megjelenítővel való kapcsolat, visszatérési értéke pedig a `gview:status(X)` hívás eredménye.

Elrendező algoritmus váltás. A Gview több elrendező algoritmust is támogat. Ezek között az alábbi függvény segítségével válthatunk:

```
gview:layout(X,fd).
```

Ahol `X` a kapcsolat azonosítója, `fd` pedig az erő alapú elrendezést azonosítja. Az `fd` helyett más algoritmus azonosítója is írható, például `layered`, a hierarchikus elrendezéshez. Visszatérési értéke pedig a `gview:status(X)` hívás eredménye.

Egy hierarchikus elrendezésű fa megjelenítése látható a 2.8. ábrán.



2.8. Ábra. Ötös mélységű fa tesztgráf.

A Gview leállítása. A megjelenítő bármikor leállítható az alábbi hívással:

```
gview:stop(X).
```

Paramétere a kapcsolat azonosítója, visszatérési értéke a `gview:status(X)` hívás eredménye.

2.3.2. Felhasználói interakció

A felhasználó a megnyitott grafikus ablakkal az egér és a billentyűzet segítségével léphet interakcióba. Ennek lehetséges módjait részletezem lentebb.

Síkbeli transzformációk. A megjelenített nézet dinamikus, értve ez alatt azt, hogy a virtuális kamerával, ami a kirajzolt gráf színterét veszi, képesek vagyunk pásztázni, nagyítani, elforgatni, stb.

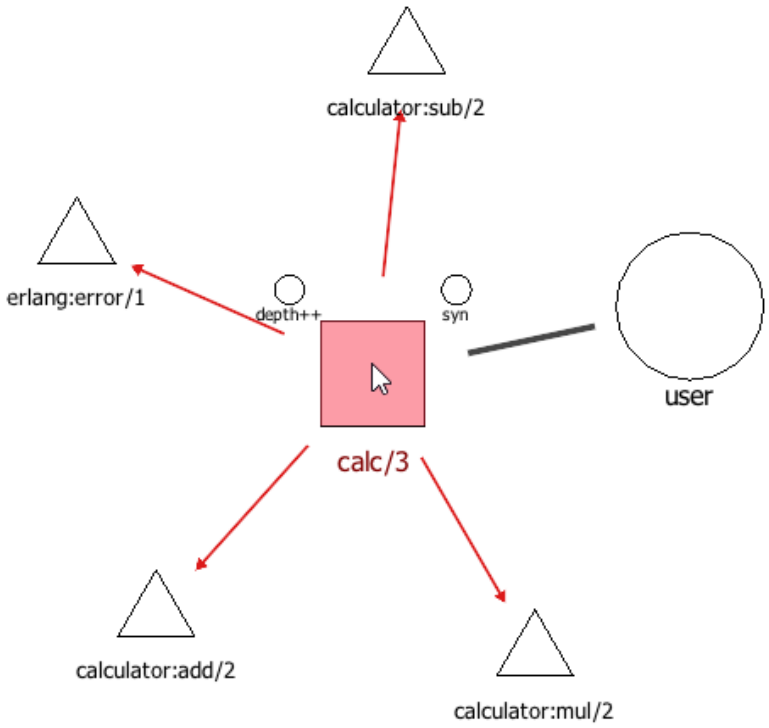
A kamera fókuszát az egér bal gombjának lenyomásával és lenyomva tartásával, majd a kurzor mozgatásával tudjuk változtatni. Ezen folyamat során az egérmutató és a kirajzolt gráf relatív helyzete nem változik, azaz a gráf követi az egér mozgását.

Nagyítani az egér görgőjével tudunk, előre görgetés során a gráf közelebb kerül a kamerához, hátra görgetéskor távolodik. A módszer természetesen működik a laptopok érintőpárnájának gesztusaival is, hiszen ezek is görgetési eseményeket váltanak ki.

A forgatáshoz az egér jobb gombját használhatjuk. Ezen gomb lenyomásával kijelölésre kerül a forgatás középpontja. Amennyiben nem engedjük fel a gombot, a mutató mozgásával tudjuk az elforgatás szögét beállítani.

Csúcsok és választógombok. Amennyiben egy csúcsra megfelelő mértékben ráközelítünk, megjelennek a csúcs válaztógombjai. Ezen választógombokra és a csúcsokra lehet az egérmutatóval kattintani.

Egy aktív csúcs, mely egy függvényt reprezentál és a két választógombja látható a 2.9. ábrán.



2.9. Ábra. Aktív csúcs és választógombjai.

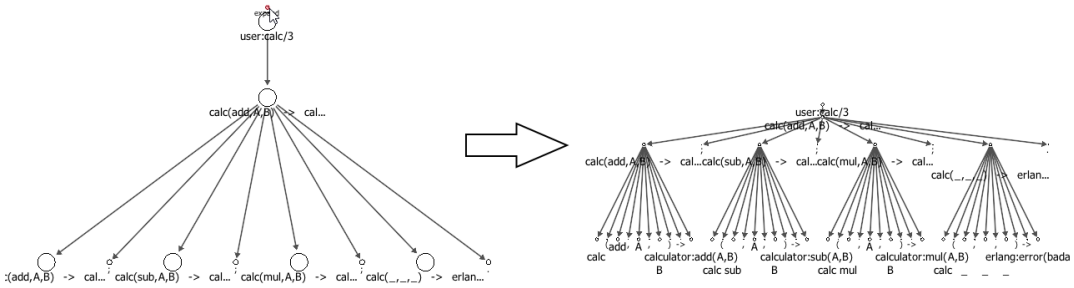
Mikor az egeret egy csúcs, vagy a csúcs címkéje fölé viszi a felhasználó, a csúcs és a címkéje bordó árnyalatot kap, ezzel jelezve, hogy aktívvá vált. Hasonlóan járhatunk el választógombok esetén.

Az egér lemozdítása az aktív csúcsról ismét fehérré teszi azt, jelezve hogy nem aktív a csúcs ezen túl.

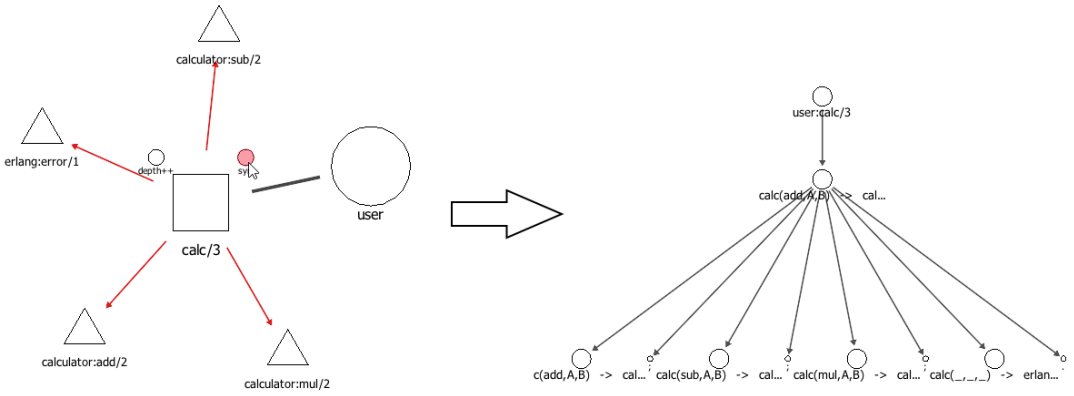
Egy csúcsra vagy választógombra mutatva és kattintva az ahhoz tartozó nézet kerül megjelenítésre. A nézet pontos típusa függ az adott csúcstól, amire kattintunk.

Például egy függvényt reprezentáló csúcsra kattintva, annak hívási nézete jön elő. Hívási gráf központi csúcsához tartozik egy mélyítés és egy szintaktikus fa választógomb. A mélyítésre kattintva, egy szinttel mélyül a hívási gráf. A szintaktikus fa gombjára kattintva a függvény szintaktikus fája kerül megjelenítésre. Ezen lehetőségekre mutatnak példát a 2.10. és a 2.11. ábrák.

Eszközleírók. Eszközleírók alatt olyan szövegbuborékokat értünk, amik akkor jelennek meg, amikor az egérmutatót hosszabb ideig egy objektum felett egy helyeben



2.10. Ábra. Szintaxis fa mélységének növelése, választógombbal.



2.11. Ábra. Függvényhívási nézetből szintaxisfa nézetbe való váltás.

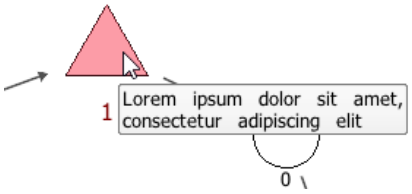
tartjuk.

Az eszközleíróban megjelenő szöveg a nézettől függ, általában az adott csúcs teljes címkéjét tartalmazza, amikor az túl hosszú, ilyenkor a címke lerövidítésre kerül.

Az eszközleíró buborékja követi az egérmutatót, amíg az el nem hagyja az adott objektumot. Ekkor a buborék eltűnik.

A buborék megjelenési és késleltetési ideje 200ms.

Egy csúcsra mutatva megjelenő eszközleíró látható a 2.12. ábrán.



2.12. Ábra. Egy csúcsra mutató egér és a megjelenő eszközleíró.

Billentyűkombinációk. A program indításától és jelenlegi állapotától függően különböző billentyűkombinációk érhetők el.

A kilépéshez az Esc gomb használható. Amennyiben hierarchikus megjelenítést alkalmazunk, a nézet 90 fokos forgatásához a CTRL+R kombináció használható. A jelenlegi elrendezés újbóli számítását az R billentyűvel kérhetjük. Nézetváltást a Backspace illetve a CTRL+Z kombinációkkal lehet kiváltani. Nézetváltás ismétlésére a CTRL+Y szolgál.

2.4. Példák nagyobb gráfokra

A RefactorErl eszközt ipari méretű alkalmazások elemzésére tervezték. Noha az előző fejezetben csak kisebb gráfokat használva illusztráltam a funkcionalitást, természetesen az eszköz képes nagy méretű forráskódok kezelésére is.

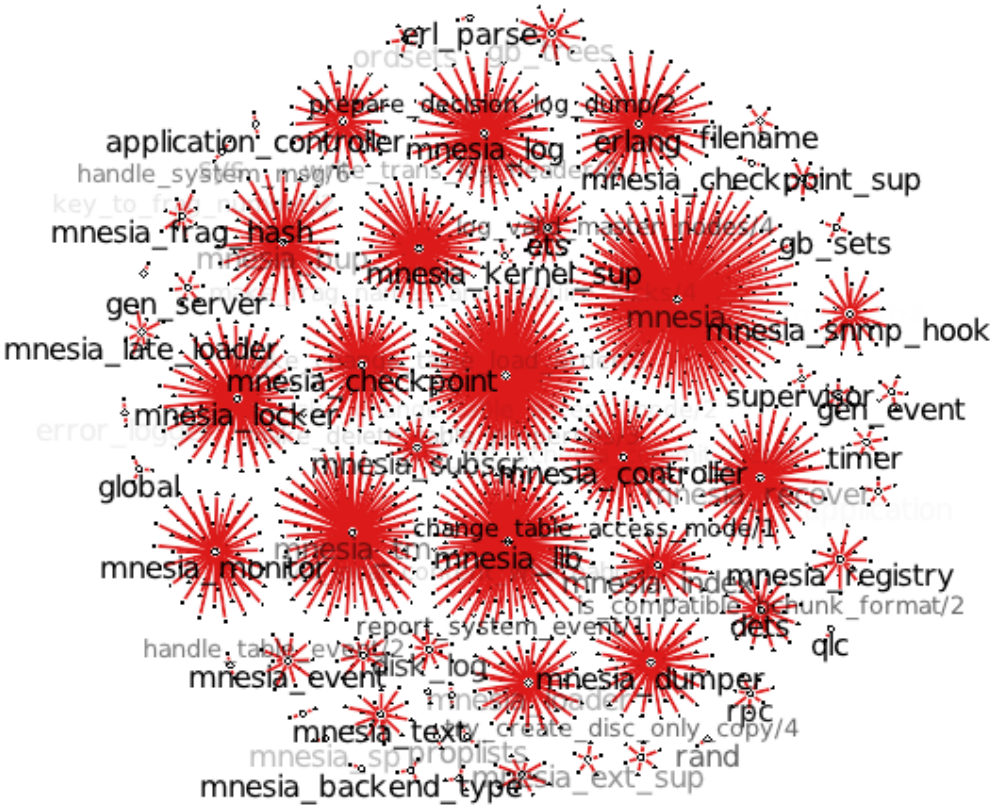
A 2.13. ábra az Erlang Mnesia [10] beépített adatbázisának betöltése után előálló modul nézetet mutatja.

