Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Analýza a vizualizace závislostí v Yocto projektech

Plzeň 2022 Michal Linha

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2021/2022

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Michal LINHA**
Osobní číslo:	**A19N0036P**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Téma práce:	**Analýza a vizualizace závislostí v Yocto projektech**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se s prostředím Yocto projects a s technologiemi které jsou s ním spojené.
2. Seznamte se s problematikou vizualizací rozsáhlých SW systémů, s důrazem na zobrazení vztahů mezi komponentami.
3. Navrhněte nástroj pro vizualizaci závislostí v receptech na sestavení linuxového jádra.
4. Implementujte prototyp navrženého řešení.
5. Navržené řešení otestujte, zejména s ohledem na přehlednost a úplnost zobrazených informací.

Rozsah diplomové práce: **doporuč. 50 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

dodá vedoucí diplomové práce.

Vedoucí diplomové práce: **Ing. Richard Lipka, Ph.D.**
Katedra informatiky a výpočetní techniky

Datum zadání diplomové práce: **10. září 2021**
Termín odevzdání diplomové práce: **19. května 2022**

L.S.

Doc. Ing. Miloš Železný, Ph.D.
děkan

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 11. října 2021

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 18.5. 2022

Michal Linha

Abstract

The main subject of this thesis is to create a tool prototype which can be used to visualize dependencies of so-called recipes in *Yocto Project*, a project used to build embedded linux distributions. In the first part of this thesis, *Yocto Project* and its parts are described. Basic forms of visual clutter removal and some dependency visualization techniques usable for software systems are also descriped in the first part of the thesis. Second part of the thesis contains proposion and implementation of the tool prototype called *Yocto Project Dependency Visualizer*. The tool was implemented in the form of a *Visual Studio Code* extension. It is able to visualize *Yocto Project* recipe dependencies directly in *Visual Studio Code* from the input file generated by *Yocto Project*. The visualization is interactive, containing functionalities like node highlighting, node removal, search for nodes and other configuration options.

Abstrakt

Předmětem této práce je vytvoření prototypu nástroje určeného pro vizualizaci závislostí tzv. receptů v *Yocto Project*, projektu, který umožňuje sestavování vestavěných linuxových distribucí. V první části práce je popsán samotný *Yocto Project* a jeho části. Dále zde byly popsány různé způsoby odstranění nepřehledností v grafech a možné způsoby vizualizace závislostí, použitelné pro softwarové systémy. Ve druhé části práce je pak popsán návrh a implementace vytvořeného prototypu nástroje *Yocto Project Dependency Visualizer* v podobě rozšíření pro *Visual Studio Code*. Nástroj dokáže zobrazit závislosti receptů *Yocto Project* přímo ve *Visual Studio Code* ze vstupního souboru vygenerovaného pomocí *Yocto Project*. Vizualizace obsahuje interaktivitu v podobě obarvování uzlů, mazání uzlů, vyhledávání uzlů a další různé konfigurace.

Obsah

1 Úvod

V dnešní době existuje spousta různých linuxových distribucí, přičemž každá může sloužit k různým účelům. Velké množství zařízení, jako jsou například ledničky nebo robotické vysavače, nepotřebuje pro svoji správnou funkčnost složitý operační systém, ale vystačí si pouze se základními funkcemi jádra a minimální funkcionalitou navíc. Tvorba sestavení takových systému ale nemusí být jednoduchá a právě pro ulehčení této činnosti slouží *Yocto Project* [1]. Základní jednotkou práce jsou recepty, pomocí kterých jsou definovány jednotlivé komponenty sestavení, které jsou mezi sebou různě závislé a tak tvoří graf.

Jelikož může být graf závislostí velmi veliký, a může obsahovat spousty protínajících se hran a dalších těžko analyzovatelných částí (tzv. vizuální šum), je vhodné jej zjednodušit. Obecně existuje několik technik pro odstranění tohoto šumu a pro vizualizaci závislostí v systémech, jako je například spojování hran nebo shlukování uzlů, úprava rozvržení jednotlivých uzlů nebo interaktivní vizualizace.

Cílem této diplomové práce je návrh a implementace prototypu nástroje, který uživatelům umožní jednoduše vizualizovat závislosti receptů v *Yocto Project*, přičemž hlavní důraz je kladen na zobrazení závislostí s co nejmenším šumem. Výsledný nástroj by měl uživatelům umožnit lépe analyzovat a vylepšovat jednotlivá sestavení systému.

V teoretické části se nachází popis samotného *Yocto Project* a jeho částí. Dále jsou popsány různé způsoby pro obecné zjednodušení vizualizací a způsoby vizualizace závislostí použitelné pro softwarové systémy od klasických, například založených na *UML*, po méně konvenční, jako je například vizualizace pomocí virtuální reality.

V praktické části je pak návrh nástroje a popis samotné implementace vytvořeného nástroje pro vizualizaci závislostí receptů v *Yocto Project*. Nakonec jsou diskutována omezení nástroje a návrhy na další možná rozšíření.

2 Yocto Project

V dnešní době existuje velké množství vestavěných, dále embedded, zařízení. Mezi tyto zařízení patří například mobily, různé set-top-boxy, a další elektronika. Tyto embedded zařízení ale pro svoji funkcionalitu mohou potřebovat operační systém. Operační systém v embedded zařízeních nemusí být příliš mohutný, stačí, aby uměl jen několik funkcí. Jako operační systém se často používá *Linux*, pro jehož sestavení může sloužit *Yocto Project* [1]. Yocto Project je open-source projekt, na kterém spolupracuje několik subjektů, a který zahrnuje nástroje a prostředí usnadňující tvorbu embedded *Linux* distribucí [2]. Mezi společnosti, které jsou součástí *Yocto Project* patří například *CISCO*, *Microsoft*, *Meta*, *Intel*, *ARM* a další [3].

2.1 OpenEmbedded

Základem *Yocto Project* je *OpenEmbedded Project*. Jedná se framework pro sestavování embedded Linux distribucí, který používá *BitBake* [5, 4]. *BitBake* je engine sloužící k plánování a vykonávání úkolů. Vytváří strom závislostí ze vstupních receptů, které jsou popsány v kapitole 2.6. Jeho cílem je umožnit *shell* a *Python* skriptům efektivní a paralelní vykonávání. Pomocí těchto skriptů poté dochází k vytvoření (s případnou kompilací zdrojových kódů) balíků, ze kterých se následně skládá samotná distribuce. Ve své podstatě je podobný nástroji *GNU Make* či jiným *make* programům. Pro svoji funkcionalitu používá různá metadata, která budou popsána v dalších kapitolách. *Yocto Project* z *OpenEmbedded* také využívá (a spravuje) *OpenEmbedded-Core*, což je vlastně set metadat, která jsou společná pro různé systémy založené na *OpenEmbedded* [6].

2.2 Referenční distribuce

Pro seznámení se s *Yocto Project* a vyzkoušení si jeho funkcionality, nebo k získání základní struktury pro začátek práce na nové embedded distribuci, lze použít referenční distribuci *Poky* [7]. *Yocto Project* umožňuje stažení této referenční distribuce z verzovacího systému *Git*. Referenční distribuce obsahuje *OpenEmbedded Core*, *BitBake* a základní metadata. Umožňuje sestavit několik distribucí, které je poté možné spustit v emulátoru *QEMU* [9, 8], který poskytuje virtuální model celého stroje [10]. Tato referenční distri-

buce je pravidelně aktualizována tak, aby vždy obsahovala nejnovější *Yocto Project* funkcionalitu [8].

2.3 Balíky

Balíky jsou výstupem build systému a jejich kombinací je vytvořena výsledná distribuce [11]. Jedná se o zabalený výstup programu *BitBake* vytvořený z jednoho receptu. Obvykle obsahuje zkompilované binární soubory [12]. Může být například vytvořen balík, který ve výsledné distribuci zajišťuje funkcionalitu příkazové řádky (*bash*), nebo balík pro *Perl* či *Python*.

2.4 Struktura projektu

Následující popis vychází ze struktury projektu referenční distribuce a popsány jsou jen základní složky. Pro základní nastavení struktury projektu a prostředí příkazové řádky slouží skript `oe-init-build-env`, který se volá příkazem `source` a je v kořenové složce projektu [13].

Jednou ze složek je složka *bitbake*, obsahující kopii nástroje *BitBake*. Obvykle se jedná o aktuální verzi tohoto nástroje. Další složkou je složka *build*, která obsahuje uživatelské konfigurační soubory a výstup build procesu. Ve složce *documentation* jsou poté uloženy soubory dokumentace a šablony a nástroje pro generování *PDF* nebo *HTML* verzí dokumentace. Složka *meta* obsahuje *OpenEmbeddedd-Core* metadata. Dalšími složkami jsou *meta-yocto-bsp*, *meta-selftest*, *meta-poky* a *meta-skeleton*, obsahující další metadata. Jedná se vlastně o vrstvy (viz kapitola 2.8). Poslední složkou je složka *scripts*, která obsahuje různé pomocné skripty, například skript pro vytvoření pull requestu do repositáře referenční distribuce [13].

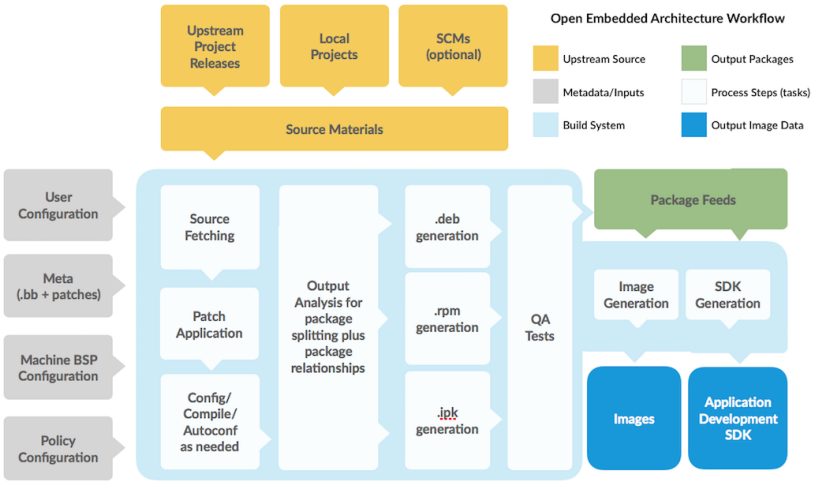
2.5 Průběh sestavení

Jako první *Yocto Project* připraví konfiguraci a další metadata, jako jsou například recepty, pomocí kterých určí z jakých balíků se bude výsledná distribuce skládat. Pro spuštění sestavení se zavolá *BitBake* příkaz se zadaným cílem, který následně provede několik kroků [4].

BitBake nejprve získá zdrojové soubory z repositářů, které jsou vývojářem specifikovány v receptech (*fetch*). Dále provede jejich extrakci (*extract*) a na extrahované zdrojové soubory aplikuje záplaty (*patch*) ze specifikovaných souborů. Dalšími kroky jsou konfigurace (*configure*), při které dojde k nastavení možností pro sestavení [14], a kompilace (*compile*) po které

následně proběhne sestavení (*build*). Posledními kroky jsou instalování (*install*), tedy zkopírování výstupních souborů do cílového umístění a vytvoření balíků (*package*) [4].

Zároveň se v průběhu celého sestavování distribuce různými způsoby kontroluje kvalita a správnost sestavení (například kontrola vygenerovaného *ELF* souboru nebo u endianity, jestli odpovídá zvolené architektuře [15]). Tyto testy jsou obsaženy ve třídě *insane* (více informací o třídách lze nalézt v kapitole 2.7) [15]. Po vytvoření balíků dojde k vytvoření samotného obrazu distribuce, a nakonec dojde k paralelnímu vytvoření *eSDK* a obrazu souborového systému [4]. Graf workflow *Yocto Project* je vidět na obrázku 2.1.



Obrázek 2.1: Workflow *Yocto Project* [11].

Zdrojové soubory je možné získat z různých systémů pro správu verzí. Jako příklad lze uvést *Git*, *SVN*, *ClearCase* a další. Zdrojové soubory je také možné získat pomocí *HTTP* nebo *FTP* protokolu anebo také z lokálního úložiště [16].

2.6 Recepty

Hlavními metadaty jsou takzvané recepty (*recipes*). Tyto recepty slouží pro popis balíku, specifikování jeho verze, závislých balíků, které můžou být využity buď během běhu distribuce nebo během jejího sestavení, cesty ke zdrojovým souborům balíku a způsob jejich získání, definování a použití záplat, způsob konfigurace a kompilace zdrojového kódu, způsob vytvoření balíku

(nebo více balíků) a kde by měl být balík nainstalován v rámci výsledné distribuce. Receptem je soubor s příponou *.bb* [17].

2.7 Třídy

Úkolem tříd je uložení informací, využitelných například v receptech (základní třída například obsahuje úkoly pro stažení zdrojových souborů, jejich extrakci, kompilaci a další). Příkladem může být třída pro *CMake*, která se stará o kompilaci zdrojových souborů pomocí nástroje *CMake*. Mezi *BitBake* třídy patří soubory s příponou *.bbclass* [17].

2.8 Vrstvy

Pro oddělení různých receptů a dalších metadat slouží vrstvy. Jejich použitím je například možné oddělit grafické uživatelské rozhraní od nastavení stroje, na kterém distribuce poběží. Vytvořené vrstvy je možné sdílet s dalšími vývojáři a ti je mohou používat ve svých projektech. Vrstvy jsou uloženy jako repositáře a jedná se v podstatě o kolekce receptů, které společně zajišťují jednu funkcionalitu. V rámci vrstvy je možné přepisovat hodnoty proměnných vrstev ležících pod danou vrstvou. Speciálním druhem vrstvy je *BSP* (*Board Support Package*), které dodávají samotní dodavatelé hardware a obsahují konfigurace nebo recepty, které jsou specifické pro určité hardware zařízení nebo platformu [17].

2.9 Přídavné soubory

Použitím přídavných souborů je možné rozšířit funkcionalitu receptů. V případě použití přídavných souborů, dojde k rozšíření nebo přepsání informací v hlavním receptu. Jedná se o soubory s příponou *.bbappend*. Těmito soubory je také možné rozšířit funkcionalitu receptů, které jsou součástí jedné vrstvy, v rámci jiné vrstvy [17].

2.10 Konfigurační soubory

Soubory s příponou *.conf* jsou konfigurační soubory. V těchto souborech jsou definovány proměnné použité během sestavování. Tyto proměnné mohou být využity pro konfiguraci stroje, distribuce, překladače a dalších. V rámci pro-

jektu existuje několik konfiguračních souborů například globální konfigurační soubor a konfigurační soubory pro jednotlivé vrstvy [17].

2.11 *Bitbake*, shell a *Python*

Jelikož je *BitBake* vytvořen v *Pythonu*, je možné například při nastavování proměnných v receptech nebo třídách použít *Python* funkce/metody a je také možné *Python* funkce v receptech vytvářet. Lze také vytvářet *shell* funkce. Z vytvořených funkcí je možné vytvářet úlohy a přiřazovat jim závislosti [18].

2.12 Nástroje z *Yocto Project*

Yocto Project se kromě *OpenEmbedded* build systému (*OpenEmbedded-Core* a *BitBake*) skládá z několika open-source nástrojů. Mezi nástroje, které slouží k vývoji distribucí patří *CROPS*, což je cross-platform framework, který slouží k vytvoření prostředí pro sestavování binárních souborů. Dalším z nástrojů je *Extensible Software Development Kit* (*eSDK*), který slouží pro vývoj a přidávání aplikací a knihoven do distribucí. Ke své funkcionalitě využívá nástroj *devtool* [19], který provádí sestavení, testování a vytváření balíků. Posledním nástrojem pro vývoj je *Toaster*, pomocí nějž je možné spouštět sestavení distribucí a sledování informací o sestavení pomocí webového rozhraní [4].

Pro produkci slouží *Auto Upgrade Helper*, který generuje vylepšení pro recepty, založené na nových publikovaných verzích receptů. *Recipe Reporting System* pak slouží pro sledování verzí *Yocto Project* receptů. Dalšími nástroji jsou *Patchwork*, sloužící pro organizaci záplat, *AutoBuilder*, jež je používán pro automatizaci testování a dodržování kvality. Nástroj *Pseudo* se používá pro spouštění příkazů v prostředích, které se chovají jako by měly *root* privilegia. Posledním nástrojem je *Cross-Prelink*, sloužící k provádění pre-linkování [4].

2.13 Vývojové prostředí pro *Yocto Project*

Embedded *Linux* distribuce je možné vytvářet v několika prostředích. Nejvhodnějším prostředím je host s nainstalovaným systémem *Linux*, na kterém je možné jednoduše spouštět nástroj *BitBake*. Mezi podporované distribuce patří *Ubuntu*, *Fedora*, *CentOS*, *Debian GNU/Linux* a *openSUSE* [20]. Podporované verze těchto systémů je možné najít v dokumentaci. Další mož-

ností je využití *CROPS*, který využije *Docker* kontejnerů pro vytvoření host stroje [4]. Pro systémy *Windows* je také možné použít *WSLv2*, (*Windows Subsystem For Linux v2*), který umožní stažení *Linux* distribuce z *Microsoft Store*, její instalaci a spuštění ve virtuálním stroji. Pomocí *WSLv2* je poté možné přistupovat k souborům ve virtuálním stroji pomocí *Windows* aplikací. *Yocto Project* ale podporuje pouze *WSLv2*, který je součástí *Windows 10* od sestavení *18917* [4]. Poslední možností je využití webového nástroje *Toaster*, pomocí kterého lze konfigurovat a pouštět sestavení a také získávat informace o sestavení [4].

Jako vhodné aplikace pro vývoj embedded systémů pomocí *Yocto Project* lze považovat *Eclipse IDE* a *Visual Studio Code*. Pro *Eclipse* navíc existoval plugin pro *Yocto Project*, jehož vývoj a podpora ale už byla ukončena ve verzi *2.7 (warrior)* [21]. Při použití *Visual Studio Code* je možné nainstalovat do něj rozšíření, které zpřehlední úpravu metadat, tedy receptů, konfigurací a dalších souborů, pomocí zvýraznění syntaxe [22].

2.14 Graf závislostí

```
3315  "db-native.do_unpack" -> "db-native.do_fetch"
3316  "db.do_build" [label="db do_build\n1:5.3.28-r1\n/home/mich
3317  "db.do_build" -> "db.do_package_qa"
3318  "db.do_build" -> "db.do_package_write_rpm"
3319  "db.do_build" -> "db.do_packagedata"
3320  "db.do_build" -> "db.do_populate_lic"
3321  "db.do_build" -> "db.do_populate_sysroot"
3322  "db.do_build" -> "gcc-runtime.do_package_write_rpm"
3323  "db.do_build" -> "glibc.do_package_write_rpm"
3324  "db.do_build" -> "libgcc.do_package_write_rpm"
3325  "db.do_build" -> "linux-libc-headers.do_package_write_rpm"
3326  "db.do_compile" [label="db do_compile\n1:5.3.28-r1\n/home/
3327  "db.do_compile" -> "db.do_configure"
```

Obrázek 2.2: Ukázka vygenerovaného grafu v *DOT* formátu.

