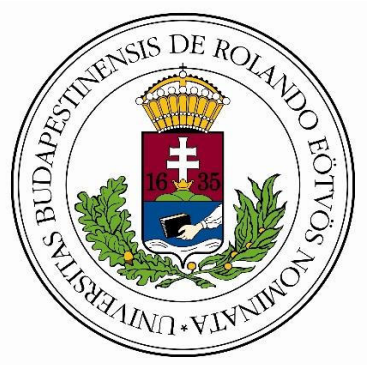


Eötvös Loránd Tudományegyetem
Informatikai Kar
Információs Rendszerek Tanszék

Vizualizációs alkalmazás fejlesztése a BCC trace moduljához

Témavezető
Dr. Laki Sándor

Szerző
Schubert Mátyás

Budapest, 2020

Tartalomjegyzék

1. Fejezet

Bevezetés

Egy program működésének megértése – komplexitástól függően – rengeteg időt vehet igénybe. A forráskód tanulmányozása mellett érdemes a program futás közbeni viselkedését is tanulmányozni, hogy megtudjuk, melyik függvények hívódnak meg, milyen hívási láncon és paraméterekkel. Ezt a módszert jelentősen megnehezíti, hogy egy program általában nem jelzi, hogy a forráskód mely részei hajtódtak végre a futás során.

Linux rendszereken több megoldás is született a programok ilyen módon való nyomkövetésére. Az egyik ilyen profilozó eszköz a nyílt forráskódú BCC [1] (BPF Compiler Collection), melynek trace moduljával programok függvényeit figyelhetjük. Az eszköz előnye, hogy a függvényhívásokról valós időben kapunk információkat. Hátránya viszont, hogy akárcsak a többi hasonló eszköznek, a BCC trace kimenetének értelmezése is nehézkes és kényelmetlen.

Feladatom célja, hogy létrehozzak egy webes felhasználói felületet a BCC trace moduljához. A felület megkönnyíti a monitorozni kívánt függvények kiválasztását, majd a futtatás során és után egy interaktív gráfot jelenít meg a hívási láncokról. A gráf segítségével megtudhatjuk, mely függvények milyen paraméterekkel és hányszor hívódtak meg.

2. Fejezet

Felhasználói dokumentáció

Az alkalmazást kizárólag Linux rendszereken lehet teljes mértékben használni, mivel a BCC által használt bővített BPF (Berkeley Packet Filters) egy Linux funkció, amely a 3.15-ös verziótól érhető el. Emiatt magát a BCC eszköztárat is csak Linux alapú operációs rendszereken lehet futtatni. Ha viszont nem akarjuk élő monitorozásra használni az alkalmazást, csupán már meglévő kimeneteit szeretnénk feldolgozni a trace modulnak, akkor az alkalmazás futtatható Windows operációs rendszereken is.

Ahhoz, hogy elindíthassuk a programot, több Python modult is installálnunk kell. Ha Linux operációs rendszerünkön még nem található a BCC, akkor azt is telepítenünk kell. Az indítás a parancssorból történik, ami után a standard kimeneten megkapjuk az URL-t, melyen elérhetjük a webes interfészt. Ezeket követően már minden funkció elérhető az interfészen keresztül.

2.1. Előkövetelmények

Az alkalmazást nem tudjuk out-of-the-box használni. Futtatás előtt meg kell győződnünk arról, hogy számítógépünkön minden szükséges programcsomag rendelkezésre áll. Az alábbi leírások segítenek az esetleg hiányzó eszközök telepítésében.

2.1.1. Python értelmező

A program Python programozási nyelven lett megírva, így futtatásához szükségünk van a Python értelmezőre, ami manapság minden modern operációs rendszerre elérhető. Az ajánlott verzió az alkalmazás futtatásához, a Python 3.7-es. Az értelmező telepítési módjairól a nyelv hivatalos letöltési oldalán [2] tájékozódhatunk.

2.1.2. Harmadik forrásból származó modulok

A forráskód több third-party Python modult is felhasznál, melyeket szintén telepítenünk. Ezek a csomagok mind elengedhetetlenek, függetlenül attól, hogy a programot milyen operációs rendszeren, vagy módon használjuk. A parancsok platformfüggetlenek, így bármilyen rendszeren kiadhatjuk őket.

```
pip3 install dash
pip3 install dash-cytoscape
pip3 install dash-bootstrap-components
pip3 install dash_daq
pip3 install pyyaml
pip3 install pexpect
```

2.1.3. BCC csomagok telepítése

A BCC telepítéséhez olyan Linux rendszer szükséges, melynek kernel verziója legalább 3.15-ös, az ajánlott kernel verzió pedig 4.1-es, vagy újabb. A további kernel konfigurációs elvásárokról a BCC hivatalos telepítési útmutatójában [3] olvashatunk, amelyben leírást kapunk arra is, hogy a különböző Linux disztribúciókra milyen parancsokkal történik a telepítés. Ubuntu rendszereken az alábbi parancsot adjuk ki a csomag telepítéséhez:

```
sudo apt-get install bpfcc-tools linux-headers-$(uname -r)
```

2.2. Futtatás

Ha sikeresen telepítettünk minden komponenst, amire szüksége van a szoftvernek, akkor elindíthatjuk az alkalmazást. A futtatás parancssorból történik, mely különböző rendszereken, különböző parancsokkal történhet. A program semmilyen parancssori argumentumot nem vár. Ha a parancssorban a program gyökérkönyvtárában vagyunk, akkor az indítás az alábbi parancsokkal történhet.

2.2.1. Indítás Linux disztribúciókon

Ha nem szeretnénk a profilozást az alkalmazáson keresztül végezni, akkor az alábbi paranccsal indíthatjuk futtathatjuk a programot.

```
./main.py
```

Az élő profilozáshoz az alkalmazás a háttérben futtatja a BCC trace modulját, amelynek a rendszer monitorozásához superuser jogosultságokra van szüksége. Így az alkalmazást is ilyen jogosultságokkal kell elindítanunk.

```
sudo ./main.py
```

2.2.2. Indítás Windows rendszereken

Windowson az alábbi paranccsal futtathatjuk a programot.

```
python3 main.py
```

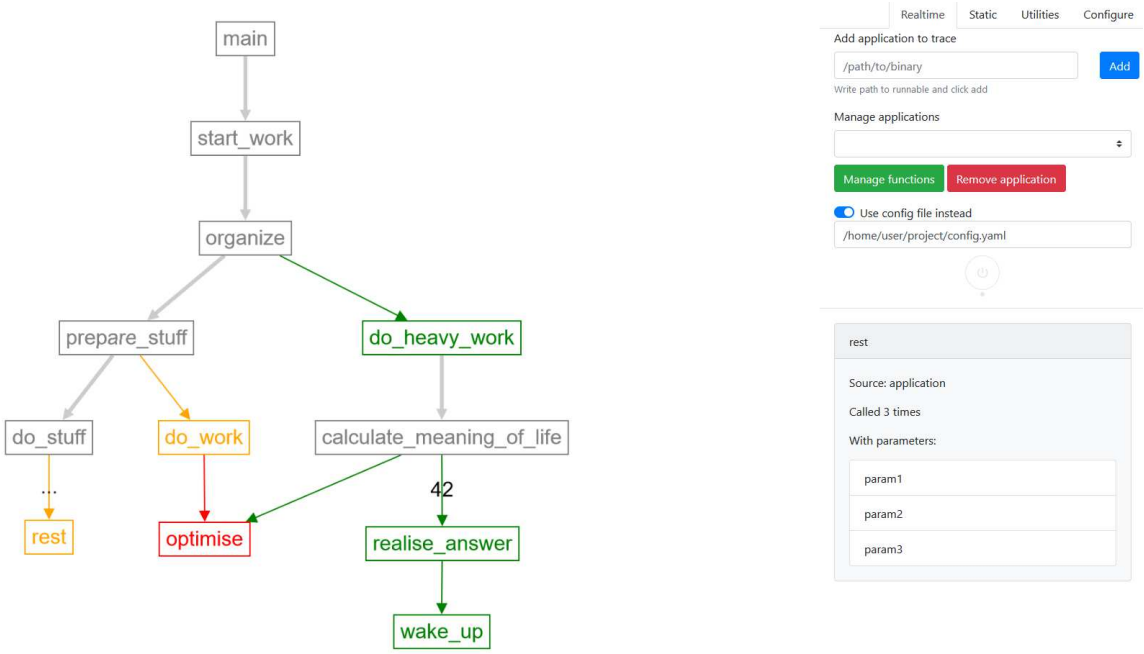
A program természetesen elindítható grafikus felületekről a `main.py` fájlra való kattintással is. Sikeres indítás után a parancssorban megkapjuk az interfész elérési útvonalát. Például:

```
Running on http://127.0.0.1:8050/
```

2.3. Felhasználói felület

Miután böngészőnkben megnyitottuk a megadott elérési útvonalat, elénk tárul az alkalmazás felhasználói felülete. Az alkalmazás teljes felülete ezen a webes interfészen érhető el. A felület a képernyő méretéhez alkalmazkodva jelenik meg. Ha futás közben változik az ablakméret, akkor követi a változást, így mindvégig biztosított a megfelelő megjelenítés. A felületen minden felirat angol nyelvű, illetve a gombok és feliratok színei is segítik a felhasználót a tájékozódásban.

A képernyőt logikailag két fő részre oszthatjuk. Jobb oldalon található a vezérlőpult, melynek segítségével elérhetjük az élő monitorozást, a statikus feldolgozást, a gráf böngészésére szolgáló funkciókat, illetve az alkalmazásra és a megjelenítésre vonatkozó különböző beállításokat. A bal oldalon található a monitorozott függvények hívási láncaiból kialakuló interaktív gráf. Közvetlenül az indítás után ez a rész még üres. A felület teljes sémáját az első ábra mutatja be.



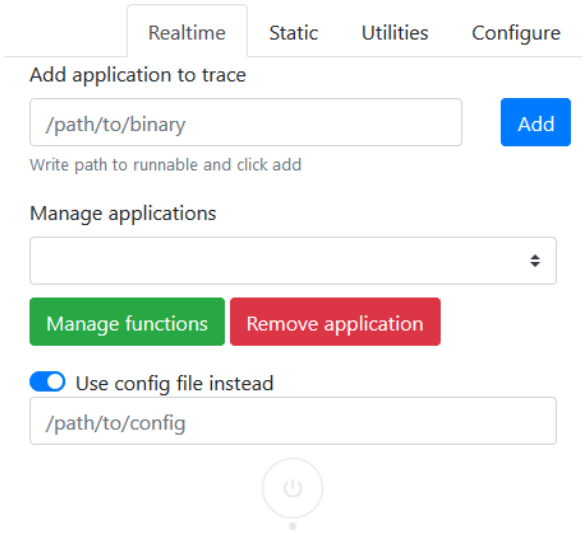
1. ábra: Az alkalmazás felhasználói felülete

2.4. Vezérlőpult

A vezérlőpult tetején található négy menüpont segítségével érhetjük el a valós idejű monitorozást (*Realtime*), egy meglévő kimenet feldolgozását (*Static*), a gráfban böngészést segítő funkciókat (*Utilities*) és az alkalmazásra vonatkozó konfigurációs beállításokat (*Configure*). A vezérlőpult alsó felén az interaktív gráfot böngészve kaphatunk információkat a kiválasztott függvényekről és függvényhívásokról.

2.4.1. Valós idejű monitorozás

A *Realtime* fülre kattintva megtaláljuk az alkalmazás legfőbb funkcióját, mellyel folyamatosan figyelhetjük az általunk megadott függvényeket. Ennek felületét láthatjuk a második ábrán. Az első beviteli mező szolgál arra, hogy megadjuk, mely alkalmazások/források függvényeit szeretnénk figyelni. Ha beírtuk a forrást, az *Add* feliratú gombra kattintva hozzáadhatjuk azt a megfigyelni kívánt források listájához.



2. ábra: Valós idejű monitorozás kezelőfelülete

A megadott forrás vagy egy bináris állomány elérési útvonala, vagy maga az állomány neve, ha az szerepel a PATH-ban. Mivel a BCC trace modul függvényekben gondolkodik, így ez nem elég ahhoz, hogy egy alkalmazás minden függvényét megfigyeljük. Minden egyes függvényt, amely ebben a forrásban található és monitorozni szeretnénk, kézzel kell hozzáadnunk a felületen.

Egy sikeres hozzáadás után forrásunk megjelenik a *Manage applications* felirat alatt található legördülő menüben. Ha itt kiválasztjuk, akkor két opciónk van a forrással kapcsolatban. A menü alatt található *Manage functions* feliratú zöld gombbal módosíthatjuk a monitorozni kívánt függvények listáját. A mellette található piros színű, *Remove application* szövegű gomb segítségével pedig eltávolíthatjuk az alkalmazást és annak minden függvényét a monitorozni kívánt elemek közül.

A menedzselni kívánt alkalmazás kiválasztása után a *Manage functions* gombot megnyomva az alkalmazáshoz tartozó kezelőfelület egy dialógusablakban nyílik meg. Az itt található felület nagyban hasonlít az előzőhöz. Az első beviteli mezőbe írhatjuk a monitorozni kívánt függvény nevét, majd az *Add* feliratú gomb segítségével elmenthetjük azt. Ez elég ahhoz, hogy a megadott függvényt megfigyeljük a monitorozás során.

Épp úgy, mint az alkalmazások esetén, itt is lehetőségünk van egy-egy függvény eltávolítására. Ha kiválasztottuk a *Manage functions* felirat alatti legördülő menüből, akkor a *Remove function* feliratú gomb segítségével kitörölhetjük a figyelni kívánt függvények közül.

A megfigyelni kívánt függvények esetén lehetőségünk van azt is monitorozni, hogy milyen paraméterekkel hívódnak meg. Ehhez a függvény kiválasztása után a *Manage parameters* gombra kell kattintanunk. Ilyenkor egy újabb, a kiválasztott függvényhez tartozó dialógus ablakon konfigurálhatjuk a kiírandó paramétereket. Ez a felület egy kicsit különbözik az előzőktől. A legördülő menü segítségével adhatjuk meg a paraméter típusát, a mellette jobbra elhelyezkedő numerikus értékeket befogadó beviteli mezőben pedig azt, hogy a függvény paraméterlistájában hányadikként helyezkedik el. Ezek megadása után az *Add* feliratú gombra kattintva hozzáadhatjuk a paramétert a függvényhez. Paraméter eltávolításához válasszuk ki azt a *Manage parameters* felirat alatt található legördülő menüből, majd kattintsunk a *Remove parameter* gombra. A paraméterek típusa alapvetően a reprezentációjukhoz kell. Minden megadható típus valamilyen C nyelvi format specifier-nek felel meg. Az egyes típusokhoz tartozó reprezentációkról a harmadik ábráról tájékozódhatunk.

Típus	**Reprezentáció**
char	`%c`
double/float	`%f`
int	`%d`
long	`%l`
long double	`%lF`
string/char	`%s`
short	`%hi`
unsigned short	`%hi`
void	`%p`

3. ábra: Típusok és reprezentációik

Több, nagy méretű program elemzése esetén rengeteg függvény, illetve paraméter megfigyelésére lehet szükség. Ebben az esetben a felhasználói felület használata ezek megadására hosszú időbe telhet. Ezen kívül, ha ugyanazon függvényeket többször is monitorozni szeretnénk az alkalmazás segítségével, kényelmes lenne, ha az alkalmazás újraindítása során a megfigyelni kívánt elemek listája nem veszne el. Mindkét nehézséget orvosolhatjuk, ha a monitorozni kívánt függvények betöltésére konfigurációs fájlokat használunk. Ahhoz, hogy ezt a funkciót elérjük, kapcsoljuk be a *Use config file instead* feliratú kapcsolót. Ekkor előtűnik egy újabb beviteli mező, melybe a konfigurációs fájl

elérési útvonalát kell megadnunk. A konfigurációs fájl egy yaml formátumú dokumentum, melyben minden monitorozni kívánt függvény és paraméter szerepel. A dokumentum első szintű kulcsai a források. Ugyanúgy, ahogy a felhasználói felületen kiválasztott források esetén, a megadott forrás itt is egy bináris állomány elérési útvonala, vagy maga az állomány neve, ha az szerepel a PATH-ban. Minden forráshoz egy lista tartozik, amelynek elemei maguk a függvények. Minden függvényhez további lista tartozhat, amelyben a paraméterek reprezentációi vannak felsorolva, a forrás kódjában szereplő sorrendben. A reprezentációkat idézőjelek közé kell raknunk. Az itt megadott *format specifier* jelölések tetszőlegesek lehetnek amíg azok léteznek a C programozási nyelvben. Tehát nem csak a felhasználói felületen feltüntetett értékeket vehetik fel. Az itt megadott értékek azt határozzák meg, hogy milyen formátumban szeretnénk látni az adott paraméter által felvett értékeket. A konfigurációs fájlban nincs lehetőségünk kizárólag tetszőleges sorszámú paraméter megadására, hanem a monitorozni kívánt paraméter előtti paramétereket is fel kell sorolnunk. Így például, ha csak a negyedik paramétert akarjuk megfigyelni, akkor is meg kell adnunk az előtte lévő kettőt is. A harmadik ábrán található részlet ad példát a konfigurációs fájl lehetséges felépítésére [4].

```
app1:
  - func1
  - func2:
    - '%s'
    - '%d'
app2:
  -  func3
```

4. ábra: A konfigurációs fájl struktúrája

A konfigurációs fájl nem használható együtt a felületen bevitt elemekkel, így, ha a konfigurációs fájl beviteléhez tartozó kapcsoló be van kapcsolva, akkor csak a fájlban leírt elemeket veszi figyelembe az alkalmazás. Kikapcsolása esetén pedig csak a felhasználói felületen bevitt elemek kerülnek elemzésre.