Amennyiben nem a teljes gráfot szeretnénk látni, akkor kattinthatunk a megtekinteni kívánt modul nevére vagy a RefactorErl segítségével is szűkíthetjük a nézetet egy modulra:

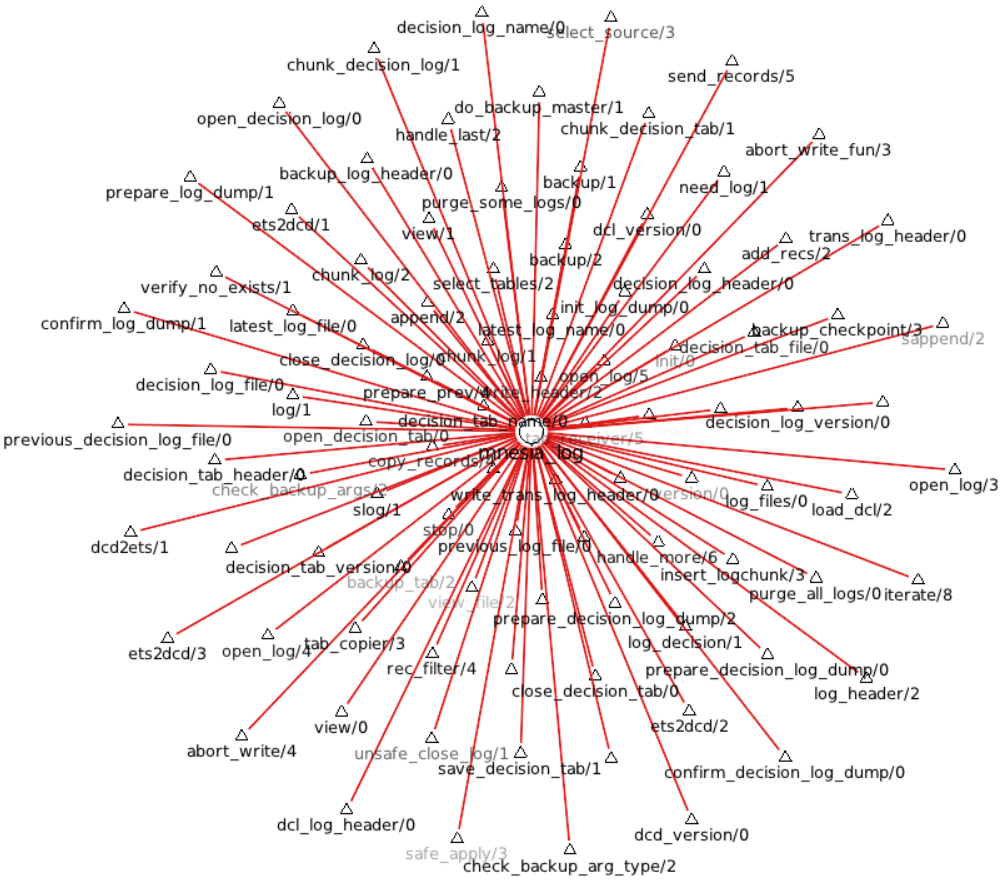
```
gview:load(P, [{modules, [mnesia_log]}]).
```

Az `mnesia_log` modulra való szűkítés eredménye a 2.14. ábrán látható.

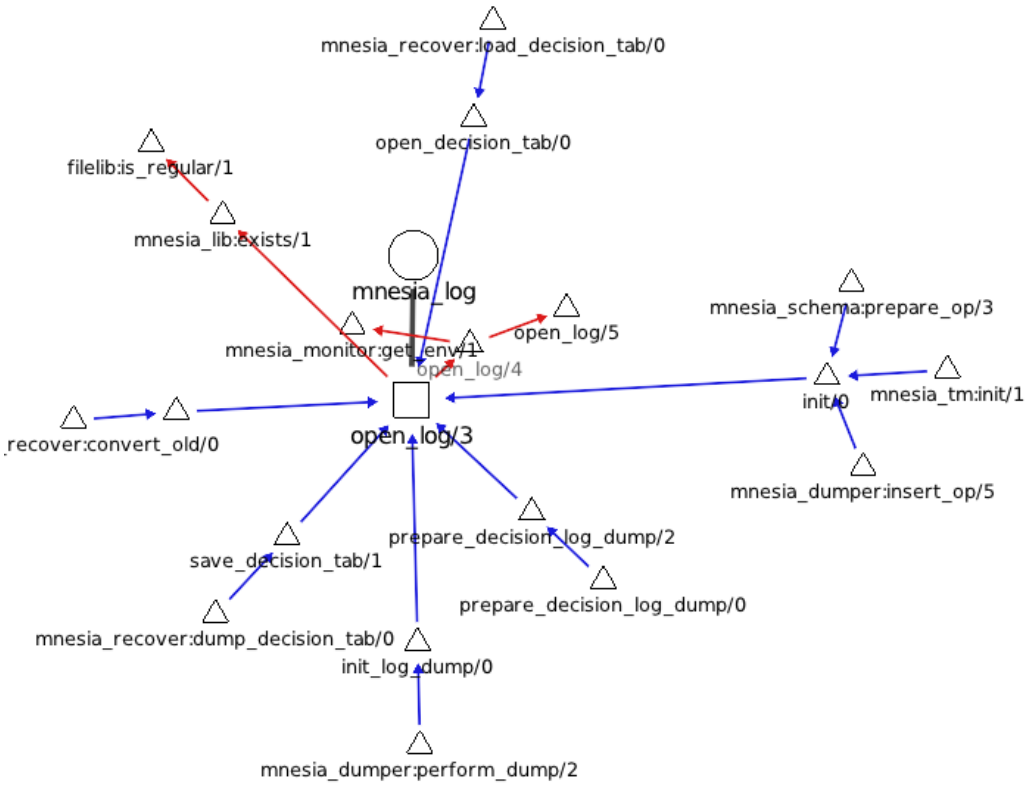
Amennyiben az `open_log/3` függvényre kattintunk a 2.14. ábrán, akkor az adott függvény függvényhívási gráfjának nézetét kapjuk (2.15. ábra). Az élek színei különböztetik meg a hívó és hívott függvényeket.



2.13. Ábra. Az mnesia alkalmazás modul nézete



2.14. Ábra. mnesia_log modul nézete



2.15. Ábra. `open_log/3` függvény hívási gráfja

3. fejezet

Fejlesztői dokumentáció

A fejlesztői dokumentáció leírja a program célját, részletezi a felhasznált technológiákat és a program felépítését és végül az eszköz tesztelését. Kifejtésre kerül a RefactorErllel való integráció is.

3.1. A kitűzött feladat

A dolgozatban bemutatott megoldás egy olyan eszköz elkészítése, mely interaktív gráfmegjelenítéssel képes támogatni meglévő eszközök kódmegértési funkcionalitásait. Az eszköz (Gview) össze lett kapcsolva a RefactorErl keretrendszerrel, így az Erlang nyelven történő fejlesztést, kódelemzést és refaktorálást, az *SPG* részeinek megjelenítésével, további kódmegértési funkciókkal tudjuk támogatni.

A Gview által a grafikus ablakban nyújtott szolgáltatások:

- a kért nézet elrendezése és megjelenítése, interaktív bejárása;
- a kapcsolódó nézetek közötti gyors váltás, a felhasználó igénye szerint;
- az elrendező algoritmustól függően, annak megjelenítési paramétereinek állítása;

- olyan nézetek esetén, ahol értelmes és célszerű, azok mélyítésére való lehetőség és
- a nézetváltás visszavonása és ismétlése.

A Gview RefactorErl interfészének konzolból és Erlang nyelven a RefactoErlből elérhető szolgáltatásai:

- új, a jelenleg futó példányoktól független Gview munkamenet indítása;
- beépített paraméteres nézet megjelenítése egy futó munkamenetben;
- tetszőleges nézet (custom view) megjelenítése egy futó munkamenetben;
- elrendező algoritmus váltása futás közben vagy futás előtt;
- nézet- és elrendezésváltás visszavonása illetve ismétlése;
- csúcscímkék megjelenítési stratégiájának váltása;
- a nézet nagyítás és eltolás alapú középre igazítása és
- a megjelenítővel való kapcsolat állapotának lekérdezése.

3.2. Felhasznált technológiák

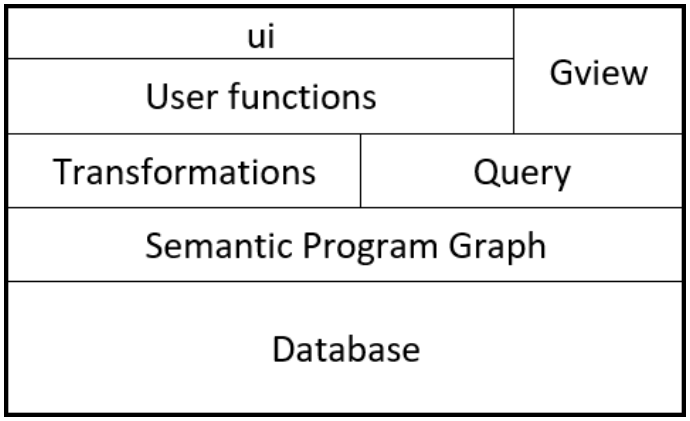
3.2.1. RefactorErl

A RefactorErl egy statikus kódelemző és refaktoráló eszköz, melyet az Eötvös Loránd Tudományegyetem Programozási Nyelvek és Fordítóprogramok tanszékén fejlesztenek 2006 óta. Az eszköz célja létező Erlang kódbázisok elemzése és refaktorálása.

Az elemző a vizsgált Erlang szoftvert egy egyedi gyökércsúccsal ellátott, irányított gráfként ábrázolja, melynek élei és csúcsai címkékkel vannak ellátva. Ezt a gráfot a Szemantikus Programgráfnak nevezzük (SPG röviden). A gráf az elemzett kód

absztrakt szintaxisfáján alapul, azonban szemantikus és lexikális információval is kiegészül. Ezen gráfban megtalálható információk részeit, nézeteit szeretnénk megjeleníteni a Gview használatával.

A Gview integrációja az *SPG* felett definiált lekérdező szolgáltatásokra épít, ahogy az a 3.1. ábrán is látható.



3.1. Ábra. A Gview integrációjának illeszkedése a RefactorErl keretrendszerébe.

A RefactorErl az elemzés eredményeként előálló SPG-t adatbázisban tárolja. A forráskód változása esetén ezt inkrementálisan módosítja. A különböző felhasználói funkcionalitások (kódmegértés támogatása, lekérdezések, refaktorálások, függőségi elemzések, stb) az SPG-re épülő transzformációs és lekérdező rétegre épülnek, nem férnek hozzá közvetlenül az adatbázishoz sem. A rendszer legkülső szintje egy úgynevezett `ui` réteg, ahol a különböző felhasználói felületek csatlakoznak az eszközhöz.

A felhasznált függvények rövid leírása. A RefactorErl által a `referl_lib` alkalmazáson[1] keresztül biztosított függvények közül az alábbiakra épít a Gview.

A `?Query:exec`[2] függvény segítségével az *SPG*-ben adott kezdőcsúcsokból kiindulva kérhetünk le kapcsolódó csúcsokat. Például a `?Mod:locals()` függvényhívást használva megkapjuk az adott modulban definiált függvényeket. Hasonlóan ehhez a függvényhívási láncot is be tudunk járni a segítségével.

[1] Erlang application

[2] A `?Query` egy RefactorErl által definiált makró, amely a `reflib_query` modul nevére fejtődik ki. A makrók definíciói header fájlokban adottak. A későbbiekben is hivatkozok hasonló módon modulnevekre.

Egy kattintott csúcs típusának kiderítéséhez a `?Graph:data` függvényt használtam. Az adatfolyam nézeteket a `?Dataflow:reach_1st` függvény segítségével valósítottam meg, mely adott kezdőcsúcsból, amennyiben az egy változót azonosít, a belőle kifele (vagy befele) befolyó adatok célját (vagy forrását) gyűjti össze.

Szintaktikus fákból álló nézeteket a `?Syn:children` függvénnyel állítok elő, amivel egy szintaktikus csúcs közvetlen gyermekeit lehet lekérdezni. Ilyen nézetek esetén az eszközleíró tartalmát a `?Syn:flat_text` függvénnyel határozom meg, ami az adott szintaktikus csúcs és leszármazottai által felölelt forráskód részletet adja vissza.

Ezeken kívül az alábbi egyszerű lekérdező függvényeket használtam:

- `?Fun:arity` – az adott függvény csúcs paraméterszáma;
- `?Fun:mod_fun_arity` – az adott függvény csúcs moduljának neve és paraméter száma;
- `?Fun:module` – az adott függvény csúcs modulja;
- `?Fun:name` – az adott függvény csúcs neve;
- `?Mod:name` – az adott modul csúcs neve.

3.2.2. Flib

Az Flib saját projektem, melynek célja számítógépes grafikához kényelmi eszközök biztosítása. A könyvtár C++ nyelven íródott és nyílt forráskódú. A forrás megtalálható GitHub-on:

`https://github.com/Frontier789/Flib`

A könyvtár négy részre tagolható, melyek közül az elsőben a *System* a többi alapjául szolgáló matematikai és alapvető működést segítő osztályok kapnak helyet, az `fm` névtérben. Ezek közül a legfontosabbak, melyeket a Gview is használ, a `vector2<T>`,

amely egy tetszőleges aritmetikai `T` típus feletti kétdimenziós vektort valósít meg. Megtalálhatók ennek specializációi, például `vec2i` és `vec2f`, illetve három- és négydimenziós változatai (`vector3<T>` és `vector4<T>`). A *System* modul másik használt tagja a `Result`, amely egy hiba leírását képes tárolni. Megtalálhatók még ezeken kívül mátrix osztályok, kvaterniók stb.

A `Graphics` modul a `System` modulra építve alapvető grafikus primitíveket tartalmaz az `fg` névtérben. A Gviewban is használt `Image` osztály képek betöltésére a CPUra, manipulálására és elmentésére alkalmas. Az itt található osztályok egy része csomagoló osztály egy-egy OpenGL objektum körül, így biztosítva objektumorientált támogatást a koncepciók használatára. Ilyen például a `Texture`, amely az `Image` GPU oldali megfelelője vagy a `Shader`, amely egy GPUn elhelyezkedő, GLSL nyelven írt árnyaló kódot reprezentáló objektum. Az erő alapú elrendezés egy speciális shader típusra építve lett párhuzamosítva, a `ComputeShader`-re.