Pomocí *BitBake* je možné vygenerovat graf závislostí. Při specifikování cíle vygeneruje *BitBake* graf závislostí úkolů ve formátu *DOT*, který lze vidět na obrázku 2.2. V tomto grafu jsou za uzly považovány úkoly vygenerované *BitBake* nástrojem a hrany jsou pak závislosti mezi úkoly. Je tak možné vidět, jak na sebe úkoly navazují. Zároveň vygeneruje seznam cílů, které se pro dokončení zadaného cíle musí vykonat. Pro zobrazení závislostí mezi recepty graficky je možné použít parametr `-u taskexp` při volání *BitBake*

pro generování grafů, což zobrazí okno v grafickém uživatelském rozhraní a umožní výběr úkolu, přičemž ve druhé půlce okna zobrazí všechny závislé úkoly a úkoly na kterých závisí. Tato funkcionalita je ale možná jen v systémech s grafickým uživatelským rozhraním. Je také možné ignorovat některé recepty (např. základní recepty), pokud není potřeba jeho zobrazení v grafu, čímž lze dosáhnout menší složitosti vygenerovaného grafu [23].

Vygenerovaný soubor ve formátu *DOT* je možné přeformátovat do grafické podoby použitím nástroje *Graphviz*[1][23]. Soubory ale mohou být příliš velké, což může stěžovat orientaci v nich. Také vykreslení do grafické podoby může trvat relativně dlouhou dobu. Odhadem může graf obsahovat stovky uzlů a hran.

Pro zobrazení závislostí je také možné použití webové aplikace *Toaster*, ve které je možné zobrazit jak recepty, na kterých daný recept závisí, tak recepty, které závisí na daném receptu. Umožňuje také zobrazení úkolů a informací o jejich běhu a dalších informací.

[1] https://graphviz.org.

3 Techniky vizualizace velkého množství dat a závislostí v software

Vizualizace software je velmi důležité téma. Softwarové projekty mohou být v dnešní době velmi rozsáhlé a mohou se skládat z velkého množství komponent. Vizualizace proto musí být přehledné a musí zobrazovat relevantní informace. Díky vizualizaci software je například možné ukázat vývojářům, kteří jsou noví na projektu, jaké komponenty projekt obsahuje, z jakých se skládá tříd a další informace [24]. Může také umožnit senior vývojářům snadnější hledání chyb, minimálně z pohledu kontroly správnosti závislostí komponent.

Při vizualizaci grafů s velkým počtem uzlů a hran mezi nimi, je nutné vzít v potaz několik omezení. Jedním z omezení je počet pixelů na zobrazovacím zařízení. To může způsobit zhoršenou čitelnost při pokusu o zobrazení celého grafu, nebo nemožnost graf zobrazit. Dalším z omezení je mozková kapacita člověka, snažícího se pochopit souvislosti v rozsáhlém grafu, což může být v podstatě nemožné. Třetím omezením je množství hran v grafu. Po překročení určité hranice ze začnou hrany křížit tak, že ani algoritmy určené pro uspořádání uzlů nedokážou graf zobrazit čitelně a tím pádem bude nemožné se v grafu orientovat. Velký počet zobrazených informací v grafu lze nazvat anglickým termínem *visual clutter*. Posledním omezením je nedostatečný výpočetní výkon, kdy může docházet ke zdlouhavému načítání grafu a k jeho obtížnému procházení v případě interaktivních grafů [25].

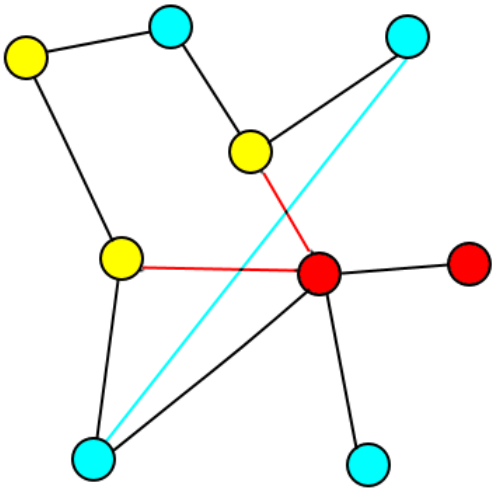
Následující kapitoly popisují některé možné přístupy k vizualizaci software, od *2D* po virtuální realitu, a možné přístupy ke zjednodušení a zpřehlednění zobrazení vizualizace. Způsoby uvedené v následujících kapitolách jsou obecné a ne vždy se týkají vývoje software. Teoreticky by ale bylo možné tyto techniky použít i pro vizualizaci závislostí v software.

3.1 Běžně používané přístupy

Existuje několik běžně používaných řešení pro redukci nadbytečných informací, které lze různě kombinovat.

3.1.1 Obarvování

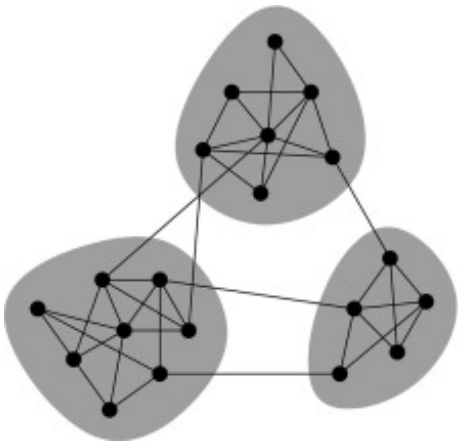
Pomocí obarvování lze rozlišit jednotlivé uzly a hrany. Při překřížení hran lze například každou obarvit jinou barvou, což usnadní orientaci a uživatel tak stále dokáže určit, kde hrana začíná a kde končí. Problém je, že paleta barev musí být omezena jen na takové barvy, na které je člověk nejvíce senzitivní [26]. Barvy také musí být zvoleny takové, aby uživatel dokázal rozlišit jednotlivé rozdíly mezi nimi. Příklad obarvování lze vidět na obrázku 3.1.



Obrázek 3.1: Zobrazené grafu s uzly, které mají různé barvy (může být dáno vlastnostmi prvků, které uzly reprezentují), a s křížícími se hranami, které jsou odděleny pomocí barev.

3.1.2 Shlukování

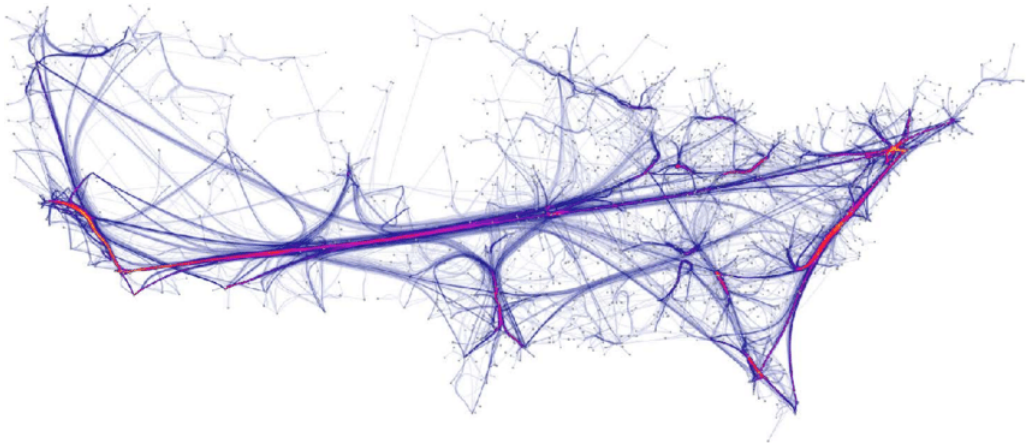
Dalším běžně používaným řešením pro odstranění nadbytečných informací nebo hran je shlukovaní. V tomto případě obvykle dochází ke shlukování uzlů nebo hran podle jejich společných vlastností do oddělených shluků. Problémem ale může být správné určení počtu takových shluků [26]. Shlukování lze vidět na obrázku 3.2.



Obrázek 3.2: Zobrazené shluky uzlů [27].

3.1.3 Spojování hran

Hrany, které by jinak vedly blízko sebe, lze spojit do jedné hrany viz obrázek 3.3. Tento způsob může zredukovat počet křížení hran, ale na druhou stranu může dojít ke snížení čitelnosti grafu. Pro tento způsob redukce nadbytečných informací je ale nutné, aby hrany byly kompatibilní, tedy aby je bylo možné spojit a aby graf neobsahoval jejich přílišné množství [26].

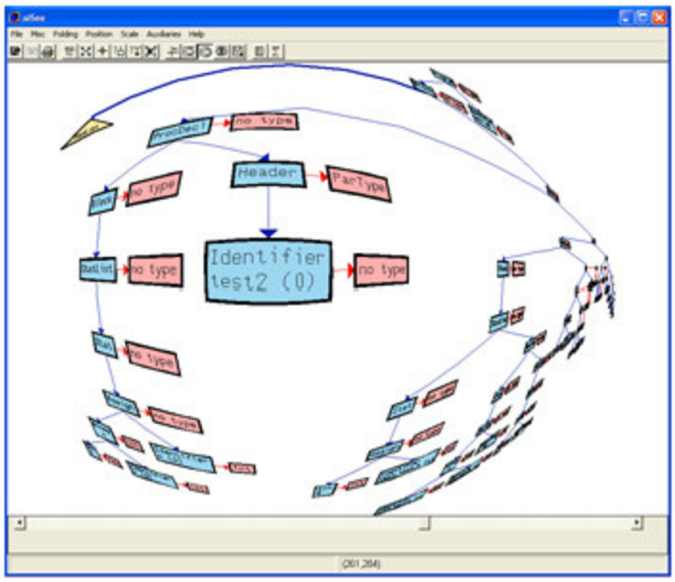


Obrázek 3.3: Zobrazení spojených hran v grafu [28].

3.1.4 Čočka

Dalším řešením redukce je zobrazení pomocí čočky. V tomto případě zůstává graf obvykle stejný, jen je pomocí čočky zvýrazněna (zvětšena) uživatelem zvolená část grafu. V případě zobrazení pomocí čočky může dojít i k přeuspořádání zvýrazněné části grafu a zobrazení více informací. Používané

implementace čočky jsou rybí oko (viz obrázek 3.4), *Magic Lens* nebo *Network Lens* [26]. Problémem ale může být zkreslení informací nebo špatná interpretace informací [26].

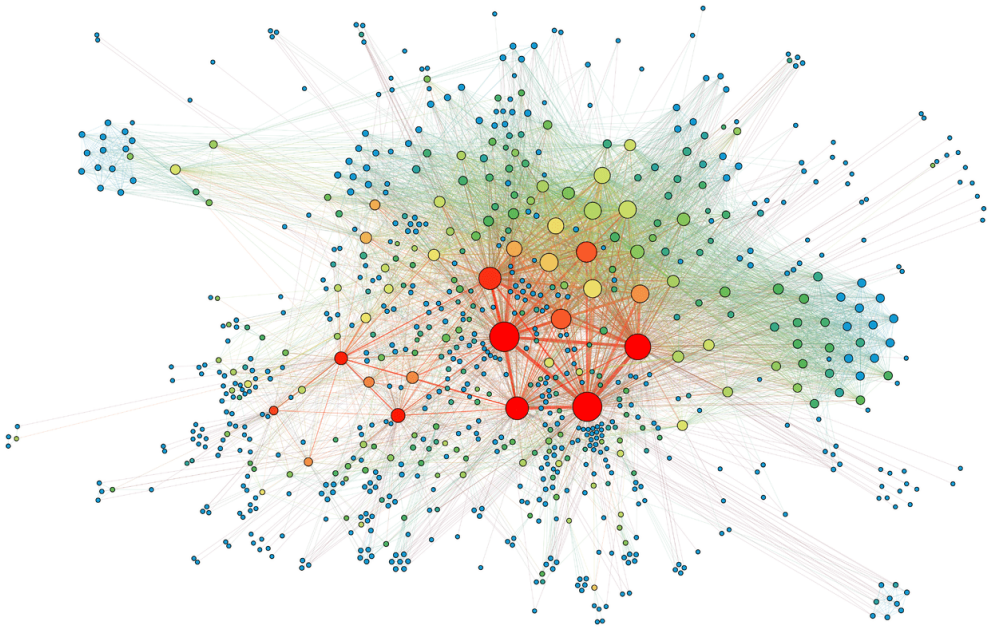


Obrázek 3.4: Zobrazení pomocí rybího oka [29].

3.1.5 Rozvržení

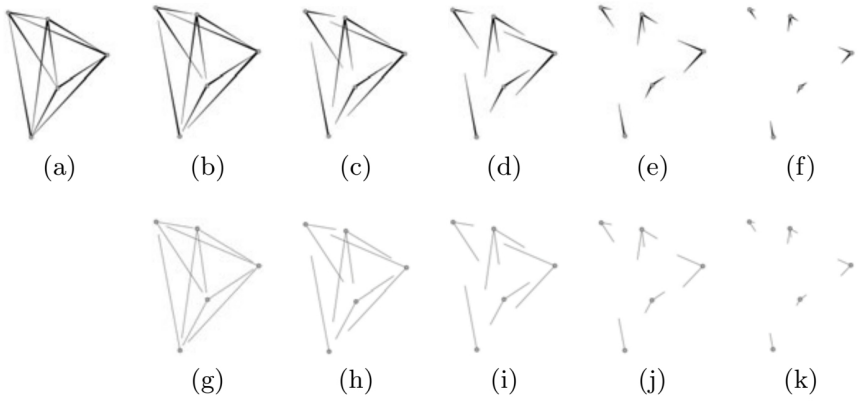
Pro zpřehlednění grafu slouží algoritmy, které určují jeho permanentní rozvržení. Takovýchto algoritmů existuje velké množství a každý si může zvolit takový, jaký mu nejvíce vyhovuje [26].

Jedním z druhů takovýchto algoritmů může být *force-directed* algoritmus. Toto rozvržení používá pouze informace ve struktuře grafu, na jeho doménu se nespoléhá. Jsou založené na fyzikálním modelu. Uzly grafu se v tomto rozvržení chovají jako nabité částice nebo magnety, které se navzájem odpuzují. Tato síla, která je odpuzuje, je silnější čím blíže uzly jsou. Hrany mezi uzly se chovají jako pružiny, které mají určitou délku a různě se natahují. Pokud je hrana (pružina), příliš napnutá, uzly se přitahují. Další částí algoritmu je energie, která způsobuje pohyb uzlů v náhodných směrech. Po určitém čase se výstup algoritmu ustálí a tak vznikne graf s *force-directed* rozložením. Existuje několik implementací takového algoritmu [30, 31]. Možný výsledek algoritmu lze vidět na obrázku 3.5.



Obrázek 3.5: Možný výsledek grafu po použití *force-directed* algoritmu [31].

3.2 Částečně vykreslené hrany pro orientované grafy



Obrázek 3.6: Zobrazení možností čar. V horní části jsou čáry zužující se, ve spodní klasické rovné čáry. a) 100% délky, b) 90% délky, c) 75% délky, d) 50% délky, e) 25% délky, f) 12,5% délky, g) 90% délky, h) 75% délky, i) 50% délky, j) 25% délky a k) 12,5% délky [32].

Jedním ze způsobů pro zjednodušení orientace v grafech, můžou být částečně vykreslené hrany grafu, což ve svém článku Evaluating Partially

Drawn Links for Directed Graph Edges [32] popisují Michael Burch, Corinna Vehlow, Natalia Konevtsova a Daniel Weiskopf. Tímto je možné předejít velkému množství křížení hran [32].

Jednou z možností částečného vykreslení hran je vykreslení pouze půlky, nebo ještě kratší části, hrany, začínající ve zdrojovém vrcholu a pokračující směrem k cílovému vrcholu. Další možností je vykreslení celého propojení, ale s krátkými mezerami [32]. Obrázek 3.6 zobrazuje možné délky a způsoby vykreslení hran.

Částečné vykreslení hran implementované v tomto článku bylo implementováno podle první, výše zmíněné možnosti s tím, že zároveň byly implementovány dva způsoby vykreslení čar. První způsob je tradiční rovná čára, druhý způsob je zužující se čára. Dále bylo implementováno několik různých délek zkrácených hran [32].

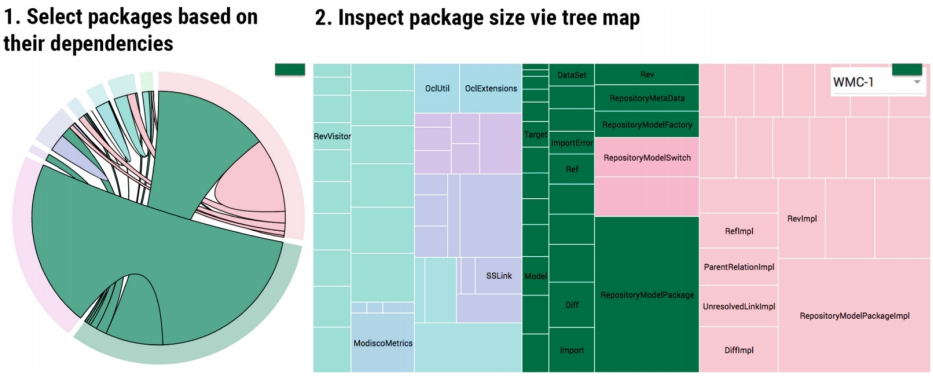
Na závěr je v článku popsána uživatelská studie, ve které uživatelé na různých vytvořených grafech, které byly vygenerovány s pomocí částečně vykreslených hran, prováděli různé úkoly. Z výsledků studie vyplývá, že způsob vykreslení hrany, tedy klasická rovná čára nebo zužující se čára, nemá na orientaci v grafu téměř žádný vliv. Dále bylo zjištěno, že optimální délka vykreslené hrany závisí na způsobu práce s grafem. Pro pouhé zjištění počtu z uzlu vycházejících hran je potřeba pouze velmi krátkých čar. Pro nalezení cesty mezi několika uzly je zapotřebí čar delších. Podle studie může dojít k urychlení práce díky grafům s částečně vykreslenými hranami, ale zároveň může dojít ke snížení přesnosti orientace v takovém grafu [32].

3.3 Interaktivní vizualizace software pomocí více grafů

Obvykle se vizualizace software provádí pomocí statických grafů. Pro usnadnění a interaktivní procházení různých grafů o software, byl v článku Interactive Visualization of Software [33] od Markuse Scheidgena, Nilse Goldammera a Joachima Fischera navržen a implementován jazyk, který lze použít pro vytváření interaktivních vizualizací software na základě výsledků jeho analýzy. Výsledkem analýzy jsou závislosti, hierarchie a metriky. Výsledná vizualizace je založená na kombinacích již existujících grafů [33].

Datům získaným z analýzy se říká *data-set*, z čehož data reprezentující softwarové komponenty jsou nazvány *data-pointy* a jsou popsány typem a značkou, a data reprezentující charakteristiky, jako například reference na další *data-pointy* nebo data s nějakou hodnotou, jsou nazvána *vlastnosti* [33].