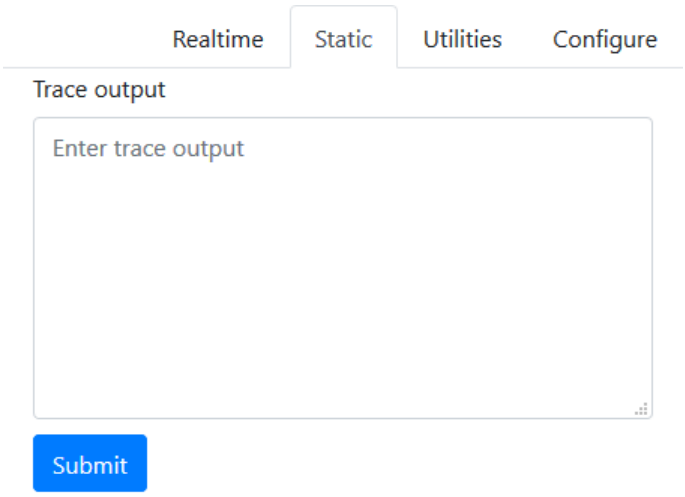
Miután valamely módon megadtuk az elemezni kívánt függvényeket és paramétereiket, elindíthatjuk a valós idejű monitorozást. Ezt az előző felhasználói felületi elemek alatti kerek bekapcsoló gomb segítségével tehetjük meg. A monitorozás sikeres indítása esetén a gomb szürkéről zöld színűre vált, az ötödik ábra szerint.

5. ábra: A monitorozást indító gomb kikapcsolt állapotban (balra) és aktív monitorozás alatt (jobbra)

Amíg a függvények figyelése tart, a bal oldalon jelenik meg a figyelt függvények hívási gráfja, amely félmásodpercenként frissül. Az alkalmazás további funkcióit nem érhetjük el amíg a monitorozás tart, illetve a valós időben kirajzolt gráf elemeit sem mozgathatjuk, mivel a gráf folyamatosan frissül. Indítás után kézzel fejezhetjük be a monitorozást a gomb újbóli megnyomásával. Ellenkező esetben a monitorozási mód magától nem áll le csak hiba esetén.

2.4.2. Statikus kimenet beolvasása és feldolgozása

A program lehetőséget ad arra, hogy már meglévő BCC trace kimeneteinket beolvassuk és adatokat nyerjünk ki belőlük. A *Static* fülre kattintva érhetjük el az alkalmazásnak ezt a másik használati módját. A valós idejű monitorozással ellentétben ez a funkció elérhető Windows rendszereken, illetve akármilyen, a BCC-t nem támogató operációs rendszer alatt is. Az ehhez tartozó felület, melyet a hatodik ábrán láthatunk, jóval egyszerűbb ahhoz képest, mint amit az élő monitorozás esetén láthattunk.



6. ábra: Felület statikus kimenet feldolgozásához

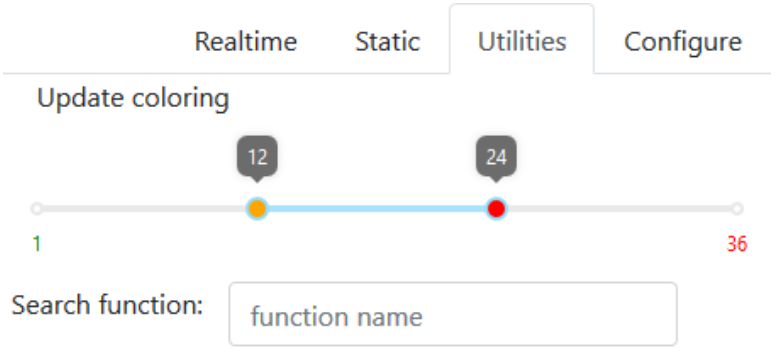
A *Trace output* felirat alatti multi-line beviteli mezőben illeszthetjük be a kívánt bemenetet, majd a *Submit* gombra kattintva fel is dolgozhatjuk azt. Ekkor rögtön megkapjuk a kimenethez tartozó teljes hívási gráfot és a benne lévő különböző adatokat. A beviteli mező vertikálisan tetszőlegesen nagyítható, de természetesen annál több sor szöveget illeszthetünk be, mint amennyi a dobozban látszólag elfér. Ekkor a bemenet jobb oldalán görgetősáv jelenik meg, amellyel a beillesztett szövegben navigálhatunk.

Egy trace kimenet mindig egy fejléc sorral kezdődik. Az alkalmazás képes kezelni azt, ha a fejlécet is benne hagyjuk a beillesztett kimenetben, de úgy is hibátlanul működik, ha levágjuk a szöveg elejéről. A fejléc a következőképpen nézhet ki:

```
PID     TID     COMM            FUNC                    -
```

2.4.3. Eszköztár

A harmadik fülre kattintva, amelyen a *Utilities* felirat olvasható, a gráffal kapcsolatos megjelenítési eszközök találhatók. Az ezeket tartalmazó felület a hetedik ábrán látható, abban az esetben, amikor már betöltöttünk egy gráfot.



7. ábra: Az eszköztár felülete, megjelenített gráf esetén

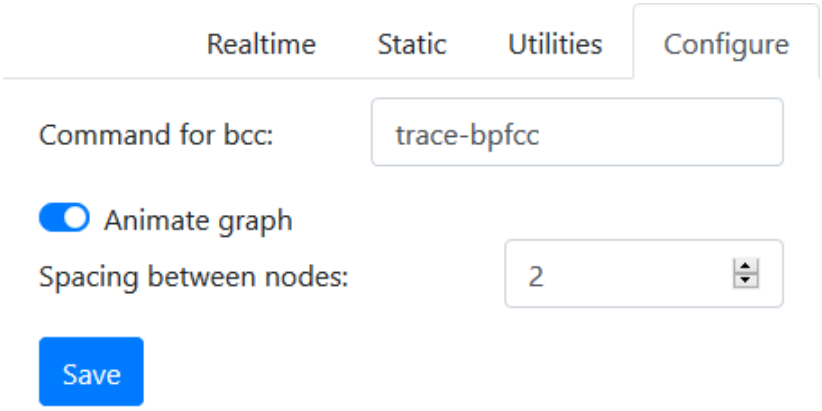
Az *Update coloring* felirat alatt található csúszka segítségével állíthatjuk be a követett függvényhívások számaitól függő színezését. A csúszka mindig 1-től kezdődik és maximuma az a szám, ahányszor a legtöbbször hívott függvényt hívták. Ezen a csúszkán egyszerre két értéket irányíthatunk, melyek sosem lehetnek egyenlőek. Az alsó fogantyú határozza meg azt, hogy egy vizsgált függvény a gráfban hány függvényhívásig maradjon zöld színű. A felső érték pedig a sárgára színezett gráfok hívásszámának a maximumát jelöli, tehát minden, az értéknél többször hívott függvény piros színű lesz. A csúszka addig nem használható, amíg nincs feldolgozva egy kimenet és nincs megjelenítve a gráf, hiszen addig a csúszka maximum értéke nem lenne egyértelmű.

Az alkalmazás lehetőséget ad függvénynevek keresésére is. Ezt a *Search function* felirathoz tartozó szöveges beviteli mező segítségével tehetjük meg. A kereső dobozba elég csak elkezdenünk begépelni a függvény nevének egy részét és minden begépelt betű automatikusan szűkíti a keresést. A gráfon így végül csak a csak a keresésnek megfelelő függvények maradnak színezettek, azok pedig amik nem felelnek meg a keresőkifejezésnek, azok szürkére váltanak. A keresés kis- és nagybetű érzékeny. Természetesen a keresési funkció, illetve a színezés beállítása együtt is alkalmazható. Ekkor minden függvény színe, amelyre illik a keresőkifejezés, az a csúszka értékeinek megfelelően változtatja a színét.

2.4.4. Beállítások

Az utolsó fül, mely a képernyő jobb szélén található és a *Configure* feliratot viseli, rejti a programra alkalmazható beállításokat. A vezérlőpultnak ez a része látható a

nyolcadik ábrán. A *command for bcc* feliratú beviteli mezőben adhatjuk meg, hogy számítógépünkön a BCC trace modulja milyen paranccsal hívható. A parancs alapértelmezett értéke `trace-bpfcc`, de ez tetszőlegesen változtatható, ha esetleg gépünkön más paranccsal használjuk az eszközt.



8. ábra: Konfigurálási felület

A parancs beállítása alatt található az *Animate graph* feliratú kapcsoló. Ennek segítségével egy kisebb grafikai beállítást kapcsolhatunk ki és be, amely azt határozza meg, hogy a gráf kirajzolódása közben az új elemek animálva jelenjenek-e meg. Ennek csak a valós idejű monitorozás alatt van jelentősége, mivel csak ekkor változhat szemünk láttára a gráf.

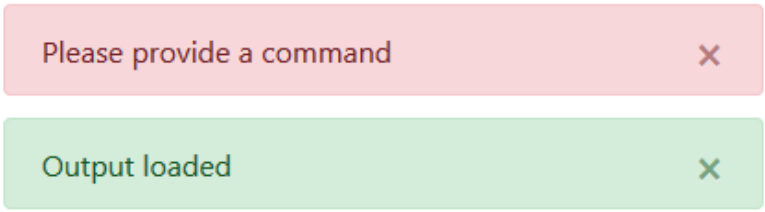
Végül az utolsó beállítás segítségével állíthatjuk a kirajzolt gráfban megtalálható függvények közötti távolságot. Hosszú függvénynevek esetén a függvényeket reprezentáló dobozok kitakarhatják egymást a gráfban. Ekkor ezt az értéket növelve kaphatunk szellősseb reprezentációt, amellyel ez a probléma orvosolható. Ugyanígy rövid függvénynevek esetén, ha fölöslegesen sok helyet foglal a gráf, az érték csökkentésével tömörebb formát kaphatunk.

Ahhoz, hogy a beállításaink életbe lépjenek, a *Save* gombra kattintással el kell tárolnunk a bevitt értékeket.

2.4.5. Értesítések

A felhasználói felület helyes használata érdekében az alkalmazás értesítésekkel lát el minket, ha valamelyik funkciót helytelenül használjuk, illetve akkor is, ha valamilyen döntést sikerül végrehajtani. Hibaüzenetet kapunk például, ha valamilyen beviteli mezőnek nem adtunk értéket, de később használni akarjuk, vagy ha hibás értéket adtunk meg. Arról is visszajelzést kapunk, ha az élő monitorozás során lép fel valamilyen hiba. Pozitív visszajelzést kaphatunk akkor, ha például sikeresen elmentettünk beállításainkat, vagy ha egy bemenetünket sikeresen fel tudta dolgozni a rendszer.

Az értesítések minden esetben tömör szöveges üzenetként jelennek meg a képernyőn. A jelzések helye mindig a jelzést kiváltó beviteli elem közelében található. Az értesítések piros vagy zöld szövegdobozban jelennek meg, amelyből az előző mindig hibát jelez, utóbbi pedig egy sikeres műveletről tájékoztat minket. A kilencedik ábrán láhatunk példát egy hibaüzenetre, illetve egy sikeres akció utáni értesítésre.



9. ábra: Példa egy hibaüzenetre (felül) és egy sikeres akció értesítésére (alul)

Habár az értesítések önmagukban is informatívak, az alábbi lista minden értesítéshez további magyarázatot nyújt:

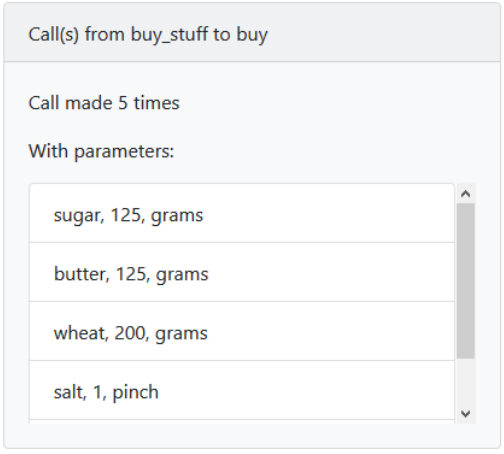
- **Application already added**: A megadott alkalmazás már szerepel a megfigyelt források listájában. Mivel egy forráshoz bármennyi függvényt hozzáadhatunk, így nem szükséges ugyanazt a forrást többször hozzáadnunk.
- **<alkalmazás neve> added successfully**: A megadott alkalmazást sikeresen hozzáadtuk a monitorozni kívánt források listájához. Így már kiválaszthatjuk azt a legördülő menüből.
- **Please provide an application name**: Úgy próbáltunk alkalmazást hozzáadni, hogy nem adtuk meg hozzá az elérési útvonalat.
- **Please select an application first**: Úgy próbáltunk meg eltávolítani, vagy módosítani alkalmazást, hogy előtte nem választottuk ki azt a források listáját reprezentáló legördülő menüből.
- **Please provide a function name**: Úgy próbáltunk függvényt hozzáadni a monitorozni kívánt függvények listájához, hogy nem adtunk meg nevet.
- **Function already added to this application**: A megadott nevű függvény már szerepel a forráshoz tartozó, megfigyelni kívánt függvények listájában. Nincs értelme egy függvényt többször hozzáadnunk a listához.
- **<függvény neve> added successfully**: A megadott függvényt sikeresen hozzáadtuk a monitorozni kívánt függvények listájához. Így már kiválaszthatjuk a legördülő menüből és adhatunk hozzá paramétereket megfigyelésre.
- **Please select a function first**: Úgy próbáltunk meg törölni, vagy módosítani függvényt, hogy előtte nem választottuk ki a legördülő menüből.

- **Please provide the type of the parameter**: Paraméter hozzáadásakor nem választottuk ki annak típusát a típusok legördülő menüjéből.
- **Please provide the position of the parameter**: Paraméter hozzáadásakor nem választottuk ki annak pozícióját a paraméterlistában.
- **Parameter already added to this position**: A megadott pozícióhoz már hozzáadtunk megfigyelni kívánt paramétert. Egy függvénynél egy adott pozíción csak egy paraméter lehet, hiszen ez az elhelyezkedését jelöli a kódban
- **Parameter successfully added**: A megadott paramétert sikeresen hozzáadtuk a monitorozni kívánt paraméterek listájához.
- **Please select a parameter first**: Úgy próbáltunk meg paramétert eltávolítani, hogy előtte nem választottuk ki a paraméterek legördülő menüjéből.
- **There was an error with tracing**:
    - **No functions to trace**: Nem adtunk meg egyetlen függvényt sem amelyet monitorozni szeretnénk, így azt nem lehet elindítani.
    - **Please provide a path to the configuration file**: Úgy próbáltunk meg konfigurációs fájllal monitorozást indítani, hogy nem adtuk meg a fájl elérési útvonalát.
    - **Could not find configuration file at <útvonal>**: A megadott útvonal hibás, mert nem található ott fájl.
    - **<útvonal> is a directory, not a file**: A megadott útvonal egy könyvtárat jelöl, nem pedig egy fájlt.
    - **Config file at <útvonal> has to be YAML format**: A konfigurációs fájl adatstruktúrája nem YAML sémájú.
    - **Could not process configuration file**: A konfigurációs fájl nem a helyes formátumot követi, így nem sikerült feldolgozni azt.
    - **Unknown error happened while processing config file**: A konfigurációs fájl feldolgozása során ismeretlen hiba lépett fel.
    - **The command was not found or was not executable: <command>**: A beállításokban megadott BCC trace parancsot nem sikerült futtatni, így a monitorozás nem lehetséges.
    - **Tracing stopped unexpected**: A monitorozás valamilyen hiba folytán idő előtt leállt. Ez főleg hibás forrás, vagy függvénynév megadása esetén történhet.

- **Output loaded**: A megadott BCC trace kimenetet az alkalmazás sikeresen feldolgozta.
- **Please provide the output of a BCC trace run**: A megadott trace kimenetet nem sikerült feldolgozni, vagy nem adtuk meg.
- **Please provide a command**: A BCC trace modult futtató parancs bemeneti mezőjét üresen hagytuk a konfiguráció mentésekor.
- **Configuration saved**: A konfigurációt sikeresen elmentette és firssítette az alkalmazás.

2.4.6. Információs panel

A baloldalt található gráf nem csupán egy statikus ábra. A gráf minden éléről és pontjáról további információkat kaphatunk, ha rákattintunk. Ekkor a vezérlőpult alsó felén jelenik meg az adott elemhez tartozó információs panel. Itt tájékozódhatunk egy függvény, vagy egy függvényhívás tulajdonságairól, attól függően, hogy a gráfban egy függvényre, vagy két függvényt összekötő élre kattintottunk-e. Egy függvényről megtudhatjuk, hogy hányszor lett meghívva a BCC trace kimenete szerint, hogy melyik binárisból, vagy objektumfájlból származik, illetve, ha paramétereit is megfigyeltük, akkor azok értékeit is megtalálhatjuk felsorolva egy listában, amikkel a függvény meg lett hívva. Ha egy két függvényt összekötő élre kattintunk, a hozzá tartozó információs panelen megtalálhatjuk, hogy ez a függvényhívás hányszor történt és milyen paraméterekkel. Egy ilyen információs panelt láthatunk a tizedik ábrán.