A grafikai osztályoktól függetlenül létezik a `Window` modul, mely az operációs rendszer natív könyvtárára építve valósít meg egységes ablakkezelési interfészt. Ilyen Windows-on a WINAPI, Linuxon pedig az Xlib. A modul tartalmaz OpenGL kontextus kezelésére alkalmas osztályt is, `GLContext` néven.

Az ablakkezelőre és a grafikus osztályokra épít a `Gui` modul. A modul célja teljes funkcionalitás biztosítása grafikus felhasználói felület készítéséhez. Az Flib GUI modellje gui elemekből (`GuiElement`) áll, amik hierarchiába szerveződnek. Az elemek képesek események kezelésére (`onEvent`), frissíthetők (`onUpdate`) és képesek önmaguk kirajzolására (`onDraw`). A hierarchia fájában minden csúcs gui elem, azonban a belső csúcsok ennél specifikusabbak, úgynevezett layout-ok (`GuiLayout`). A layoutok az eseményeket minden gyermeküknek továbbítják, csakúgy, mint a frissítési és rajzolási kéréseket. A közös adatokat, amiket minden elemnek el kell tudnia érni, egy GUI kontextus tárolja (`GuiContext`). Grafikus ablakba a GUI kontextust megvalósító GUI ablak által kerülhetnek ki az elemek (`GuiWindow`).

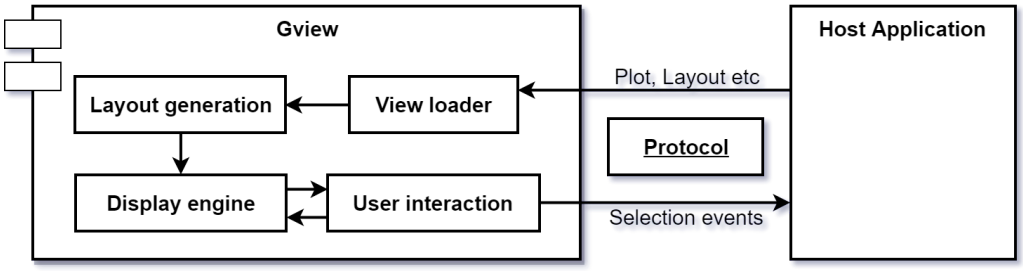
A Gview erősen épít az Flib GUI funkcionalitására, a megjelenítést egy `GuiWindow`-ban végzi, több osztálya is úgy került megtervezésre, hogy az Flib által biztosított GUI hierarchiába illeszkedjen.

3.3. A Gview felépítése

A Gview C++ nyelven íródott, a megvalósítás során a teljes feladathoz meghatározott részfeladatok megvalósítására osztályokat vezettem be. A Gview négy modulra tagolódik, amik rendre az adatátvitelért, az elrendezésért, a megjelenítésért és az interkacióért felelősek. Erről ad áttekintést a 3.2. ábra.

3.3.1. A kód szerveződése

Az alább részletezett osztályok mindegyike saját fájllal és fejállománnyal rendelkezik. Az implementációs fájlok (*.cpp*) az *src* mappában találhatók. A fejállományok az *include* mappában találhatók. A fordítást a gyökérben található *makefile* írja le.



3.2. Ábra. A Gview négy modulja, melyek az eszköz különböző funkcionalitásait adják.

A program indításakor a `main.cpp` fájlban található `main` függvény kerül meghívásra. Ezen függvény a parancssori argumentumok kezelése után, az azok által meghatározott alkalmazás osztályt példányosítja és futtatja.

A C++ oldali működést meghatározó osztályok leírása a 3.3.2. alfejezetben található.

A C++ kódban található osztályok egy része csupán adattárolásra használatos, ezek részletesen a 3.3.3. alfejezetben kerülnek ismertetésre.

Ezen adattípusok segítségével történik a kommunikáció az Erlang oldali és a C++ oldali kód között, melynek C++ oldali osztályai a 3.3.4. alfejezetben vannak kifejtve.

Az importált gráfok csupán a gráf kinézetét írják le, elrendezést nem határoznak

meg. Ezen feladat az elrendező algoritmusokat megvalósító osztályok feladata, melyek a 3.3.5. alfejezetben kerülnek ismertetésre.

Az elkészült elrendezés a fogadott megjelenítési adatokat tényleges geometriává alakítását és kirajzolását a 3.3.6. alfejezetben kifejtett osztályok valósítják meg.

Végül, a felhasználói interakciókat a 3.3.7. alfejezetben tárgyalom részletesen.

3.3.2. Alkalmazás osztályok

A következő modulok adják az alkalmazás gerincét. Ezek funkcionalitása közvetlenül határozza meg az alkalmazás viselkedését.

Az alkalmazás osztályok öröklődési UML diagramját mutatja a 3.3. ábra.

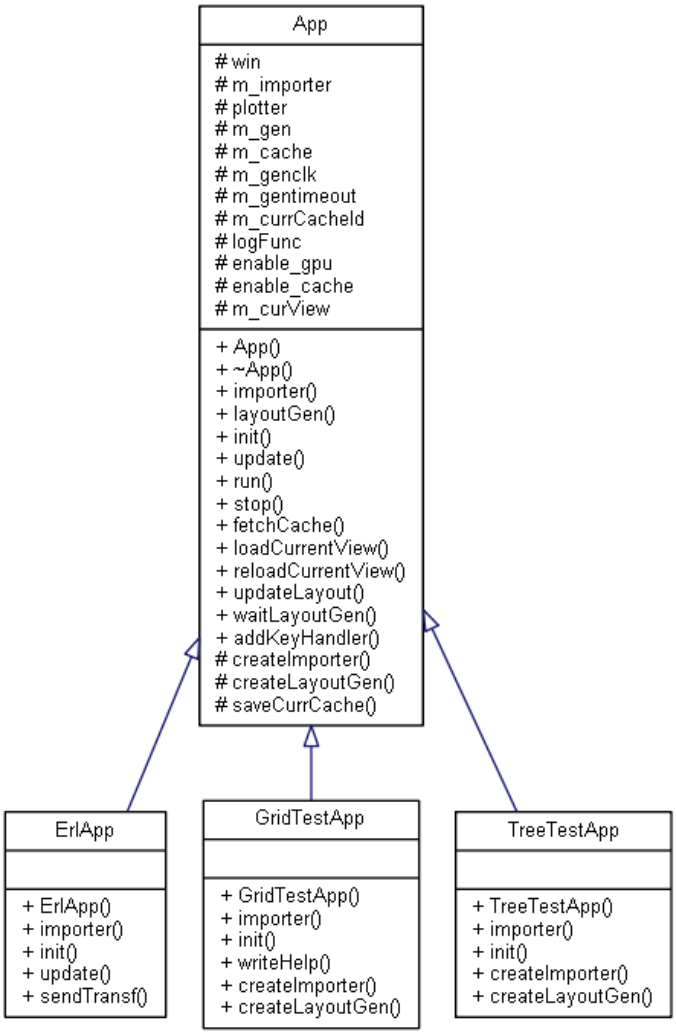
App. Az `App` osztály a viselkedés osztályok alapja, tartalmazza a közös funkcionalitásokat. A belőle leszármazó osztályok adják az alkalmazás egyes működési módjait.

Konstruktorának paramétere egy *log* függvény és két logikai érték, melyek megszabják, hogy az alkalmazás használhat-e GPU-t és, hogy használhat-e gyorsítótárat.

Az `App` osztály tárol egy importálót és egy elrendezés generálót. Ezeket lekérdezni az `importer()` és a `layoutGen()` tagfüggvények hívásával lehet. Az alkalmazás incializációja az `init` függvényben történik, ennek visszatérési értéke a lehetséges hibaleírója. Az alkalmazás periodikusan meghívja az `update` függvényét, melyet a leszármazott osztályok implementálhatnak. Ezen négy függvény felüldefiniálható a leszármazott osztályokban, ezzel módosítva az alkalmazás működését.

Az alkalmazás futtatása a `run` függvénnyel történik, megállítása a `stop` függvénnyel.

Egy adott azonosítójú nézethez, a `fetchCache` függvénnyel kérhetjük le a tárolt elrendezést az alkalmazásban tárolt adattárból.



3.3. Ábra. Az `App` osztály és a belőle leszármazó osztályok UML diagramja.

A jelenlegi nézet be- és újra töltésére szolgál rendre a `loadCurrentView` és a `reloadCurrentView` tagfüggvény.

Leszármazott alkalmazások. Az `App` osztályból származik az `ErlApp` osztály, mely `ErlImporter`-t használ importerként és futása során kezeli az Erlangból érkező üzeneteket. Az osztály ezért felülírja az `importer`, az `init` és az `update` függvényeket.

További osztályok is származnak le az `App` osztályból, `GridTestApp` és `TreeTestApp`. Ezen osztályok célja és működése közel azonos: teszt alkalmazások, melyek speciális imprtálókat használnak. Ezen importálók a `GridImporter` és a `TreeImporter`.

3.3.3. Adatleíró osztályok

Ezek az osztályok nem egy konkrét feladatot vagy algoritmust valósítanak meg, hanem összetett adatokat tárolnak.

CacheManager. Az osztály célja a számításigényes elrendezések perzisztálása. Az elrendezéseket `string`-ekkel azonosítjuk.

Az oszály példányosítható egy `string` paraméterrel, ekkor a paraméter által azonosított fájlból megpróbálja betölteni az ott tárolt mentéseket.

Jobb megoldás, ha a `loadFromFile`-t használjuk fájlba mentett cache betöltésére, mivel ez a visszatérési értékében jelzi az esetleges hibát.

A tárolt elrendezések elmenthetők fájlba a `saveToFile` tagfüggvénnyel, melynek paramétere a fájl neve, visszatérési értéke az esetleges hiba leírója.

A `loadFromFile` és a `saveToFile` függvények rendelkeznek fájlnév nélküli változattal, ekkor a legutóbbi `setFile` hívással beállított fájl kerül használatra.

Az elmentett elrendezés a `load` függvénnyel tölthető be, mely visszaadja a paramé-

ter azonosítóhoz tárolt elrendezés `Layout`-ját. Amennyiben nincs tárolva elrendezés egy azonosítóra, de mégis lekérdezzük a `load` függvénnyel, úgy egy üres elrendezést kapunk eredményül. A `check` függvénnyel lehet ellenőrizni, hogy egy adott azonosítóhoz van-e mentett elrendezés. A paramétere az elrendezés azonosítója és egy logikai értékkel tér vissza.

Új elrendezést felvezetni a tárba a `save` függvénnyel lehetséges, paraméterei az elrendezés azonosítója és az elrendezés. A tárolt elrendezések megsemmisíthetők a `clear` függvény meghívásával.

Layout. A megjelenítendő nézet egy lehetséges elrendezését tárolja, ennek megfelelően csupán egy kétdimenziós vektorok vektorát tárolja `positions` néven.

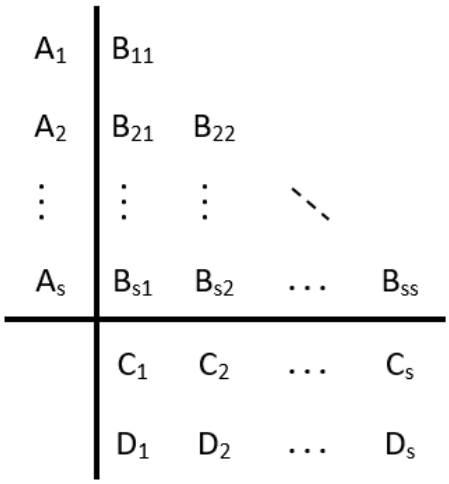
Kényelmi függvénye a `size` függvény, ami visszaadja a csúcsok számát.

LogLevel. Egyszerű `enum` típus, melynek célja egy *log* művelet jelentőségének a mértékét tárolja el.

Lehetséges értékei:

- `LOG_NOTHING = -1` – nincs üzenet;
- `LOG_ERROR = 1` – kritikus üzenet, mindenképp megjelenítendő;
- `LOG_WARNING` – valószínűleg hibát okozó körülményt jelez;
- `LOG_INFO` – összetett művelet sikeres befejezését jelzi;
- `LOG_DATA` – adatátviteli hibakeresésre használatos üzenetek;
- `LOG_DEBUG` – általános hibakeresési üzenetek.

RKDesc. Beágyazott Runge-Kutta módszer Butcher-tábláját tárolja el. A bővített Butcher-tábla szerkezete a 3.4. ábrán látható. Mivel az erő alapú elrendezésnek nincs szüksége a B együtthatómátrixra, az `RKDesc` ezt nem tárolja.



3.4. Ábra. Bővített Butcher-tábla.

Az osztály publikus adattagjai:

- `double h0` – a kezdő lépésköz;
- `double N_h` – a magasabb rendű módszer konvergenciarendje;
- `double N_e` – a beágyazott módszer konvergenciarendje;
- `double eps` – maximális engedélyezett hiba lépésenként, pixelekben;
- `vector<double> A,C_H,C_E` – rendre az A, C és D vektorok a Butcher-táblában.

View. A `View` osztály feladata egy megjelenítendő gráf minden grafikus adatának tárolása, ennek megfelelően jelentős méretet ölthet, és érdemes csak ritkán lemásolni.

A gráf ábrázolása éllistásan történik, a `graph` vektor adattag tárolja minden csúcsra az őt leíró adatokat (egy `Node` példányt).

A `Node` adattípus egy csúcs összes adatát tárolja; az őt reprezentáló alakzat méretét és alakját (`visuals`), a címkéje szövegét és az eszközleírójának szövegét (`label`), a pozícióját és "fontosságát" (`body`), a hozzá tartozó választógombok címkéit és azonosítóit (`selectors`) illetve a belőle kifele mutató élek vektorát (`edges`).