Nejprve byl autory článku vybrán set grafů, které lze použít pro vytvoření vizualizace, dále byl vytvořen způsob projekce dat z *data-setu* do grafů, čehož bylo dosaženo podobným způsobem jako v *XML XPath*. Nakonec byly vybrány čtyři principy kombinací grafů, které jsou založeny na grafech, výběrových skupinách, zdrojích dat a tocích dat mezi nimi. Každý graf má typ a dále může mít konfiguraci (osy, rozsahy atd.), vzor a další specifické vlastnosti. Vzor a specifické vlastnosti určují jak jsou data v grafu zobrazena. Data do grafu přichází ze zdrojů, což může být například *JSON* soubor, nebo ze skupin, které mají dvě funkce, a to možnost kombinace *data-pointů* z různých grafů a možnost předávat tyto kombinace dalším grafům. Nelze ale dodávat data grafu ze skupiny která je v něm obsažena. Navržený jazyk poté pracuje s výše popsaným meta-modelem a umožňuje kombinovat grafy. Uživatel poté může například vybrat data v jednom grafu a sledovat změny v grafu druhém [33]. Použití lze vidět na obrázku 3.7.



Obrázek 3.7: Zobrazení použití více grafů [33].

Navržený způsob vizualizace byl implementován jako knihovna webových komponent. Pro samotnou vizualizaci byla použita webová knihovna *D3.js*, která umožňuje převádět datové modely do *DOM*, v jehož stromu jsou uloženy grafické elementy společně s *data-pointy*. Tím pádem je možné získat data z grafu zpět, což jiné knihovny nemusí umožňovat. Další výhodou *D3.js* je jednoduchá stylizace komponent grafu [33].

Uživatelé mohou díky vytvořené aplikaci vytvářet komplexní vizualizace. Aplikace je zároveň rozšiřitelná a uživatel do ní může implementovat i jiné typy grafů [33].

3.4 Interaktivní vizualizace software s pomocí konverzačního uživatelského rozhraní

Obyčejné vizualizace, například s využitím *UML* digramů mohou zobrazit celé prostředí projektu. Může se ale stát, že uživatele zajímá pouze nějaká určitá část programu. To může například znamenat, že by uživatel chtěl zobrazit pouze komponenty, které tvoří grafické uživatelské rozhraní a jejich závislosti. Pro toto vytvořili Stefan Bieliauskas a Andreas Schreiber v článku A Conversational User Interface for Software Visualization [24] prototyp nástroje, který pomocí konverzačního rozhraní umožní uživateli požádat aplikaci o zobrazení určité části vizualizace. Tento nástroj je také možné využít v konverzaci několika osob, kdy nástroj aktivně zpracovává věty z konverzace uživatelů a na základě zpracovaných dat automaticky zobrazuje část grafu o které se uživatelé baví [24].

Konverzační uživatelské rozhraní umožňují uživatelům komunikaci s počítači způsobem bližším normální lidské komunikaci [24].

Mezi konverzační uživatelské rozhraní můžou patřit asistenční systémy. Tyto systémy se snaží problémy řešit delegací práce na příslušné subsystémy. Příklady asistenčního systému mohou být *Siri*, *Cortana* nebo *Alexa*. Asistenční systémy zároveň obvykle obsahují rozhraní pro přidávání nové funkcionality [24].

Dalším možným případem konverzačního uživatelského rozhraní mohou být chatboti. Tyto komponenty se v rámci určité komunikační platformy chovají podobně jako lidé. To znamená, že zpracovávají věty zadané uživatelem a poté hledají vhodné odpovědi podle jejich znalostí. Chatboti jsou vhodné převážně pro řešení specifických úkolů jako je například odpovídání na otázky o počasí. Chatbot zároveň ukládá historii konverzací, nebo alespoň klíčová slova z ní, aby mohli uložené informace použít v budoucí konverzaci. Když se například uživatel zeptá na aktuální počasí v Plzni, tak si chatbot uloží Plzeň jako požadovanou lokaci, a poté při otázce na počasí na další dny použije chatbot tuto informaci pro zobrazení předpovědi v Plzni [24].

Autoři článku vytvořili 3 základní způsoby použití nástroje. Prvním z nich je využití pro jednoduché zobrazení vizualizace. Jako příklad zmiňují, potřebu manažera projektu zobrazit informace o software v rámci běžné firemní komunikační platformy. Tímto by mohl také získat informace a vizualizaci o změnách v architektuře software [24].

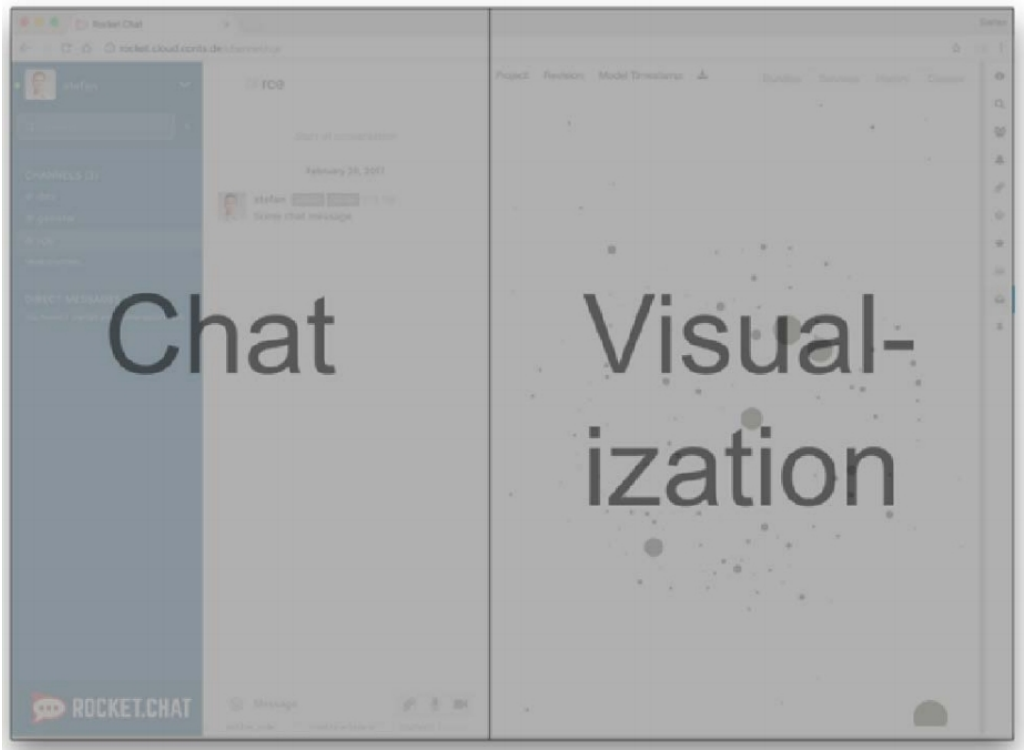
Druhou možností je zobrazení vizualizace novým členem týmu. Ten by mohl s nástrojem komunikovat a dotazovat se na různé části systému, které mu nástroj zobrazí, např. balíky, které obsahují dotazovanou funkcionalitu,

a balíky, které s nimi mají nějakou závislost [24].

Výše zmíněné využití může být také použito pro snadnější hledání chyb v software tím, že dojde k zobrazení pouze těch balíků, které mají něco společného s dotazovanou (chybnou) funkcionalitou [24].

Poslední možností je zobrazování vizualizace pro distribuovaný tým. Členové týmu by tak mohli probírat nějakou funkcionalitu, přičemž chatbot by mohl na základě informací získaných z konverzace vytvářet vizualizaci probírané funkcionality. Toto by mělo umožnit všem členům týmu lépe vidět čeho se konverzace týká [24].

Vytvořený nástroj pracuje s *Java OSGi* aplikacemi. To znamená, že jedna z částí architektury je *OSGi-API*, které slouží pro získávání informací o třídách, balících a dalších částech zkoumaného software [24].



Obrázek 3.8: Základní rozvržení grafického uživatelského rozhraní [24].

Hlavními částmi jsou pak komunikační software a chatbot, jenž zpracovává zprávy uživatelů, ve kterých hledá klíčová slova a spojení podle regulárních výrazů. Další částí je vizualizační komponenta. Tato komponenta slouží pro zobrazení svazků (*bundles*) pomocí grafu. V popisované verzi nástroje jsou velikosti zobrazovaných svazků určeny podle balíků v nich obsažených. Dále jsou zobrazeny jejich importované a exportované závislosti. Vizualizační komponenta přijímá data od chatbota, pomocí nichž provádí filtraci

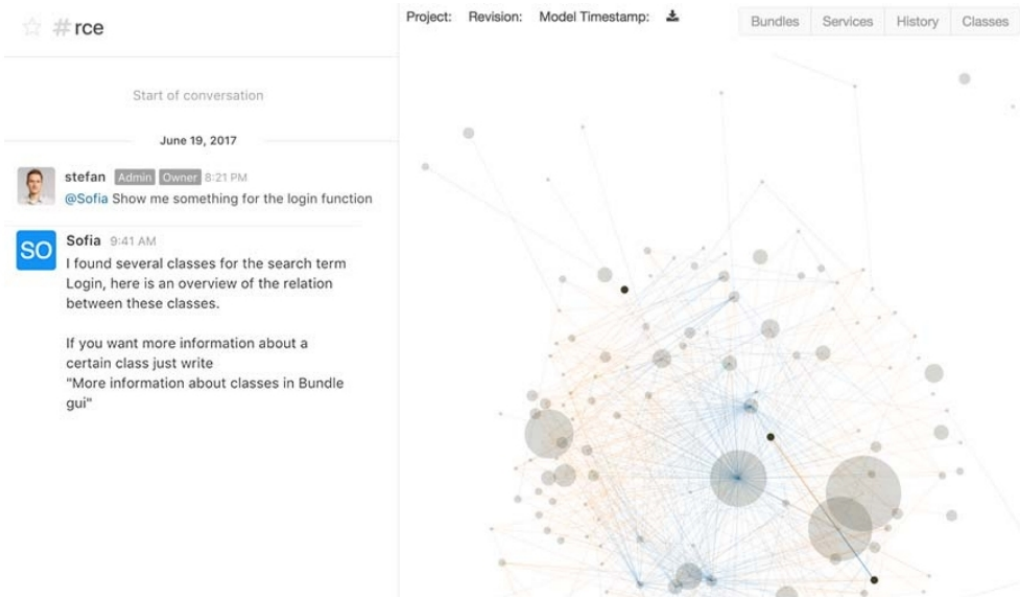
vizualizace [24].

V komunikačním software jsou pak spuštěny jak chatbot, tak vizualizační komponenta. Chatbot a vizualizační komponenta jsou však na sobě nezávislé nástroje [24].

Nástroj je rozdělen do dvou oken, kde nalevo je komunikační software a napravo pak zobrazená vizualizace [24]. Rozložení grafického uživatelské rozhraní lze vidět na obrázku 3.8.

S nástrojem je možné pracovat dvěma způsoby. Prvním způsobem je přímá komunikace, kdy uživatel oslovuje přímo chatbota, který poté předává data vizualizační komponentě, která provede vizualizace viz obrázek 3.9. Zde můžou nastat dvě možné situace. První je, že uživatel už ví, na jakou část systému se chce chatbota zeptat, a tak se jej zeptá, přitom dojde k přímému zobrazení všech odpovídajících částí a jejich závislostí, přičemž zvýrazní ty, které přímo odpovídají hledané části. Druhou možností je obecná otázka o nějaké funkcionalitě. V tomto případě dojde k zobrazení všech svazků, které obsahují dotazovanou funkcionalitu a také všech svazků, které mají třídy s hledanou funkcionalitou [24].

Druhým způsobem je pasivní komunikace, kdy chatbot zpracovává veškerou komunikaci v místnosti a snaží se co nejpřesněji zobrazit část, které se komunikace týká. Poté dochází k zobrazení stejné vizualizace všem účastníkům komunikace [24].



Obrázek 3.9: Ukázka funkcionality nástroje [24].

Některé způsoby použití je možné nahradit jinými metodami. V případě zjišťování informací o chybách je někdy lepší se zeptat přímo vývojáře dané

funkcionality, to samé může platit i o zaučování nového člena týmu. Navržený nástroj ale může tyto metody zpřesnit a usnadnit. Jelikož se jedná o prototyp, je možné jeho funkcionalitu nadále rozšířit, například možností odpovídat na složitější dotazy [24].

3.5 Interaktivní vizualizace závislostí předmětů na univerzitě

Článek Interactive visualization of dependencies [34] od Camila Arango Moreny, Waltera F. Bischofa a H. Jamese Hoovera se sice nevěnuje přímo vizualizaci software a jeho komponent, ale věnuje se vizualizaci závislostí předmětů na univerzitě. Nicméně je možné, že by popisovaná funkcionalita fungovala i ve vizualizaci software. Předměty na univerzitě mají nějaké prerekvizity, tedy předměty, které musejí být splněny před zapsáním určitého předmětu, a předměty, které musejí být splněny před nebo souběžně s určitým předmětem, tedy tzv. korekvizity. Popisovaný způsob vizualizace byl vytvořen z důvodu relativně velkého množství chybně vytvořených studijních plánů studenty univerzity [34].

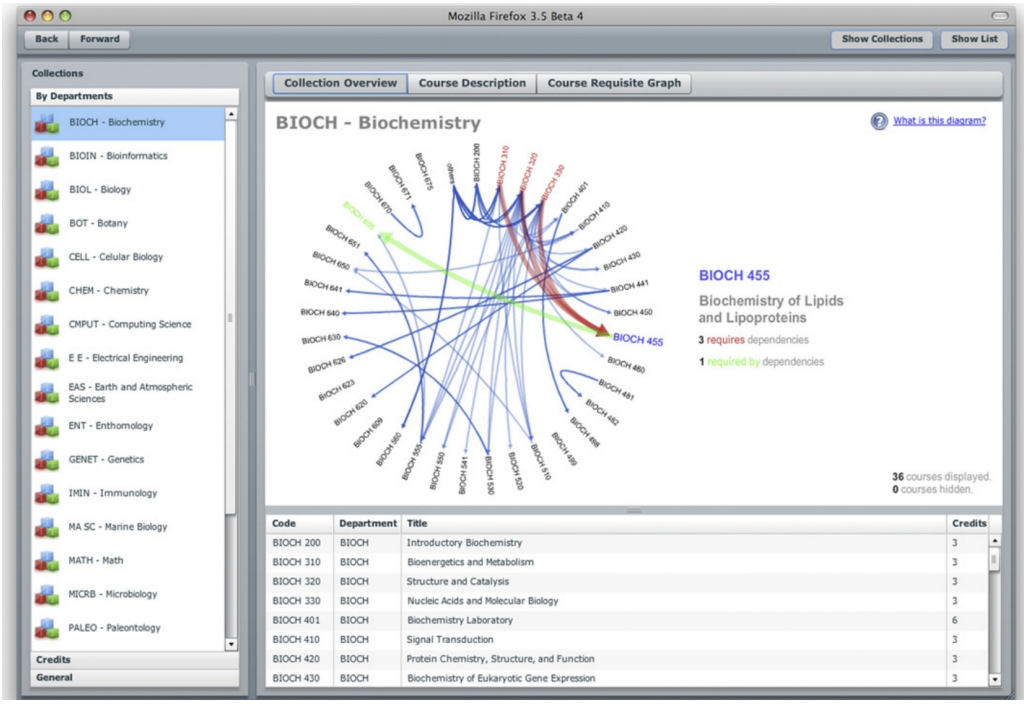
Autoři článku vytvořili prototyp nástroje pro vizualizaci závislostí předmětů. Nástroj je založen na *Flare toolkitu*, který umožňuje vytvářet interaktivní vizualizace a zobrazovat je pomocí *Flash pluginu* (což jej v dnešní době činí nepoužitelným) [34].

Každé prerekvizity a korekvizity mají přiřazené podmínky. Tyto podmínky mohou být takové, že student musí splnit všechny závislé předměty, jakýkoliv jeden předmět nebo minimálně N předmětů [34].

Navržený nástroj lze rozdělit na několik částí. První částí je prohlížeč předmětů, kde jsou předměty rozděleny do kolekcí podle jejich společných vlastností, kterou může být například fakulta. Prohlížeč předmětů lze vidět na obrázku 3.10. Po zvolení kolekce dojde k zobrazení dalších částí a tou je kruhová vizualizace závislostí předmětů v kolekci a seznam předmětů v kolekci. Po zvolení předmětu se uživateli zobrazí další dvě části nástroje. První je pohled na informace o předmětu se schránkovým digramem rekvizit předmětu. Druhou částí po zvolení předmětu je kruhový diagram závislostí předmětu [34].

Oba zmíněné kruhové diagramy jsou si velmi podobné. Předměty jsou poskládány v kruhu a jsou seřazeny abecedně. Jednotlivé závislosti jsou poté spojeny zahnutými šipkami. Po kliknutí na předmět dojde k červenému zvýraznění šipek požadovaných předmětů a zelenému zvýraznění šipek předmětů, které zvolený předmět vyžadují. Dojde také k vypsání počtu závislých

předmětů. Šipky v kruhových diagramech také mění svoji průhlednost podle toho, jakého typu podmínky závislost je a podle dalších parametrů [34].



Obrázek 3.10: Ukázka prohlížeče předmětů [34].

Autoři článku vytvořili uživatelskou studii pro otestování funkcionality nástroje a pro získání dalších informací. Účastníci uživatelské studie přijali vytvořený nástroj s tím, že jeho použití zrychluje hledání předmětů a zjednodušuje orientaci v závislostech. Navrhovaným vylepšením byla možnost textového vyhledávání, které autoři nestihli implementovat z časových důvodů [34].

Jelikož je článek z roku 2011 a pro zobrazení využívá *Flash plugin*, jehož podpora je v dnešní době již ukončena, byl by nástroj v popsané podobě spíše nepoužitelný a pro jeho implementaci by musela být zvolena jiná technologie.

3.6 Interaktivní vizualizace softwarových komponent pomocí virtuální reality

Díky dostupnosti relativně levných, ale kvalitních, brýlí pro virtuální realitu je možné zkoumat jejich využití i pro vizualizaci závislostí softwarových komponent [35]. Andreas Schreiber a Marlene Brüggemann v článku Interactive Visualization of Software Components with Virtual Reality Headsets [35]

vytvořili nástroj pro *OSGi* aplikace (které jsou založeny na komponentách), který umožňuje uživatelům prozkoumávat závislosti jednotlivých komponent právě pomocí virtuální reality [35].

Systémy modulárních elektrických komponent byly použity jako metafora pro vizualizaci závislostí komponent. V těchto systémech je možné jednotlivé komponenty upravovat pomocí přepínačů a sliderů, propojovat je mezi sebou a také jednotlivé komponenty odebírat [35].