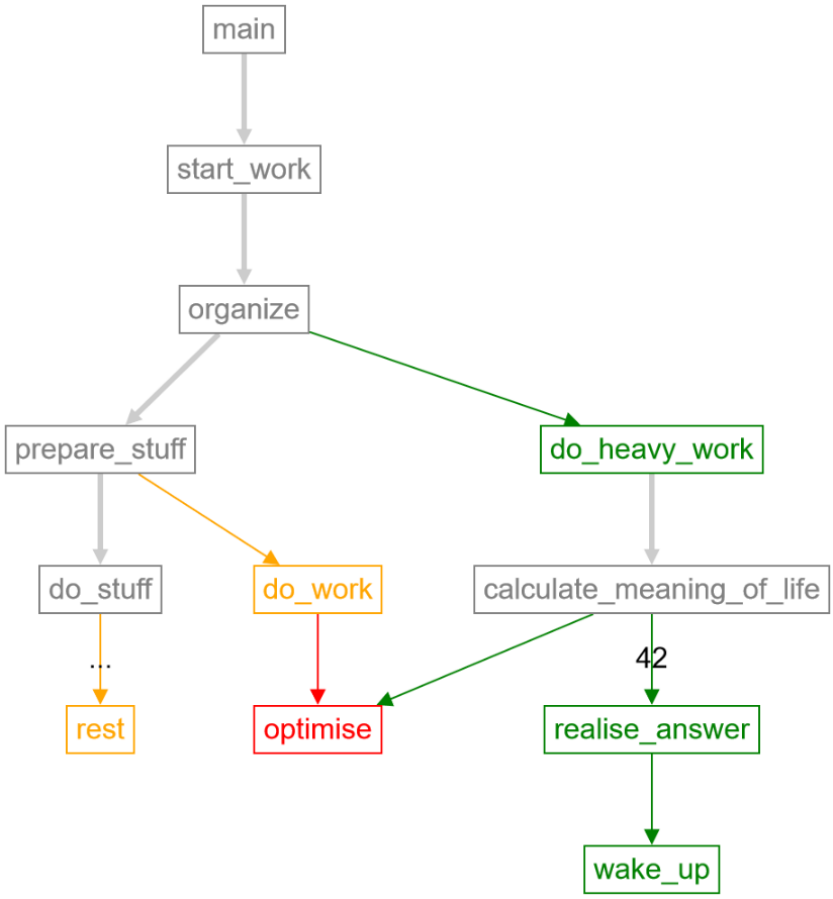
10. ábra: Információs panel, amelyet egy élre kattintva láthatunk

A paramétereket felsoroló lista egy eleme az egy híváshoz tartozó paraméterek egy sorban. Tehát minden hívás egy külön elem a listában, melybe az összes használt paraméter értéke beletartozik. Ha egy függvény, vagy függvényhívás rengetegszer fordul

elő különböző paraméterekkel, úgy a paramétereket felsoroló lista egy görgetősávval válik böngészhetővé.

2.5. Gráf

Az alkalmazásnak két fő használati módja van, de azok eredménye mindkét esetben egy, a kimenetek feldolgozásából kinyert adathalmaz előállítása, melynek fő reprezentációjára a megfigyelt függvények hívási láncaiból kialakuló gráf. Ez a gráf a felhasználói felület bal oldalán helyezkedik el egy sikeres feldolgozást követően, vagy a monitorozás alatt és után. A gráf reprezentációjára a tizenegyedik ábrán láthatunk példát.



11. ábra: Példa az alkalmazás által kirajzolt gráfra

A gráf elemei szabadon mozgathatók, ha éppen nem végzünk élő monitorozást. A gráf maga is mozgatható, nagyítható és kicsinyíthető, így átláthatjuk a megfigyelt függvények által kialakított egész hívási gráfot, vagy koncentrálhatunk csupán egy kisebb területre is. A gráf pontjainak alapértelmezett elrendezése automatikus, de alapvetően a képernyő tetején láthatjuk a hívási láncok elején elhelyezkedő függvényeket, majd egyre lefelé haladunk a hívási láncon és az ábrán, míg annak legalján főleg a megfigyelt függvények találhatóak.

2.5.1. Függvények

A gráf pontjai a megfigyelt függvényeket reprezentáló színes dobozok, amelyeken az adott függvény neve szerepel. Ezeket a pontokat szabadon mozgathatjuk a gráfnak fenntartott területen, így kialakítva a számunkra logikus és átlátható ábrát.

A függvény színe a gráfban lehet szürke, zöld, sárga és piros, melyek több jelentéssel bírhatnak. Szürkével jelöljük azon pontokat melyek a hívási lánc részei, de nem monitoroztuk őket külön. Ezekről a függvényekről nem tudjuk hányszor lettek meghívva a megfigyelés során, így ők nem kapnak az ehhez az értékhez tartozó színt sem. A további három színt csupán a külön elemzett függvények vehetik fel. Ezek a színek könnyítik meg felhasználónak, hogy azonosítsa egy megfigyelt program hotspot-jait azáltal, hogy vizuális visszajelzést kap a gyakran és a kevesebbszer előforduló függvényekről a megfigyelés során. A zöld szín hívatott a legkevésbé használt függvényeket jelölni, míg a sárga mutatja, hogy mely függvények lettek átlagosan sokszor meghívva. A piros szín jelöli a legtöbbet meghívott függvényt, illetve további gyakran használt függvényeket. A vezérlőpultban lehet beállítani, hogy hány hívástól milyen színt vegyenek fel a függvényekhez tartozó gráfpontok.

Egy függvény színe nem csak akkor lehet szürke, ha nem figyeltük meg, de akkor is, ha a vezérlőpultban található kereső funkciót használjuk. Ebben az esetben egy megfigyelt gráf is elvesztheti a színét, ha neve nem tartalmazza a keresőkifejezést.

2.5.2. Élek

A gráf élei a kimenet alapján feltérképezett függvényhívásokat reprezentálják. Egy élnek mindig egy függvény a kiindulópontja, és mindig egy függvény a végpontja. Az éleknek van iránya, melyet az él egyik végén található kis nyíl jelez. Ebből adódóan a hívási láncokból kialakuló gráf egy irányított gráf, amely segítségével pontosan meg tudjuk állapítani, hogy mely függvény melyiket hívta a monitorozás során. Ha az *A* függvényből jövő él *B* függvényre mutat, az azt jelenti, hogy a megfigyelés során *A* függvény hívta *B* függvényt.

Az élek színe ugyanúgy, akárcsak a pontok esetén, mutatja a hívás gyakoriságát. Természetesen itt csak az adott két függvény közötti hívások számát vesszük figyelembe, így előfordulhat, hogy egy függvény és a hozzá tartozó hívások eltérő színűek. Egy él szürke, ha a függvény, amibe tart szintén szürke, tehát ha a hívott függvényt nem figyeltük meg, vagy neve nem tartalmazza a keresőkifejezést. Az élek, akárcsak a függvények, szabadon mozgathatók. Ekkor az él mindkét végén megtalálható pont vele együtt mozog.

Az éleken megjelenhetnek a hívás paraméterei, ha azokat megfigyeltük. Ekkor, ha az adott hívás csupán egyszer fordult elő a megfigyelt program futása során, úgy a paraméterek rögtön megjelennek a gráfon. Ha több mint egyszer szerepel a hívás, úgy az élen három pont (…) jelzi, hogy a híváshoz több paraméter tartozik. Ekkor az információs panel használatával tudhatjuk meg, hogy mik voltak ezek a paraméterek.

2.5.3. A gráfban nem megjelenített információk

Az interaktív gráf elemeire lehetőségünk van rákattintani, hogy további információkat tudjunk meg róluk. Kattintás után a vezérlőpult alsó felében megnyílik egy információs panel, amely további információkat szolgáltat a kiválasztott elemről.

Az alkalmazás használata során lehetséges, hogy két különböző bináris függvényeit is vizsgáljuk. Ekkor előfordulhat, hogy a két különböző alkalmazásban ugyanazzal a névvel szerepeljenek függvények. Mivel minden függvényhívásról elmentjük, hogy melyik forrásból származik, így ekkor a gráfban is külön jelennek meg ezek a függvények, még akkor is, ha a nevük azonos. Arról, hogy melyik függvény melyik forrásból származik, az információs panel tájékoztat.

3. Fejezet
Fejlesztői dokumentáció

A dolgozat célja egy webes felhasználói felület készítése, egy amúgy parancssorból futtatható eszköz számára. A felület vezérléséhez, illetve annak adatokkal való feltöltéséhez háttérszámításokat is végeznünk kell. Ebből adódóan többrétegű alkalmazást készítünk, amelyben a felület alá modell is tartozik. A különböző rétegek egyértelműen elkülönülnek és így kicserélhetők, így biztosítva a rugalmas fejlesztési lehetőségeket és az alkalmazás bővítését más eszközök támogatására.

3.1. Tervezés

3.1.1. Programszerkezet

A programot *Model View ViewModel* (MVVM) architektúrában valósítjuk meg. Ennek megfelelően létrehozzuk a Nézet (`View`) rétegét, melyben a felhasználói felület és az általa felhasznált vizuális komponensek találhatóak. A Modell (`Model`) felel az üzleti logikáért, vagyis az adatok feldolgozásáért és különböző számítások végzéséért. A Nézetmodell (`ViewModell`) köti össze a nézetet a modellel, tehát fő feladata, hogy a nézet számára értelmezhető módon prezentálja a modell által előállított adatokat. Megvalósítjuk a modell számára adatokat eltároló Perzisztenciát (`Persistence`), amely az alkalmazás számára hosszútávú adattárolást biztosít. A rétegeket irányító Környezet (`App`) réteg összefogja az alkalmazás komponenseit és felel azok megfelelő példányosításáért.

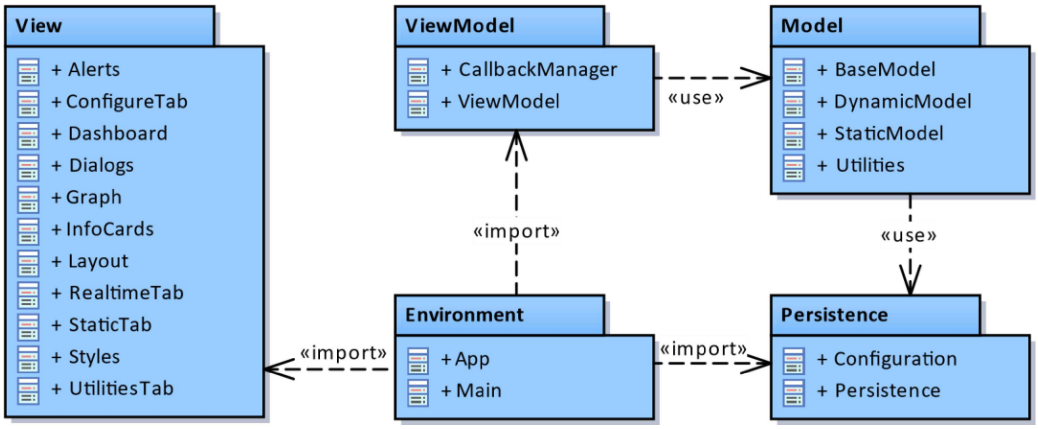
Az alkalmazásnak két felhasználási módja van. Az egyik az élő monitorozást teszi lehetővé, amíg a másik, egy statikus szöveget képes feldolgozni. A két különböző funkcióhoz két különböző modell tartozik. Mivel azt, hogy éppen melyik funkciót akarjuk használni futás közben döntjük el, így a szükséges modellt mindig dinamikusan hozzuk létre. Ezáltal a modellt és a hozzá tartozó perzisztenciát nem a környezet réteg határozza meg. A program csomagszerkezete az alábbi ábrán látható.

3.1.2. Implementáció

A programot Python programozási nyelvben valósítjuk meg, amelyet csupán a webes megjelenítés finomítása miatt egészítünk ki egyetlen rövid CSS állománnyal. Az alkalmazás a nyelv által nyújtott előnyöket kihasználva nem tisztán objektum orientált, egyes moduljai csupán függvényeket tartalmaznak. A rétegek, a környezeten kívül mind

külön könyvtárakba vannak elhelyezve, így egyértelműsítve a határaikat. A rétegek közötti kapcsolatokat függőség-befecskendezéssel oldjuk meg.

A környezet felel az alkalmazás indításáért és annak előkészítéséért. Ehhez szükségünk van a konfigurációra, a nézetre és a nézetmodellre. A nézetmodell dinamikusan kezeli a modelleket, melyek saját maguknak hozzák létre a perzisztenciát. A rétegek által alkotott csomagdiagrammot az alábbi ábrán láthatjuk.



12. ábra: Az alkalmazás csomagdiagramja

A webes vizuális felületet és a hozzá tartozó webszervert a Dash [5] keretrendszer segítségével valósítjuk meg, amelyben a felület komponenseit tisztán Python kóddal, objektum orientáltan tudjuk létrehozni és kezelni.

3.1.3. Dash keretrendszer

A Dash egy Python programozási nyelvhez készült keretrendszer, kifejezetten grafikus felületű webes alkalmazások fejlesztésére. A keretrendszer legnagyobb erénye, hogy a grafikus felületet objektum orientáltan, Pythonban készíthetjük el, anélkül, hogy HTML, vagy JavasScript kódot kellene írnunk. Ugyanakkor, a sajátos eseménykezelés miatt, a webapp indulásakor a felhasználói felület minden olyan elemének ismertnek kell lennie, amelynek egyedi azonosítója van. Ebből adódóan a dinamikus felhasználói felület generálás jelentősen limitált.

A felület mindenfajta elemét egy-egy osztály reprezentálja, amit a keretrendszer valamelyik modulja szolgáltat. Így például, ha alkalmazásunkban egy `<div>` címkéjű HTML elemet szeretnénk létrehozni, úgy kódunkban a `Div` osztályt kell példányosítanunk melyet a `dash_html_components` modulból érhetünk el. Minden ilyen példánynak egyedi azonosítója és egyéb, nevesített tulajdonságai vannak, melyekre hivatkozhatunk eseménykezeléskor. Legfontosabb közülük a `children`, amely az elem tartalmára hivatkozik. Ez a tartalom lehet egy újabb elem, vagy több másik

elem listája is. Így az egész felületet felépíthetjük elemek egymásba ágyazásával. A felületen végbemenő minden változás valójában egy felületi elem valamely tulajdonságának módosítására vezethető vissza.

A keretrendszer rendhagyó módon kezeli az eseményvezérlést. A Dash-ben az eseményeket *callback*-eknek hívjuk. Minden *callback*-nek kell, hogy legyen legalább egy kiváltó oka (`Input`) és legalább egy hatása (`Output`). Mind a kiváltó okok, mind pedig a hatások a felhasználói felületen található elemek valamilyen attribútumának a megváltozását jelentik. Ezeken kívül lehetőségünk van tetszőleges elemek attribútumainak az állapotát (`State`) is felhasználni az esemény kezelésekor. Minden elemre az elem egyedi azonosítójával hivatkozunk, amelyhez az elem típusától függően különböző attribútumok tartozhatnak. A callback az őt beállító függvény névterében van, így hivatkozhat annak paramétereire, az ott beállított változókra.

3.2. Perzisztencia

A perzisztencia réteget a `persistence` csomag tartalmazza. Ez a réteg felel az alkalmazás hosszútávú adattárolásáért. A rétegben két modul és azokon belül egy-egy adattároló osztály található. A klasszikus perzisztencia felel a gráf adatainak tárolásáért, amíg a konfiguráció az alkalmazás alapvető beállításait tartalmazza.

3.2.1. Konfiguráció

A felhasználói felület vezérlőpultján *Configuration* feliratú menüre kattintva érhetjük el az alkalmazásra vonatkozó beállításokat. Ezeket a felhasználó tetszés szerint módosíthatja, így az alkalmazás egyes részei személyre szabhatók. Ezen beállításokat tartalmazza a `Configuration` osztály, melyet a `configuration` modul tartalmaz.

Az osztály konstruktorában beállítható mind a három konfigurálható beállítás, de ezeknek mind van alapértelmezett értéke, így az osztály példányosítható alapbeállításokkal is, amelyekkel rögtön használható. Az osztály interfészt biztosít a konfigurációs értékek beállításának és a lekérdezésének. A BCC trace parancsát meghatározó érték és a gráfra vonatkozó beállítások között logikai határt húzunk, így eme két csoport számára biztosítunk egy-egy *setter* függvényt. Minden beállítási értékhez külön *getter* függvény tartozik.

3.2.2. Perzisztencia

Az alkalmazás legfontosabb feladata, hogy a BCC trace kimenetek feldolgozása során kapott információkból gráfot rajzoljon ki a felhasználó képernyőjére. Az ehhez a gráfhoz tartozó adatokat az alkalmazás nem tudja teljes mértékben a nézetben tárolni.