Az `edges` vektor elemei `Edge` típusúak, így nem csak azt tárolják, hogy az él másik végén melyik csúcs helyezkedik el (`to`), de az él grafikus tulajdonságait, mint a vastagság, élvégződések, szín (`visuals`) és az él "fontosságát" is (`strength`).

A tárolt gráf jelenlegi elrendezését a `getLayout` tagfüggvénnyel lehet lekérdezni, visszatérési értéke az elrendezés. Új elrendezés a `setLayout`-tal állítható be, paramétere az új elrendezés.

Az osztály által biztosított lekérdező függvények:

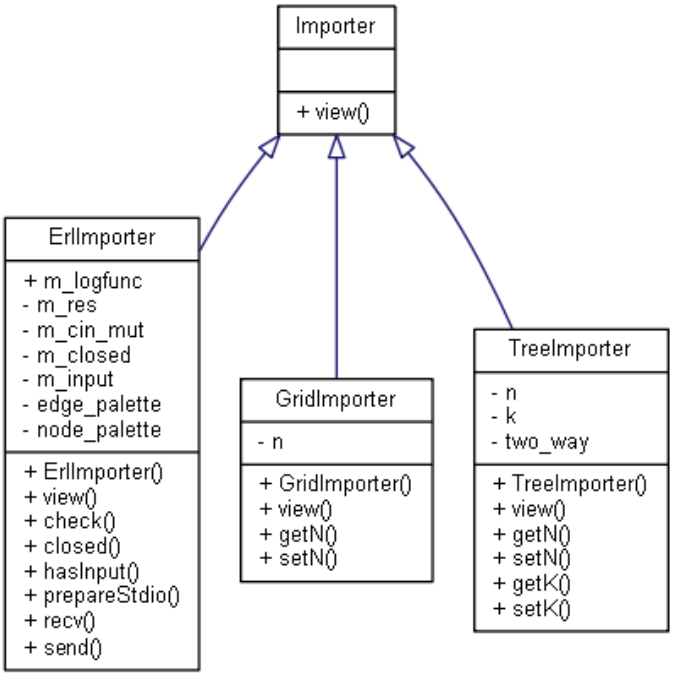
- `size_t size() const` – a csúcsok számát adja vissza;
- `rect2f aabb(mat4 transf) const` – tengelyirányú befoglaló téglalapot számol adott transzformáció mellett;
- `vector<string> selectorLabels(size_t nodeId) const` – adott csúcshoz tartozó választógombok címkéjét kérdezi le;
- `size_t probe(mat4 transf,vec2 mp) const` – adott transzformáció mellett megvizsgálja, melyik csúcson helyezkedik el egy pont a képernyőn;
- `bool isJoint() const` – megvizsgálja, hogy a gráf összefüggő-e;
- `string getHash() const` – kiszámít egy nagy valószínűséggel egyedi hasító értéket a gráf szerkezetéből.

3.3.4. Gráf importáló rész

Egy-egy nézet leírójának betöltése, illetve az események közvetítése a hosztalkalmazás irányába az `Importer` absztrakt osztályt megvalósító leszármazott osztályok feladata.

Az importáló osztályok öröklődési UML diagramját mutatja a 3.5. ábra.

Az `Importer` osztály által definiált tisztán virtuális `view()` függvény használható a legfrissebb nézeti gráf betöltésére. A kommunikációs protokoll megvalósításából

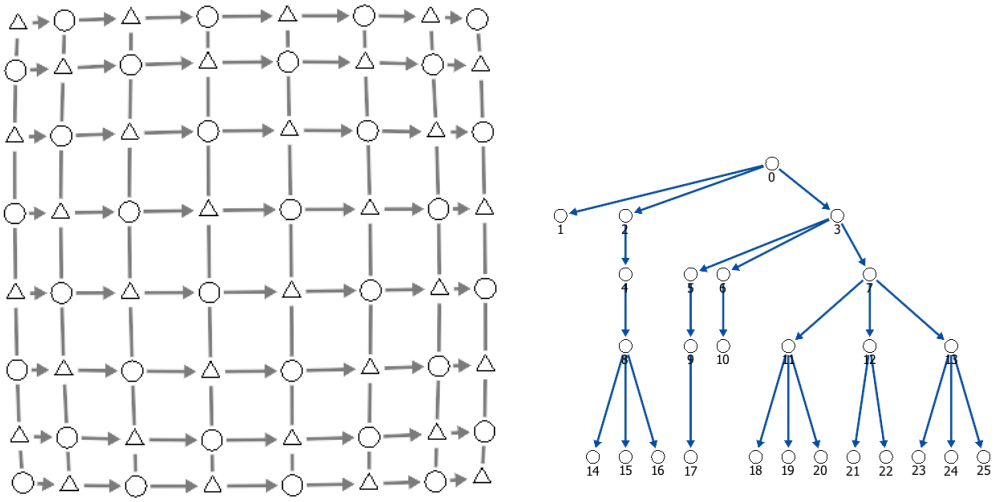


3.5. Ábra. Az `Importer` osztály és a belőle leszármazó osztályok UML diagramja.

adódó interfész függvények nem jelennek meg az `Importer` osztályban, hiszen nem minden beviteli mód tudja támogatni üzenetek fogadását, mint például a teszt importálók, amik egy végtelen nagy fix gráfot jelenítenek csupán meg.

A TreeImporter és GridImporter tesztelők. A `TreeImporter` és a `GridImporter` osztályok megvalósítják az `Importer` absztrakt osztályt, tesztgráfok generálását valósítva meg. Mindkét osztály rendelkezik paraméterekkel, amelyeket *setter* és *getter* metódusokkal lehet beállítani illetve lekérdezni. A két importáló rendre egy $N * N$-es rács-gráf és egy determinisztikus véletlenszerű fagráfot generál mikor meghívjuk a `view()` függvényüket. A `GridImporter` paramétere a rács mérete, míg a `TreeImporter` két paramétere a generált fa mélysége és sűrűsége állítható. A 3.6. ábrán látható példa a két tesztgráfra.

ErlImporter. Az `ErlImporter` megvalósítja az `Importer` absztrakt osztályt, Erlang Portokon keresztül történő kommunikációra alapozva. Az osztály példányosítása a *standard input és output* birtokbavételét eredményezi, mivel ezeken a leírókon



3.6. Ábra. Nyolcszor nyolcas rács tesztgráf és ötös mélységű fa tesztgráf.

keresztül zajlik majd a kommunikáció az Erlang oldali kóddal. A megvalósított `view()` függvény a hosztalkalmazás által elküldött legutóbbi gráfleírót fogadja.

A `send` és `recv` tagfüggvények feladata a kétirányú kommunikáció biztosítása, mindegyik változatuk rendre adat küldésre és fogadásra használható. Ezek a függvények tipikusan a megfelelő `App` megvalósításban használatosak. A függvények nincs visszatérési értéke, lehetséges paraméterezésik:

- void recv (View &view, Delegate< void, LogLevel, string > logFunc)
- void recv (size_t &length)
- void recv (vector< size_t > &arr)
- void recv (vector< float > &arr)
- void recv (string &str)
- void recv (void *data, size_t bytes)
- void send (size_t num)
- void send (string str)
- void send (const vector< float > &arr)
- void send (const void *data, size_t bytes)

3.3.5. Elrendezésgenerálás

Az megoldott feladat központi kérdése hatékony, interaktív és informatív elrendezések meghatározása. Ennek megfelelően mindegyik elrendező algoritmus saját osztállyal rendelkezik.

Az elrendező osztályok öröklődési UML diagramját mutatja a 3.7. ábra.

LayoutGen. A `LayoutGen` osztály az összes elrendező algoritmusra jellemző közös interfészt és funkcionalitást tartalmazza. Elindít egy *worker* szálat, melyen az elrendező algoritmus számításai futhatnak, így nem zavarva a *GUI* szál működését.

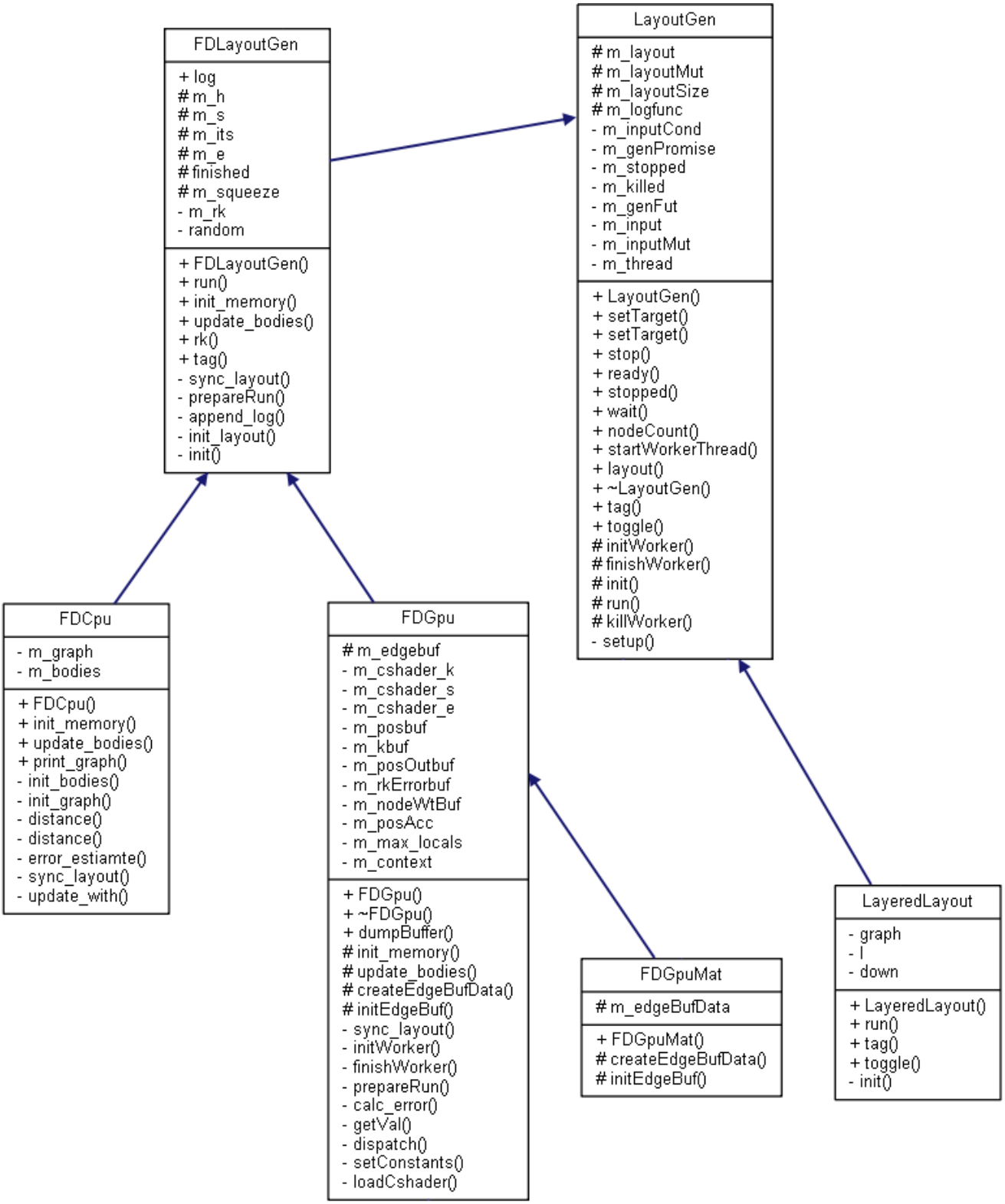
Az elrendező algoritmusok átvesznek egy *log* függvényt a konstruktorukban, mely segítségével a detektált hibákat log fájlba tudják írni.

A `setTarget` függvény új nézet érkezéséről informálja az elrendező motort, opcionálisan egy *cachelt* elrendezést is átadva. Visszatérési értéke a függvényben potenciálisan keletkezett hiba.

A `startWorkerThread` paraméter és visszatérési érték nélküli függvény elindítja és inicializálja a számítási szálat, az adott algoritmus igénye szerint. Az osztály példányosítása után az első `setTarget` hívás előtt kell egyszer meghívni a tulajdonosnak.

A futó elrendezési algoritmust meg lehet állítani, le lehet kérdezni, hogy befejezésre került-e illetve, hogy leállt-e, továbbá lehet várni a befejeződésére. Ezen funkciókat az alábbi függvények látják el, értelem szerint a feladat angol nevével ellátva:

- void stop ()
- bool ready () const
- bool stopped () const
- void wait (Time timeout)



3.7. Ábra. A `LayoutGen` osztály és a belőle leszármazó osztályok UML diagramja.

Lekérdezhető az elrendező algoritmus által a legutóbbi lépésben szinkronizált elrendezés a `layout` metódussal, melynek visszatérési értéke egy `Layout`. Amennyiben a számításokat végző szál éppen szinkronizál, úgy a hívás blokkolódik, amíg az be nem fejeződik, tehát ez egy szálbiztos művelet.

Az algoritmusok rendelkeznek továbbá egy egyedi címkével, melyet a `tag` függvénnyel kérdezhetünk le. További tulajdonságok állíthatók a `toggle` függvénnyel, azonban ez első sorban az Erlangból érkező kérések feldolgozására használatos.