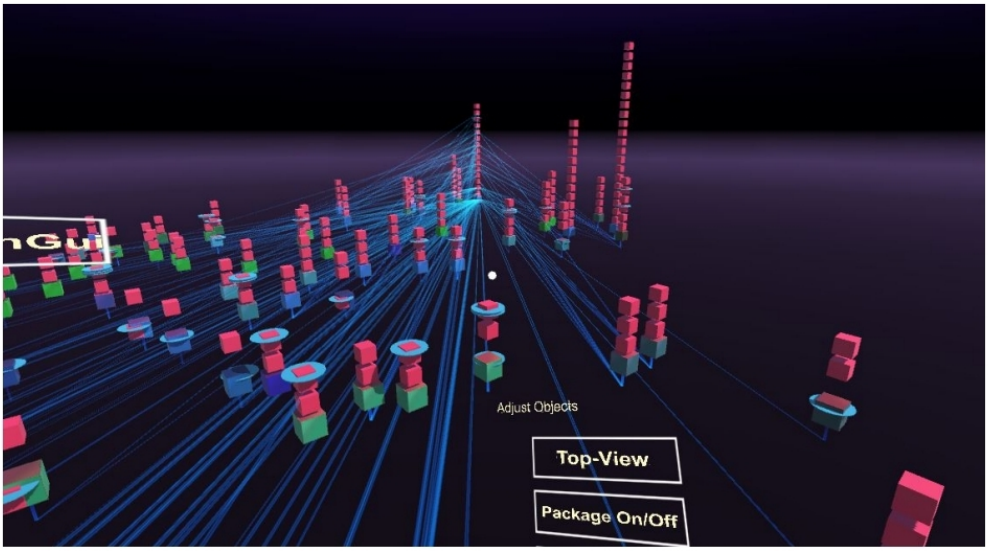
Balíky a jejich svazky jsou reprezentovány pomocí krychlí. Jednotlivé krychle jsou poté poskládány na sebe s tím, že spodní krychle reprezentuje svazek a má jinou barvu a velikost než krychle ostatní. Díky tomuto je jednoduché poznat do jakého svazku balík patří a také pomocí výšky kolik balíků daný svazek obsahuje [35].

Dále je nutné reprezentovat služby, které jednotlivé balíky poskytují. Ty jsou reprezentovány jako červené kruhy (disky), kolem jednotlivých balíků. V případě více služeb poskytovaných jedním balíkem dojde opět k jejich vyskládání na sebe s tím, že je mezi nimi mezera [35].

Dalším prvkem, který je možné zobrazit, jsou třídy obsažené v jednotlivých balících. Pro jejich reprezentaci byl zvolen válec s průhledným textem vedle něj, který udává její název. Pro zvolený balík se vždy vytvoří spirála válců, které poté reprezentují jeho všechny třídy. Spirála začíná ve zvoleném balíku a její další části mají vždy o něco větší poloměr a úhel, čímž je zajištěno to, že se jednotlivé třídy různých balíků neprotínají [35].

Pomocí hran jsou poté reprezentovány závislosti balíků a jejich svazků. Zobrazeny jsou hrany spojující importované balíky a balíky exportované a mají rozdílné barvy. Do jednoho svazku vedou hrany z balíků, které svazek importuje, a každý balík, který je svazkem exportován je spojen se svazky, které ho importují [35].

Rozložení jednotlivých prvků je možné uskutečnit několika způsoby. Prvním je náhodné rozložení a druhým je shlukování podle symbolických jmen svazků. Dále je možné prvky rozložit do kvadratického pole nebo do speciálně poskládaných řad tak, aby bylo vidět co nejvíce svazků. Je možné se také na scénu podívat shora pro zobrazení větší části vizualizace [35]. Ukázku grafického uživatelského rozhraní lze vidět na obrázku 3.11.



Obrázek 3.11: Ukázka vytvořené vizualizace se zobrazením závislostí zobrazené třídy [35].

Pro navigaci v prostoru byl vytvořen speciální režim, který spustí pohyb dopředu. Pohybem hlavy se poté mění směr pohybu v zobrazení [35].

Systém byl vytvořen pro *VR* brýle, které nepotřebují žádné další periferie, ovládání je tedy možné pouze tlačítkem přímo na *VR* brýlích [35].

Pro zvolení nějakého prvku ve vizualizaci je nutné se na tento prvek dívat delší dobu. Pro přesné zaměření slouží zaměřovač ve tvaru kruhu nebo křížku[35].

Mezi jednotlivé interaktivní prvky patří například možnost zobrazení nebo skrytí balíků ve svazku, možnost rotace scény, zobrazení nebo skrytí hran závislostí, možnost zobrazení pouze jednoho svazku a podrobnějších informací o něm a další [35].

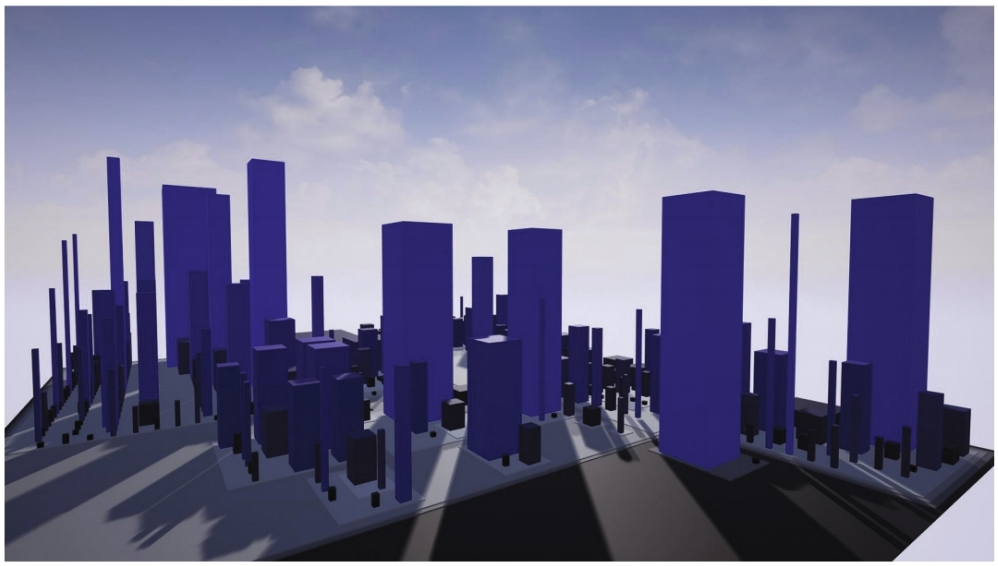
Nástroj byl implementován pomocí herního enginu *Unity* a byl vytvořen pro velké množství *VR* platforem a to i pro *Google Cardboard* [35].

Použití nástroje je vhodné hlavně pro získání přehledu o *OSGi* aplikaci, ale je možné ho rozšířit o více informací. Nejobtížnějším prvkem při použití *VR* technologií je navigace prostředím [35].

Na téma vizualizace software pomocí virtuální reality existuje velké množství článků. Je například možné vizualizovat software jako město (viz obrázek 3.12) [36, 37].

Ugo Erra, Nicola Capece, Simone Romano, a Giuseppe Scanniello právě toto v článku Visualising a Software System as a City Through Virtual Reality [36] popisují. Vygenerované město je podobné *tree map* grafu. Jednotlivé

balíky vytváří bloky, které jsou na sebe vrstvené, ale jejich výška není příliš veliká a je jednotná, a mají různé odstíny šedé barvy podle úrovně zanoření. Třídy jsou poté zobrazeny jako hranoly, které mají na rozdíl od balíků odstíny modré barvy v závislosti na počtu řádek kódu, kde tmavší znamená více řádek kódu. Rozměry hranolu jsou dány počtem atributů třídy a výška je dána počtem jejích metod. Zobrazení dále obsahuje značky s názvy balíků a tříd, u kterých se zároveň zobrazí i jejich metriky. Dále vizualizace pomocí ray tracingu umožňuje selekci tříd pro zobrazení detailních informací [36]. Článek ale nezmiňuje zobrazení závislostí a zároveň v něm není popsána žádná studie, pomocí které by mohlo být vyhodnoceno, jak užitečný takovýto přístup k vizualizaci je.



Obrázek 3.12: Ukázka vytvořené vizualizace v podobě města [36].

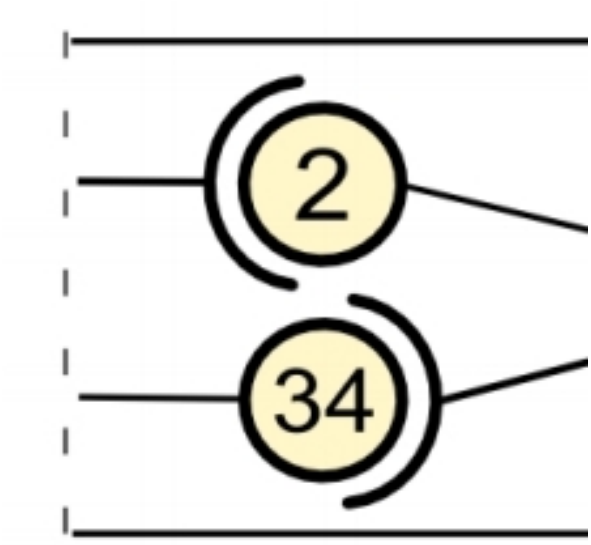
3.7 Interaktivní vizualizace založená na *UML* pro velké softwarové diagramy

UML (*Unified Modeling Language*) slouží pro vizualizaci závislostí tříd, komponent a dalších informací. Při velkém počtu tříd ale, stejně jako u dalších klasických forem vizualizace, tedy s hranami a uzly, dochází ke zhoršení orientace v grafu z důvodu velkého množství, často i překrývajících se, čar. Je také těžké zvolit jestli uživatel chce zobrazit celý diagram a nebo jen výběr těch nejdůležitějších informací, které není vždy snadné určit. Problémem *UML* také je, že v grafu nelze žádné informace schovávat a zobrazovat později, graf je totiž statický [38].

Pro řešení tohoto problému byla v článku An Interactive UML-like Visualization for Large Software Diagrams [38] od Lukáše Holého, Iva Malého, Ladislava Čmolíka, Kamila Ježka a Přemka Brady navržena notace založena na *UML* a vytvořen nástroj, který tuto notaci používá [38].

Jednou z částí nástroje je interaktivní zvýrazňování komponent a jejich propojení. Po zvolení komponenty se zvýrazní všechny hrany, které komponentu propojují s požadovanými komponenty, červeně a všechny hrany, které přichází od komponent, které komponentu vyžadují, modře. U ostatních komponent a hran dojde ke zmenšení jejich výraznosti. Nástroj také umožňuje textové vyhledávání komponent, při kterém taktéž dochází ke zvýraznění nalezených komponent [38].

Další částí nástroje je shlukování rozhraní. Zde byla změněna notace *UML* diagramu a to tak, že oproti klasickému provedení, kde každé poskytované rozhraní představuje socket do kterého se komponenty připojí, se v nástroji všechna komponentou poskytovaná rozhraní sloučí do jednoho propojení, viz obrázek 3.13. Dále dojde k odstranění zobrazení jmen rozhraní. Výpis všech rozhraní lze poté zobrazit kliknutím na sloučené rozhraní. Díky této funkcionalitě dojde k odstranění velkého množství propojení [38].

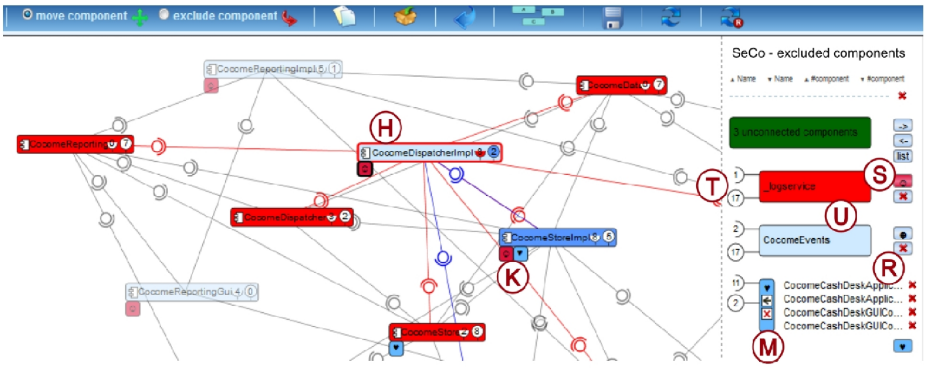


Obrázek 3.13: Ukázka socketu [39].

Odstranění komponent s nejvíce propojeními může způsobit odstranění velkého množství hran grafu a tím zlepšení jeho čitelnosti. Této skutečnosti využívá další část nástroje, která byla pojmenována oblast oddělených komponent. Nejvíce propojené komponenty a jejich hrany jsou tedy odstraněny z grafu, ale aby informace o jejich propojeních byly stále dostupné,

jsou přesunuty do oblasti oddělených komponent, která je vedle diagramu. Oblast oddělených komponent je seznam prvků, které jsou tvořeny shluky propojení (požadovaných, zobrazených jako kolečka a poskytovaných, zobrazených jako *sockety*), komponentami a symbolem prvku, který může být poté zobrazen u komponenty v grafu, čímž dá uživateli najevo, že tato komponenta je propojena s některou z oddělených komponent. Shluky propojení obsahují počty propojení. Mezi komponenty, které je vhodné do této oblasti přesunout, patří ty, které uživatel zná, nebo ty, které jsou propojeny skoro s každou komponentou a jejichž zobrazení tedy nemá příliš velký smysl [38].

Poslední funkcí nástroje je vytváření logických shluků komponent. Komponenty, které jsou použity pro implementaci jedné funkcionality aplikace nebo pro ně existuje jiný důvod, je možné přidat do jednoho společného prvku v oblasti oddělených komponent a tento prvek nějak pojmenovat. Tento prvek lze poté vrátit do grafu jako jednu speciální komponentu a zobrazit její hrany (propojení). U této speciální komponenty je ale možné provést další akce. Je možné její rozpuštění na jednotlivé komponenty nebo její zobrazení jako seznam komponent, u kterých je opět možné využití funkce interaktivního zvýrazňování. Speciální komponentu je také možné vrátit zpět do oblasti oddělených komponent [38].



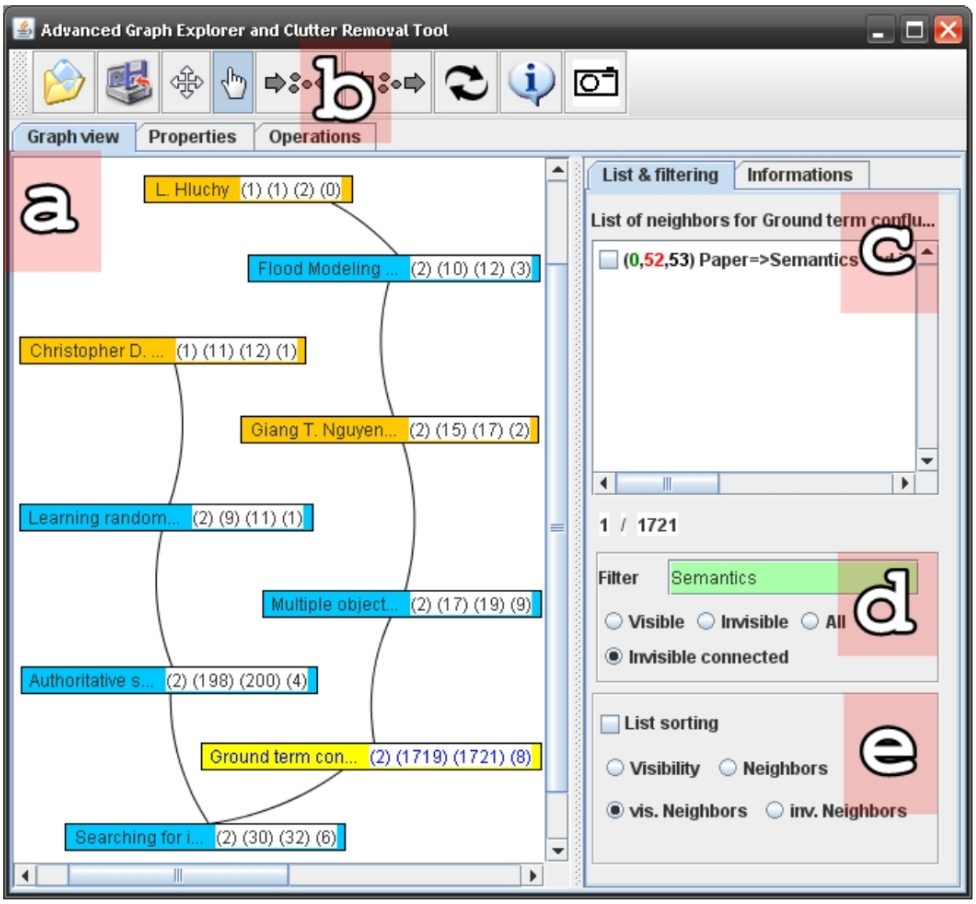
Obrázek 3.14: Ukázka uživatelského rozhraní vytvořeného nástroje [38].

Jedná se o webovou aplikaci, jejíž schopností je zobrazit graf závislostí komponent jakékoliv komponentně založené aplikace (musí být vytvořena v podporovaném komponentovém modelu). Pro vytvoření grafu používá aplikaci *ComAV* a ten zobrazí v prohlížeči pomocí *HTML5*. Na obrázku 3.14 lze vidět ukázku grafického uživatelského prostředí nástroje [38].

Byla provedena uživatelská studie, ve které došlo k porovnání vytvořeného nástroje a klasického *UML* diagramu, ve které bylo na vzorku *12* účastníků zjištěno, že práce s vytvořeným nástrojem je rychlejší než při použití klasického *UML* diagramu [38].

3.8 Filtrování nadbytečných uzlů založené na propojenosti a viditelnosti

Článek Graph clutter filtering based on connectivity distance and visibility [26] od autorů Jána Mojžiše a Michala Laclavíka se zabývá redukcí hran a uzlů v grafu pro zjednodušení práce s rozsáhlými grafy, ve kterých dochází k překrytí uzlů nebo hran dalšími hranami, čímž dochází ke zhoršení orientace v grafu. Dalším problémem může být příliš málo prostoru na zobrazovacím zařízení. Vytvořená metoda je založena na zobrazení hran mezi viditelnými uzly [26].



Obrázek 3.15: Zobrazení vytvořené aplikace. Část *a* zobrazuje hlavní obrazovku vizualizace, část *b* tlačítka pro rozložení a složení uzlu, část *c* seznam sousedů, část *d* filtr a část *e* řazení seznamu [26].

Cílem filtrace je, aby pro každý uzel byly vybrány pouze ty uzly, které mají alespoň jedno propojení s viditelným uzlem. Viditelný uzel je takový,

který je zobrazen na zobrazovacím zařízení [26].

Byla navržena pravidla, kde první říká, že v cestě mezi dvěma viditelnými uzly může být pouze jeden neviditelný uzel. Délka cesty mezi dvěma viditelnými uzly je aktuálně *2*. Druhé pravidlo říká, že pro každý uzel ze setu propojených sousedů lze nalézt set jeho viditelných sousedů [26].

Pro demonstraci filtrace byl vytvořen nástroj v programovacím jazyce *Java*, jehož grafické uživatelské rozhraní lze vidět na obrázku 3.15. Základem nástroje je framework *Jung*. Pomocí vytvořeného nástroje je množné zobrazit seznam sousedů uzlu s filtrováním nebo bez a informaci o jejich viditelnosti. U každého uzlu se zobrazí počet viditelných sousedů, počet neviditelných sousedů, celkový počet sousedů a počet neviditelných uzlů, které mají hranu s již viditelnými uzly. Uživatel může zvolit, jestli chce rozvinout uzel a zobrazit jeho sousedy nebo jeho sousedy zneviditelnit [26].