Egyfelől azért, mert egyes adatokat bonyolult lenne a Dash keretrendszer által szolgáltatott lehetőségekkel tárolni. Másfelől pedig azért, mert a lehető legkevésbé szeretnénk a keretrendszerre támaszkodni olyan esetekben, amelyek nem az általa szolgáltatott nézetet érintik. Ezáltal függetlenné válhatunk a nézettől és rugalmasan akár le is cserélhetjük azt.

A `persistence` modulban található `Persistence` osztály hivatott arra, hogy minden adatot tartalmazzon arról a gráfról, amellyel a felhasználó éppen dolgozik. Az osztály tisztán adattárolást végez, illetve az adatok megfelelő kezelését rejti magába. Ennek megfelelően az adatok betöltését speciálisan megvalósító metódusokat és ezen adatok lekérdezését biztosító függvényeket tartalmazza.

A pontok és élek betöltését a perzisztenciába a `load_nodes` és a `load_edges` függvények végzik. Segítségükkel élő monitorozás során az adatokat inkrementálisan bővíthetjük adatvesztés nélkül. Így ugyanarra a pontra, vagy élre vonatkozó újabb elérhető információk esetén a perzisztencia képes a megfelelő módon hozzáadni ezen adatokat az adott elemekhez.

3.3. Modell

A program modell rétege felel minden számításért és az *üzleti logikáért*. A teljes modellt a `model` könyvtár tartalmazza, amelyben megtalálható a három modelltípus, illetve a közös, több réteg által is használt függvények eszköztára. A modell feladata elsődlegesen a beérkező adatok feldolgozása, majd azokból, vagy azok által létrejövő gráf adatok elmentése a perzisztenciába. A modellek ezen kívül több olyan függvényt is tartalmaznak, melyekkel a perzisztenciában tárolt adatokat kérdezheti le a nézetmodell. Mivel minden adatot a perzisztenciában tárolunk, így egy állapottárolás nélküli modellt valósítunk meg.

3.3.1. Ős modell

A program két különböző funkciójának megvalósításához két különböző modell létezik. Azonban a két modell interfésze és a működésekhez szükséges beállítások nagyban megegyeznek egymással. Éppen ezért a két modell egy közös ősből származik, amely definiálja a közös függvényeket és változókat. Ez a közös ős a `base` modulban található `BaseModel` osztály.

A `BaseModel` osztály konstruktorában várja a konfigurációt, amelyet privát adattagként eltárol. Ezen kívül konstruktorában létrehozza az adattárolásra alkalmas `Persistence` típusú adattárolót. Mivel a statikus és a dinamikus funkciót megvalósító

modell is az ősosztály konstruktorát használja a sajátjában, ebből következően a perzisztencia élettartama az őt létrehozó modellével egyezik meg. Az osztály további függvényei olyan alapvető lekérdező és beállító metódusok, melyeket a futás közben mindkét típusú modell esetén felhasználunk, ezáltal a típusoktól függetlenek. Ilyen például a perzisztenciát, vagy a konfigurációt kezelő parancsok, vagy a gráf állapotát lekérdező függvények.

3.3.2. Statikus modell

A statikus modell az alkalmazás azon funkcióját valósítja meg, amelyben egy már meglévő BCC trace kimenetet dolgozunk fel és alakítunk át gráffá. A statikus modellt a `static` modulban található `StaticModel` implementálja. Konstruktorában csupán az ősosztály konstruktorát hívja meg a paraméterben kapott konfiguráció egyeddel, más belső változóra, vagy beállításra nincs szüksége. Az osztály további része is hasonló egyszerűséget mutat. Egyetlen függvénye a `load_text`, amely a nyers trace kimenetet várja, majd feldolgozza azt az `utils` modulban található függvények segítségével. A feldolgozott gráfot pedig betölti a perzisztenciába.

3.3.3. Dinamikus modell

Az élő monitorozást végző dinamikus modell jóval nagyobb mértékben bővíti ki az ősosztály funkcionalitását a statikus modellhez képest. A dinamikus modell típusa a `DynamicModel` osztály, melyet a `dynamic` modul tartalmaz. Az osztály inicializáláshoz itt is csak a konfiguráció példányára van szükségünk, amelyet az ősosztály konstruktorába adunk tovább, viszont ez a modell további belső változókat tartalmaz.

Az élő monitorozáshoz a dinamikus modell elindít egy BCC trace monitorozást a megfigyelni kívánt függvényeknek megfelelő paranccsal. Ahhoz, hogy egyszerre tudjuk futtatni a trace modult és feldolgozni a kimenetét, illetve folyamatosan frissíteni a felhasználói felületet, a monitorozást egy külön szálon futtatjuk. A külön szálban indított folyamat kimenetének folyamatos feldolgozását egy ciklusban oldjuk meg, amelynek ciklusváltozója az osztály egy adattagja. Ezáltal ezt a ciklust tetszőleges időpontban kívülről is le tudjuk állítani, ami elengedhetetlen, mivel egyébként a trace modul csupán hiba esetén terminál, más esetben örökké fut. Mivel a modell nem tudja közvetlenül értesíteni a nézetet és az eseménykezelőt az indított *thread* állapotáról, így az állapot jelzésére külön változókat hozunk létre, amelyeket az eseménykezelő a monitorozás alatt minden frissítéskor lekérdez. Két ilyen állapotjelző változót hozunk létre. Az egyik a szál indításának előkészítése közben fellépő hibákat tárolja, míg a másik a szálban futás közben fellépő hibákat menti el.

A külön szálon indított folyamatok kimenetének valós idejű feldolgozása nehézkes az általuk kiírt sorok *buffer* tartományai miatt. Ahhoz, hogy a trace kimenetét zökkenőmentesen, azonnal megkapjuk, a párhuzamosan indított *thread*-ben a pexpect [6] modul segítségével indítjuk el a BCC trace modult. a pexpect segítségével tetszőleges parancsot futtathatunk a `spawn` osztállyal, majd kommunikálhatunk az elindított folyamattal. Alapvetően ilyenkor ez a folyamat blokkolná a főszálat, ezért helyezzük az egész logikát egy külön szálra. Jelen esetben csupán a folyamat kimenete érdekel minket valós időben, amelyet az osztályon hívott `expect` függvény segítéségével kérhetünk le, amelynek paraméterben adhatjuk meg, hogy az adott kimenetet meddig kérjük le. A paraméter segítségével sorokra bonthatjuk a kimenetet. A kimenet sorait hívási láncokba csoportosítjuk, melyek határai a kimenetben található üres sorok. A hívási láncokat azonnal feldolgozzuk a `utils` modul függvényeivel és a kapott részgráfot betöltjük a perzisztenciába. Fontos, hogy az `expect` függvény blokkol, tehát ha éppen nincs feldolgozandó kimenet, akkor a szálon használt ciklus nem fut tovább. Ugyanakkor a függvénynek megadható egy időzítő, ami után TIMEOUT típusú kivételt dob. Az időzítő elég rövidre vételével és a kivétel lekezelésével érhetjük el, hogy ha a ciklusváltozó hamissá válik, akkor a következő iterációban kilépjünk abból és ne ragadjunk benne, kimenetre várva. Ez kritikus a ciklus és így a szál elegáns leállításának szempontjából.

A fő logika elindításához szükséges adatokat kétféleképpen állíthatjuk elő. Külön függvény kezeli a felületen beállított függvények kollekciójának feldolgozását és a konfigurációs fájllal való betöltést. A dinamikus modellnek ezen két belépési pontján kívül további speciális *getter* függvényei, illetve egy, a monitorozást leállítását vezérlő függvénye van.

3.3.4. Eszköztár

A program több olyan függvényt is tartalmaz, melyeket logikailag nem helyes egyetlen osztály részfüggvényeként sem megvalósítani. Tipikusan ilyenek a szövegeket *parse*-oló függvények. Habár ezeket implementálhatnák az egyes modellek, a külső forrásból érkező adatok struktúráját nem a program határozza meg, így jobb, ha ezek feldolgozását kiemeljük. Így az adatstruktúrák megváltozása esetén nem szükséges a modellhez nyúlnunk, elég az eszköztárban definiált függvényeket az új követelményekhez igazítanunk. Emiatt technikailag a modellek logikája független a külső forrásból származó adatok formájától.

Ezek a függvények és további, egyetlen osztályhoz sem köthető eszközök lettek kiemelve a `utils` modulba. Ugyan osztályok alá nem tudjuk őket besorolni, a függvények többsége számításokat végez és logikát valósít meg, így a `utils` modul a modell réteg része. Itt találhatóak a két használati funkció bemenetét feldolgozó függvények. A `flatten_trace_dict` a felületen összeállított forrásokat, függvényeket és paramétereket hozza a BCC trace parancsában használható formára.

Ugyanilyen formára hozza az `extract_config` függvény a konfigurációs fájl tartalmát. A konfigurációs fájl formátuma yaml, melynek betöltését a Python-hoz elérhető `yaml` modul segítségével végezzük. Ezeken kívül többnyire egyszerű, nem sok számítást végző függvényeket találhatunk itt, egyet kivéve.

A legtöbb szövegfeldolgozó függvény a modulban csupán az adat egyik formájáról a másikra konvertál. Ugyanakkor a dinamikus és a statikus feldolgozás közben is szükségünk van ezen adatok értelmezésére is, hogy az azokból kinyert információt már használható formában tudjuk a perzisztencia számára prezentálni. Ezt az értelmezést hajtja végre a `parse_stack` függvény, amely egy trace kimenet által meghatározott hívási lánc sorait várja egy listában. A függvény reguláris kifejezések segítségével azonosítja a sorok típusát és értelmezi az adott sorban közölt adatokat. Ezeket azután pontokká és élekké alakítja, melyeket az egész hívási láncra összegyűjt. Egy pontot egyértelműen meghatároz a hozzá tartozó függvény neve és annak forrása. Egy élt pedig egyértelműen meghatároz, hogy melyik két pont között található. Ezzel az a probléma, hogy így az élekre párok párjaiként kellene hivatkoznunk, amely átláthatatlan kódot eredményezne. Ezért ehelyett minden pontot a függvény és a forrás nevéből létrehozott *hash* értékkel azonosítunk. A kimenet teljes feldolgozása után a pontokat és éleket egy speciális `namedtuple` típusú objektumban adjuk vissza, melynek `nodes` adattagja tartalmazza a pontokat és `edges` adattagja az éleket.

A szövegek feldolgozása során különféle hibákkal találkozhatunk, melyek közül több beépített, de több olyan is van, melyek csupán az alkalmazás logikája szerint számítanak hibás eredménynek. Ezek egységes lekezelésére a modulban létrehozott `ProcessException` kivételosztályt használjuk, melyet egyedi szöveggel láthatunk el, így értesítve a felhasználót az egyes hibákról.

3.4. Nézetmodell

A nézetmodell felelős a nézet és a modell összekapcsolásáért, tehát feladata a modell irányítása a nézetből kapott parancsok által és a nézet módosítása a modellből származó adatok továbbadásával. A réteget implementáló fájlokat a `viewmodel` könyvtár tartalmazza. Ezt a réteget két részre osztjuk. Ide tartozik a klasszikus nézetmodellt megvalósító `ViewModel` osztály, illetve ide soroljuk az alkalmazás eseményvezérlését, amelyet a `CallbackManager` osztály implementál. Az eseményvezérlést további komponensekre bontjuk, így a különböző komponenseket vezérlő események elkülönülnek egymástól.

3.4.1. Eseménykezelés

A nézet elemei mind a Dash keretrendszer által definiált osztályokból származnak. Ezen elemekhez szintén a keretrendszer nyújtja az eseménykezelés módját. Minden eseményt valamilyen nézetben található elem tulajdonságának megváltoztatásával váltunk ki. A keretrendszer szabályai szerint minden eseménynek hatással kell lenni a nézet valamely elemének valamely tulajdonságára. Ezáltal a legtöbb munkát az eseményen belül, a felület frissítése előtt kell elvégeznünk.

Minden eseményt egy `app.callback` dekorátorral ellátott függvény valósít meg. A dekorátorban az app, az alkalmazásban példányosított webapp. Erre a webappra lesz értelmezve az adott esemény. Emiatt fontos, hogy a webapphoz már az esemény beállítása előtt hozzá legyen rendelve a nézet és abban az összes azonosító. A dekorátorban felsorolunk minden tulajdonságot az őt tartalmazó elem azonosítójával együtt egy párban, amelyet az esemény megváltoztat. Minden ilyen pár egy `Output` objektumot alkot. Az eseményt kiváltó tulajdonságokat és elemeket hasonló módon `Input` objektumokba csomagoljuk. Végül az eseményt nem kiváltó, de az esemény számára jelentőséggel bíró tulajdonságokhoz és elemekhez `State` objektumokat használunk. A három felhasznált osztályt a `dash.dependencies` modulból importáljuk. A függvény, amely magát az eseményt írja le, tetszőleges névvel látható el, de paraméterei pontosan az `Input` és `State` objektumokban megadott tulajdonságok, sorrendben. Így a tulajdonságok értékeire ezen paraméterek által tudunk hivatkozni. Fontos követelménye a keretrendszernek, hogy egy elem egy adott tulajdonsága csupán egyetlen *callback* `Output`-jaként szerepelhet. Ez azt jelenti, hogy minden arra a tulajdonságra vonatkozó eseményt egyetlen függvénybe kell gyúrnunk. Szerencsére ezt megkönnyíti, hogy a `callback_context` változó segítségével lekérdezhetjük az

eseményt kiváltó `Input` elem azonosítóját és így ennek ismeretében dönthetünk az elvégzendő számításokról és az `Output` tulajdonságok értékéről.

A függvény alapvetően egy teljesen átlagos Python függvényként viselkedik, így tetszőleges számításokat végezhetünk benne. Ugyanakkor visszatérési értékei mindig az `Output` objektumokban megadott tulajdonságok új értékei kell, hogy legyenek, sorrendben. Lehetőségünk van `None` objektumot is visszaadni, viszont ilyenkor pontosan erre az értékre frissítjük az adott tulajdonságot, amely csak különleges esetekben indokolt. Ha azt akarjuk, hogy az esemény ne váltson ki változást a felületen, akkor a speciális `PreventUpdate` kivételt kell kiváltanunk. Ez azért fontos az alkalmazásban, mert több helyen alkalmazzuk különböző események láncolatát. Ilyen láncolat akkor jön létre, mikor egy esemény kimenete egy olyan tulajdonság, amely egy másik esemény bemenete. Ilyenkor mivel az első esemény megváltoztatja az adott tulajdonságot, azzal implicit aktiválja a második eseményt. Ez akkor is igaz, ha az első esemény `None` objektumot állított be. Így azt, hogy a láncolat ne induljon be, a kivétel segítségével érjük el.

Az események függvényeit és a hozzájuk tartozó dekorátorokat egy további függvénybe csomagoljuk. Ez azért előnyös, mert ekkor a becsomagoló függvény paramétereihez hozzáfér a dekorátor és az esemény is. Ennek segítségével tudnak a dekorátorok a környezetben létrehozott webappra hivatkozni. Ezen kívül ilyen módon további információkat is megoszthatunk az események között, mint például a nézetmodellt, amelybe egyes események információkat, vagy parancsokat küldenek tovább. Egy esemény addig nem az alkalmazás része, amíg a csomagolófüggvényét meg nem hívjuk. Így tulajdonképpen a csomagolófüggvényeik meghívásával inicializáljuk az eseményeket.

Az alkalmazás eseményeit a `CallbackManager` osztály fogja össze és állítja be, amely a `callbacks` modulban található. Az osztály konstruktora webappot, illetve a nézetmodellt várja. Ezeket nem kell eltárolnunk osztályszintű változókba, csupán az eseményeket becsomagoló függvényekhez használjuk fel őket. Ezeket a függvényeket a `setup_callbacks` függvény hívja meg, melyet a konstruktorban rögtön használunk. A valós idejű monitorozáshoz temporálisan tárolnunk kell a felhasználó által megadott forrásokat, függvényeket és a hozzájuk tartozó paramétereket. Ezt a `to_trace` osztályszintű változó segítségével valósítjuk meg. Ezt a változót több esemény is módosítja, hozzáadnak, vagy eltávolítanak belőle elemeket a felhasználói utasítások

alapján. A monitorozás során pedig ezt a változót adjuk át a valódi nézetmodellnek, hogy utána a modell feldolgozhassa azt.

A `CallbackManager` osztály önmagában egyetlen eseményt sem tartalmaz. Minden eseményt külön modulokba emeltünk ki, hogy megtalálásuk és karbantartásuk egyszerűbb legyen. Összesen hét ilyen modult valósítunk meg az alkalmazásban.

Az alkalmazások kezelésekor felugró ablak eseményeit az `app_dialog_callbacks` modul szolgáltatja, melyben olyan események találhatóak, mint például egy függvény hozzáadása az alkalmazáshoz, vagy éppen annak törlése.

Az alkalmazás konfigurációját beállító menühöz tartozó események a `configure_callbacks` modulban találhatóak és a beállítások elmentéséért, vagy ellenőrzéséért felelnek.