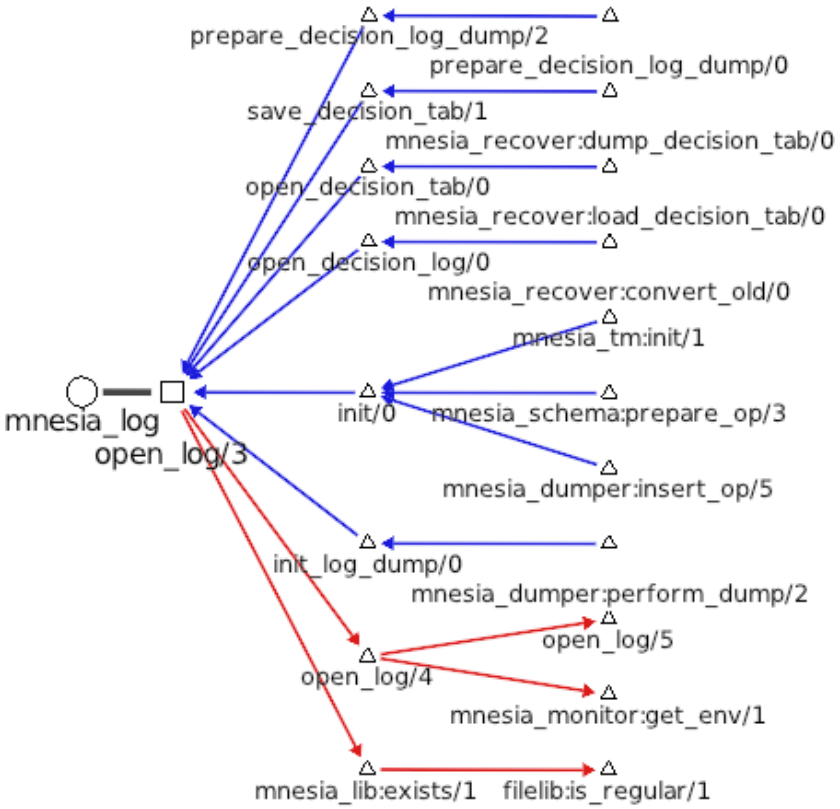
A leszármazott osztályoknak a `run`, az `initWorker`, a `finishWorker` és az `init` függvényeket kell megvílósítaniuk, melyek rendre a számítási szálon futtattandó elrendezés-előállító függvény, a számítási szálon esetlegesen szükséges kontextust felépítő és lebontó függvények és a *GUI* szálon tárolt adatokat a számítási szálra mozgató eljárások. Az `initWorker` és a `finishWorker` rendelkeznek alapértelmezett üres megvalósítással, így nem tisztán virtuálisak.

LayeredLayout. A `LayeredLayout` megvalósítja a hierarchikus rétegelt elrendezés algoritmusát, implementálja az ősosztály tisztán virtuális metódusait.

A `run` metódus végrehajtja az elrendező algoritmus lépéseit, majd az elkészült elrendezést szinkronizálja a *GUI* szállal és kilép. Nincs szükség köztes szinkronizációra, hiszen az algoritmus milliszekundumok alatt lefut.

Az `init` függvény megvalósítása a nézetleíróból kimásolja a gráf szerkezetére vonatkozó adatokat, hogy azokon szabadon tudjon dolgozni a számítási szálon futó algoritmus. Az osztály `tag` függvénye a konstans `"layered"` szöveget adja vissza. A 3.8. ábrán látható egy példa hierarchikus elrendezésre.

FDLayoutGen. Az erő alapú elrendezés két megvalósítása jelenik meg a dolgozatban: egy a CPU-ra tervezett és egy masszívan párhuzamosított. Ezen két megvalósítás különböző környezetet, erőforrásokat igényel, ezért a közös funkcióik kiemelésre kerültek az `FDLayoutGen` osztályba. Mindkét megoldás a Runge-Kutta [11, 12] módszercsaládra épít, mint matematikai eszközre a fizikai rendszer szimulációja során.



3.8. Ábra. Az Mnesia alkalmazás open_log/3 függvényéhez tartozó hívási gráf hierarchikus elrendezése.

Az osztály példányosításához a *log* függvényen kívül egy Runge-Kutta módszer leíró objektumra is szükség van, mivel az algoritmus paramétere, hogy pontosan melyik módszert alkalmazza.

Az `FDLayoutGen` képes elmenteni futás közben az algoritmus lépésenként generált adatait, mint a lépésköz vagy az approximációs lépések száma, hogy később elemezhető legyen. Ezen funkcióját a konstruktor `do_logging` paramétere szabályozza.

A `run` függvény felállít egy kezdeti elrendezést, majd az erő alapú elrendezés algoritmusát futtatva lépéseket tesz a fizikai rendszer fixpontja felé. A belső ciklus pontosan addig fut, amíg egy előre meghatározott közelségbe nem kerül két egymást követő iteráció vagy valaki le nem állítja a generálást. Minden lépésben megtörténik az aktuális elrendezés szinkronizációja, így a hosszú ideig generálódó elrendezések (nagy gráfok esetén) a felhasználó szeme előtt alakulnak, ezzel mutatva, hogy az alkalmazás dolgozik.

A `run` függvényben az elrendezés frissítése az `update_bodies` függvényben történik, amit a leszármazott osztály valósít meg. Az implementáció számára szükséges memóriát a leszármazott osztályok az `init_memory` függvény implementálásában tudják lefoglalni, melynek paramétere az új nézet, visszatérési értéke az esetleges hiba.

Lekérdezhető az algoritmus által használ Runge-Kutta módszer leírója a `rk` függvénnyel, a `tag` függvény visszatérési értéke `"force_directed"`.

FDCpu. Az erő alapú elrendezés algoritmusát valósítja meg, a CPU-n. Leszármazik a `FDLayoutGen` osztályból és megvalósítja annak `init_memory` és `update_bodies` függvényeit.

Konstruktora az `FDLayoutGen` konstruktorával megegyező paraméterezésű.

Definiál egy `print_graph` függvényt, mely a paraméterben megadott kimeneti folyamra kiírja a feldolgozott gráf szerkezetét, hibakeresés céljából.

FDGpu. Az `FDGpu`, hasonlóan a `FDCpu`-hoz, az erő alapú elrendezést valósítja meg, azonban a GPU-ra optimalizálva. Leszármazik a `FDLayoutGen` osztályból és megvalósítja annak `init_memory` és `update_bodies` függvényeit.

Konstruktora az `FDLayoutGen` konstruktorával megegyező paraméterezésű.

Az osztály rendelkezik két tisztán virtuális metódussal, melyek a konkrét adattárolási sémát kill kialakítsák, `createEdgeBufData`, melynek paramétere a jelenlegi gráf leírója és `initEdgeBuf`, melynek nincs paramétere. Mindkét függvény visszatérési értéke a potenciális hiba leírója. Az első az élek tárolásához használt grafikus memória lefoglalását végzi. A második ezen memóriába tölti fel a kezdeti elrendezést.

FDGpuMat. Az `FDGpuMat` megvalósítja a `FDGpu` osztály két virtuális függvényét. A konkrét módszer, amivel tárolja a gráf élet a GPU-n az egy textúrára ültetett szomszédsági mátrix.

Konstruktora az `FDLayoutGen` konstruktorával megegyező paraméterezésű.

3.3.6. Megjelenítés

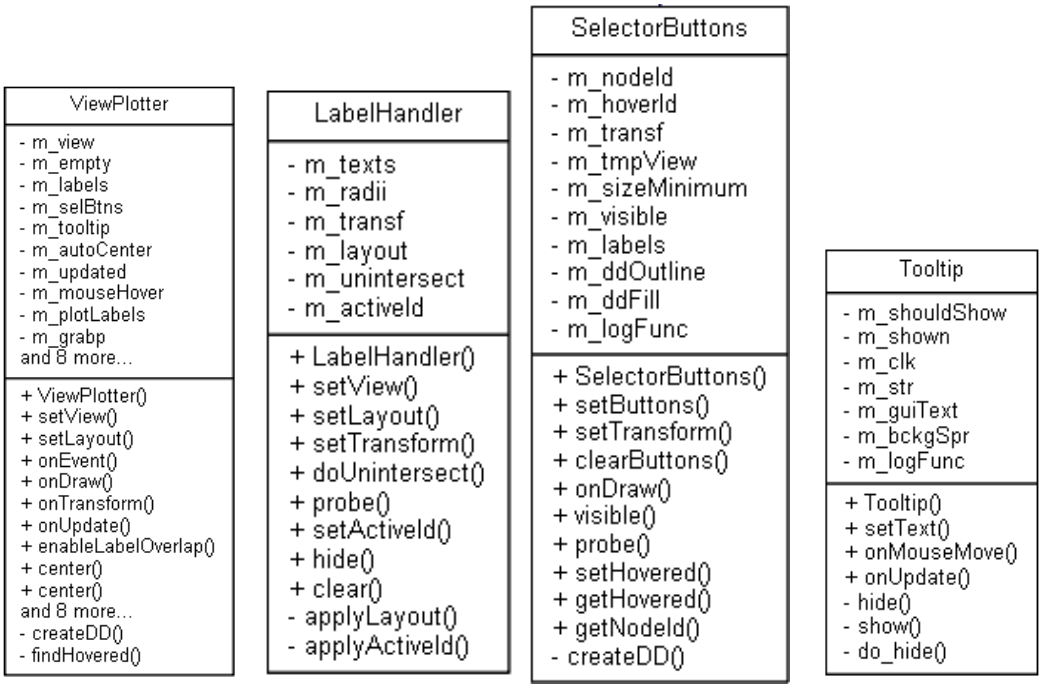
Az elrendező algoritmusok által kiszámított elhelyezkedés, melyet `Layout` típusú változóban tárolunk, csupán a csúcsok helyét írja le, nem a valódi geometriát, amit meg szeretnénk jeleníteni.

A megjelenítéshez használt osztályok UML diagramját mutatja a 3.9. ábra.

A megjelenítéshez először el kell készíteni a geometriát az elrendezés alapján (tesszellálni) majd azt felküldeni a GPU-ra és kirajzolni.

ViewPlotter. A `ViewPlotter` osztály fogja össze a megjelenítés különböző feladatait megvalósító osztályokat, ez az osztály felelős magáért a gráf kirajzolásért.

Az osztály leszármazik a `GuiElement`-ből, tehát az *Flib* GUI hierarchiájába illeszt-



3.9. Ábra. A megjelenítéshez használt négy osztály UML diagramja.

hető, illetve leszármazik a `TransformListener`-ből is, így a felhasználói transzformációs eseményekről, mint az eltolás és forgatás, értesítést kap.

Egy `ViewPlotter`-ben új nézet megjelenítéséhez előbb jelezni kell, hogy változott az aktív nézet, a `setView` függvénnyel, melynek egyetlen paramétere az új nézet leírója.

Ezután `setLayout` hívásokkal adhatjuk át az új elrendezést a megjelenítőnek, ami elvégzi a geometria kiszámítását és a megjelenítést a jelenlegi nézeten.

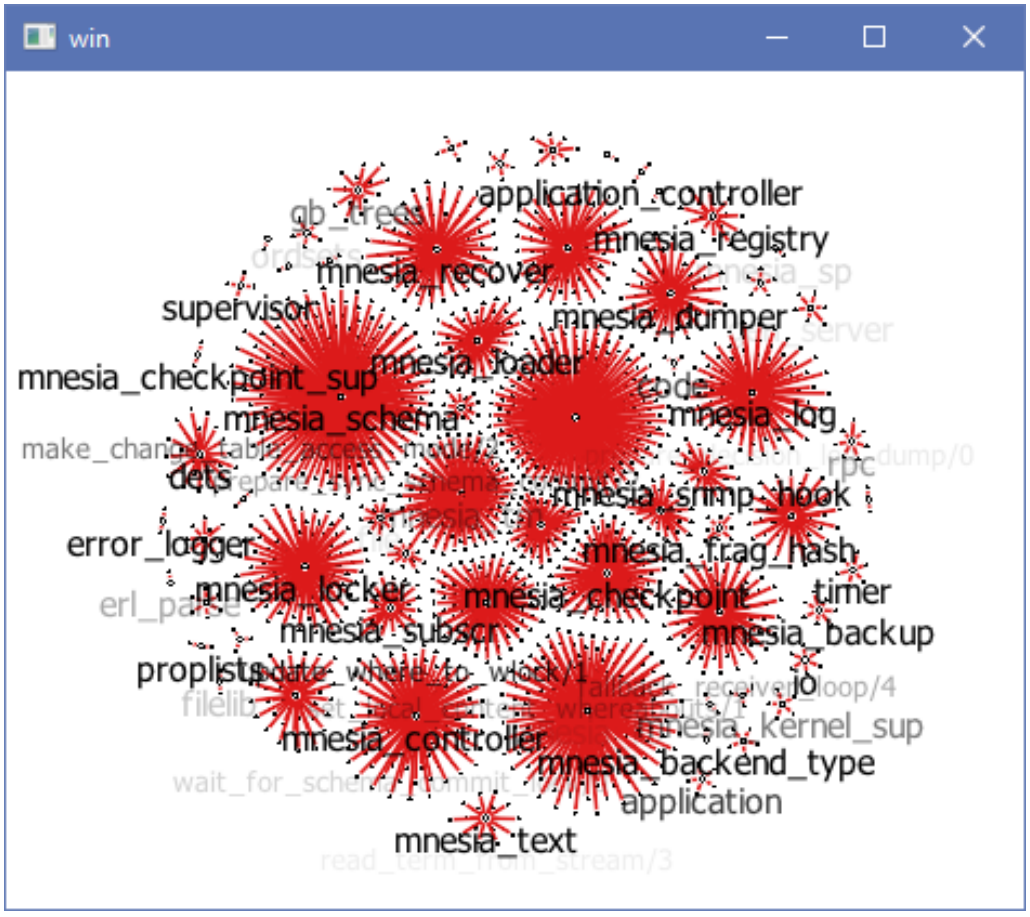
Ahhoz, hogy beilleszkedjen a GUI hierarchiába, az osztály felüldefiniálja az alábbi függvényeket, rendre az eseménykezelésre, a rajzolási felszólításra, a transzformációs frissítésekre és az időleges frissítésekre:

- bool onEvent(fw::Event &ev)
- void onDraw(fg::ShaderManager &shader)
- void onTransform(bool userAction)
- void onUpdate()

A csúcsokhoz tartozó címkék átfedését le lehet tiltani, ekkor az amúgy átfedő csúcsok lentebb kerülnek, illetve engedélyezni az `enableLabelOverlap` tagfüggvénnyel, melynek paramétere az átlapolás engedélyezése. A címkék le is tilthatók teljesen az `enableLabels(false)` hívással.