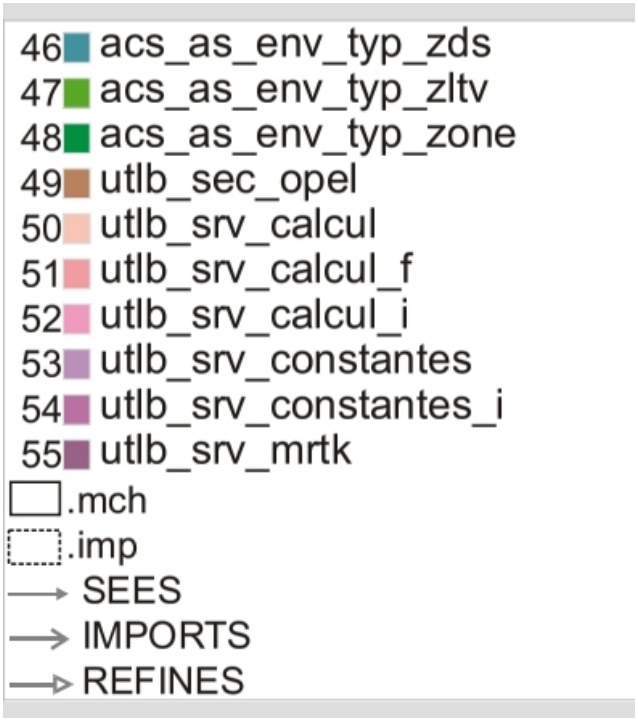
Z výsledků testování vyšlo najevo, že článkem popsaná metoda nemusí být vždy efektivní. Po kombinaci s dalšími filtry se ale její efektivita zvýšila [26].

3.9 Využití koláčových diagramů a stromové struktury pro vizualizaci rozsáhlých softwarových architektur

Mireille Samia a Michael Leuschel v článku Towards pie tree visualization of graphs and large software architectures [40] navrhují pro odstranění přílišného množství hran vizualizaci pomocí koláčových diagramů a stromové struktury. Cílem je zpřesnění vizualizace a zjednodušení pochopení struktury vizualizované architektury, a to hlavně díky přesnému zobrazení všech uzlů a všech jejich propojení [40].

Vizualizace se skládá z několika částí a to z uzlů, propojení mezi uzly, které je provedeno pomocí koláčových diagramů, a legendy [40].

Legenda, kterou je možné vidět na obrázku 3.16, obsahuje seznam všech uzlů a každému přiřazuje barvu a číslo. Různé barvy jsou použity pro různé druhy uzlů (například různé druhy modulů) a každý vrchol v této druhové skupině má jiný odstín dané barvy. Tímto je dosaženo sloučení uzlů podobných typů. Čísla byla uzlům přiřazena pro usnadnění rozlišování uzlů v případě uživatelů s poruchou barevného vidění. Dále jsou v legendě zobrazeny a popsány různé typy závislostí [40].

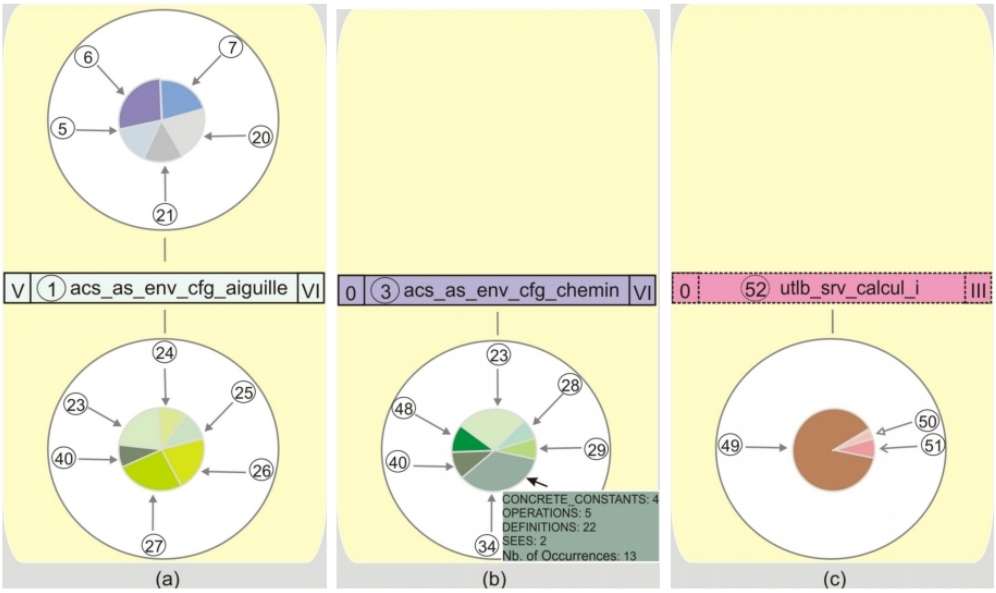


Obrázek 3.16: Legenda navržené vizualizace [40].

Každý uzel je v grafu zobrazen samostatně různým způsobem podle jeho typu. Například pokud se jedná o stroj, je zobrazen jako obdélník s nepřerušovanou obvodovou čarou. V obdélníku je pak římskými číslicemi zobrazeno, kolik vstupních (vlevo) a kolik výstupních (vpravo) propojení uzel obsahuje (v případě žádných propojení je použita *0*) [40].

Propojení jsou zobrazena v koláčovém diagramu, kde každý výřez představuje jeden uzel, se kterým má daný uzel propojení. Barva výřezu je stejná jako barva propojeného uzlu a jeho velikost reprezentuje nějakou metriku. U každého výřezu je ještě kruh (opět vykreslený nepřerušovanou čarou v případě, že se jedná o stroj), který obsahuje číslo propojeného uzlu a od tohoto uzlu vede šipka k výřezu. Šipka může mít různé styly zobrazení podle typu závislosti mezi uzly (viz legenda). Diagramy jsou nad nebo pod obdélníkem uzlu. Diagram nad uzlem zobrazuje vstupní propojení a digram pod uzlem zobrazuje výstupní propojení [40].

Po kliknutí na uzel je možné zobrazit souhrn informací o uzlu. Lze například zobrazit počet výskytů uzlu nebo počet proměnných v uzlu. Ukázku vizualizace lze vidět na obrázku 3.17 [40].



Obrázek 3.17: Zobrazené uzly (obdélníky), u každého uzlu jeho vstupní závislosti (nahoře), výstupní závislosti (dole) a u jednoho z uzlů zobrazení souhrnu informací [40].

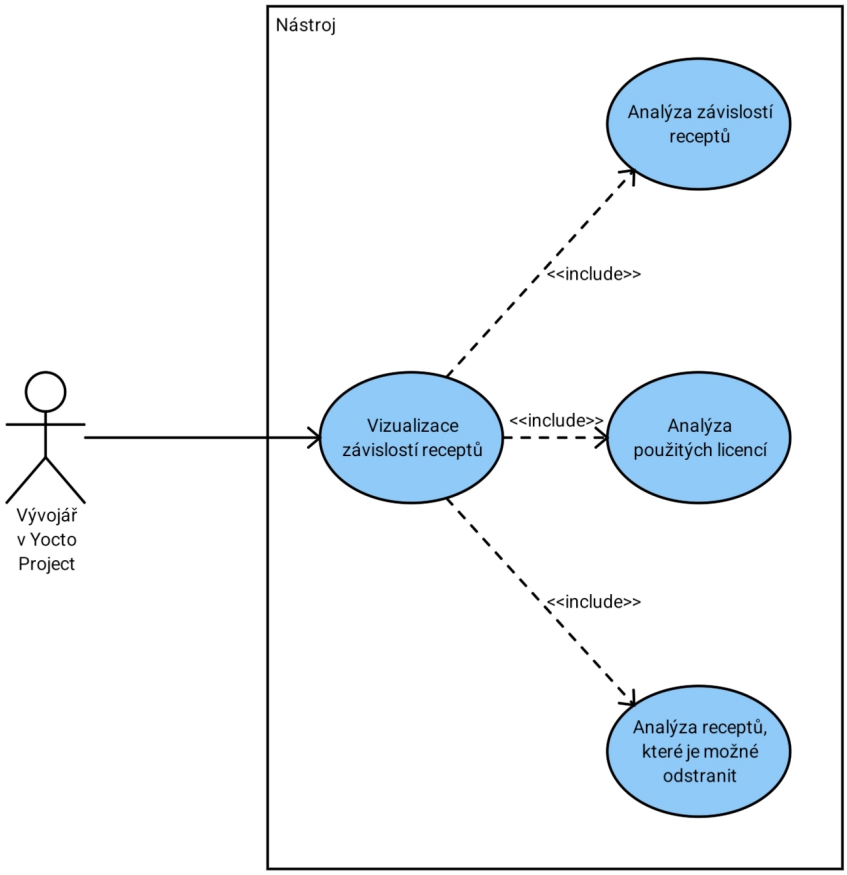
Popsané řešení může zpřehlednit práci s grafem softwarové architektury. Dalším rozšířením může být například přidání detailnějšího přehledu [40].

4 Návrh

Hlavním cílem této diplomové práce bylo navržení a vytvoření prototypu nástroje (dále jen nástroj) pro vizualizaci závislostí v receptech pro sestavení linuxového jádra v *Yocto Project*. Nástroj by měl sloužit vývojářům v *Yocto Project*, kteří by měli jeho pomocí získat možnost jednoduše sledovat závislosti mezi jednotlivými recepty a díky tomu analyzovat sestavení a následně jej vylepšovat.

4.1 Případy užití nástroje

Jednotlivé případy užití nástroje lze vidět na obrázku 4.1.



Obrázek 4.1: Diagram případů užití.

4.1.1 Analýza jednotlivých závislostí

Prvním a základním případem užití je klasické zobrazení závislých receptů, ve kterém bude možné jednotlivé recepty označovat a zobrazovat dodatečné informace o nich, zvýrazňovat recepty, se kterými má označený vrchol hranu, a nakonec pak jednotlivé recepty z vizualizace odstraňovat pro případné zlepšení orientace v grafu. Toto by mělo umožnit vývojáři základní analýzu závislostí receptů.

4.1.2 Analýza licencí jednotlivých receptů

Dalším případem užití bude analýza licencí jednotlivých receptů. Základní vizualizace by měla být stejná jako v prvním případě s tím rozdílem, že by místo zvýraznění vybraného receptu a receptů s ním propojených mělo docházet ke zvýraznění receptů podle typu licence specifikované v receptu. Uživatel by měl v tomto případu užití získat možnost zjistit, jak daný recept a jeho typ licence ovlivní celkové licencování výsledného sestavení.

4.1.3 Analýza receptů vhodných k odstranění

Posledním případem užití bude možnost určit, jak odstranění jednoho receptu celkově ovlivní sestavení systému. Stejně jako ve druhém případu užití by základ vizualizace měl být stejný jako v prvním případě užití s následujícím rozdílem - po označení jednoho receptu by mělo dojít k rekurzivnímu zvýraznění receptů, které tento recept vyžadují a které vyžadují zvýrazněné vyžadované recepty. Díky tomuto bude možné určit, které recepty nejvíce ovlivňují celkové sestavení a které recepty jsou vhodné k odstranění.

4.2 Základní návrh nástroje

Nejprve je nutné navrhnout, jakým způsobem se bude nástroj používat. Jako nejvhodnější se jeví tři možnosti.

První možností je implementace samostatné aplikace, kterou by uživatel spustil a pomocí které by vygeneroval vizualizaci. Druhou možností je implementace nástroje jako pluginu pro vývojové prostředí *Eclipse*. Uživatel by tak mohl vizualizaci vygenerovat přímo ve zmíněném vývojovém prostředí. Třetí možností je podobná druhé možnosti, s tím rozdílem, že zde by nástroj byl implementován jako rozšíření do open-source textového editoru *Visual Studio Code*.

Pro implementaci je nakonec zvolena možnost implementace nástroje jako rozšíření pro *Visual Studio Code*, jelikož bude uživateli umožněno zobrazení vizualizace přímo v prostředí ve kterém pracuje na sestavení systému. Zároveň *Visual Studio Code* umožňuje stažení rozšíření pro zvýraznění *BitBake* syntaxe. Další výhodou *Visual Studio Code* je jeho dostupnost na více platformách. Důvodem pro nezvolení druhé možnosti je ukončení vývoje *Yocto Project* pluginu pro *Eclipse IDE* [21]. První možnost nebyla zvolena z důvodu pro uživatele snadnějšího zobrazení vizualizace v prostředí ve kterém vyvíjí bez nutnosti přepínání se mezi jednotlivými programy.

Visual Studio Code pro vyvíjení rozšíření nabízí snadné nastavení prostředí a během vývoje umožňuje snadné testování vytvářeného rozšíření. Pro vývoj rozšíření využívá webové technologie *Node.js*. Nástroj bude vyvíjen v programovacích jazycích *TypeScript* a *JavaScript*.

4.3 Struktura projektu nástroje

Struktura projektu pro nástroj bude založena na struktuře, která je vytvořena po zavolání příkazu určeného pro prvotní nastavení vývojového prostředí *Visual Studio Code* rozšíření.

Dále pak zdrojový kód bude rozdělen do několika částí. První část se bude starat o získávání dat pro vizualizaci a další potřebné informace k zobrazení a bude implementována v jazyce *TypeScript*. Druhá část se bude starat o základní zobrazení, která budou popsána pomocí *HTML*, a bude také implementována v jazyce *TypeScript*. Třetí částí jsou *JavaScript* soubory, které se využívají v jednotlivých *HTML* zobrazeních. Komunikaci mezi *JavaScript* soubory a *TypeScript* soubory bude umožňovat *Visual Studio Code API* [41]. Čtvrtá část bude sloužit pro hlavní nastavení funkcionality rozšíření a pro projení jednotlivých částí dohromady a bude implementována v jazyce *TypeScript*. Poslední částí jsou pomocné funkce a proměnné, a je opět implementována v jazyce *TypeScript*.

V kořenové složce projektu bude také několik konfiguračních souborů, z nichž nejdůležitější bude soubor `package.json`, ve kterém dochází k definici použitých příkazů, kontextových menu, oken a dalších prvků.

4.4 Užívání nástroje

Vytvořený nástroj bude možné nainstalovat do *Visual Studio Code* na všech možných platformách. Jeho hlavní užití ale bude na linuxových distribucích užitých pro sestavování embedded linuxových systémů. Zároveň ale nástroj

bude podporovat práci s *WSLv2* na systému *Windows*.

Jelikož se jedná o prototyp, bude hlavní důraz nástroje kladen na výslednou vizualizaci a její další možnosti tak budou omezeny. Nástroj tedy například nebude schopný vytvářet vizualizace bez předem vytvořeného vstupního souboru pomocí nástroje *BitBake*.

Výslednou základní vizualizaci bude možné vytvořit pomocí příkazu z palety příkazů. Pokročilejší nastavení bude možné provést z bočního panelu v seznamu rozšíření.

Uživatel bude mít možnost zvolit si, jaký typ analýzy bude chtít ve výsledné vizualizaci provádět, podle případů užití popsaných v kapitole 4.1, případně nastavit další parametry vizualizace, jako je například možnost zvolit si, jakého typu *BitBake* úkolu budou závislosti vizualizovány. Podle zvolené možnosti poté dojde k vygenerování grafu s danými druhy zvýrazňování prvků, kde každý bude umožňovat odstraňování receptů z grafu a jejich vrácení zpět, a dále možnost otevřít recept daného prvku grafu.

Nástroj zatím nebude umožňovat žádné ukládání práce a zobrazení více grafů najednou.

4.5 Návrh vizualizace

Vizualizace si jako vzor bere řešení z článku An Interactive UML-like Visualization for Large Software Diagrams [38] od Lukáše Holého, Iva Malého, Ladislava Čmolíka, Kamila Ježka a Přemka Brady. Zobrazen tedy bude graf s orientovanými hranami a vrcholy, kde jednotlivé vrcholy budou reprezentovat recepty a hrany závislosti mezi nimi. Vrcholy bude možné odebírat a vracet zpět. Vrcholy budou reprezentovány pomocí obdélníků s názvem receptu uvnitř. Hrany pak budou reprezentovány čárami s šipkami.

Teoreticky by bylo možné použít i vizualizaci pomocí samotného *DOT* formátu, ale jelikož je vygenerovaný graf příliš velký, bylo obtížné jej v již existujících nástrojích zobrazit, a protože snahou bylo vytvořit vizualizaci co nepřehlednější, bylo rozhodnuto, že se takovýto postup nepoužije.

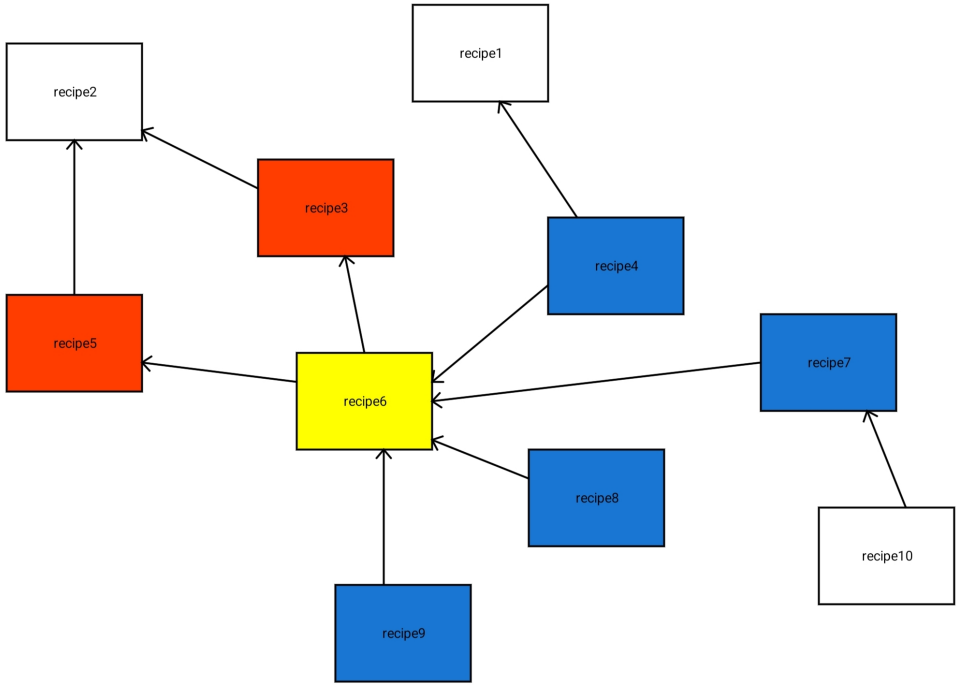
Jelikož se rozšíření pro *Visual Studio Code* vytvářejí pomocí webových technologií, bude pro vytvoření grafu použita knihovna *D3.js* a pro rozvržení grafu bude použit *force-directed* algoritmus přímo zabudovaný ve zmíněné knihovně. Uživatel bude mít možnost nastavit jednotlivé parametry algoritmu.

Zároveň se po zvolení vrcholu zobrazí podrobnější informace o něm, minimálně tedy dojde k zobrazení cesty k receptu a licence. Uživatel také bude moci otevřít samotný soubor receptu pro zvolený vrchol.

Dalším řešením pro eliminaci přílišného množství hran bude uživatelova možnost zvolit si, pro jaký typ *BitBake* úkolu se graf vytvoří. Nevýhodou takového řešení ale může být nutnost analýzy většího počtu grafů a tím pádem zhoršení orientace v systému. Dosažené snížení počtu hran by ale na druhou stranu mělo orientaci v grafu zjednodušit.