A függvények kezeléséért felelős dialógusablak eseményeit a `func_dialog_callbacks` modulban találhatjuk, melyek a paraméterek hozzáadását és eltávolítását valósítják meg.

A gráf ábrájának frissítéséért és az információs panelek megjelenítéséért a `graph_callbacks` modulban található események felelnek. A `display_info_card` függvény által becsomagolt `update_node_info` esemény jeleníti meg a megfelelő paneleket a vezérlőpult alsó felén. Itt láthatjuk, hogy az esemény a paneleket mindig újra létrehozza a `NodeInfoCard` és a `EdgeInfoCard` osztályok megfelelő felparaméterezésével és példányosításával. Ez egy különleges eset, amikor nem csupán egy létező elem valamely tulajdonságát változtatjuk meg, hanem tulajdonképpen egy újabb elemmel bővítjük a nézetet.

Az eseménykezelés egy másik különlegesebb módja is itt található, melyet a két másik esemény valósít meg. Az `update_elements` esemény a gráf elemeit, az `update_style` pedig a gráf színezését állítja be. Élő monitorozás közben a gráf színezése automatikusan, a gráf adatai alapján beállított értékek alapján történik. Azt, hogy az elemek megváltozásakor a színezés is ahhoz mérten változzon, az események láncolásával oldjuk meg. Az `update_style` esemény egyik kiváltó okának a gráf elemeinek változását adjuk meg. Ezáltal, ha a gráf elemei változnak, a gráf színeit újra kiértékeljük és általuk frissítjük a gráf kinézetét. Ehhez a `Graph` osztály statikus `stylesheet` függvényét használjuk.

Azt is érdemes megemlítenünk, hogy a gráf elemeit nem csupán felhasználói akció által kiváltott esemény tudja megváltoztatni. Az élő monitorozás megvalósításához a

keretrendszer `Interval` osztályát használjuk, amely bizonyos időközönként aktiválódik. Ezáltal automatikusan, felhasználói beavatkozás nélkül tudjuk frissíteni a gráfot. Az Interval osztály azért is elengedhetetlen az élő monitorozáshoz, mert amint azt már többször megfogalmaztuk, minden esemény bemenetének egy nézetben található elem tulajdonságának megváltozása kell. Sajnos, sem a modell, sem a nézetmodell nem ilyen elem, tehát nem tudnak eseményeket aktiválni. Ezért ezzel az időzítő osztállyal, mintavételezéssel oldjuk meg a modellben végbement változások felügyeletét.

A `realtime_callbacks` modulban találhatjuk az élő monitorozáshoz használt vezérlőpult eseményeit. Az itt található eseményekkel vezérelhetjük a monitorozást, vagyis így az ahhoz használt időzítő indítását és leállítását.

A `static_callbacks` modulban a trace kimenetek statikus feldolgozását kezelő eseményt találjuk. Végül a `utilities_callbacks` eseményei állítják be, hogy az eszköztár elemei használhatók-e, vagy sem. Ez attól függ, hogy létezik-e gráf, amire az eszközöket használhatnánk.

3.4.2. Nézetmodell

A modellel kommunikáló nézetmodellt a `ViewModel` osztály valósítja meg, amelyet a `viewmodel` modul tartalmaz. A nézetmodell és vele együtt az alatta lévő rétegek sem használnak fel modulokat a Dash keretrendszerből. Az osztály főleg getter függvényeket tartalmaz, melyek segítségével beállíthatjuk a nézetet, illetve setter függvényeket, amelyekkel az események vezérlik az alkalmazás logikáját. Az osztály függőség befecskendezéssel, konstruktorában kapja meg a konfigurációt.

Másik adattagja a modell valamilyen implementációja. A nézetmodell induláskor a `BaseModel` osztályt használja, amely nem absztrakt, ugyanakkor csupán alapvető tulajdonságokkal rendelkezik. Ezáltal a gráf és egyes vezérlőpulton található elemek már a futás indulásának pillanatától kezdve kaphatnak kezdeti adatokat, amelyeket felhasználhatnak. Az alkalmazás két használati módjának megfelelően két modellt definiálunk. Az élő monitorozásért felelő `DynamicModel` osztályt, illetve a statikus elemzést végző `StaticModel` osztályt. Mindkét osztály a `BaseModel` leszármazottja és azt további egyedi funkcionalitással bővítik ki. A futás során a nézetmodell dinamikusan hozza létre az alkalmazás modelljét, attól függően, hogy éppen melyikre van szükség.

A nézetmodellben található getter függvények segítségével kapja meg a gráf a pontjait és az éleit is, mely adatokat a nézetmodell a Dash keretrendszer által előírt formátumba konvertálja. A nézetmodell szolgáltatja a megfelelő színeket a gráf színezési

szabályaihoz és az információs panelek feltöltéséhez szükséges adatokat is. Egyes getter függvények csak bizonyos modellek esetén értelmezhetőek. Mivel a Python nem erősen típusos nyelv, ezért megengedi ilyen függvények definiálását. Feltehetjük, hogy a program futása során soha nem áll elő olyan helyzet, amikor egy olyan függvényt hívunk a modellen, amelyet az nem definiál, így ezeket a függvényeket ugyanúgy definiáljuk, mint bármelyik másikat. Tipikusan ilyen függvények az élő monitorozás státuszára vonatkozó lekérdezések, melyeket csak élő monitorozás közben használunk fel.

A felhasználó a vezérlőpulton irányíthatja a nézetmodellt. Ehhez megfelelően setter, illetve a modellt inicializáló és kezelő függvények találhatóak az osztályban. Többek között ezen függvények felelnek különböző konfigurációk elmentéséért és érvényesítéséért, vagy modellek indításáért és leállításáért, amelyekhez az eseménykezelő szolgáltat adatokat.

3.5. Nézet

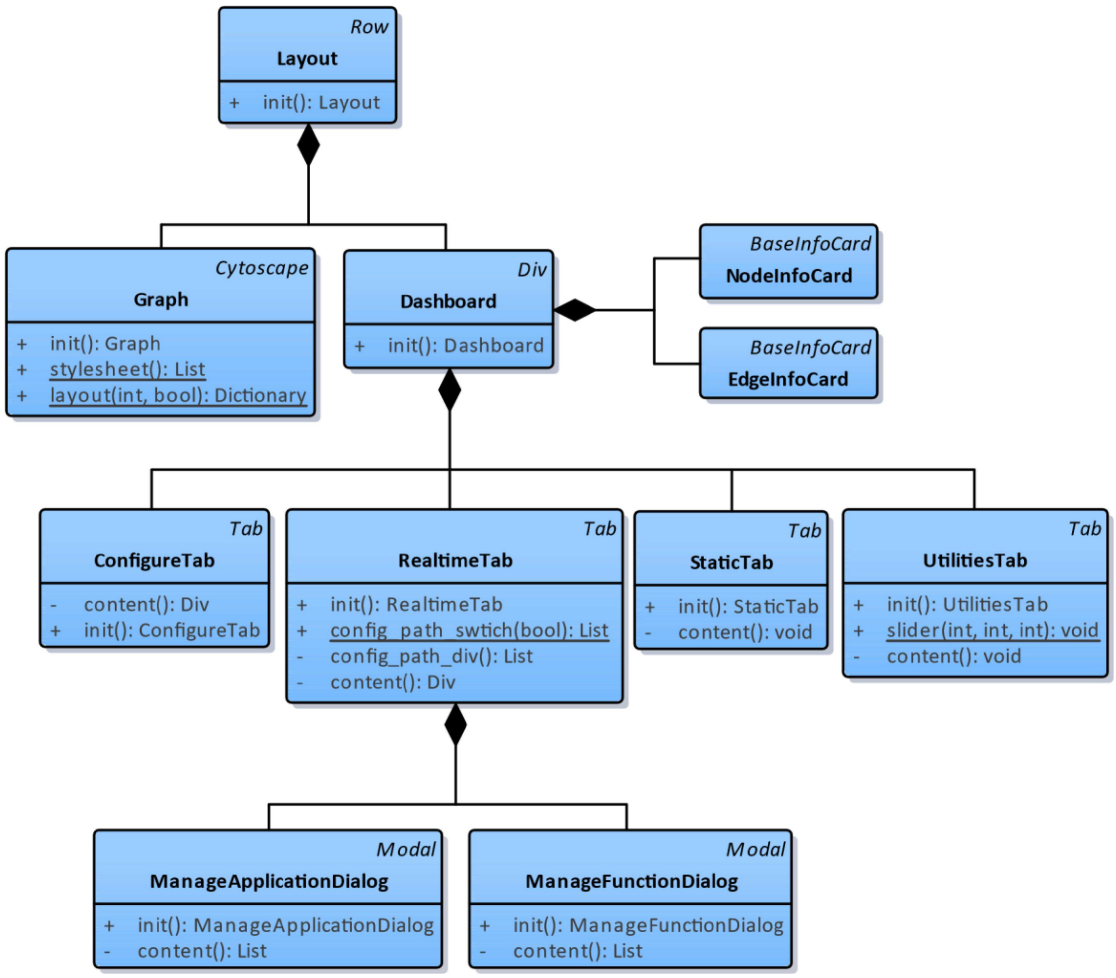
A nézet réteg felel az alkalmazás minden, megjelenítéssel kapcsolatos komponenséért. A nézethez tartozó minden modul egy önálló komponenst vagy komponenseket tartalmaz, melyek modul szinten újrahasznosíthatók és cserélhetők. Az alkalmazás megjelenéséhez a Bootstrap sémát [7] használjuk, így a legtöbb vizuális elem a `dash_bootstrap_components` csomagból származik.

A nézet minden osztályát a Dash keretrendszer valamelyik osztályából származtatjuk, így valamilyen HTML, vagy JavaScript komponensnek felelnek meg. A nézet minden eleme egy egyedi azonosítóval rendelkezik, ami által az eseménykezelő hivatkozni tud rá. Ebből adódóan elméletileg minden nézetben szereplő elem, amelyre az eseménykezelő hivatkozik, *Singleton*. Egyes osztályok tartalmaznak folyamatosan frissülő elemeket, melyek az adatokat az eseménykezelőből kaphatják. Mivel az eseménykezelő a nézet elemeire csupán azonosítójukkal hivatkozik, így a nézet egyetlen példányosított objektumát sem mentjük el változóba. Csupán a legfelső réteg egy példányát átadjuk a webappnak.

Az olyan elemeket, melyek beállításait nem tudjuk egyszerűen attribútumaikon kezelni, külön statikus függvénybe emeljük ki. Ezen függvények teljesen új elemeket tudnak előállítani, melyek tulajdonságait paramétereik szerint állítják be. Fontos, hogy minden ilyen függvényhez tartozó paraméternek legyen alapértelmezett értéke, így üresen, az elem első használata előtt is legyárthatjuk azt, hogy ismert legyen a webapp indításakor. Természetesen az ilyen módon előállított elemek esetén felmerülhet, hogy

így egyszerre több elem is létezhet megegyező azonosítóval egy időpontban a futás során, vagy esetleg egy sem. Emiatt ezeket a függvényeket fokozott elővigyázatossággal használjuk.

A nézetet a `Layout` osztály fogja össze, amely a gyökere a nézet osztályainak függőségei által alkotott fának. Az alábbi diagramon a nézet állandó elemeinek osztálydiagramját láthatjuk, amelyen nem tüntetjük fel a hibaüzeneteket és a stílusokat.



13. ábra: A nézet réteg osztálydiagramja

3.5.1. Stílusok

Lehetőségünk van arra, hogy módosítsuk a Dash keretrendszer által létrehozott elemek kinézetét. Ezt megtehetjük a CSS file-ban, vagy megtehetjük rögtön az elem implementálásakor a `style` attribútumának beállításával. Mivel a második lehetőség egyértelműbb és átláthatóbb kódot eredményez, így a lehető legtöbb vizuális finomhangolást ezzel a módszerrel végezzük el. Ezen kívül így több különböző elemnek is adhatunk ugyanolyan kinézeti beállításokat, amikkel megelőzzük a kódismétlést.

Az összes ilyen stílusbeállítást a `styles` modul tartalmazza. A modulban csak függvények találhatóak, melyek mindegyike egy `dictionary` típusú objektumot ad vissza. A modulban találhatóak egyszerű komponensekhez tartozó stílusok, de többségében a gráf elemeire vonatkozó kinézeti beállításokat tároljuk itt.

A gráfra vonatkozó stílusok a Cytoscape.js [8] könyvtár stíluskonvencióját használják. Eszerint minden stílus tartalmaz egy `selector` kulcsot, amely meghatározza, hogy mely elemekre vonatkozik az adott stílus. A második kulcs a `style`, amelyhez pedig a kinézet leírását megadó `dictionary` tartozik. A gráfhoz tartozó stílusokat két részre bonthatjuk. Az egyik csoport a gráf pontjaira vonatkozik, míg másik az éleire. Mindkét csoportban megtalálható az alap stílus, amely minden pontra vagy élre vonatkozik. Ezeket egészítik ki, vagy írják felül a további stílusok, melyek a gráfelemek különböző színezését írják le. Így mind a két csoportban megtalálhatók a zöld, sárga, illetve piros színű elemekre vonatkozó stílusbeállítások.

3.5.2. Értesítések

Az `alerts` modul tartalmazza a felületen megjelenített összes értesítést. Minden értesítés valójában az `Alert` osztály egy megvalósítása. Alapvetően két típusú értesítést különböztetünk meg, hibaüzeneteket (`ErrorAlert`) és sikeres akciók utáni értesítéseket (`SuccessAlert`). A kettő között csupán a színük különbözik, más tulajdonságaik mind megegyeznek, így mindkét sémát egy alapsémából (`BaseAlert`) származtatjuk, amely pedig az `Alert` osztályból származik le. Egy kivételével minden értesítés a két típus valamelyikébe tartozik.

A monitorozás több hiba miatt is sikertelen lehet. Emiatt az ehhez tartozó értesítés (`trace_error_alert`) üzenete változhat, ami miatt annak értelmezésére hosszabb időt vehet igénybe. Így az ehhez tartozó hibaüzenetet egyedi módon jelenítjük meg.

Magukat az értesítéseket a program futása során sohasem változtatjuk. Csupán a megjelenítésükre megszabott elem `children` tulajdonságát állítjuk át egy újonnan létrehozott példányra. Ebből adódóan egyik értesítés példánynak sem adunk azonosítót, hiszen nem kell az egyedekre ilyen módon hivatkoznunk.

3.5.3. Dialógus ablakok

Az alkalmazás két dialógus ablakot tartalmaz, melyek megvalósítását a `dialogs` modul tartalmazza. Az egyik egy adott alkalmazás kezelésére ad lehetőséget (`ManageApplicationDialog`), míg a másikkal egy adott függvényt módosíthatunk (`ManageFunctionDialog`). Mindkét ablak a `Modal` osztály egy-egy megvalósítása.

Az alkalmazás futása során akármennyi különböző forrást és függvényt is kezelnünk kellhet ilyen ablakokkal. Ugyanakkor ezen ablakok tartalmára szükségünk is van, hogy a bennük eltárolt adatokat elmenthessük és felhasználhassuk. Ezért minden ilyen ablaknak egyedi azonosítót kellene adnunk, ráadásul a webapp indításakor tudnunk kellene, hogy pontosan hány ilyen ablakra lesz szükség. Ez természetesen lehetetlen, ezért valójában a futás teljes időtartama alatt ugyanaz a két ablak szolgál ki minket, csupán dinamikusan változtatjuk a tartalmukat. Az ablakok teljes tartalmát az osztályok `content` függvénye tartalmazza, amely az elemek listájával tér vissza. Az elemek közül az ablak fejlécét és a felkínált függvények, vagy paraméterek listáját kell folyamatosan frissíteni, amit megtehetünk csupán az adott elemek azonosítójával és megfelelő attribútumuk beállításával. Így a `content` függvény csupán a kód átláthatóságát szolgálja, nem olyan statikus függvény, amelyet az eseményvezérlés felhasznál.

3.5.4. Valós idejű monitorozás

A valós idejű monitorozáshoz kialakított vezérlőpult implementációját a `realtime_tab` modulban találhatjuk. Az itt definiált `RealtimeTab` osztály tartalmazza a monitorozás kezeléséhez szükséges beviteli mezőket. Az osztály a Tab osztályból származik, így a vezérlőpulton megjelenítve kinyitható menüként funkcionál a három másik hasonló funkcióval együtt.

A dialógus ablakokhoz hasonlóan, jelen esetben is a content függvény szolgáltatja az elem tartalmát. Ebben vannak definiálva a különböző egyszerű beviteli elemek, de mivel a futás elejétől ismernünk kell a felület minden elemét, így itt kerül példányosításra a monitorozás beállításához nélkülözhetetlen két dialógus ablak is. A gráf elemeinek valós idejű frissítéséhez rendkívül fontos az `Interval` típusú változó definiálása, amely segítségével időközönként frissíteni tudjuk a felület elemeit. Így bár ez technikailag nem vizuális elem, mégis itt kerül létrehozásra.

A `content` függvény nem vár paramétereket melyek megváltoztathatják és az eseménykezelő nem is hivatkozik rá. A függvény által definiált elemek között ugyanakkor több olyan van, amely a futás során változhat. Ezek közül a konfigurációs fájl használatához megjelenített kapcsoló letiltása sajnos nem ugyanúgy működik, mint más elemek tulajdonságainak a beállítása. Így az egész kapcsolót ki kell emelnünk egy statikus függvénybe (`config_path_swtich`), amivel a kapcsolóból két verziót tudunk létrehozni és az adott események során az egyiket, vagy másikat beállítani a vezérlőpulton.