A nézet középre igazítható (transzformációk alaphelyzetbe állítása) illetve egy adott téglalapra fókuszálható a `center` függvény nulla és egy paraméteres változatával.

Az *Mnesia* adatbáziskezelő moduljai és függvényei láthatók a 3.10. ábrán, középre igazítva.



3.10. Ábra. Az *Mnesia* adatbáziskezelő moduljai és függvényei.

Az osztály `loadShaders` függvénye betölti a *technikához* használt shadereket, eredménye a lehetséges hiba leírója.

LabelHandler. A `LabelHandler` feladata a megjelenített gráf csúcsaihoz tartozó címkék kirajzolása. Ennek megfelelően, az osztály az Flib GUI hierarchiájába illesz-

kedik, tehát leszármazik a `GuiElement` osztályból.

Konstruktora átveszi a GUI kontextust és egy paramétert arra, hogy a címkék átfedését engedélyezze-e.

Új nézet betöltése esetén, hasonlóan a `ViewPlotter` osztályhoz, előbb a `setView` tagfüggvény hívásával kell jelezni, majd a `setLayout` tagfüggvény hívásaival lehet az elrendezést frissíteni. A két függvény paraméterezése azonos a `ViewPlotter`-ban feltüntetett, annyi eltéréssel, hogy a `setView` nem másolja le a paraméterét.

Mivel a transzformációs eseményeket számon tartjuk a `ViewPlotter` osztályban, ezért elegendő, ha a `LabelHandler` csak a transzformációs mátrix legújabb értékét kapja meg minden iterációban. Erre a célra a `setTransform` függvény használható, melynek egyetlen paramétere az új transzformációs mátrix.

A címkék átfedése engedélyezhető illetve letiltható a `doIntersect` tagfüggvénnyel.

A `probe` tagfüggvény paramétere egy kétdimenziós pont, az ablak koordinátarendszerében, visszatérési értéke pedig annak a címkének a sorszáma, amire a pont transzformáltja illeszkedik. Amennyiben több ilyen címke is létezik, a nagyobb területű címke kerül választásra. Ha nincs ilyen címke, `size_t(-1)`-el tér vissza a hívás.

Az egér által mutatott csúcs, azaz az aktív csúcs címkéje megkülönböztető színt kaphat, ezért a `setActiveId` függvénnyel közölhető a `LabelHandler`-el, hogy melyik az aktív csúcs. Paramétere az aktív csúcs sorszáma.

A `hide` tagfüggvény elrejti a címkéket, a `clear` pedig meg is semmisíti a címkéket megvalósító `GuiText` objektumokat.

SelectorButtons. Az aktív csúcs körül megjelenő választógombok kezelésére használatos osztály. Feladata a gombok elhelyezése, megjelenítése és a gombokkal történő esetleges felhasználói interakció kezelése.

Szintén *GUI* elemként került megvalósításra, így konstruktora átveszi a *GUI* kon-

textust.

A `setButtons` függvénnyel állíthatjuk be, hogy melyik csúcson és milyen választógombok jelenjenek meg. Paraméterei a gombok címkéi egy vektorban, a csúcs azonosítója és leírója.

Az osztály a `LabelHandler`-hez hasonlóan a `setTransform` függvényén keresztül fogad el transzformációs változásokat, melyekhez igazítja a kirajzolást.

A megjelenített gombok eltüntethetők a `clearButtons` tegfüggvény használatával.

A `LabelHandler`-hez hasonlóan lekédrezhető az egérmutató alatt elhelyezkedő választógomb sorszáma a `probe` tagfüggvénnyel. Paramétere az egérmutató koordinátái az ablak koordinátarendszerében, visszatérési értéke azonban nincs, az eredményről a `getHovered` függvénnyel tájékozódhatunk.

Az, hogy az osztály jelenleg jelenít-e meg gombokat a `visible` függvénnyel kérdezhető le.

Tooltip. A `Tooltip` osztály feladata az eszközleíró (tooltip angolul) megjelenítése, mozgatása és elrejtése.

Az osztály *GUI* elemként van megvalósítva, konstruktora átveszi a GUI kontextust paraméterként. Továbbá leszármazik a `MouseMoveListener` osztályból, így automatikusan frissítéseket kap az egérmutató helyzetéről, amit le tud reagálni.

Az aktuálisan megjelenített eszközleíró szöveget a `setText` függvénnyel lehet beállítani, melynek paramétere egy UTF-32 kódolású string az Flib-ből, így lehet tetszőleges UNICODE szöveg. Üres szöveggel való meghívása elrejti az eszközleírót, amíg ez meg nem történik, látható marad.

Az `onMouseMove` és `onUpdate` függvények az Flib-bel való integrációhoz szükségesek. Az `onMouseMove` az egér mozdításakkor hívódik meg, míg az `onUpdate` fix időközönként rendszeresen.

ViewTesselator. A `ViewTesselator` feladata az adott nézethez, melyhez az elrendezés már elkészült, kiszámítani a megjelenítendő geometriát. Ez egy dinamikusan használandó osztály, azaz példányai rövid életűek.

A konstruktor átveszi a megjelenítendő nézet leíróját, melyre konstans referenciát tárol. Ezután az osztály adattagjait kell feltöltenünk:

- `mat4 transf` Az alkalmazandó transzformáció
- `vec2 wsize` Az ablak belső mérete, az ezen kívül eső geometria levágható
- `float zoom` A transzformáció nagyítása
- `float spacing` Arányszám az élek és csúcsok közötti szabad hely mértékére
- `float arrowSize` Arányszám az élek végződéseinek méretére
- `size_t active` Az aktív csúcs azonosítója

Az adattagok kitöltése után a paraméter nélküli `tess` függvény hívásával elkészül a geometria. Az elkészült geometriát két `Mesh` változóból olvashatja ki a felhasználója: `line_mesh` és `tris_mesh` adattagokból. Ezen objektumok közvetlenül feltölthetők a GPU-ra az Flib segítségével.

A tesszellációt követően megsemmisíthető az osztály példánya.

3.3.7. Felhasználói interakció kezelése

A felhasználói interakciót az Flib által nyújtott lehetőségekkel kezeljük le.

A nézet transzformációja a `TransformListener`-ből leszármazó `ViewPlotter` által kerül kezelésre, a felüldefiniált transzformációs függvények segítségével. A kattintásokat is ez az osztály kezeli, hasonló módon.

A billentyűkombinációk kezelésére bármelyik GUI elem feliratkozhat, így működési módtól függően eltérő lehet az elérhető kombinációk halmaza.

A közös kombinációkat az `App` osztályban kezeljük le, mint az escape gombbal való kilépés. A futási módokra specifikus billentyűkombinációk a megfelelő `App` megvalósításban szerepelnek.

A megjelenítés során a `ViewPlotter`, a `LabelHandler` és a `ViewTessellator`, illetve a `SelectorButtons` osztályokra hagyatkozik a tényleges megjelenítéshez, ő csak a munkát hangolja össze.

3.4. Integráció a RefactorErllel

A RefactorErl oldalon a Gview-val való használathoz elkészítettem a megfelelő Erlang modulokat.

A megjelenítendő gráf szerkezetét és megjelenítési tulajdonságait leíró adattípust a 3.4.1. szakaszban részletezem.

Ezt az adatszerkezetet felhasználva a `gvcomm` modul, alacsonyszintű üzenetek továbbítására építve, magasabb szintű szolgáltatásokat valósít, mint például gráf küldése.

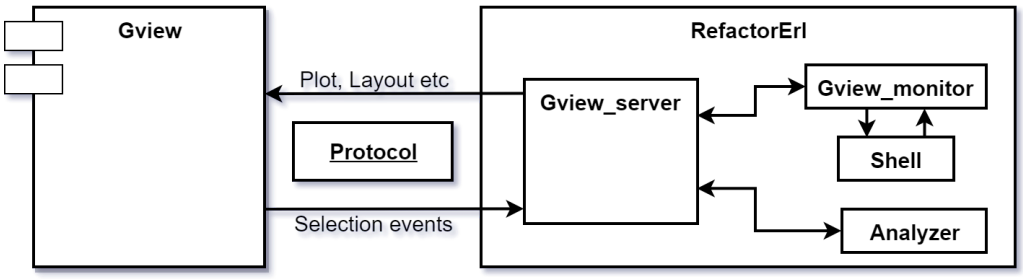
A `gvcomm` által nyújtott kommunikációs felületre épít a `gview_server`, mely a megjelenítővel való kapcsolatot tartja fent.

A `gview_monitor` feladata a `gview_server` felügyelete, annak állapotáról való tájékoztatás.

A `gview_collector` a gráf nézetek összeállítását végzi.

Ezen modulokat a `gview` modul fogja össze, mely biztosítja a felhasználói interfészt az Erlangból történő használatra.

Az említett modulok részletes leírása a 3.4.2., 3.4.3., 3.4.4., 3.4.5. és a 3.4.6. alfejezetekben található. Az Erlang oldali működést a 3.11. ábra foglalja össze.



3.11. Ábra. A külső megjelenítő kapcsolata a RefactorErlen belüli kezelővel.

3.4.1. Gráfleíró típus

Megjelenítés előtt az Erlang oldali kód előállítja a megjelenítésre szánt gráf leírását. Ezen leírás nem csak a gráf szerkezetét, de annak kinézetét is tartalmazza. Ez a gráfleíró egy Erlang `map` típusú érték.

A `map`-ek kulcs-érték párok tárolására és azok hatékony elérésére használt adatszerkezetek.

A gráfleíró `map` elemei opcionálisak, a lehetséges kulcsokat és jelentésüket a 3.1. táblázat tartalmazza.

kulcs	típus	jelentés	alapérték
nodes	[any]	Csúcsazonosítók listája	[]
node_palette	[npale]	Csúcstípusok	[]
edge_palette	[epale]	Éltípusok	[]
labels	[string]	Csúcscímkék listája	[]
node_wts	[int]	Csúcssúlyok listája	[]
node_tys	[int]	Csúcsok típusindexekei	[]
edges	[{any,any}]	Élek listája	[]
edge_wts	[int]	Élek súlya	[]
edge_tys	[int]	Élek típusindexei	[]
selectors	[[id]]	Csúcsonként a választógombok	[]
tooltips	[string]	Eszkezleírók szövege	[]

3.1. Táblázat. A gráfleíró `map` kulcsai és értelmezése

Az `epale` egy él kinézetét leíró típus, a `{float,atom,atom,[int,int,int]}`-el egyezik meg. A rendezett négyes első eleme az él vastagsága, második és harmadik eleme az él kezdetének és végének az alakja (lehetséges értékek: `circle`, `square`, `triangle`, `none`), az utolsó elem pedig és az él színének *RGB* kódja.

Az `npale` egy csúcs kinézetét leíró típus, `{int,atom}`-el egyezik meg. A rendezett pár első eleme a csúcs méretét, a második pedig a formáját adja meg. A csúcs formája `circle,square,triangle,none` értékeket veheti fel.

A választógombok `id` típusa kétféle lehet, `{string,any}` vagy `string`. A `string` mindkét esetben az eszközleíróban megjelenő szöveget tárolja.

Az élek párok listájaként kerülnek reprezentálásra, ezért a párok elemei csak létező csúcsok azonosítói lehetnek.

A 3.1. lista egy példa egy nyolc elemű kör lehetséges leírására. A példagráfban a csúcsok címkéi felváltva `1`-ek és `0`-ák. Az élek vékony, szürke nyilak, az `a` csúcs a többitől nagyobb háromszög, a többi csúcs kör alakú. Az `a` csúcsnak van csak eszközleírója, ami a `"Hello"` szöveg, megjelenítéskor idézőjelek nélkül.

3.1. Felsorolás. Példa gráfleíróra Erlang nyelven, ami egy 8 hosszú kört ír le.

```
1  #{
2      nodes => [a,b,c,d,e,f,g,h],
3      labels => [1,0,1,0,1,0,1,0],
4      edge_palette => [{3,none,triangle,[80,80,80]}],
5      node_palette => [{7,circle},{10,triangle}],
6      tooltips => ["Hello"],
7      node_tys => [1,0,0, 0,0,0, 0,0,0],
8      edges => [{a,b},{b,c},{c,d},
9                {d,e},{e,f},{f,g},
10               {g,h},{h,a}]
11 }
```

A gráfok küldését az alább részletezett `gvcomm` modul végzi. Gráfok fogadásáért a C++ oldali kódban az `ErlImporter` osztály felelős, mely a 3.3.4. fejezetben került kifejtésre.

3.4.2. gview_server

A modul feladata a c++ oldali kód kezelése, innen történik az indítása a *gview.exe*-nek, illetve a vele való kapcsolattartás.

A `start/2` függvény két paramétere a Gview elérési útja és a futtatható állomány neve. Hívására elindítja a Gview-t és inicializálja a kapcsolatot, ezután a visszaadott folyamatazonosítón keresztül várja az üzeneteket.

Ezt a modult a `gview` modul használja, nem önálló használatra készült.