4.5.1 Základní vizualizace

Základní vizualizace bude umožňovat označení vrcholů kliknutím na ně, přičemž dojde ke zvýraznění zvoleného vrcholu žlutou barvou a ke zvýraznění vrcholů, které zvolený vrchol vyžaduje, barvou modrou a vrcholů, které zvolený vrchol vyžadují, barvou červenou. Tato vizualizace bude sloužit pro základní analýzu závislostí. Návrh základní vizualizace lze vidět na obrázku 4.2.

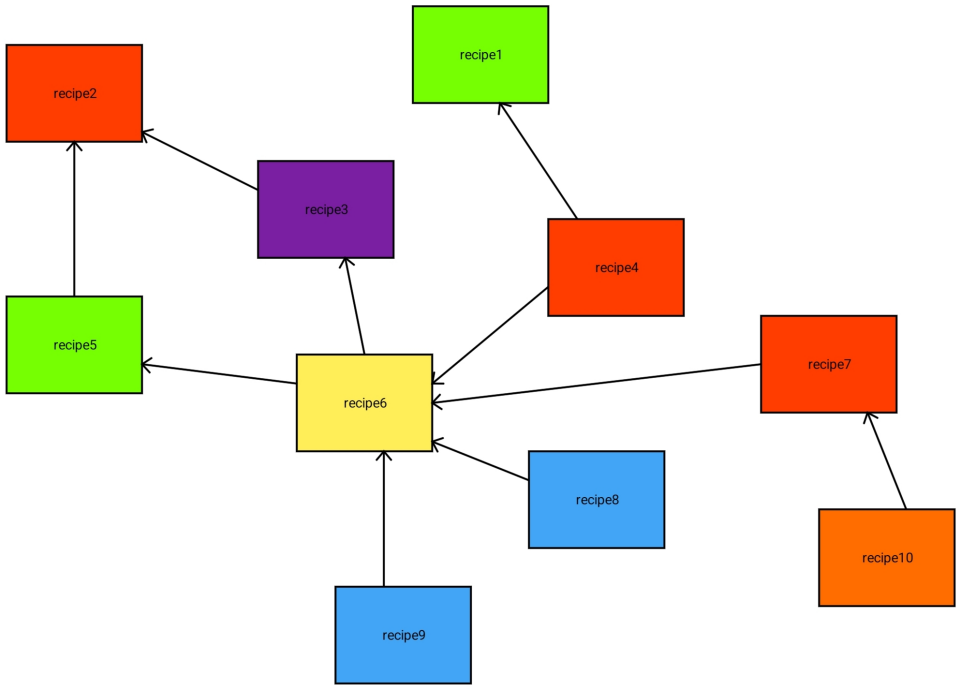


Obrázek 4.2: Návrh základní vizualizace.

4.5.2 Označení licencí

Dalším typem vizualizace, který bude nástroj umožňovat, bude zvýraznění vrcholů podle licencí jejich receptů. Zde bude vizualizace různými barvami obarvovat vrcholy podle deseti nejčastějších licencí. Ostatní vrcholy budou označeny základní barvou. V tomto případě nebudou po zvolení vrcholu

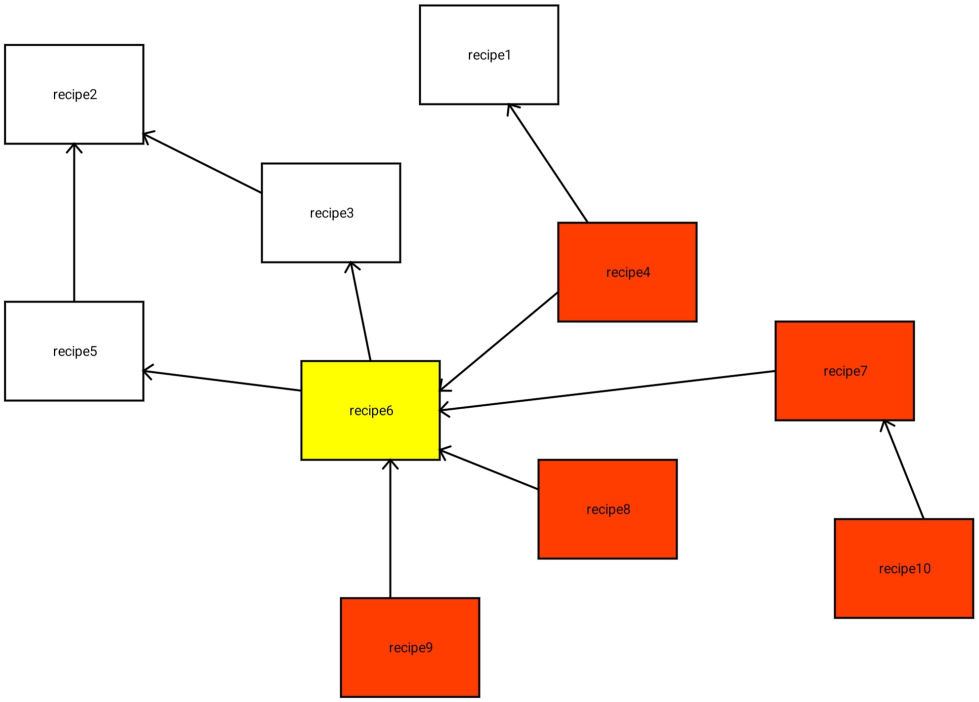
zvýrazněny další závislé vrcholy, jelikož tato funkcionalita bude sloužit pouze pro analýzu licencí. Tento typ vizualizace lze vidět na obrázku 4.3.



Obrázek 4.3: Návrh vizualizace zvýraznění licencí.

4.5.3 Rekurzivní zvýraznění receptů

Posledním typem vizualizace bude možnost zvolení jednoho vrcholu, přičemž dojde ke zvýraznění zvoleného vrcholu žlutou barvou. Dále pak dojde ke zvýraznění všech vrcholů, které na zvoleném vrcholu závisí červenou barvou. Tato akce je rekurzivní. Pro každý závislý vrchol a jeho všechny závislé vrcholy se tedy akce provede znovu. Výsledkem bude obarvení červenou barvou všech vrcholů, které přímo i nepřímo závisí na zvoleném vrcholu (viz obrázek 4.4). Pomocí tohoto typu vizualizace bude možné zkoumat, jak odstranění jednoho receptu z konfigurace ovlivní celkové sestavení systému.



Obrázek 4.4: Návrh vizualizace pro rekurzivní zvýraznění receptů.

4.6 Získávání dat

Jak již bylo zmíněno v kapitole 4.4, nástroj nebude umožňovat automatické získávání dat z daného *Yocto Project* projektu. Jako vstup nástroje bude použit *.dot* soubor vytvořený programem *BitBake*, který obsahuje všechny potřebné informace.

Vygenerovaný *.dot* soubor obsahuje graf v *DOT* formátu, který reprezentuje závislosti mezi jednotlivými *BitBake* úkoly. Dále obsahuje záznam úkolu s elementem *"label"*, který uchovává další informace, z nichž nejdůležitější je cesta k receptu.

Dalším zdrojem informací budou i samotné recepty.

4.6.1 Získávání dat k vizualizaci

Jako data pro vizualizaci budou považovány řádky z *.dot* souboru ve formátu:

$$\textit{"recipe1.do\_task"} \text{ -> } \textit{"recipe2.do\_task"},$$

kde *recipe1* and *recipe2* jsou dva rozdílné recepty a *do_task* může být libovolný typ úkolu. Dále budou důležité záznamy ve formátu:

$$\textit{"recipe.do\_task" [label="recipe do\_task\nversion\n/path\_to\_recipe"]},$$

kde *recipe* je libovolný recept, *version* je verze receptu a */path_to_recipe* je cesta k receptu. Před cestou k receptu také ještě může být řetězec *"virtual:native:"*.

Soubor v *DOT* formátu tedy bude načten a jeho řádky budou postupně procházeny. Podle uživatelem zvoleného typu *BitBake* úkolu budou z jednotlivých řádků získávány informace, které poslouží pro vytvoření reprezentace grafu. Budou tedy brány v potaz pouze řádky, kde na levé straně bude *do_task* podle zvoleného úkolu. Dále budou zpracovávány řádky s cestou receptu, která bude vždy uložena v instanci reprezentující daný recept.

Speciálním a také základním případem ale bude vytvoření grafu z řádek tak, že na levé straně řádky reprezentující závislost bude muset recept mít úkol typu *"do_prepare_recipe_sysroot"* a na pravé straně úkol typu *"do_populate_sysroot"*. Tyto řádky by měly reprezentovat závislosti receptu na levé straně právě procházené řádky podle atributu *DEPENDS*, který by měl specifikovat, na jakých receptech první recept závisí. Závislosti by se tak měly přiblížit hodnotám z *Yocto Project* nástroje *Toaster*.

Zpracovaná data budou ukládána do objektů, ze kterých se nakonec vytvoří *JSON* řetězec, který bude použit jako vstup pro *D3.js* knihovnu.

4.6.2 Získávání dat z receptů

Soubory receptů v sobě mají mimo jiné nastavené proměnné ve formátu:

$$\textit{variable} = \textit{"value"}$$

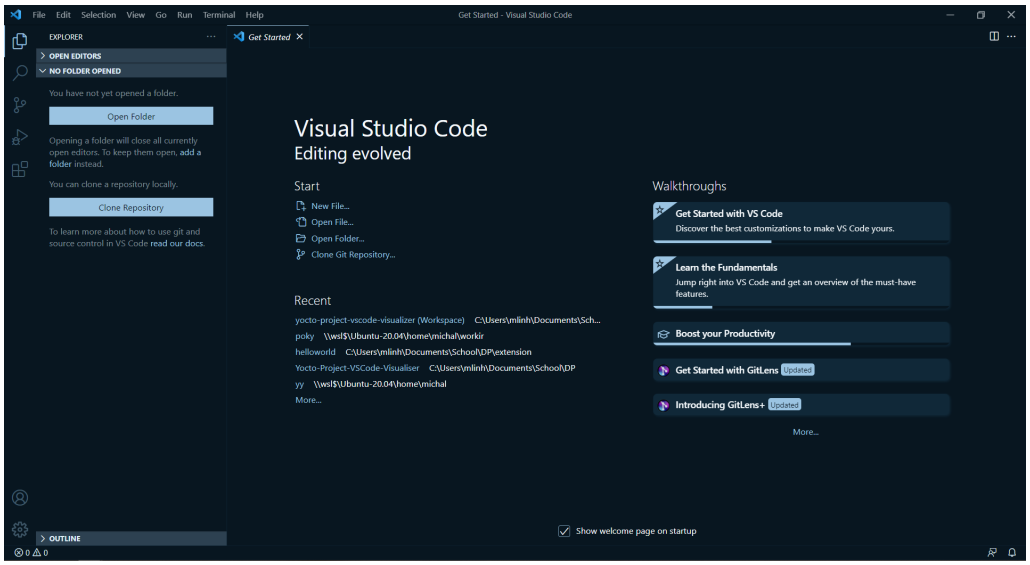
nebo

$$\textit{variable} \mathrel{+}= \textit{"value"}.$$

Bude proto možné jednotlivé recepty načíst a opět procházet po řádkách a získávat o nich dodatečné informace. Nejdůležitější proměnnou pro nástroj

bude proměnná *LICENSE*, která obsahuje informace o licenci receptu. Řádek s takovou proměnnou tedy bude zpracován. Licence budou ale brány takové, jaké jsou specifikované v receptech. Může se tedy stát, že recept bude specifikovat více licencí, nebo bude svou licenci odkazovat na jiný recept a nebo v případě, že se bude skládat z více částí, může mít každá část jiné licencování. Dalšími zajímavými proměnnými vhodnými k zobrazení uživateli by mohly být *DESCRIPTION*, *HOMEPAGE* nebo *SUMMARY*.

4.7 Návrh uživatelského rozhraní

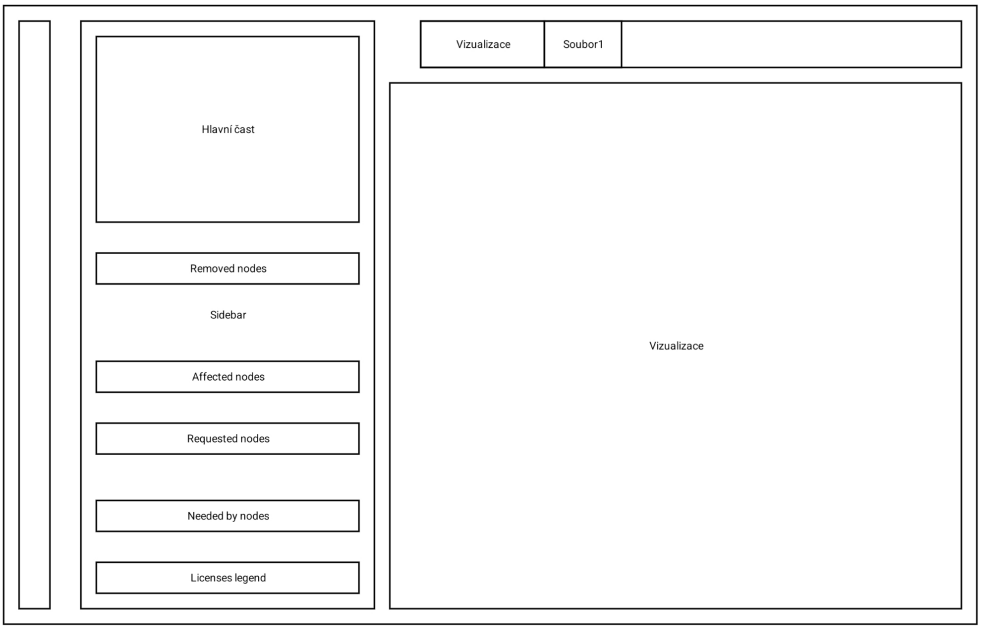


Obrázek 4.5: Ukázka grafického uživatelského rozhraní *Visual Studio Code*.

Jelikož je nástroj implementován jako rozšíření pro *Visual Studio Code*, bude jeho grafické uživatelské rozhraní z velké míry určeno tímto programem. Grafické uživatelské rozhraní *Visual Studio Code* je možné vidět na obrázku 4.5. Nástroj bude složen z několika částí. První a hlavní částí bude záložka se samotnou vizualizací, která bude otevřena po žádosti uživatele o vygenerování grafu. Druhou částí bude boční menu, neboli sidebar. V tomto menu bude uživateli nabídnuta možnost nastavit parametry vizualizace a další informace, a také zde budou stromová zobrazení (pouze s jednou úrovní), která budou obsahovat různé seznamy receptů. Zároveň zde bude legenda k vizualizaci licencí. Jednotlivé položky v sidebaru bude možné přesunout do pravé části programu, různě je skrýt, nebo přeskládat tak, jak to umožňuje *Visual Studio Code*. Návrh rozložení jednotlivých částí lze vidět na obrázku 4.6.

4.7.1 Hlavní menu

Hlavní menu bude obsahovat možnosti nastavení parametrů *force-directed* algoritmu, který je součástí knihovny *D3.js*. Dále bude obsahovat možnost nastavení typu *BitBake* úkolu, pro který se vygenerují propojení mezi jednotlivými recepty. Další částí hlavního menu bude možnost zvolit si, jaký typ vizualizace uživatel chce vytvořit, tedy základní, pro analýzu licencí a pro analýzu odstranitelných receptů. Pro vygenerování vizualizace bude v hlavním menu tlačítko.

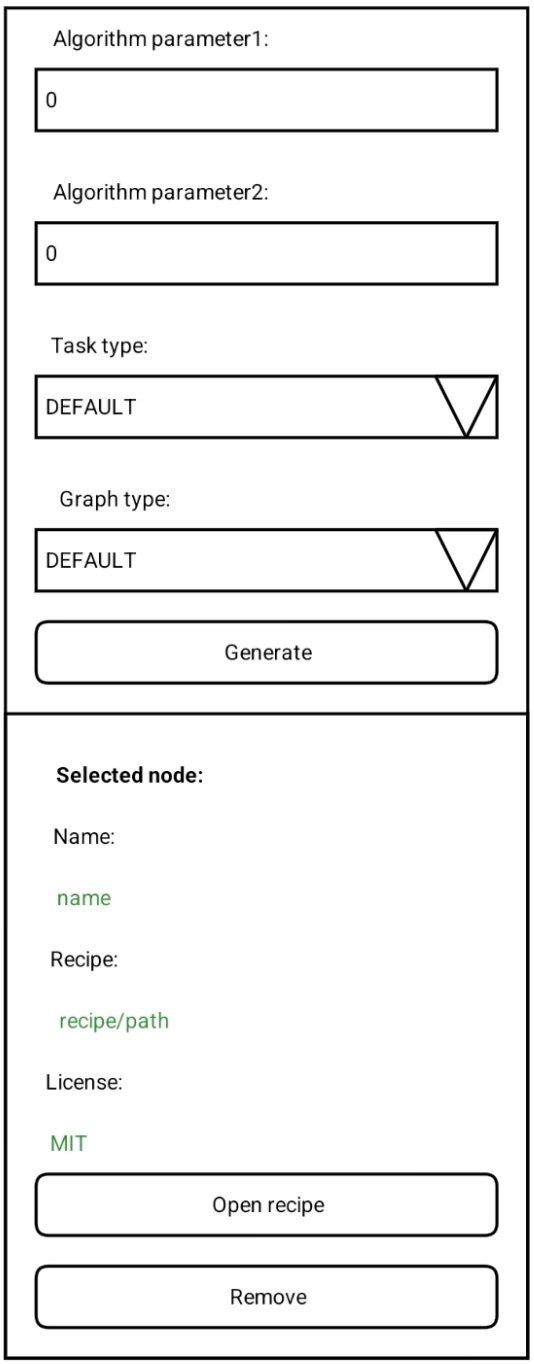


Obrázek 4.6: Návrh rozložení grafického uživatelského rozhraní nástroje.

Druhou částí hlavního menu budou informace o zvoleném receptu. Po kliknutí na vrchol v grafu se zde zobrazí informace o názvu, receptu, cestě k samotnému souboru receptu a informace o použité licenci. Pod těmito informacemi budou tlačítka pro otevření souboru receptu v nové záložce a pro odstranění vrcholu z vizualizace.

Pro vytvoření základní vizualizace také bude sloužit příkaz z palety příkazů *Visual Studio Code*, kterou lze obvykle zobrazit kombinací kláves *CTRL+SHIFT+P*, nebo klávesou *F1*.

Celý návrh hlavního menu je zobrazen na obrázku 4.7.



Obrázek 4.7: Návrh hlavního menu.