3.5.5. Statikus kimenet feldolgozás

A vezérlőpult tetején a második fülre kattintva érhetjük el a statikus kimenetek feldologázására létrehozott lapot. Az ezt megvalósító osztály a `StaticTab`, amelyet a `static_tab` modulban találunk. Az osztály, akárcsak a `RealtimeTab`, szintén a Tab osztályból származik, így ugyanúgy is jelenik meg. Az osztály minden elemét tudjuk vezérelni csupán attribútumaik segítségével, így az osztálynak egyetlen kiemelt statikus függvényre sincsen szüksége.

3.5.6. Eszköztár

A kirajzolt gráf színezésének módosításához a vezérlőpult harmadik menüjéről elérhető mezőket használjuk. Ezen két beviteli mezők egyike a hívások száma alapján a színezést beállító csúszka, másik a keresőmező, amellyel függvények nevére szűrhetünk. Mindkettő tulajdonságai változnak a futás során, de a csúszkának szinte minden attribútumát külső adatok állítják be. Emiatt ezt az elemet statikus függvénybe (`slider`) emeljük ki és folyamatosan újat gyártunk belőle.

A pult többi elemét kezelni tudjuk csupán attribútumaik segítségével, így azok a `content` függvényben definiálódnak. Az egész eszköztárat összefoglaló osztály a `UtilitiesTab`, amely a Tab osztály leszármazottja és a `utilities_tab` modulban található.

3.5.7. Beállítások

Az utolsó menüpont a beállítások kezelésére kialakított laphoz tartozik. Ezt a komponenst a `ConfigureTab` osztály definiálja, amely a `configure_tab` modulban található. Az osztály a Tab osztály leszármazottja és csupán egy `content` függvényt tartalmaz, amely meghatározza a beállítások kezeléséhez szükséges elemeket.

3.5.8. Információs panelek

A kirajzolt gráf elemeire kattintva különböző információs panelek jelennek meg dinamikusan a vezérlőpult alsó felén. A panelek definíciói az `info_cards` modulban találhatók. Attól függően, hogy a gráf egy pontjára, vagy élére kattintottunk, más-más információkat kapunk az adott elemről. Emiatt mindkét fajta elemhez külön osztályt készítünk, amelyek különböző módon állítják elő az információs panel tartalmát. A pontok információs paneljét a `NodeInfoCard`, az élekét pedig az `EdgeInfoCard` osztályok valósítják meg. Mivel a panelfajták csupán a tartalmukban különböznek, így mindkettőt egy közös ősre, a `BaseInfoCard` osztályra vezetjük vissza. Az ősosztály pedig a `Card` osztály leszármazottja, amely meghatározza vizuális karakterisztikáit.

Az információs paneleket a futás során dinamikusan hozzuk létre. Ez azt jelenti, hogy minden alkalommal, mikor egy elemre kattintunk, egy újabb panelt hozunk létre. Ha még egy elemre sem kattintottunk, akkor természetesen nem jelenik meg semmilyen panel a vezérlőpult alján. Ilyenkor valójában semmilyen panel nem is létezik a felületen. Ez azt jelenti, hogy a webapp indulása előtt az alkalmazás nem tud a panelek létezéséről és azonosítójukat sem ismeri. Ez alapvetően csak akkor lenne gond, ha a panelekkel további interakciós lehetőségeink lennének. Viszont szerencsére a panelekhez nem kell eseményeket hozzárendelnünk, így azonosítóikra sosem kell hivatkoznunk. Ezért a paneleket azonosító nélkül hozhatjuk létre, bármi akadály nélkül.

3.5.9. Vezérlőpult

A vezérlőpultot a `Dashboard` osztály valósítja meg, amelyet a `dashboard` modul tartalmaz és a `Div` osztályból származtatjuk. A vezérlőpulton található a négy lapból álló lapozófelület, amelyeket a `Tabs` objektummal valósítunk meg. Alatta helyezzük el az információs paneleknek kialakított részt, ami egy egyszerű `Div` objektum, amelynek `children` attribútumát üresen hagyjuk, hiszen induláskor egyetlen panel sem létezik. Futás során ezt az elemet használjuk a különböző panelek megjelenítésére.

3.5.10. Gráf

A felhasználói felületen, a vezérlőpulttól balra helyezkedik el a hívási láncokat reprezentáló interaktív gráf. Ahogy a felhasználói felület többi elemét, úgy ezt az ábrát is a Dash keretrendszer segítségével valósítjuk meg. Ugyanakkor a gráf nem olyan egyszerű komponens, melyet a keretrendszer HTML, vagy Bootstrap moduljaiban megtalálunk. A keretrendszer külön megoldást nyújt kifejezetten gráfok létrehozására. A gráfot, a Dash kifejezetten interaktív gráfok alkotására kialakított `dash_cytoscape` [9] moduljával valósítjuk meg. A `graph` modulban található `Graph` osztály implementálja az alkalmazásban megjelenített gráfot. Ez az osztály a `Cytoscape` osztályból származik, így ugyanúgy, ahogy a többi nézetben létrehozott osztály, a vizuális elemet explicit valósítja meg.

Az alkalmazás futása során a gráfnak három attribútuma változik. A gráf pontjait és éleit az `elements` attribútum listaként kezeli, amelyet az eseményvezérlő dinamikusan tölt fel a megfelelő elemekkel a nézetmodellből kinyert információkkal. Természeten inicializáláskor még egyetlen eleme sincs a gráfnak. A gráf másik két attribútumát a külön statikus függvényekbe szervezzük ki.

A gráf elrendezéséért a `layout` attribútum felel. A Dash keretrendszer több lehetőséget is kínál a gráf pontjainak automatikus elrendezésére. Sajnos ezen sémák közül egyik sem felel meg az igényeinknek. Lehetőségünk van azonban harmadik forrásból származó elrendezési sémákat is használni, melyeket a konstruktorban elhelyezett `load_extra_layouts` függvénnyel importálhatunk be. Ezek közül a gráf a `dagre` elrendezést használja, ami jól kezeli az irányított gráfokat, amelyeknél a pontokat egy sorrendi logika szerint kell elrendezni. Ez az elrendezés biztosítja azt, hogy a hívási láncok elején található függvények feljebb láthatók és ahogy haladunk a hívási láncon, úgy a függvények egyre lejjebb helyezkednek el az ábrán. A gráf elrendezéséhez az alapsémán kívül hozzátartozik a pontok közötti távolság, illetve, hogy a gráf változása esetén az ábra változása animált, folytonos mozgással történjen-e. Ezeket a tulajdonságokat a felhasználó kézileg is beállíthatja. Mivel a gráf elrendezése egy dictionary típusú objektum, amelyben minden erre vonatkozó beállítás egyszerre szerepel, így ezeket csak egyszerre tudjuk módosítani. Így az elrendezésért a `layout` statikus függvény felel, amelynek paraméterei segítségével állíthatjuk be a pontok távolságát és az animálást.

A gráf pontjai és élei különböző színeket vehetnek fel, melyek az adott elemre vonatkozó hívások számától függenek. Ezeket a színeket a gráf `stylesheet` attribútuma egy listában fogadja, amelyben a `styles` modulban létrehozott speciális stílusok szerepelnek. Ezeket a stílusokat a `stylesheet` statikus függvény paramétereivel tudjuk személyre szabni.

3.5.11. Alaprajz

A `Layout` osztály felel a felület alapvető elrendezéséért, amely a `layout` modulban foglal helyet. Az osztály a `Row` osztályból származik. Segítségével speciális `Col` objektumokat tudunk vízszintesen egymás mellé helyezni. Egy-egy ilyen `Col` objektumba csomagoljuk a `Graph` és a `Dashboard` egy-egy példányát. Azzal, hogy a konstruktorban példányosítjuk ezt a két objektumot, a nézet felépítése miatt valójában az összes vizuális komponenst is létrehozzuk, amelyeknek azonosítóját induláskor ismernünk kell. Így az egész nézet csupán ezen osztály példányosításával létrehozásra kerül és használható.

3.6. Környezet

Az alkalmazás indításáért, illetve egyes rétegek példányosításáért a környezet réteg felel. Ebben a rétegben nem végzünk számításokat, csupán az alkalmazás rendszerének felállításához szükséges lépéseket tesszük meg. Így használat közben ez a réteg már explicit nem felel semmiért.

3.6.1. Belépési pont

Az alkalmazást a `main` nevű futtatható állomány segítségével indíthatjuk el. Indítás után példányosítjuk az alkalmazást, majd elindítjuk azt. Mivel maga az alkalmazás nem a belépési pont része, hanem különálló entitásként kezeljük, így a script könnyen bővíthető több alkalmazás futtatásának támogatására is.

3.6.2. Alkalmazás

Az alkalmazás legfelső rendszerező és indító osztálya az `App` osztály. Az osztály konstruktorában példányosítja az alkalmazás konfigurációját, nézetmodellét és nézetét, illetve a felülethez tartozó webappot, melyek között a kapcsolatokat is felépíti.

Az alkalmazás konfigurációja (`Configuration`) a futás egésze során perzisztens és minden rétegnek szüksége van rá valamilyen formában. A nézetmodell (`ViewModel`) a konstruktorában kapja meg a konfigurációt. Mindkét osztályból létrehozott egyed azonnal használható alapszinten a példányosítás után, ugyanúgy, ahogy a nézet (`Layout`) is. Ezek után már előállíthatjuk a webes szolgáltatásokért felelős webappot.

A webapp a `Dash` osztály egy példánya, amelyet a Dash keretrendszer szolgáltat. Az osztály önmagában tartalmazza és így elrejti a webes felület felállításához szükséges logikát, ebből következően elegánsan és tömören használhatjuk. A modern és átlátható megjelenés érdekében beállítjuk a vizuális komponensek egységes sémáját. A webszerver indítása előtt beállítjuk a nézetet is, melyet a teljes megjelenítésért felelős osztály (`Layout`) egy egyedének átadásával teszünk meg. Ez azért fontos, mert indításkor a webappnak ismernie kell a felület minden interaktív elemének azonosítóját. Utolsó lépésként beállítjuk az alkalmazás címét, amelyet megjelenít böngésző.

A konstruktorban az előzőleg felsorolt adattagokon kívül létrehozzuk az eseményvezérlőt is. A felülettel való interakció lehetőségeit a webapp indítása előtt kell beállítanunk. Az eseményekért felelős adattag példányosításához az osztály (`CallbackManager`) konstruktorának szüksége van a webappra, amihez beállítjuk az eseményeket, és a nézetmodellre amire az adott akciók hatással lehetnek, illetve amiből

a nézet számára adatokat adhatunk vissza. Az eseménykezelőnek nincsen közvetlenül szüksége a nézetre, mivel valójában a nézet komponenseihez tartozó egyedi azonosítókra hivatkozva kezeli őket, ezért nem hivatkozik magukra a példányokra.

Az osztály `start` metódusa indítja el a szervert és teszi elérhetővé a webes felhasználói felületet. Mivel a keretrendszer által beépített webszervert használjuk, így be kell állítanunk, hogy a klienssel folytatott RestAPI kérések ne jelenjenek meg a futás során a parancssorban. Ezt az indítás egy nevesített paraméterével érhetjük el.

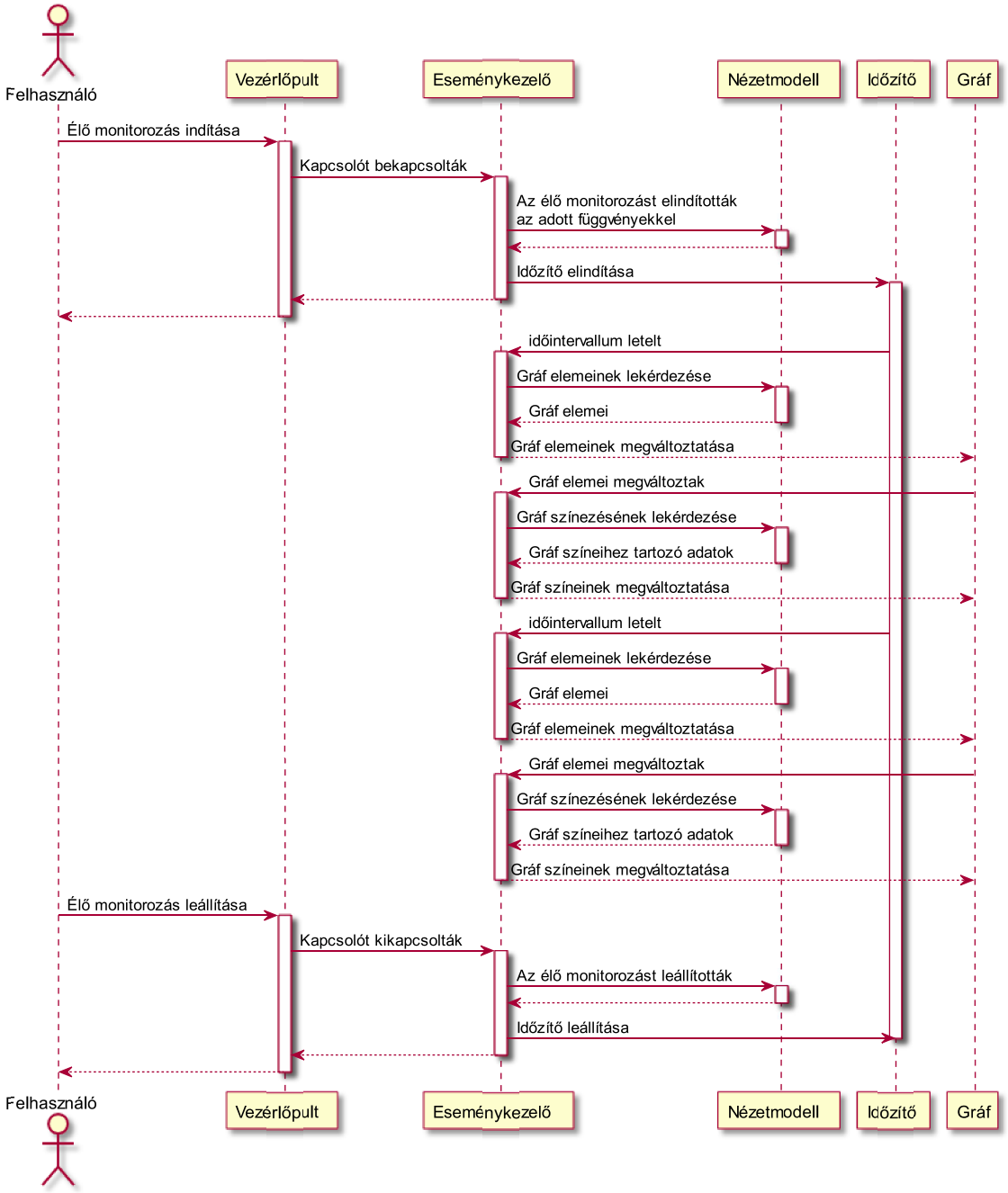
3.7. Gyakorlati példák

Az alkalmazás futása során több komponens kommunikál egymással aszinkron módon. Azért, hogy jobban megérthessük a háttérben húzódó logikát, az alábbi szekvencia diagram gyűjtemény nyújt segítséget. Az alábbi ábrák az élő monitorozást mutatják be absztrakt módon. Az ábrákon a könnyebb érthetőség érdekében nem a kódban szereplő osztálynevekkel jelöljük az egyes komponenseket. Helyettük a komponensek feladatát magyarul leíró nevekkel dolgozunk. Ugyanígy a komponensek közötti kommunikációt sem egyszerű függvénynevekkel írjuk le, hanem a hívásokat és üzeneteket logikusan leíró szövegekkel.

Az tizennegyedik ábra részletesen bemutatja hogyan frissül az alkalmazás felületén megjelenített gráf. Azért, hogy a diagram átlátható maradjon, csupán a nézetmodellig mutatja be a rétegeket. A bemutatott eseményeknél úgy tekintjük, hogy a felhasználó a felületen állítja össze a megfigyelni kívánt elemeket. Ugyanakkor ez csupán az első eseménykezelő és nézetmodell közötti kommunikációt befolyásolja. Konfigurációs fájl esetén ekkor az eseménykezelő nem rögtön a függvényeket adná tovább a nézetmodellnek, hanem a konfigurációs fájl elérési útvonalát. A megfigyelni kívánt elemek kiválasztása nincs feltüntetve a diagramon, úgy tekintjük, hogy a felhasználó azt a munkát már helyesen elvégezte és semmi sem hiányzik a monitorozás megindításához.

A diagramon láthatjuk, hogy a gráf folyamatos frissítését a nézetben található időzítő végzi bizonyos időközönként. Az időzítő aktiválja az eseménykezelő megfelelő *callback*-jét, amely feltölti a gráfot a nézetmodellből kinyert elemekkel. A gráf elemeinek frissülése újra aktivál egy eseményt, amely ezúttal a gráf színezéséhez szükséges adatokat kéri le és általuk frissíti a gráfot. Ekkor valósul meg az események korábban leírt láncolata. Az időzítő egészen addig aktív marad, amíg a felhasználó le nem állítja a

monitorozást. Ilyenkor az időzítővel együtt a nézetmodell felé is kommunikáljuk a monitorozás leállását és így az ábra által bemutatott logika is véget ér.