3.4.3. gview_monitor

Feladata a `gview_server` felügyelete, amennyiben az meghibásodik, ez továbbra is elérhető, és lekérdezhető az állapota.

Az előző modulhoz hasonlóan a `start/1` függvénnyel indíthatjuk el a monitort, ennek egyetlen paramétere a már elindított `gview_server` folyamat azonosítója.

Ezt a modult is a `gview` modul használja, nem önálló használatra készült.

3.4.4. gview_collector

Feladata az egyes nézetek előállítása, ez a modul biztosítja a RefactoErllel való kommunikációt.

Exportált függvényei mind nézet leírót adnak eredményül (azaz `::  map` típusúak) Ezen függvények:

- `funcs_of_modules_view/1` Modulok és a bennük definiált függvények nézetét állítja elő. Paramétere a modulok listája.
- `funcalls_view/1` Függvényhívási nézetet állít elő. Paramétere a függvény és

a hívási mélység, egy tuple-ben.

- `funcalls_view/2` Függvényhívási nézetet állít elő. Paraméterei a függvény és a hívási mélység.
- `fun_syn_view/2` A szintaktikus fa egy részét állítja elő. Paramétere a függvény, ami a nézet gyökere és az előállítandó mélység.
- `clause_or_expr_view/3` Klózhoz vagy Kifejezéshez tartozó szintaxisfa nézet előállítása. Amennyiben az adott csúcs egy változó, úgy adatfolyam nézetet állít elő. Paraméterei a csúcs, az előállítandó szintaxisfa mélysége és az adatfolyam iránya (`true` - visszafele,`false` - előre).

3.4.5. gvcomm

Feladata az alacsony szintű Erlang Port funkcionalitásra magasabb rendű funkcionalitások kiépítése, mint gráf küldése.

A *send*-del kezdődő függvények első paramétere minden esetben a port azonosítója, amin keresztül a Gview-val kapcsolódik a program, visszatérési értékük pedig az `ok` atom. A modul által biztosított függvények:

- `send_int_list/2` Elküldi a paraméterben kapott egész számok listáját. A számok a tetszőleges négybájtos egészek lehetnek.
- `send_float_list/2` Elküldi a paraméterben kapott lebegőpontos számok listáját.
- `send_text/2` Elküldi a paraméterben kapott szöveget. Amennyiben nem listát kap, azt szöveggé alakítja, és úgy küldi el.
- `send_layout/2` Elküldi a paraméterben kapott elrendezési algoritmus azonosítóját.
- `send_graph/2` Elküldi a paraméterben kapott gráfleírót.

- `recv_id/1` A paraméterül kapott porton fogad egy négybájtos egész számot, ez a szám a visszatérési érték.
- `decode_floats/1` Bájtok listáját alakítja vissza lebegőpontos számok listájává, amit eredményként visszaad. A paraméterlista hossza négy többszöröse kell, hogy legyen.
- `uint4/1` Négy bájtból dekódol egy egész számot, amit eredményként ad. Paramétere négyelemű lista.

3.4.6. gview

Ez a modul biztosítja a gráfmegjelenítő interfészét. Ezen keresztül használja a RefactorErlt használó fejlesztő.

A modul publikus függvényei:

- `start/0` Elindít egy új gráfmegjelenítő példányt és a kapcsolat azonosítóját adja eredményül.
- `start/1` Hasonló a `start/0`-hoz, paramétere a megjelenítő program neve.
- `start/2` Hasonló a `start/0`-hoz, paraméterei rendre a megjelenítő elérési útja és a program neve.
- `status/1` Egy megjelenítővel való kapcsolat állapotát kérdezi le. Paramétere a kapcsolat azonosítója, melyet a `start` hívások valamelyike adott vissza. Visszatérési érteke lehet az `ok` atom, ekkor a megjelenítő rendeltetésszerűen fut, `{exit,normal}` pár, ekkor a megjelenítő szabályos módon leállt (felhasználó kérésre), illetve `{error,E}` ahol `E` tetszőleges hibaleírója, ekkor az `E` hiba következtében leállt a megjelenítő, végül `timeout`, ami azt jelöli, hogy időn belül nem érkezett válasz a megjelenítőtől (esetleg rossz azonosítót adtunk meg).
- `status/2` Hasonló a `status/1`-hez, második paramétere a válaszra várás maximális idője.

- `stop/1` Leállít egy futó megjelenítőt, paramétere a megjelenítővel való kapcsolat azonosítója. Visszatérési értéke a `status/1` hívás eredménye.

- `undo/1` Egy futó megjelenítőn betölti az előző nézetet, paramétere a megjelenítővel való kapcsolat azonosítója. Visszatérési értéke a `status/1` hívás eredménye.

- `redo/1` Egy futó megjelenítőn betölti a legutóbb visszavont nézetet, paramétere a megjelenítővel való kapcsolat azonosítója. Visszatérési értéke a `status/1` hívás eredménye.

- `load/2` Betölt egy futó megjelenítőn egy nézetet. Első paramétere a megjelenítővel való kapcsolat azonosítója, második lehet `{funcall,F}`, ahol `F` egy függvény csúcs, ekkor függvényhívási nézet kerül betöltésre, `modules`, ekkor az összes modult tartalmazó modulnézet töltődik be, illetve `{modules,Mods}`, ahol `Mods` modulok listája, ekkor ezen modulok modulnézete töltődik be. Visszatérési értéke a `status/1` hívás eredménye.

- `plot/2` Betölt egy tetszőleges nézetet egy futó megjelenítőbe, paramétere a megjelenítővel való kapcsolat azonosítója és a megjelenítendő gráfleíró. Visszatérési értéke a `status/1` hívás eredménye.

- `center/1` Középre igazítja egy futó megjelenítőben a megjelenített gráfot. Paramétere a megjelenítővel való kapcsolat azonosítója. Visszatérési értéke a `status/1` hívás eredménye.

- `layout/2` Megváltoztatja egy futó megjelenítő által használt elrendezési algoritmust. Paramétere a megjelenítővel való kapcsolat azonosítója és az elrendező algoritmus azonosítója, ami lehet `fd` vagy `layered`, melyek rendre az erő alapú és a hierarchikus elrendezést jelentik. Visszatérési értéke a `status/1` hívás eredménye.

- `drop_cache/1` Egy futó megjelenítőben az elmentett elrendezések kidobását eredményezi. Paramétere a megjelenítővel való kapcsolat azonosítója. Visszatérési értéke a `status/1` hívás eredménye.

- `labels_mode/2` Megváltoztatja egy futó megjelenítőben a címkék elhelyezési stratégiáját. Paramétere a megjelenítővel való kapcsolat azonosítója és a stratégia neve, ami lehet `hide`, `normal` vagy `move`, melyek rendre a címkék elrejtését, egyszerű elrendezését és átlapoltság mentes elhelyezését jelentik. Visszatérési értéke a `status/1` hívás eredménye.

3.5. Tesztelés

A Gview gráfmegjelenítő alkalmazás fejlesztése során folyamatos tesztelésen esett át, az így feltárt hibák javítása megtörtént. A fejlesztés tehát a teszt alapú fejlesztés elveit követve ment végbe. Az implementációt követően különböző működés szerinti csoportosítással végzett tesztek is készültek. Az alábbiakban ezen tesztek egy része szerepel.

3.5.1. Egységtesztek

A RefactoErl integráció tesztelése az EUnit [13] könyvtár segítségével történt, melyben tetszőleges tesztek készíthetők Erlang modulokhoz. Azon modulok, melyekhez EUnit teszt áll rendelkezésre `modulname:test()` hívással tesztelhetők fordítás után, ahol `modulname` a tesztelendő modul neve.

A dolgozathoz mellékelt CD lemezen megtalálható a közel 100 teszt, amik a fejlesztés során készültek.

A modulok tesztelése során azok gyakran küldenek üzenetet a megjelenítő számára, amit az egységteszteken nem szeretnénk ténylegesen futtatni. Az Erlang üzenet küldése azonban ugyanúgy működik Porton keresztül, mint folyamat számára való küldéskor. Ezt kihasználva a teszt hívások általában saját folyamatuk azonosítóját adják át ezeknek a hívásoknak, majd a választ megfelelően kibontva ellenőrzik annak tartalmát. A válasz elmaradása esetén sikertelen a teszt.

3.2. Felsorolás. Az Erlang oldali kód teszteléséhez használt port utánzó függvények

```
1  receive_as_port(_R) ->
2          S = self(),
3          receive
4                  {S, {command, C}} -> C
5          after 0 ->
6                  error
7          end.
8
9  send_as_port(Bytes) ->
10         self() ! {self(), {data, Bytes}},
11         self().
```

Erre a célra készítettem két függvényt, amik portok viselkedését képesek utánozni, forrásuk a 3.2. listán látható.

Az első függvény egy portnak küldött üzenetből veszi ki a tényleges üzenetet. A második egy porttól érkező üzenetet csomagol ki.

gvcomm A `gvcomm` modul feladata különböző kommunikációs függvények biztosítása. A tesztelése ezen függvények speciális esetekben való helyes működését vizsgálja.

A vizsgált függvények és eseteik:

- `receive_as_port`
  - Üres lista küldése
  - Kis számok (0-255) küldése
  - Nagy számok (>65536) küldése
- `send_text`
  - Üres szöveg küldése
  - Egy karakter küldése
  - Rövid szöveg küldése
  - Különleges (újsor) karaktert tartlamazó szöveg küldése

- `send_layout` Mindkét elrendezés értékének küldését teszteltem.
- `recv_id`
  - Nulla fogadása
  - Kis szám (<255) fogadása
  - Nagy szám (>256) fogadása
- `send_graph` Átfogóbb tesztelés a használati esetek tesztjeiben történik meg.
  - Üres gráf (#{}) küldése
  - Csak a csúcsok megadása
  - Csúcsok és élek megadása
  - Csúcsok, élek és címkék megadása
- `decode_floats` Mivel a lebegőpontos aritmetika pontatlan lehet, itt nem tökételes egyezést, hanem minimális eltérést megengedő összehasonlítást végeztem.
  - Üres lista dekódolása
  - Egész számok dekódolása
  - Nagy számok dekódolása
  - Tört számok (pl 12.1212) dekódolása
- `uint4`
  - Nulla dekódolása
  - Kis számok (0-255) dekódolása
  - Nagy számok (>65536) dekódolása

gview_server A `gview_server` feladata a megjelenítővel való kapcsolattartás, ehhez egy folyamatosan üzenetekre várakozó folyamatot tart fent.

Mivel a tesztelés során a megjelenítő tényleges futtatása nem történik meg, a modul belső üzenetfeldolgozó rekurzív függvényét, a `loop/2` függvényt közvetlenül indítjuk

el egy új folyamatként. Az átadott paraméter nem a megjelenítő portja, hanem a tesztelő saját folyamatának azonosítója. Így a teszt során a feldolgozó úgy tekint a tesztelőre, mint a megjelenítőre és neki küldi vissza válaszait, amiket ellenőrizni tudunk.

A tesztelt üzenetek:

- `{send, M}` Az `M` szöveg direkt küldése a megjelenítőnek. A leggyakoribb üzenet.
  - Üres üzenet küldése
  - Hosszabb üzenet küldése
- `{set_layout, L}` A megjelenítő által használt elrendező algoritmus váltása
  - `L = fd`
  - `L = layered`
- `{plot, G}` A `G` gráf megjelenítése. A tesztek során csak nagyon egyszerű gráfokkal teszteltem, mivel a használati eset tesztjei intenzíven tesztelik.
- `{undo}, {redo} és {reload}` Rendre az elrendezésváltás visszavonása, ismétlése és a jelenlegi nézet újraküldése.
  - Egy ilyen üzenet küldése
  - Több egymás utáni azonos
  - Különböző üzenetek egymás után
- `"undo", "redo" és "reload"` Az előző üzenetekhez hasonló, azonban a megjelenítő irányából érkeznek.
  - Egy ilyen üzenet küldése
  - Több egymás utáni azonos
  - Különböző üzenetek egymás után
- `stop` Leállítja a futó szervert.

gview_monitor A `gview_monitor` esetében hasonlóan kell eljárnunk, mint a `gview_server` esetében, hiszen ez a modul a `gview_server` figyelésére lett létrehozva.

A különbség, hogy itt az üzenetek nem a megjelenítőnek vannak címezve, ezért nem kell ki- és becsomagolnunk őket.

A teszteléshez két különböző kimenetelű eseménysorozatot vizsgáltam meg, az elsőben egy szabályos indítás, használat és leállítás jelenik meg:

1. A `gview_monitor` elindítása.
2. Kiemelt szereppel nem rendelkező üzenetek küldése a monitornak.
3. Ezen üzenetek bármilyen típust is tartalmazhatnak.
4. A monitor állapota ezek után `ok`.
5. Leállítjuk a monitort, szabályosan.
6. A monitor az állapot lekérdezésre ezután nem válaszol.

A második tesztben a szimulált megjelenítő meghibásodik és leáll. A megjelenítő rendeltetésszerű leállása hasonló lefolyást eredményez, annyi különbséggel, hogy a végső státuszkérés eredményében a kilépés oka `normal`.

1. A `gview_monitor` elindítása.
2. Kiemelt szereppel nem rendelkező üzenetek küldése a monitornak.
3. A monitor állapota ezek után `ok`.
4. Szimuláljuk a megjelenítő összeomlását. A monitor erre nem válaszol, hiszen ez egy rendszerüzenet.
5. Az állapotlekérdezés ezután visszaadja a hibát, akár egymás után többször is.
6. Leállítjuk a minotort, szabályosan.
7. A monitor az állapot lekérdezésre ezután nem válaszol.

gview_collector A `gview_collector` feladata gráf nézetek előállítása kérésre. A használati tesztek során intenzív tesztelésen esik át.