4.7.2 Seznamy receptů

Kromě hlavního menu bude sidebar obsahovat několik seznamů receptů. Mezi tyto recepty bude patřit seznam odstraněných receptů, seznam vyža-

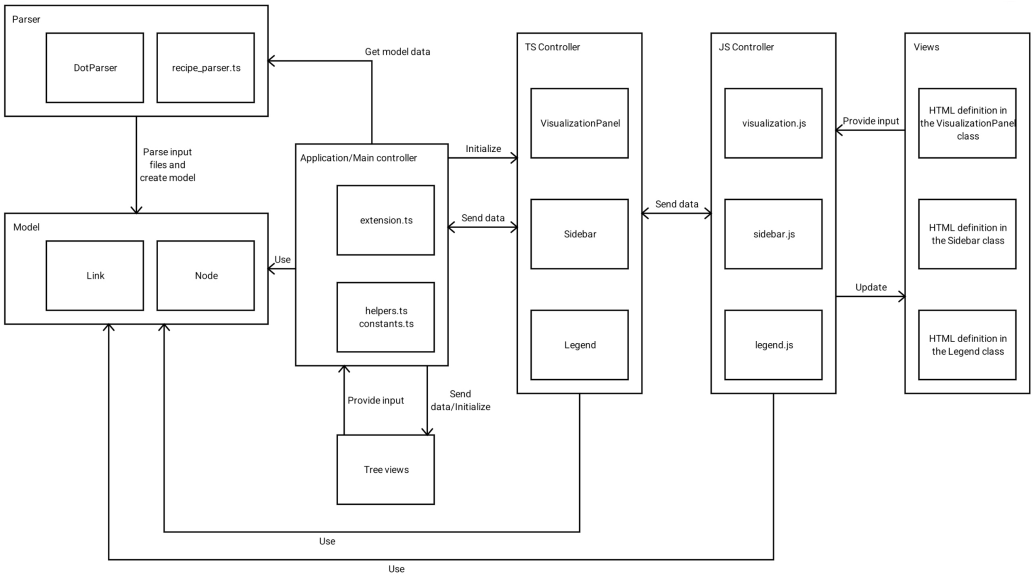
dovaných receptů pro zvolený recept, seznam receptů vyžadujících zvolený recept a seznam receptů, které by byly ovlivněny odstraněním zvoleného receptu. Každý recept v těchto seznamech bude mít v kontextovém menu možnost otevření souboru daného receptu. Recepty ze seznamu odstraněných receptů bude možné vracet zpět do vizualizace, opět pomocí kontextového menu. U receptů vyžadovaných zvoleným receptem a receptů vyžadujících zvolený recept bude možnost je zvolit a zvýraznit ve vizualizaci přímo z těchto seznamů. Seznam receptů, které by byly ovlivněny odstraněním zvoleného receptu bude viditelný pouze tehdy, zvolí-li uživatel analýzu receptů vhodných k odstranění.

4.7.3 Legenda

Pokud uživatel zvolí jako typ vizualizace analýzu licencí, dojde k zobrazení legendy, ve které bude označeno, jaká barva patří jakému typu licence. Toho bude dosaženo tak, že dojde k vypsání deseti nejpoužívanějších licencí příslušnou barvou písma.

5 Implementace

Jak již bylo zmíněno v kapitole 4.2, nástroj byl implementován v jazycích *TypeScript* a *JavaScript* a hlavní konfigurace byla provedena v *JSON* souboru. Zdrojové soubory jsou uloženy ve složce *src* a jsou rozděleny do několika podsložek. Nejdůležitějšími podsložkami jsou složky *view*, *parser*, *model*, *tree_providers*, a *support*. Dále složka *src* obsahuje hlavní *TypeScript* soubor `extension.ts`. Ve složce *view* jsou obsaženy soubory, které reprezentují jednotlivé části grafického uživatelského rozhraní (pro svou funcionalitu využívají *WebViews*, které zobrazují *HTML* obsah), ve složce *tree_providers* jsou poté soubory, které zajišťují funkcionalitu seznamů receptů. Složka *parser* obsahuje funkcionalitu získávání dat z *.dot* souboru a ze souborů receptů. Složka *model* pak obsahuje třídy reprezentující jednotlivé datové prvky vizualizace (vrcholy a hrany). Složka *support* obsahuje pomocné skripty a definice konstant. Složka *src* také obsahuje složku *test* ve které jsou implementované testy. Další důležitou složkou je pak složka *media*, která mimo jiné obsahuje *JavaScript* soubory použité v jednotlivých souborech ze složky *view*. Celý nástroj nese název *Yocto Project Dependency Visualizer*. Implementace také využívá několik knihoven, které jsou staženy pomocí manažeru balíčků *npm*. Logickou strukturu implementace nástroje lze vidět na obrázku 5.1.



Obrázek 5.1: Struktura implementace nástroje.

Pro inspiraci během implementace byly použity ukázkové implementace rozšíření z oficiálního *Git* repositáře [42] a video na platformě *YouTube* od autora Ben Awad [43].

5.1 Založení a struktura projektu

Pro založení projektu je nutné nainstalovat pomocí *npm Yeoman* [44] a generátor rozšíření pomocí příkazu n`npm install -g yo generator-code`. Po nainstalování je možné zavolat příkaz `yo code` pomocí kterého lze vygenerovat základní strukturu projektu.

Jednotlivé kroky generátoru nabízí možnost zvolení čeho se projekt týká a v jakém jazyce bude vytvořený, nastavení jména projektu, identifikátoru projektu (může být stejný jako jméno) a popisu projektu. Dále pak nabízí možnost inicializace *Git* repositáře, možnost zvolit, zda bude zdrojový kód zabalen pomocí balíčku *webpack* (zvoleno ano pro zmenšení počtu souborů ve výsledném balíku rozšíření) a zvolení manažeru balíčků (zvolen *npm*). Po nastavení možností dojde k vytvoření projektu a dotazu, zda chce uživatel tento projekt otevřít.

Po založení obsahuje projekt tři složky a několik souborů.

Složka *.vscode* obsahuje konfigurační soubory pro *Visual Studio Code* jako je například soubor *launch.json*, který definuje, jak spustit rozšíření a jak spustit testy. V souboru *tasks.json* jsou pak definovány úkoly, které jsou použity v souboru *launch.json*. V souboru *settings.json* je pak nastavení pro samotné *Visual Studio Code*. Soubor *extensions.json* pak obsahuje seznam doporučených rozšíření. Složka *node_modules* obsahuje stažené *Node.js* moduly. Poslední složkou je složka *src*, která obsahuje zdrojové soubory a testy.

Mezi soubory v kořenové složce projektu patří například soubor s popisem rozšíření *README.md*, *CHANGELOG.md* s popisem změn verzí, *tsconfig.json* pro *TypeScript* nastavení a další konfigurační soubory. Nejdůležitějším souborem je soubor *package.json*, ve kterém jsou definovány informace o projektu (jméno, verze, apod.), aktivační události rozšíření, cesta k hlavnímu zdrojovému souboru rozšíření, příkazy, které se v rámci rozšíření volají, jednotlivé vizuální prvky (*views*, *menus*, apod.), různé skripty a vývojové závislosti a další.

5.2 Třídy nástroje

Hlavní implementace nástroje (bez složky *test*) obsahuje celkem 10 tříd a jedno rozhraní. Všechny třídy a rozhraní jsou implementované v jazyce

TypeScript. Zdrojové *.ts* soubory, které je obsahují mají stejná jména jako třídy, tedy například třídu `DotParser` lze nalézt v souboru *DotParser.ts*.

5.2.1 Třídy `DotParser`, `Node`, `Link` a rozhraní `GraphElement`

Třídu `DotParser` je možné najít ve složce *parser* a obsahuje funkcionalitu sloužící pro získávání dat z *.dot* souboru. Obsahuje metody pro postupné procházení *.dot* souboru podle zadaných parametrů a ukládání relevantních dat do dalších datových struktur, ze kterých nakonec vygeneruje *JSON* řetězec.

Mezi datové struktury, které mimo jiné třída `DotParser` využívá, patří třídy `Node`, `Link` a `GraphElement` ze složky *model*. Třída `Node` slouží pro reprezentaci vrcholů, nebo-li receptů, třída `Link` pak reprezentuje propojení mezi jednotlivými vrcholy. Rozhraní `GraphElement`, které nic neobsahuje, je implementováno třídami `Node` a `Link` a slouží pro sjednocení těchto dvou tříd tak, aby je bylo možné uložit do jedné mapy společně.

5.2.2 Třídy `ConnectionsTreeDataProvider`, `RemovedTreeDataProvider` a `NodeTreeItem`

Třídy `ConnectionsTreeDataProvider` a `RemovedTreeDataProvider` jsou si velmi podobné a obsahují funkcionalitu využitou pro seznamy receptů. Obsahují metody pro přidávání a odstraňování prvků ze seznamu a pro vyčištění celého seznamu. Obě třídy lze nalézt ve složce *tree_providers*.

V téže složce lze také nalézt třídu `NodeTreeItem`. Tato třída slouží pro reprezentaci prvku ze seznamu receptů. Obsahuje tedy data, která se mají u jednotlivých prvků zobrazit, jako je například název receptu, nebo *tooltip* po najetí myší na prvek.

5.2.3 Třída `VisualizationPanel`

Třída `VisualizationPanel` se nachází ve složce *view* a jedná se o implementaci panelu pro vizualizaci. Je v ní definován jeho základní vzhled pomocí *HTML*, ve kterém je použit *JavaScript* soubor `visualization.js` a také obsahuje metodu ve které je implementováno přijímání zpráv od daného *JavaScript* souboru. Dále obsahuje metody pro otevření a skrytí okna.

5.2.4 Třídy `Sidebar` a `Legend`

Další ze tříd je třída `Sidebar`, kterou lze nalézt ve složce *view*. Jejím úkolem je zobrazení hlavního menu s možnostmi nastavení parametrů vizualizace, tlačítkem pro vygenerování vizualizace a s informacemi o zvoleném receptu z vizualizace a dalšími akcemi pro zvolený recept. Stejně jako ve třídě `VisualizationPanel` obsahuje definici komunikace s *JavaScript* souborem, který je použit v její *HTML* definici.

Třída `Legend` je, stejně jako třída `Sidebar`, uložena ve složce *view*. Tato třída slouží pro zobrazení legendy při zvolení analýzy licencí jako typu grafu. Opět obsahuje *HTML* definici a metody, které slouží pro odesílaní dat o legendě do příslušného *JavaScript* souboru.

5.3 Získávání dat

Jak již bylo zmíněno, získávání dat je implementováno ve složce *parser*. Samotná implementace je provedena ve třídě `DotParser`. Další část implementace získávání dat lze najít ve skriptu `recipe_parser.ts`.

5.3.1 Získávání dat pro graf

Získávání dat pro graf je implementováno v metodách `parseDotFile()`, `parseDotFileDefault()` a `parseDotFileTaskType()` ve třídě `DotParser`. Metoda `parseDotFile()` pouze volá zbylé dvě metody podle zadaných parametrů. Metoda `parseDotFileDefault()` je volána nezvolí-li uživatel specifický *BitBake* úkol, jinak je zavolána metoda `parseDotFileTaskType()`, které se jako parametr předá zvolený *BitBake* úkol.

V metodách `parseDotFileDefault()` a `parseDotFileTaskType()` dále dojde k otevření *.dot* souboru, jehož řádky jsou potom postupně procházeny. Řádky jsou poté rozděleny pomocí metody `split()` podle řetězce *" -> "*, přičemž dojde k uložení jednotlivých částí do pole. Pokud část na indexu *0* obsahuje řetězec *"do_prepare_recipe_sysroot"* v případě metody `parseDotFileDefault()` nebo řetězec s hodnotou uživatelem zvoleného *BitBake* úkolu v případě metody `parseDotFileTaskType()`, a pokud zároveň část na indexu *0* neobsahuje řetězec *"label="* v případě obou metod a část na indexu *1* obsahuje řetězec *"do_populate_sysroot"* v případě metody `parseDotFileDefault()`, dojde ke zjištění jmen závislých receptů. Název prvního receptu se zjistí z první části rozdělené řádky a to tak, že se řetězec *".do_prepare_recipe_sysroot"*, nebo řetězec s názvem *BitBake* úkolu s tečkou před ním v případě metody `parseDotFileTaskType()`, na-

hradí prázdným řetězcem a také dojde k odstranění uvozovek. Jméno druhého receptu se v případě metody `parseDotFileDefault()` získá obdobným způsobem jako jméno prvního receptu, pouze se zde nahrazuje řetězec *".do_populate_sysroot"*. V případě metody `parseDotFileTaskType()` dojde k rozdělení druhé části řádky metodou `split()` podle řetězce *"."*, z tohoto rozdělení se použije část na indexu *0* a u té dojde k odstranění uvozovek. Tuto část lze najít popsanou pseudokódem v algoritmu 1.

Pokud se názvy receptů nerovnají dojde k jejich případnému přidání do seznamu vrcholů jako instance třídy `Node` s novým *ID*, nebo ke zjištění jejich *ID*. Podle těchto *ID* je poté vytvořena instance třídy `Link`, která reprezentuje závislost prvního receptu na receptu druhém. Tato instance je nakonec přidána do seznamu hran.

Pokud první část řádky obsahuje řetězec *"label="*, dojde k rozdělení řádky podle řetězce *" "*. Pro zjištění jména příslušného receptu poté dojde k rozdělení první části řádky podle řetězce *"."* a následně k odstranění uvozovek. Jako zdrojový řetězec pro získání cesty k receptu slouží část řádky na indexu *2*. Z té jsou odstraněny nepotřebné řetězce a výsledek je rozdělen podle řetězců *"\n"* a *"/"*. Z tohoto rozdělení je použita poslední část, jelikož reprezentuje cestu k souboru receptu. Následně dojde k načtení instance třídy `Node` daného receptu ze seznamu vrcholů pokud již existuje, nebo k vytvoření nové s novým *ID* a přiřazení cesty k receptu dané instanci třídy `Node`. Zároveň se rozhodne, zda uživatel zvolil režim analýzy licencí a pokud ano, dojde k načtení licence daného receptu. Pokud ne, licence se načítají až v případě zvolení vrcholu uživatelem pro snížení času vytváření grafu. Tato část metody je popsána pseudokódem v algoritmu 2.

Seznamy vrcholů a hran se poté přidají do mapy. Výstupem metod `parseDotFileDefault()` a `parseDotFileTaskType()` je *JSON* řetězec, jež je vygenerovaný z této mapy.

Jako příklad mějme následující obsah *.dot* souboru, ze kterého chce uživatel získat informace o závislostech pro *BitBake* úkol *do_build*:

```
digraph depends {
"acl.do_build"[label="acl
do_build\n:2.2.53-r0\n/path/acl_2.2.53.bb"]
"acl.do_build"-> "attr.do_package_write_rpm"
"binutils-native.do_patch"[label="binutils-native
do_patch\n:2.36.1-r0\nvirtual:native:/path/binutils_2.36.bb"]
"binutils-native.do_patch"->
"quilt-native.do_populate_sysroot"
}
```

Algorithm 1 Algoritmus pro získání vrcholů a hran pro graf

```
task1          ▷ Specifikovaný BitBake task nebo
                 "do_prepare_recipe_sysroot".
task2          ▷ Cokoliv nebo "do_populate_sysroot".
procedure PARSEDOTFILELINKS
    nodes          ▷ Seznam vrcholů.
    links          ▷ Seznam hran.
    index = 1      ▷ Index vrcholů.
    data = loadFile()
    for line in data do
        ld = line.split(" -> "))      ▷ Rozdělení řádky do pole.
        if
            ld[0].includes(task) and not
                ld[0].includes("label=") and
                ld[1].includes(task2)
then
            recipe1 = removeTaskAndQuotes(ld[0], task1)
            recipe1 = removeTaskAndQuotes(ld[0], task2)
            if recipe1 not equal recipe2 then
                if not nodes.find(recipe1) then
                    nodes.add(new Node(recipe1, index))
                    source = index
                    index++
                else
                    source = nodes.find(recipe1).index
                end if
                if not nodes.find(recipe2) then
                    nodes.add(new Node(recipe2, index))
                    target = index
                    index++
                else
                    target = nodes.find(recipe1).index
                end if
                links.add(new Link(source, target))
            end if
        end if
    end for
end procedure
```

Algorithm 2 Algoritmus pro získání cest k receptům

```
task1        ▷ Specifikovaný BitBake task nebo "do_prepare_recipe_sysroot".
task2        ▷ Cokoliv nebo "do_populate_sysroot".
procedure parseDotFileRecipePaths(index)
    nodes            ▷ Seznam vrcholů.
    links            ▷ Seznam hran.
    index = 1        ▷ Index vrcholů.
    data = loadFile()
    for line in data do
        ld = line.split(" -> "))      ▷ Rozdělení řádky do pole.
        if ld[0].includes("label=") then
            ld = line.split(""))      ▷ Druhé rozdělení řádky do pole.
            recipe = ld[0].split(".")
            recipe = removeQuotes(recipe)
            label = removeQuotes(ld[0])
            label = removeBrackets(label)
            label = removeString(label, "label=")
            ld = label.split("\n", ":")
            recipePath = ld[ld.length - 1]
            if not nodes.find(recipe) then
                nodes.add(new Node(recipe, index, recipePath))
                source = index
                index++
            else
                nodes.find(recipe).setRecipe(recipePath)
            end if
        end if
    end forreturn index
end procedure
```

V první řádce dojde ke zjištění, že obsahuje řetězec *"label="* a tak je rozdělena na jednotlivé části podle řetězce " "a jednotlivé prvky jsou uloženy do seznamu. První prvek z tohoto seznamu (*"acl.do_build"*) je rozdělen podle řetězce *"."* a tím dojde ke zjištění jména receptu (první část toho rozdělení), ze kterého jsou odstraněny nadbytečné uvozovky. Druhá část seznamu (*"[label="acl do_build\n:2.2.53-r0\n/path/acl_2.2.53.bb"]"*) je dále rozdělena podle řetězců *"\n"* a *":"* a poslední část tohoto rozdělení (*"/path/acl_2.2.53.bb"]"*) je cesta k souboru receptu (po odstranění zbylé

hranaté závorky a uvozovky). Po tomto dojde k vytvoření instance třídy `Node` se jménem receptu *"acl"* a s cestou k souboru daného receptu a *ID 1*, jelikož je seznam vrcholů zatím prázdný.

Ve druhé řádce není obsažen řetězec *"label="* a je v něm obsažen řetězec *do_build*. Řádka je tedy rozdělena podle řetězce *" -> "*. Z první části (*"acl-native.do_build"*) dojde k odstranění řetězce *".do_build"* a uvozovek, čímž je zjištěno jméno prvního receptu. Druhá část rozdělení je dále rozdělena podle řetězce *"."* a název druhého receptu je poté po odstranění uvozovek první část tohoto rozdělení. Jména receptů se nerovnají a tak zpracovávání pokračuje dál. Poté je zjištěno, že seznam vrcholů již obsahuje vrchol s názvem *"acl"* a je tak načteno jeho *ID*, které je *1*. Vrchol s názvem *"attr"* v seznamu ještě není, tak je do něj přidán s ID *2*. *ID* prvního vrcholu je poté použito jako *ID* zdrojového vrcholu a *ID* nově přidaného vrcholu je použito jako *ID* cílového vrcholu. Z těchto *ID* je následně vytvořena instance třídy `Link` a ta je přidána do seznamu hran.

Třetí řádka obsahuje řetězec *"label="* a tak je zpracována stejně jako jako řádka první, čímž do seznamu vrcholů přibude vrchol nový s *ID 3*.

Poslední řádka neobsahuje řetězec *"label="* ani řetězec *do_build* a tak není zpracovávána.

5.3.2 Získávání dat z receptů

Aktuálně je v nástroji implementováno pouze získávání informací o licencích v receptech. To je implementováno ve funkci `parseRecipe()` v souboru `recipe_parser.ts`. Po načtení souboru dojde k jeho procházení po řádkách. Každá řádka je rozdělena metodou `split()` podle znaku *'='*. Pokud je délka pole s rozdělenými hodnotami větší než *1*, dojde ke kontrole, zda není hodnota na indexu *0* po odstranění bílých znaků rovna řetězci *"LICENSE"*. Pokud ano, u hodnoty na indexu *1* se poté odstraní uvozovky a bílé znaky a hodnota se uloží do slovníkové struktury. Po tomto se procházení zastaví.