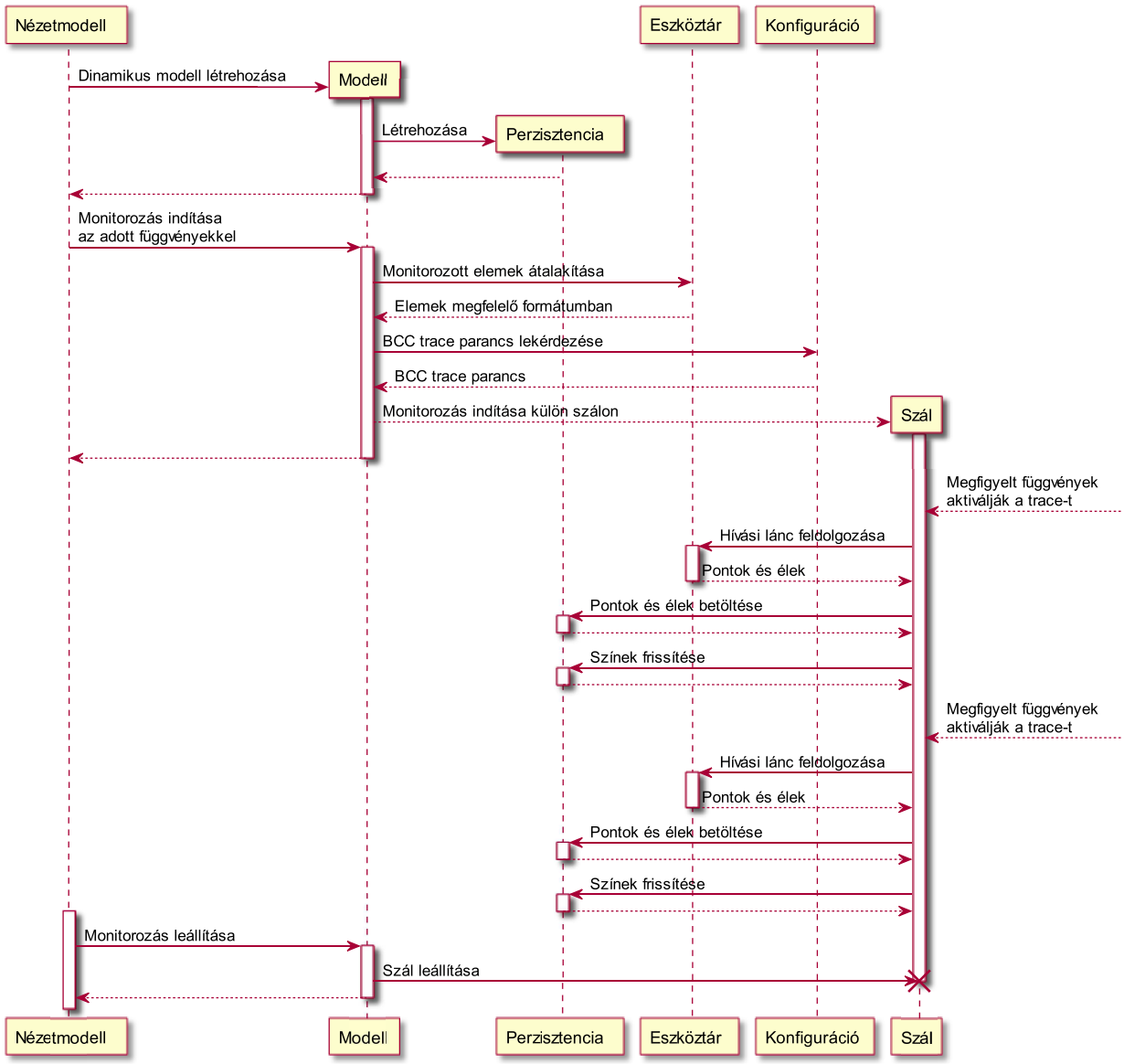


14. ábra: Az alkalmazás frontend részének viselkedése monitorozás közben

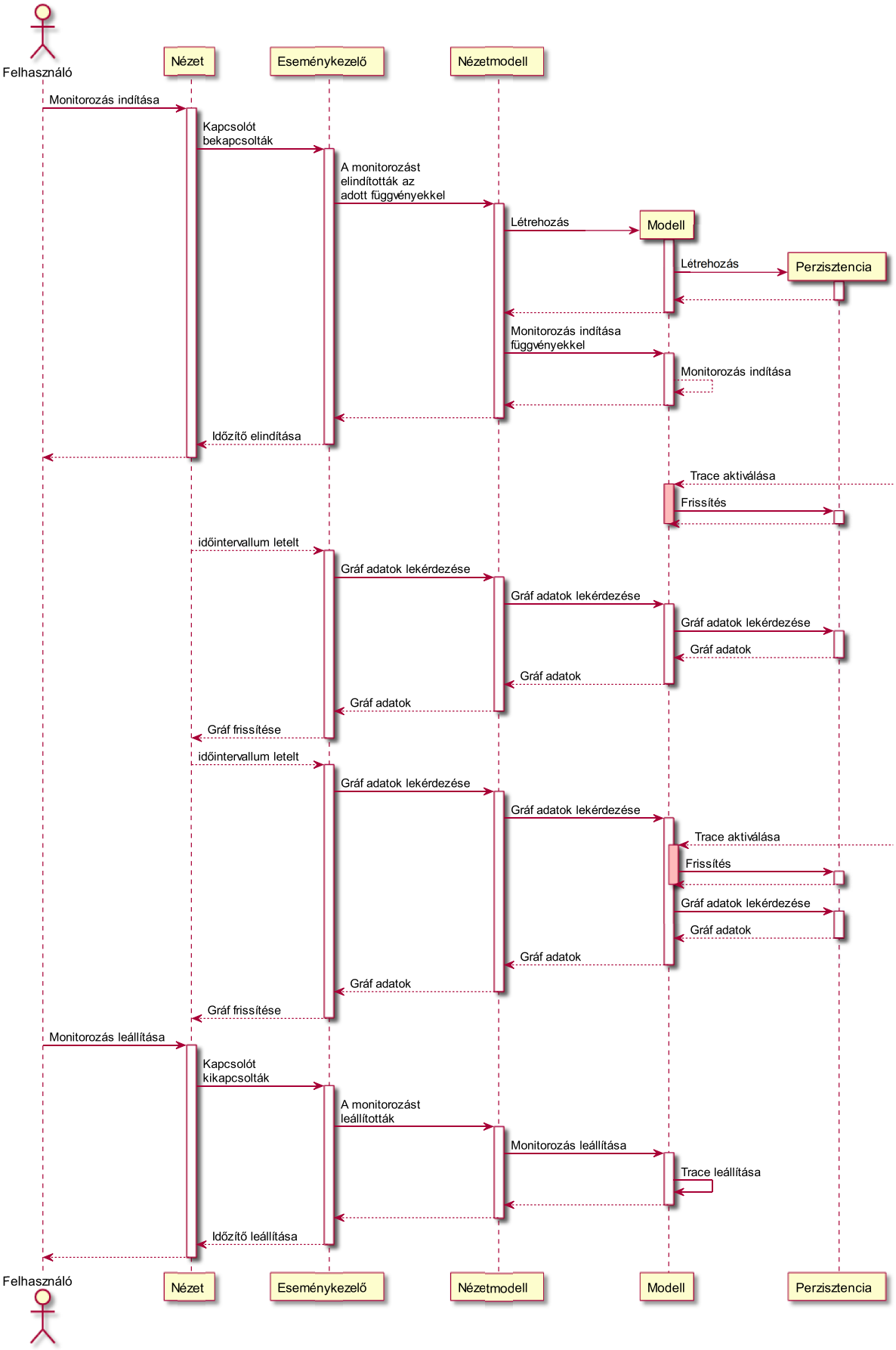
A háttérben egy hasonló folyamat játszódik le a modellben. A tizenötödik ábra a program *backend* részének működését mutatja be a nézetmodellbe beérkező hívástól kezdődően. Az előző ábrához illeszkedően a monitorozást a nézetmodelltől kezdődően követhetjük nyomon, így az abba beérkező hívásokat már nem tüntetjük fel.

Az ábrán láthatjuk, hogy a modell és a perzisztencia létrehozása dinamikusan történik. A megfigyelni kívánt függvények betöltése után a modell létrehozza a trace

modult futtató, párhuzamosan futó szálat. A trace modult a megfigyelt függvények aktiválják, melyekről az információkat a kernelből olvassa ki. Ugyanakkor a trace modul belső működése az alkalmazás szempontjából elhanyagolható. A szál a kiolvasott kimenetet feldolgozza, majd betölti a perzisztenciába. Bár az ábrán ez nincs feltüntetve, de a szálnak nem okoz problémát, ha a trace kimenetének feldolgozása közben újabb hívási láncok érkeznek be. Ha ilyen versenyhelyzet áll elő, a trace kimenete csupán bővül, nem veszik el. Ilyen esetben a szálban futó ciklus következő iterációjában olvassuk ki az azóta beérkezett hívási láncokat. A monitorozás befejezésekor kilépünk a ciklusból és befejezzük a trace futtatását. Ezáltal a szál magától terminál.



15. ábra: Az alkalmazás modelljének viselkedése monitorozás alatt



16. ábra: Az alkalmazás rétegeinek együttműködése monitorozás során

Végül az utolsó, felületesebb diagram egy átfogó bemutatást nyújt arról, hogyan működik az alkalmazás egésze a funkció használata közben. A tizenhatodik ábrán csupán az egyes rétegeket ábrázoljuk külön, a bennük lévő elemeket nem. Így például a nézetbe tartozó vezérlőpultot, gráfot és időzítőt egyetlen életvonalon reprezentáljuk. Ugyanígy a trace futtatására létrejövő párhuzamosan futó szálat is a modell részének tekintjük. Ezen kívül nem tüntetjük fel a monitorozásban csupán segédszerepet játszó konfigurációt és eszköztárat sem, ahogy a velük folytatott kommunikációt sem tüntetjük fel.

A diagramon jól látható a modell és a perzisztencia létrehozása, majd a monitorozás indítása. A szál indulása után aszinkron módon frissíti a perzisztenciát az újonnan elérhető adatokkal. Ezeket az eseményeket külön színnel jelöltük, mivel ekkor a modell külön szálon futó része aktiválódik. Ezzel párhuzamosan tudja a nézet lekérdezni a gráf adatait és azoknak megfelelően frissíteni a megjelenített gráfot.

3.8. Tesztelés

Az alkalmazás minden osztályát és függvényét teszteljük. Ugyanakkor automata tesztekkel csak az alsóbb réteget tudjuk vizsgálni. A Dash keretrendszerhez tartozó komponensek tesztelését manuálisan végezzük és jegyzőkönyvben dokumentáljuk. Az automata tesztelést két részre bontjuk, egység és integrációs tesztekre. Egység tesztek minden tesztelhető függvényre elérhetőek, míg integrációs tesztekkel a komplexebb, több függvényt felhasználó logikákat ellenőrizzük.

Az automata tesztek a `tests` könyvtárban találhatóak. Ezen belül az egységtesztek a `unit`, az integrációs tesztek pedig az `integration` könyvtárban érhetők el. A teszteléshez a pytest modult használjuk, illetve a tesztesetek megfelelő szeparálásának biztosításáért több függvény tesztjeinek esetében *mockoljuk* az általuk felhasznált osztályokat és függvényeket. Mockolásra a Python beépített `unittest.mock` modulját használjuk.

A tesztek futtatására a `run-tests.sh` állományt használjuk. Ez futtatja a teszteket, illetve előkészíti az integrációs tesztek környezetét, majd a tesztelés végeztével lebontja azt. Az egységtesztek futtathatók egyszerűen a pytest paranccsal akár Windows rendszereken is. Az integrációs tesztek esetében ugyanakkor felhasználjuk a BCC trace eszközt, így ezek a tesztek csak Linux rendszereken fognak lefutni. Az eszköz futtatása miatt ezeket a teszteket *superuser* jogosultságokkal kell futtatnunk, amit szintén ellenőriz a tesztek helyes futtatására létrehozott bash szkript. Az integrációs tesztek feltételezik,

hogy a program gyökérkönyvtárából indítják őket. A szkript egy C programkód fordítását is végzi, így szükséges hozzá, hogy számítógépünkre fel legyen telepítve a GCC fordító.

3.8.1. Egységtesztek

Az egységtesztek esetében további könyvtárakat találunk, melyek rétegek szerint csoportosítják a teszteket. Minden teszt neve az általa tesztelt modul neve `test_` prefixszel ellátva. Egységtesztekkel teszteljük a nézetmodell réteg klasszikus nézetmodell osztályát, illetve a teljes modell- és perzisztencia réteget. Összesen 122 darab egységteszt érhető el ezekre a komponensekre a `unit` könyvtárban, amelyek lefedik a kód teljes tartalmát. A legtöbb függvényre számos teszt érhető el, melyek helyes bemenő adatokkal és *edge-case* esetekkel is ellenőrzik a függvények helyes működését.

A `persistence` könyvtárban találhatók a konfigurációra vonatkozó tesztek, a `test_configuration` állományban. Az itt található tesztek ellenőrzik a konfiguráció alapértelmezett értékeit és a benne található értékeket azok beállítása után. Ezen kívül teszteljük a konfigurációban található adatok lekérdezésére vonatkozó függvényeket is.

A konfiguráció tesztjei mellett találhatjuk meg a perzisztenciára vonatkozó teszteket is a `test_persistence` állományban. Ellenőrizzük a perzisztenciában található alapértelmezett adatokat, majd azok feltöltését különböző pontok és élek kombinációjával. Azért, hogy minden esetet lefedhessünk, az adatok betöltése sokszor nem üres perzisztenciába történik, hanem abba mesterségesen állítunk be elemeket előzetesen. Ez azért fontos, hogy tesztelhessük a perzisztencia helyes működését, vagyis azt, hogy ha a betölteni kívánt adatok már létező elemekhez tartoznak, akkor azon elemeket bővíti ki az új adatokkal. Továbbá teszteljük a konfiguráció *setter* és *getter* függvényeit a perzisztencia többféle előzetes beállítása alapján.

A modell rétegben mind a három modell osztályt külön teszteljük és az eszköztár függvényei is egy külön fájlt kapnak. Azért, hogy függetlenek maradjunk a perzisztencia rétegtől, az abban található osztályokat minden esetben mock osztályokkal helyettesítjük, amikor arra szükség van. Ez azt jelenti, hogy a modell működésének ellenőrzése esetén a perzisztencia rétegben található osztályok függvényeinek csupán a paraméterezését vizsgáljuk, hiszen csak ezek függenek a modelltől. Abban az esetben, ha az ilyen függvényének visszatérési értéke is számít, akkor a visszatérési értéket is mockoljuk valamilyen `dummy` értékkel, majd ezen érték alapján tesztelünk.

A `test_base_model` állomány tartalmazza az ős modellre vonatkozó teszteseteket. Az ős modellben minden függvény a perzisztenciára, vagy a konfigurációra

vonatkozó valamilyen beállító, vagy lekérdező függvény. A beállító függvényeket a mockolt függvények megfelelő paraméterezésének vizsgálatával, a lekérdező függvényeket pedig a mockolt függvények előre beállított visszatérési értékei alapján teszteljük.

A `test_static_model` tartalmazza a statikus modell tesztjeit. Ez a modell nem sokban bővíti ki az ős modell funkcionalitását. Természetesen azokat a függvényeket, amelyeket az ős modellben már teszteltünk, nem fogjuk erre a modellre újra letesztelni, mivel azok működése nem változik. Az inicializálás utáni állapoton kívül csak a szöveg betöltését végző függvény működését ellenőrizzük, mivel ez az egyetlen dolog, amit a statikus modell hozzáad az ős modellhez függvények szempontjából. Itt hasonlóan az ős modell osztály tesztjeihez, csupán a perzisztencia függvényeinek paramétereit vizsgáljuk. A szöveg feldolgozásához azonban az eszköztár egyes függvényeire is szükségünk van, így az abszolút önállóság érdekében ezen függvényeket is mock verziókkal helyettesítjük.

A dinamikus monitorozás osztályát a `test_dynamic_model` Python szkript tartalmazza. A többi tesztben említett általános dolgokat itt is teszteljük, úgy, mint az inicializálás, vagy a lekérdező és beállító függvények. Ugyanakkor a dinamikus modell esetén speciális figyelmet kell fordítanunk az élő monitorozás helyes elemi működésének tesztelésébe. Azért, hogy ennek a bonyolult logikának az egyes komponenseit külön-külön tudjuk tesztelni, függetlenül a futtatási környezettől, a felhasznált pexpect modul osztályait, függvényeit és kivételeit minden esetben mock objektumokkal helyettesítjük. Ezután ezen objektumok paraméterezését teszteljük, illetve előre beállított visszatérési értékük feldolgozását a tesztelt függvények által.