Mivel a csúcsok azonosítójának szerkezete implementációs részlet, a lekérdezések pontos eredményére nem tesztelhetünk. Ehelyett, ahol lehet lekérdezzük a megfelelő csúcsok azonosítóját és azt vizsgáljuk, hogy a kiszámolt nézetbe a megfelelő csúcsok kerültek. Továbbá függvényhívási és szintaktikus nézetek esetén a mélység növelésével nem tűnhetnek el bekerült csúcsok, csupán új csúcsok adódhatnak hozzá, élekkel ugyanígy.

Azt, hogy egy lista tartalmaz egy másikat úgy döntöm el, hogy a két listát rendezem, majd párhuzamosan lépkedek rajtuk, a tartalmazandón csak akkor lépve, mikor a másikban annak fejeléméhez érkezek, egészen addig, amíg az egyik el nem fogy. Az ehhez használt függvény kódja a 3.3. listán látható.

3.3. Felsorolás. Rendezett listák tartalmazásának vizsgálata

```
1 contains_sorted(_,[]) -> true;
2 contains_sorted([],_) -> false;
3 contains_sorted([A|As],[B|Bs]) when A =:= B ->
4         contains_sorted(As,Bs);
5 contains_sorted([_A|As],Bs) ->
6         contains_sorted(As,Bs).
```

Az alábbi tesztesetek feltételezik, hogy szigorúan csak a fentebb bemutatott `user.erl` és `calculator.erl` van csak betöltve a RefactorErlbe.

- `funcs_of_modules_view([])` eredménye `[]`.
- Az alábbi hívások eredményében található csúcsokra való teszt:
  `funcs_of_modules_view([calculator])`,
  `funcs_of_modules_view([user])`,
  `funcalls_view({calculator,add,2,2})`,
  `funcalls_view({calculator,fib,1,2})`,
  `funcalls_view({user,calc,3,2})`.
- További tesztek arra vonatkozólag, hogy a `fun_syn_view/2` illetve társa, a `funcalls_view/1` által generált nézetek a mélység növelésével csak bővülnek,

mind csúcsok, mind élek tekintetében.

gview Az utolsó modul a tesztelés sorában. Feladata a felhasználói interfész biztosítása, amin keresztül a gráfmegjelenítő utasítható, többek között nézetváltásra, leállásra vagy más elrendező algoritmus használatára.

Mivel a modul `start` függvénye elindít egy új monitort és az interfész függvények ezzel a folyamattal kommunikálnak, a tesztelés során a függvényhívások hatására elküldött üzeneteket vizsgáltam meg. Ezen üzenetek egymástól függetlenek, a legtöbb hívás akár többet is küldhet egymás után. Számos hívás után történik egy `status/1` kérés is, ekkor előre elküldök egy üzenetet a tesztetlő folyamatnak, amit a `status/1` le tud kezelni, mint választ a monitortól.

3.5.2. Használati tesztek

Ebben a szakaszban olyan teszteket írok le, melyeket nagyon nehéz lenne automatikusan elvégezni, ezért kézzel győződhetünk meg helyes eredményükről.

Ezek a tesztek egymást követő lépéseket írnak le, melyek a program összességét, a komponensek együttes működését teszik próbára.

Első lépésben egy futó RefactorErl Erlang shellbe töltsük be az *Mnesia* adatbáziskezelő forrását. Ennek lépéseiről a RefactorErl webes bevezetőjében olvashatunk.

1. Az alábbi értékadással indítsunk el egy új megjelenítőt:

   ```
   X = gview:start().
   ```

   Megjelent a fehér ablak, tehát a hívás sikeres.

2. Nyissuk meg a `mnesia_log` modulhoz tartozó modulnézetet:

   ```
   gview:load(X,{modules,[mnesia_log]}).
   ```

   Az ablakban megjelent a piros tengersünre hasonlító alakzat, tehát a hívás sikeres.

3. Az ablakban az egérgörgővel közelítsünk rá a gráfra és mutassunk azon csúcsra amely az `open_log/4` függvényt reprezentálja!

   A program kiemeli a csúcsot bordó színnel, tehát érzékeli a rámutatást.

4. Kattintsunk a kijelölt függvényre a bal egérgombbal!

   Megnyílik az `open_log/4` kettő mélységű hívási gráfja, a művelet sikeres.

5. Közelítsünk rá a négyzet alakú `open_log/4` függvényre és mutassunk rá az egérmutatóval!

   Megjelenik két választógomb a csúcs felett, kattintsunk a `syn` feliratúra.

   Megjelent az `open_log/4` szintaxisfája, a művelet sikeres.

6. A konzolba írjuk be az alábbi parancsot, amivel az elrendezési algoritmust átváltjuk hierarchikus elrendezésre:

   ```
   gview:layout(X,layered).
   ```

   A grafikus ablakban az elrendezés átvált hierarchikusra, így a teszt sikeres.

7. A megjelenítő ablakában nagyítsunk rá ismét az `open_log/4` függvényre, majd kattintsunk a az egyetlen megjelent választógombra. A gomb felirata `expand`.

   Az `open_log/4` szintaktikus fájának egy szinttel mélyebb nézete jelent meg, tehát a művelet sikeres.

8. Ismételjük meg az előző pontot.

9. Kattintsunk a negyedik rétegen az `Fname` csúcsra!

   Az `Fname` adatfolyam diagramja jön elő, a művelet sikeres.

10. Kattintsunk az egyetlen értékre, amit az `Fname` felvehet!

    A kifejezés szintaxisfája jelenik meg, tehát a művelet sikeres.

11. Az ablakot fókuszban hagyva, nyomjuk meg a CTRL+R billentyűkombinációt!

    Az elrendezés elforgatva jelenik meg, tehát a művelet sikeres.

12. Nyomjuk meg ismét az előző kombinációt, majd a CTRL+Z kombinációt hatszor, így visszajutva az `open_log/4` hívási gráfjához.

A visszalépések során a helyes gráfok, a helyes elrendezéssel jelennek meg, tehát a visszavonás sikeres.

13. A konzolon írjuk be az alábbi függvényhívást:

    ```
    gview:status(X).
    ```

    Az eredmény `ok`, ami azt jelzi, hogy a megjelenítő épségben van és fut, ami megfelel a valóságnak.

14. A konzolon indítsunk egy új megjelenítőt, az előző bezárása nélkül, majd töltsük be rajta az `mnesia_text` modult:

    ```
    P = gview:start(), gview:load(P,{modules,[mnesia_text]})
    ```

15. Kattintsunk egy tetszőleges függvényre a második ablakban!

    Az első ablak tartalma nem változott meg, a másodikban pedig megjelent a függvény hívási gráfja, a teszt sikeres.

16. Az első ablakban nyomjuk meg a CTRL+Y billentyűkombinációt háromszor!

    A második ablak változatlan, az elsőn ismét az `open_log/4` szintaktikus fáját láthatjuk, a művelet sikeres.

17. A második ablakban nyomjuk meg az Esc billentyűt, majd a konzolba írjuk be az alábbi két hívást:

    ```
    gview:status(P). gview:status(X).
    ```

    Az első eredménye `{exit,normal}` a másodiké `ok`, melyek rendre a megjelenítő szabályszerű leállását és a megjelenítővel való kapcsolat épségét jelentik. A teszt sikeres.

18. A megmaradt megjelenítőt zárjuk be a konzolról az alábbi függvénnyel, majd ellenőrizzük a `status` hívással, hogy kilépett-e:

    ```
    gview:stop(X). gview:status(X).
    ```

    A megjelenítő ablak eltűnt, a `gview:status` hívás eredménye `timeout`, melynek jelentése, hogy a monitor is leállt. A próba sikeres.

19. Indítsunk három megjelenítőt az alábbi kifejezéssel:

    ```
    L = [gview:start() || _<-lists:seq(1,3)].
    ```

    Mind a három megnyílt, tehát a hívás sikeres.

20. Kérjük mindhárom megjelnítőben a modulnézet megjelenítését:

    ```
    [gview:load(I,modules) || I <- L].
    ```

    Mindhárom megjelenítő elkezdte az elrendezés elkészítését, tehát a próba sikeres.

21. Zárjunk be kettő megjelenítőt az ablak bezárógombjával, majd várjunk amíg az utolsón kialakul a végleges elrendezés, vagy legalább közel kerül hozzá. Ezután zárjuk be a harmadik megjelenítőt is és indítsunk egy újat:

    ```
    Y = gview:start(), gview:load(X,modules).
    ```

    Az újonnan indított megjelenítő nem a kezdő állapotból indítja az elrendezés generálását, hanem az előző megjelenítő által hagyott elrendezést finomítja tovább, tehát az adatok perzisztálása működik.

4. fejezet

Befejezés

A kódmegértési funkciók támogatására egyre növekszik az igény, ezért erre a fontos problémára az adatvizualizáció eszközével szeretnék megoldást kínálni. A dolgozatomban bevezetett alkalmazás, Gview, egy interaktív gráfmegjelenítő alkalmazás, mely képes az elemzett szoftver struktúrájának részeit dinamikus megjeleníteni, így azt felfedezhetővé teszi a felhasználója számára. Az alkalmazás képes tetszőleges forrásból származó gráfok kirajzolására, több elrendezési algoritmussal is. Jelenleg az erő alapú elrendezés és a hierarchikus elrendezés támogatott.

A megjelenített nézetek között könnyű és gyors felhasználói interakció által válthatunk. A gráf csúcsai körül választógombok helyezhetők el, mely által egy csúcshoz kapcsolódó több nézet is könnyedén elérhető. A csúcsokhoz eszközleírókat is társíthatunk, amik az egérmutató egy helyben tartásakor jelenik meg.

A dolgozatomban bemutattam az alkalmazás RefactoErllel való integrációját, a RefactorErl beli Gview interfész részletes leírásával.

Végül az elkészült szoftver tesztelését is elvégzem.

A szakdolgozatomban bemutatott szoftver részét képezi a 2018 májusában és 2019 májusában bemutatott TDK dolgozataimnak is [14, 15]. A TDK dolgozatok eredményét tudományos folyóiratban is publikáltuk [16], illetve további egy folyóiratcikk

és egy konferenciacikk jelenleg elbírálás alatt van.

4.1. Jövőbeli munka

A jövőbeli munka a jelenlegi eredményekre alapozva, azok továbbfejlesztése. Ez részben a jelenlegi RefactorErl integráció újabb funkciókkal való bővítését jelenti. Másrészt viszont a Gview több hosztalkalmazással, akár különböző kommunikációs csatornán keresztül is, való összekapcsolását is kitűztem célul. Valamint, szeretnék további elrendező algoritmusokat megvalósítani, mint például a Stress-Majoring gráf elrendező algoritmus.

Irodalomjegyzék

[1] Zoltán Porkoláb, Tibor Brunner, Dániel Krupp, and Márton Csordás. Codecompass: An open software comprehension framework for industrial usage. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 361–369, New York, NY, USA, 2018. ACM.

[2] István Bozó, Dániel Horpácsi, Zoltán Horváth, Róbert Kitlei, Judit Kőszegi, Máte Tejfel, and Melinda Tóth. RefactorErl - Source Code Analysis and Refactoring in Erlang. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools, ISBN 978-9949-23-178-2*, pages 138–148, Tallin, Estonia, October 2011.

[3] James Gettys, Robert W Scheifler, Hideki Hiura, Inc Bill McMahon, Ron Newman, Shigeru Yamada, and OSSI Fujitsu. Xlib- c language x interface. Technical report, Protocol Version, 1985.

[4] A GCC fordító weboldala. `http://gcc.gnu.org`. Megnyitva: 2019. 05. 22.

[5] A RefactorErl elérhetősége. http://plc.inf.elte.hu/erlang/refactorerl-releases.html. Megnyitva: 2019. 05. 28.

[6] Az Erlang telepítési oldala, Windows specifikus szekció. http://erlang.org/doc/installation_guide/install-binary.html#windows. Megnyitva: 2019. 05. 28.

[7] A RefactorErl telepítéséhez szolgáló részletes utasítások. http://pnyf.inf.elte.hu/trac/refactorerl/wiki/Install. Megnyitva: 2019. 05. 28.

[8] Az Flib GitHub oldala. `https://github.com/Frontier789/Flib`. Megnyitva: 2019. 05. 22.

[9] A Gview hivatalos GitHub oldala. https://github.com/Frontier789/Gview. Megnyitva: 2019. 05. 28.

[10] Håkan Mattsson, Hans Nilsson, and Claes Wikström. Mnesia - a distributed robust dbms for telecommunications applications. *practical aspects of declarative languages*, pages 152–163, 1999.

[11] C. Harper. *Introduction to mathematical physics*. Prentice-Hall physics series. Prentice-Hall, 1976.

[12] J.R. Dormand and P.J. Prince. A family of embedded Runge-Kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1):19–26, 1980.

[13] Richard Carlsson and Mickaël Rémond. Eunit: a lightweight unit testing framework for erlang. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, pages 1–1. ACM, 2006.

[14] Mátyás Komáromi. Visualising software dependencies to support code comprehension. TDK dolgozat, Kari Tudományos Diákköri Konferencia, Informatikai Szekció, ELTE, Budapest, Hungary, 1. díj, 2019.

[15] Mátyás Komáromi. Gview: Efficient graph visualisation for RefactorEr. TDK dolgozat, Kari Tudományos Diákköri Konferencia, Informatikai Szekció, ELTE, Budapest, Hungary, 1. díj, 2018.

[16] Mátyás Komáromi, István Bozó, and Melinda Tóth. An Efficient Graph Visualisation Framework for RefactorErl. *Studia Universitatis Babeș-Bolyai Informatica*, 63(2):21–36, 2018.