Ugyanígy mockoljuk a külön szál létrehozását is, illetve annak belső működését is. A dinamikus modell úgy lett kialakítva, hogy a szálban található ciklus is könnyen tesztelhető, mivel a ciklusmag teljes egészében ki lett emelve egy külön függvénybe. Így a szál olyan egységekre van bontva, amelyek egymástól teljesen függetlenül is működőképesek, így működésük önállóan ellenőrizhető. A tesztek során nem ellenőrizzük a gráf elemeinek adott hash értékeket, mivel azok minden futás alkalmával eltérhetnek. Ahol erre mégis szükség lenne, ott dummy hash értékeket használunk.

A modell réteghez tartozik az eszköztár is. Így a három modellre írt tesztállomány mellett található az eszköztár függvényeire vonatkozó teszteseteket tartalmazó `test_utils.py` fájl is. A modul egységtesztelése sokkal egyszerűbb a modellekéhez képest, mivel az eszköztár már eredetileg is teljesen önálló függvényekből áll. Ezáltal egységtesztelésük során nem kell külön energiát befektetni az elkülönítésükbe.

Egyetlen teszteset sem igényel mockolást. A függvények tesztelése ezáltal egyértelmű, bizonyos bemenetek esetén teszteljük a függvény kimenetét.

A legmagasabb szintű réteg, amelyet egységtesztekkel tesztelünk, a nézetmodell. Teszteseteit a `viewmodel` könyvtár `test_viewmodel` állományában találhatjuk. A nézetmodell esetén elsősorban a modell rétegtől való függetlenséget kell biztosítanunk, így az ahhoz tartozó osztályokat és függvényeket mockoljuk. A megfelelő helyettesítéseknek köszönhetően a tesztesetek rövidek és kompaktak, hiszen a nézetmodell csupán egy közvetítő réteg a modell és a nézet között, így komponensei mind egyszerű beállító és lekérdező függvények.

3.8.2. Integrációs tesztek

Az integrációs tesztek a logika egészét, a komponensek együttműködését tesztelik. Ezáltal minden teszt a nézetmodellből indul, mivel itt kezdődik a program logikája. A statikus feldolgozás esetén feltöltjük, élő monitorozás esetén beindítjuk a modellt a megfelelő paraméterekkel, majd a tesztesetek végén a modellben keletkezett adatokat vizsgáljuk. Ennek megvalósításához külső erőforrásokat is felhasználunk.

A három tesztesetet tartalmazó három modul az `integration` könyvtárban található. Egy-egy teszt érhető el az élő monitorozás két módjára, a harmadik teszt pedig a statikus feldolgozást ellenőrzi. A mellettük található `resources` könyvtárban találhatóak a tesztekhez szükséges statikus erőforrások.

Egyik teszteset sem tartalmaz mock objektumokat, így a tesztek teljes mértékben az alkalmazás valódi logikáját tesztelik. Minden tesztesetnél ugyanazt a gráfot kell előállítania a modellnek, így az adatok vizsgálata ki van emelve az `asserts` modul `assert_results` függvényébe. A modulban megtalálhatóak az elvárt értékek is.

A dinamikus modellt kétféleképpen indíthatjuk el. Ha a felhasználói felület segítségével állítjuk össze a megfigyelni kívánt függvények kollekcióját, akkor a nézetmodell egy `dictionary` típusú objektumot ad át a modellnek, amely a monitorozni kívánt függvényeket strukturáltan tárolja. Ezt az esetet hivatott tesztelni a `test_dynamic_with_dict` állományban található egyetlen teszteset, amelynek neve röviden `test_dynamic_model_with_trace_dict`.

A teszteset előre megadott függvényeket ad át a modellnek monitorozásra. Ugyanakkor a függvények természetesen valamilyen forráshoz kell, hogy tartozzanak. Azért, hogy a program működését a külvilágtól a lehető legelszigeteltebben tesztelhessük, egy külön, a tesztkörnyezet által létrehozott alkalmazást adunk meg forrásnak.

Az alkalmazás kódját a külső erőforrások között `test_application.c` dokumentumban találhatjuk meg. Ha a teszteket a megadott indító szkripttel futtatjuk, akkor az lefordítja ebből a programkódból az alkalmazást, amely a monitorozás tesztelésekor már rendelkezésre áll.

A monitorozás indítása után megvárjuk, hogy működőképessé váljon a BCC trace eszköz, majd közvetlenül a teszt kódjából indítjuk el a monitorozott alkalmazást. Miután az lefutott, leállítjuk a monitorozást és ellenőrizzük a programban létrejött adathalmazt. Az ellenőrzés során csupán a gráf pontjait és éleit ellenőrizzük, hiszen az azokból következő többi adat helyességét az egységtesztekkel már ellenőriztük.

A dinamikus monitorozást konfigurációs fájl segítségével is elindíthatjuk. Ebben az esetben a konfigurációs fájl tartalmát kell kiegészítenünk az általunk létrehozott tesztalkalmazás elérési útvonalával, majd ezt kiírni egy ideiglenes konfigurációs fájlba. Ezek után elindítjuk a monitorozást. Ettől a ponttól kezdve a kettő élő monitorozást tesztelő teszteset működése azonos.

Végül a harmadik teszteset a modell statikus kimenet feldolgozását ellenőrzi. Ezt a `test_static` állomány `test_static_model` tesztesete implementálja. A feldolgozandó statikus kimenetet a `test_static_output` nevű külső erőforrásfájl tartalmazza. Az ebben található kimenet pontosan az a kimenet, amit a BCC trace produkál az általunk írt teszt alkalmazás megfigyelésekor, ha azt pontosan olyan paraméterezéssel tesszük, mint az élő monitorozás tesztelésénél. Ezáltal miután ezt a szöveget betöltöttük a nézetmodellbe, a kialakult gráf meg kell, hogy egyezzen az élő monitorozások alatt kialakult gráfokkal. Így természetesen itt is a külön modulba kiemelt egységes ellenőrzést használjuk.

3.8.3. Manuális tesztek

A nézet elemeit és az azokhoz tartozó eseményeket kézzel teszteljük, így ezekhez nem érhetők el automata tesztesetek. Az alábbi jegyzőkönyv írja le, hogy az alkalmazás mely funkciója, milyen eredményt kell, hogy produkáljon.

Legelőször az alkalmazás indítását ellenőrizzük. A belépési pont indítása után kapott URL cím segítségével a böngészőben elérjük az alkalmazás felületét. Itt láthatjuk a vezérlőpultot és azt, hogy még nincs semmilyen gráf megjelenítve. A vezérlőpult négy menüpontjára kattintva láthatjuk a hozzájuk tartozó műszerfal tartalmát. Ezek közül az eszköztár beviteli mezőin ellenőrizzük, hogy képesek vagyunk-e adatot bevinni, ugyanis ezek a mezők a gráf betöltése előtt ki vannak kapcsolva.

Miután teszteltük az alkalmazás felületét annak használata előtt, ellenőrizzük az élő monitorozás funkciót. Az Add gomb megnyomása esetén forrás megadása nélkül hibaüzenetet kapunk. Ennek megadása után viszont sikeresen hozzáadódik a források legördülő menüjéhez a megadott forrás, amiről értesítést is kapunk. Az add gomb újabb megnyomása esetén megint hibaüzenetet kapunk, hiszen a forrást már hozzáadtuk megfigyelésre. Ezek után elindítjuk a monitorozást, amelyre hibaüzenetet kapunk, mivel nem adtunk meg egy függvényt sem monitorozásra. Az adott függvények esetén a monitorozást sikeresen elindíthatjuk. Ezután a megadott függvények forrását futtatva láthatjuk a dinamikusan kirajzolódó gráfot. A monitorozás leállítása után a gráfot átalakítjuk, mozgatjuk és lekérdezzük az elemekhez tartozó információs paneleket, melyeknek megvizsgáljuk igazságtartalmát. Ezek után kipróbáljuk a monitorozást több forrással, függvénnyel és paraméterekkel is.

Az élő monitorozást kipróbáljuk konfigurációs fájl használata esetén is. A fájl használatához rendelt kapcsolóra rányomva láthatjuk, hogy előtűnik a fájl elérési útvonalához tartozó beviteli mező. Ennek üresen hagyása, helytelen adatokkal való feltöltése, vagy a megadott konfigurációs fájl helytelen tartalma esetén a monitorozás indításakor az előzménynek megfelelő hibaüzenetet kapunk. Helyes fájl megadása után a monitorozás során láthatjuk a konfigurációs fájlban megadott függvényekhez tartozó hívási gráfot. A monitorozás ezen módját is kipróbáljuk több függvénnyel, forrással és paraméterekkel is. A monitorozások közben ellenőrizzük, hogy elérjük-e a többi, lezárt funkciót és beviteli mezőt.

Az élő monitorozás után teszteljük a statikus betöltést is. Itt tulajdonképpen nem létezik helytelen bemenet, ugyanakkor, ha nem megfelelő formátumú szöveget adunk meg feldolgozásra, semmilyen gráfot nem kapunk. Ezen kívül teszteljük a statikus feldolgozást különböző valós trace kimenetekkel. Ellenőrizzük a kirajzolt gráfot egy és számos hívási lánc esetén, több függvényt, forrást és paramétereket tartalmazó kimenetek esetén is. Végül teszteljük a helyes működést úgy, hogy benne hagyjuk a trace kimenet fejlécét és úgy is, hogy nem.

A statikus feldolgozás és az élő monitorozás esetén is teszteljük, hogy új kimenet betöltése, vagy monitorozás indítása esetén az előző gráf megszűnik, annak adatai nem szivárognak át az újabb adatok közé.

Az eszköztár működését már teszteltük létező gráf nélkül. Így most azzal a feltétellel vizsgáljuk őket, hogy már létezik gráf, amelyen láthatjuk hatásukat. A színeket beállító csúszka esetén teszteljük a rajta található fogókat. Ellenőrizzük, hogy ezeket

keresztülvihetjük-e egymáson és hogy tehetjük-e őket egyenlő pozícióba. Ezek után teszteljük a gráf pontjainak és éleinek színeit a csúszka használata során. Ekkor azt is teszteljük, hogy a nem megfigyelt függvényekhez tartozó pontok, illetve a beléjük vezető élek minden esetben szürkék maradnak-e. A csúszka után teszteljük a keresőmezőt. Mikor még nem kerestünk semmire, minden elem az eredeti színét mutatja. Kereséskor a keresésnek nem megfelelő elemek szürke színűre változnak. Ellenőrizzük, hogy a keresés érzékeny-e kicsi és nagybetűkre, illetve lehetséges-e a függvénynevek bármely töredékére, belső részeire keresni. Végül teszteljük, hogy az eszköztár két eleme együttesen is a várt hatást okozzák-e.

A manuális tesztelést a konfigurációs beállítások vezérlésével fejezzük be. Teszteljük, hogy trace parancs megadásának hiánya esetén milyen hibaüzenetet kapunk, illetve, hogy ekkor mentésre került-e az üres parancs. Teszteljük a konfigurációt további parancsok esetén is, amely során azt tapasztaljuk, hogy az élő monitorozás továbbra is működik-e. Helytelen parancs esetén nem működik. Ezek után ellenőrizzük az animációkhoz tartozó kapcsoló mindkét állapotát élő monitorozással. Végül egy már előállt gráfon teszteljük a pontok közötti távolságot beállító beviteli mezőt. Azt is teszteljük, hogy ez a két beállítás újonnan létrehozott gráfok esetén is továbbra is megmarad.

4. Fejezet
Összefoglalás

A megvalósított alkalmazás, a feladatnak megfelelően, egy webes felhasználói felületet biztosít a BCC trace modulnak. Az alkalmazás segítségével könnyen értelmezhetjük a modul kimenetét, amelyet amúgy csak fáradságos munkával tudnánk feldolgozni, sok idő alatt. Ezáltal hatékonyan elemezhetjük a monitorozott programok belső működését akár valós időben, azok futása közben.

Az alkalmazás lehetőséget biztosít függvények élő megfigyelésére, amelyeket két módon is megadhatunk. Ezen kívül már létező trace eredményeinket is feldolgozhatjuk az alkalmazás segítségével. Az alkalmazás kihasználja a modul több funkcióját is. Megadhatunk több függvényt és forrást is monitorozásra. A függvényekhez megadhatunk megfigyelni kívánt paramétereket, amelyekről a kirajzolt ábra szintén tájékoztat bennünket.

A program belső szerkezete jól elkülönített rétegekből áll, amelyekben a program egyes komponenseit önálló modulok és osztályok valósítják meg. A program kódja egyszerűen átlátható és megérthető. Ezek által a program gond nélkül karbantartható, komponenseit rugalmasan cserélhetjük és bővíthetjük. Így az alkalmazásba a BCC további moduljait gördülékenyen integrálhatjuk.

A Dash keretrendszer segítségével a webes felület egységes kinézetet kapott, amely informatív és egyszerűen kezelhető. A megjelenített gráf interaktívan böngészhető, kedvünk szerint mozgatható, nagyítható és pontjait tetszés szerint rendezhetjük. A gráf elemeinek színezésével és a keresőfunkció segítségével a függvényeket tovább szűrhetjük, ezzel is elősegítve az adatok könnyebb értelmezését a felhasználó számára.

5. Fejezet

További fejlesztési lehetőségek

Természetesen az alkalmazást nem csak további BCC modulokkal fejleszthetjük tovább. A program a jelenlegi állapotában is rengeteg, viszonylag egyszerűen megvalósítható fejlesztési lehetőséget kínál. Az alábbi felsorolás tartalmaz pár ötletet, amellyel az alkalmazás biztosan bővülni fog még a dolgozat leadása után.

A BCC trace segítségével több különböző forrás függvényeit figyelhetjük meg. Az alkalmazásban azt, hogy melyik függvény melyik forráshoz tartozik, az információs panelek segítségével tudhatjuk meg. Ugyanakkor ezzel nehézkes megállapítani, hogy mely a forráshoz tartozó összes függvény és átlátni a források közötti átmeneteket. Erre lenne nagy segítség egy olyan nézet kialakítása, amelyben gráfban a függvények források szerint lennének csoportosítva. Erre létező megoldást kínál a Dash keretrendszer, így ennek megvalósítása egyértelmű.

Jelenleg élő monitorozás esetén kétféle módszer áll rendelkezésünkre a monitorozni kívánt függvények megadására. Ugyanakkor ezen módszerek együttes alkalmazására nincsen lehetőség. Ennek egyfajta megoldása lenne, ha a módszerrel előállított kollekciók átalakíthatók lennének a másik típusra. Így a felhasználói felületen összeállított elemeket elmenthetnénk konfigurációs fájlba, vagy a konfigurációs fájl tartalmával tölthetnénk fel a vezérlőpult megfelelő mezőit.

Az alkalmazás segítségével akárhányszor végezhetünk élő monitorozást. Ugyanakkor minden monitorozáskor az előző adatok elvesznek. Ez általában megfelelő számunkra, ugyanakkor egyes esetekben hasznos lenne, ha egy már meglévő gráfot egészíthetnénk ki újabb hívási láncokkal. Ehhez kapcsolódik az a szintén jelenleg még nem implementált funkció, hogy a már feldolgozott adatokat elmenthessük további használatra, majd később betölthessük őket.

Jelenleg a megfigyelni kívánt függvények nevét manuálisan kell begépelnünk a vezérlőpult megfelelő mezőjébe élő monitorozás esetén. Ugyanakkor az adott forráshoz tartozó függvények nevét automatikusan is listázhatná az alkalmazás. Linux alatt lehetőség van egy programban található függvények lekérdezésére. Ezt felhasználva kilistázhatjuk a felhasználónak, mely függvények közül választhat, így jelentősen meggyorsítva és megkönnyítve az alkalmazás használatát, ráadásul a függvények nevét sem tudná a felhasználó elgépelni.

Forrásjegyzet

[1] „BPF Compiler Collection (BCC),” [Online].
Available: https://github.com/iovisor/bcc. [Hozzáférés dátuma: 23 10 2019].

[2] „Downloading Python,” [Online].
Available: https://wiki.python.org/moin/BeginnersGuide/Download.
[Hozzáférés dátuma: 07 09 2019].

[3] „Installing BCC,” [Online].
Available: https://github.com/iovisor/bcc/blob/master/INSTALL.md.
[Hozzáférés dátuma: 14 08 2019].

[4] C. E. I. d. N. Oren Ben-Kiki, „YAML Ain’t Markup Language,” [Online]
Available: https://yaml.org/spec/1.2/spec.pdf. [Hozzáférés dátuma: 21 09 2019].

[5] „Dash User Guide,” [Online]. Available: https://dash.plot.ly.
[Hozzáférés dátuma: 18 11 2019].

[6] „Pexpect version 4.7,” [Online]. Available: https://pexpect.readthedocs.io/en/stable.
[Hozzáférés dátuma: 28 11 2019].

[7] „Dash Bootstrap Components,” [Online].
Available: https://dash-bootstrap-components.opensource.faculty.ai.
[Hozzáférés dátuma: 02 10 2019].

[8] „Cytoscape.js,” [Online]. Available: https://js.cytoscape.org.
[Hozzáférés dátuma: 05 11 2019].

[9] „Dash Cytoscape,” [Online]. Available: https://dash.plot.ly/cytoscape
[Hozzáférés dátuma: 20 11 2019].