Pokud řádka s licencí nebyla nalezena, nebo pokud byla licence specifikována jiným způsobem, je hodnota licence nastavena na *"none"*.

Získávání dat lze rozšířit o další hodnoty a ty lze pak ukládat do slovníkové struktury.

Jako příklad může sloužit soubor s následujícím obsahem:

```
SUMMARY = "Utilities for managing POSIX Access Control Lists"

HOMEPAGE = "http://savannah.nongnu.org/projects/acl/"
BUGTRACKER = "http://savannah.nongnu.org/bugs/?group=acl"
```

```
SECTION = "libs"

LICENSE = "LGPLv2.1+ & GPLv2+"
LICENSE_${PN} = "GPLv2+"
```

Jednotlivé řádky jsou rozděleny podle znaku *'='*. Následně jsou z první části rozdělení (např. *"LICENSE_ ${PN} "*) odstraněny bílé znaky. Pokud je po tomto odstranění rovna hodnota levé strany řetězci *"LICENSE"*, dojde ve druhé části k odstranění uvozovek a bílých znaků a tato část je poté brána jako receptem použitá licence. Z výše uvedeného souboru je tedy použita řádka *"LICENSE = "LGPLv2.1+ & GPLv2+""* a jako licence zvolena *"LGPLv2.1+ & GPLv2+"*.

Nástroj neřeší, zda zadaná hodnota proměnné s licencí dává smysl, tedy zda se třeba neodkazuje na jiný recept.

5.4 Komunikace mezi *JavaScript* soubory a *TypeScript* soubory

Jelikož je některá funkcionalita implementována v *JavaScript* souborech a některá v *TypeScript* souborech, je nutné zajistit, aby tyto typy souborů mezi sebou komunikovaly. Například základní obsluha stisknutí nějakého tlačítka byla implementována v *JavaScript* souboru, ale hlavní akce se provede v *TypeScript* souboru (např. otevření nového okna rozšíření).

K tomuto slouží *Visual Studio Code API* [41], které lze použít i v *JavaScript* souborech. Pro jeho použití stačí inicializovat proměnnou následujícím způsobem:

```
const vscode = acquireVsCodeApi();.
```

Tato proměnná byla poté využita pro posílání zpráv pomocí metody `postMessage()` do tříd, která definují zobrazení, ve kterých je použit daný *JavaScript* soubor. V samotných třídách jsou poté definovány akce, které se po přijetí zpráv provádějí, pomocí metody `onDidReceiveMessage()`.

V některých případech je v nástroji také nutné posílat zprávy z tříd reprezentujících zobrazení. Pro toto je opět použito posílání zpráv pomocí metody `postMessage()` ze samotných *WebView*, která náleží třídám `Sidebar`,

`Legend` a `VisualizationPanel`. V *JavaScript* souborech jsou poté zprávy zpracovávány pomocí metody `addEventListener()`.

5.5 Vizualizace

Nejdůležitější částí nástroje je samotné zobrazení vizualizace. To je implementováno v souboru `visualization.js`, který je použít v *HTML* definici ve třídě `VisualizationPanel`. Jak již bylo zmíněno v kapitole 4.5, o samotnou vizualizaci se stará webová knihovna *D3.js*. Vstupem pro vizualizaci je *JSON* řetězec vytvořený třídou `DotParser` a do souboru `visualization.js` je předán pomocí *hidden input HTML* elementu, který je definován ve třídě `VisualizationPanel`.

5.5.1 Základní vizualizace

Nejdříve dojde k načtení *JSON* dat, které je implementováno ve funkci `initData()`, a k vytvoření *SVG* elementu, do kterého se bude vizualizace generovat, což je provedeno pomocí *D3.js* knihovny ve funkci `initSVG()`. Dále se pomocí knihovny *D3.js* vytvoří jednotlivé grafické prvky vizualizace pro neodstraněné recepty. Pro vrcholy to jsou obdélníky a textové elementy s názvy receptů. Toto je provedeno ve funkcích `nodesUpdate()` a `labelsUpdate()`. Pro hrany se vytvoří přímky se šipkami, což je provedeno ve funkci `linksUpdate()`. Funkce `nodesUpdate()`, `labelsUpdate()` a `linksUpdate()` se poté volají i po jakékoliv změně v datech pro vizualizaci (např. odstranění vrcholu). Samotná vizualizace se poté inicializuje ve funkci `initSimulation()`. Zde dojde k vytvoření a inicializaci simulace, nastaví se zde jednotlivé síly a parametry a přidají se do vizualizace vrcholy a hrany z *JSON* řetězce. Mezi jednotlivé parametry patří vzdálenost hran, což se dá považovat jako síla pružiny v popisu algoritmu z kapitoly 3.1.5, počet iterací simulace, odpudivá síla hran a síla, která všechny vrcholy přitahuje ke středu plátna. Dále se specifikuje, že po skončení simulace se zavolá funkce `simulationTicked()`. Ta provede nastavení pozic jednotlivých grafických elementů.

Zároveň s tím dojde k inicializaci matice propojení vrcholů. Tato matice ukládá, jak jsou vrcholy mezi sebou propojeny. Toho je využito v zobrazení vyžadovaných vrcholů a vrcholů vyžadujících daný vrchol. Dále je této matice využito při rekurzivním zvýrazňování receptů. Toto je implementováno ve funkci `initMatrix()`.

Po kliknutí na vrchol dojde k průchodu všech vrcholů a označení vrcholů vyžadujících zvolený recept a vrcholů, které zvolený recept vyžaduje.

Oba typy závislostí mají jinou barvu. Obtažení jednotlivých vrcholů se nastaví podle toho, jakým směrem vede jejich propojení se zvoleným vrcholem. Pokud se vrchol rovná zvolenému vrcholu, dojde k jeho zvýraznění jinou barvou.

5.5.2 Zvýraznění licencí

Po vygenerování vizualizace dojde ve funkci `initData()` k vytvoření seznamu všech licencí a počtu jejich výskytů postupným procházením všech vrcholů. Ze seznamu jsou poté odstraněny prvky s hodnotou *"none"* a prázdným řetězcem. Nakonec je seznam seřazen sestupně, podle počtu výskytů jednotlivých licencí. Pro samotné zvýraznění licencí je použita podobná implementace jako označování vrcholů s tím rozdílem, že funkcionalita obarvování licencí se volá ve funkci `nodesUpdate()` a je aplikována na všechny vrcholy. Obarvení vrcholů je vždy podle deseti nejpoužívanějších licencí.

Při obarvovaní dojde k nastavení atributu barvy obtažení vrcholů podle deseti nejpoužívanějších licencí, ostatní jsou obtaženy barvou základní.

5.5.3 Rekurzivní zvýraznění receptů

Pro rekurzivní zvýraznění receptů je opět použita stejná vizualizace jako v případě základní vizualizace. Rozdíl v tomto případě je takový, že po zvolení analýzy vrcholů k odstranění, dojde, po kliknutí na vrchol, k zavolání funkce `addAffectedNodes()`. Funkce `addAffectedNodes()` je rekurzivní a volá se pro každý vrchol, který vyžaduje zvolený vrchol. V této funkci se vrchol, pro který byla zavolána, přidá do seznamu ovlivněných vrcholů a následně se opět volá pro každý vrchol, který vyžaduje tento vrchol.

Dále jsou opět procházeny všechny obdélníky a pokud vrchol patřící obdélníku patří do seznamu ovlivněných vrcholů, je obdélník zvýrazněn. Pokud se vrchol patřící obdélníku rovná zvolenému vrcholu, je barva jeho obtažení nastavena jinou barvu než základní a než ovlivněné vrcholy, jinak je barva obtažení nastavena na základní.

5.5.4 Export do *SVG*

Export do *SVG* je implementován ve funkci `exportSVG()` a to tak, že nejprve dojde k získání aktuálního stavu *SVG* elementu (výška, šířka, transformace). Poté se elementu grafu, který patří pod *SVG* element, nastaví transformace na střed a výška a šířka *SVG* elementu se nastaví na výšku a šířku elementu grafu. Celý *SVG* je poté serializován pomocí třídy `XMLSerializer` na řetě-

zec. Tento řetězec je poté odeslán do třídy `VisualizationPanel`, která zobrazí dialogové okno pro uložení *SVG* do souboru a po uživatelově potvrzení přidá k řetězci s *SVG* daty *XML* hlavičku a soubor zapíše do souborového systému.

5.5.5 Vyhledávání vrcholů

Oproti návrhu byla po testování do nástroje přidána možnost vyhledávání vrcholů podle zadaného řetězce. Během toho dojde k vyčištění všech informací o zvoleném vrcholu (pokud nějaký zvolen je) a k odstranění všech zvýraznění ve vizualizaci. Řetězec pro vyhledávání nemusí obsahovat přesný název vrcholu. Pro nalezení vrcholů dojde k průchodu všemi vrcholy aktuálně viditelnými ve vizualizaci a pokud jejich názvy obsahují zadaný řetězec, jsou zvýrazněny. V případě použití analýzy licencí také dojde k odstranění zvýraznění jednotlivých vrcholů podle jejich licencí, to se zobrazí až poté, co uživatel zvolí nějaký vrchol (např. ten, který hledal).

6 Testování

Vytvořený nástroj byl otestován třemi způsoby. Algoritmy pro získávání dat ze souborů byly otestovány jednotkovými testy. Grafické uživatelské rozhraní bylo otestováno pomocí scénářů pro funkcionální testování. Nakonec bylo provedeno uživatelské testování pro získání zpětné vazby.

6.1 Jednotkové testování

Pro implementaci jednotkových testů byla použita vzorová implementace, která byla automaticky vytvořena při zakládání projektu rozšíření a využívá framework *Mocha* [45], který je určen pro testování *JavaScript* souborů, ale lze jej použít i pro *TypeScript* soubory. Testy lze najít ve složce *scr/test*. V této složce je pak dále složka *data*, která obsahuje vstupní *.dot* a *.bb* soubory pro testování. Ve složce *suite* jsou pak soubory `index.ts`, který slouží pro inicializaci *Mocha* frameworku, a `extension.test.ts`. Oba tyto soubory byly vytvořeny inicializačním nástrojem rozšíření. Změny byly provedeny pouze v souboru `extension.test.ts`, ve kterém byly implementovány samotné testy. Ve složce *scr/test* je ještě soubor `runTest.ts`, který byl opět vytvořen inicializátorem a slouží pro spuštění testů.

Jednotkovými testy byly otestovány pouze veřejné metody, které jsou ve třídě `DotParser`, a funkce `parse_recipe()`, nacházející se v souboru `recipe_parser.ts`. Jako vstupní testovací soubor pro otestování funkcionality třídy `DotParser` byl použit zkrácený *.dot* soubor vygenerovaný pro cíl *core-image-minimal* v rámci referenční distribuce *Poky* z větve *hardknott*. Tento soubor obsahuje pouze závislosti pro první dva recepty (*acl-native* a *acl*).

Otestováno bylo, zda třída `DotParser` vrací správné výsledky pro základní získávání dat z grafu, pro získávání dat z grafu s informacemi o licencích, pro získávání dat z grafu pro jeden zvolený BitBake task (ostatní by měly fungovat stejně) a testování pro neexistující vstupní soubor.

Dále bylo otestováno, zda funkce `parse_recipe()` vrací správné hodnoty pro soubory se specifikovanou licencí, pro soubory bez specifikované licence, pro soubory s licencí specifikovanou pod řádkami specifikující například licenci nějaké specifické knihovny a pro neexistující soubory.

Při testování docházelo k problémům s *npm* příkazem pro kompilaci testů, protože kontroloval stažené moduly a v modulu *d3* nacházel chyby. Tento nástroj ale zároveň kompiloval soubory s testy. V konfiguračním sou-

boru `launch.json` tak při spouštění testů byl tento příkaz odstraněn. Při prvním spuštění testů po změně jejich implementace ale musel být vrácen zpět, aby je zkompiloval.

6.2 Testování uživatelského rozhraní a celkové funkcionality

Testování uživatelského rozhraní a zároveň i celkové funkcionality bylo provedeno pomocí testovacích skriptů prováděných ručně autorem diplomové práce. Pro testovací účely byla vytvořena složka obsahující složku *build* s *.dot* souborem a složku *recipes* se soubory receptů. Všechny soubory byly vytvořeny ručně tak, aby bylo možné snadno otestovat funkcionalitu. Testování bylo rozděleno na 4 skripty.

První skript testuje funkcionalitu základní vizualizace. Tento skript obsahuje otestování vygenerování grafu bez konfigurace, možnosti zvolení vrcholu v grafu a akcí provedených po zvolení vrcholu (správné vypsání informací o vrcholu, správné označení vrcholu a vrcholů se kterými má zvolený vrchol hrany a vyplnění seznamů s propojenými vrcholy), otestování možnosti otevřít soubor receptu zvoleného vrcholu, odstranit z vizualizace a vrátit zpět, možnosti zvolit vrchol ze seznamů propojených vrcholů a možnost otevřít soubor receptu z kontextových menu seznamů.

Druhý skript slouží pro otestování funkcionality zvýraznění vrcholů ovlivněných odstraněním zvoleného vrcholu. Vstupní *.dot* soubor obsahuje cyklické závislosti (které by jinak graf obsahovat neměl), aby došlo k otestování všech zastavovacích podmínek hledání ovlivněných vrcholů. Skript testuje, zda správně funguje možnost zvolit typ analýzy, zda jsou zvýrazněny očekávané vrcholy a zda jsou vrcholy správně přidány do seznamu ovlivněných vrcholů. Zároveň se testuje, zda je možné vrcholy volit ze seznamu ovlivněných vrcholů a možnost otevřít soubor receptu z kontextového menu tohoto seznamu.

Další skript byl vytvořen pro otestování správnosti funkcionality pro zvýrazňování licencí. Zde se hlavně testuje, zda jsou všechny vrcholy správně obarveny a zda je správně zobrazena legenda.

Poslední skript slouží pro otestování funkčnosti různých tlačítek a zobrazování informativních nebo chybových hlášek. Testuje se zde například, zda dojde k správnému zobrazení chybové hlášky při nedostupnosti vstupního souboru, při snaze otevřít soubor receptu pro vrchol nebo vrchol smazat v případě, že žádný nebyl zvolen. Dále se testuje, zda správně funguje vyhledávání receptů, nastavování parametrů a možnost spustit generování grafu

pomocí palety příkazů ve *Visual Studio Code*.

U první testované verze bylo zjištěno, že je obtížné hledat recepty ve vizualizaci podle jména. Na základě toho byla do nástroje implementována funkcionalita vyhledávání uzlů.

Všechny testovací skripty lze najít v příloze B.

6.3 Uživatelské testování

Pro získání zpětné vazby od uživatelů byla vytvořena jednoduchá uživatelská studie. Pro tuto studii bylo vygenerováno několik vstupních *.dot* souborů pro různé recepty v rámci referenční distribuce *Poky* na větvi *hardknott*. Recepty byly voleny tak, aby vygenerované soubory měly různý počet hran a vrcholů. Tyto soubory pak byly uloženy do složek podle jmen zvolených receptů. Ve složkách je vždy podsložka *build* s *.dot* souborem.

Uživatelé poté měli za cíl splnit několik úkolů. Mezi úkoly patří nalezení nejčastěji vyžadovaných receptů ve vizualizaci se základním nastavením (pro demonstraci jednoduchosti nalezení těchto uzlů pouhým pohledem), nalezení všech receptů, které vyžadují určitý recept se základním nastavením, nalezení všech receptů, které ovlivní odstranění určitého receptu, nalezení všech receptů závisejících na určitém receptu a nalezení všech receptů, jejichž odstranění neovlivní žádné jiné recepty. Hlavním cílem uživatelů ale nebylo splnit úkoly přesně podle zadání, ale zjistit, jak se s nástrojem pracuje a které vlastnosti by mohly být vylepšeny. Vizualizace v nástroji nebyla porovnávána s již existujícím nástrojem a tak uživatelům ani nebyl měřen čas potřebný k dokončení úkolů a ani správnost jejich výsledků.

Uživatelům se líbila interaktivnost grafu, snadno pochopitelné uživatelské rozhraní a možnost zvolení vrcholu se seznamů závislých nebo ovlivněných receptů. Jako vlastnosti, které by mohly být zlepšeny uživatelé uvedli zaměření na nalezený vrchol při vyhledávání podle jména receptu a automatické zvolení nalezeného vrcholu, což nebylo v prototypu implementováno z toho důvodu, že může být nalezeno více receptů, které obsahují daný řetězec. Dále navrhovali zobrazení počtu závislých uzlů a různé klávesové zkratky, což by určitě byla funkcionalita implementovaná v plné verzi nástroje. Uživatelům se také nelíbilo, že pokud při vyhledávání není nalezen žádný uzel, uživatel nedostane žádné upozornění. Na základě toho byla tato možnost do nástroje přidána.

Otestování funkcionality zvýraznění licencí nebylo v těchto scénářích provedeno, jelikož vyžaduje přístup k receptům, takže by tester musel mít přístup ke stažené referenční distribuci.

7 Omezení a návrhy na další rozšíření

Vytvořený nástroj je pouze prototypem a proto neobsahuje veškerou funkcionalitu, která by z nástroje udělala reálně použitelný nástroj.

Nástroj vyžaduje existující *.dot* soubor ve složce *build*, která je v kořenu otevřeného adresáře. Zároveň nástroj neumí pracovat s pracovními prostory (workspace), které mají projekt (složku), pro který by uživatel chtěl vytvořit vizualizaci na jiné pozici než na první. Proto je pro uživatele bezpečnější, když v pracovním prostoru bude mít otevřenou pouze složku s *Yocto Project*. Dalším omezením je nemožnost automatického vytvoření vstupního *.dot* souboru.

Uživatel nemůže zároveň zobrazit více grafů (např. pro různé typy *BitBake* úkolů) a musí vždy graf generovat znovu, což může ztížit například hledání všech závislostí. Řešením může být otevření více instancí *Visual Studio Code*.

I přes výrazné snížení počtu hran mohou být grafy velmi rozsáhlé. To je ale dáno především samotnou rozsáhlostí některých projektů.

Ve vytvořeném nástroji nelze v současné implementaci měnit jednotlivá barevná schémata jelikož se jedná o prototyp a cílem bylo demonstrovat možnou podobu vizualizace.

Ve vygenerovaném grafu je možné se ztratit. Například při oddálení grafu je možné se různě posouvat do stran a během tohoto ztratit graf z obrazovky. Řešením je pak nutnost přegenerování grafu a tím pádem ztráty provedených změn.

Nástroj v aktuální podobě neumožňuje ukládání rozpracované práce. Po zavření okna s vizualizací nebo po vypnutí a opětovném zapnutí *Visual Studio Code* tak uživatel musí graf vygenerovat znovu a znovu provést provedené změny.

Získané informace o licencích nemusí být vždy vypovídající. Některé recepty definují licence tak, že místo licence specifikují jméno receptu, který poté specifikuje licenci. Dále také může dojít ke specifikování různých licencí pro různé prvky (receptem vytvořená knihovna, recept samotný, atd.). V celkové licenci je poté specifikováno více licencí a celý tento řetězec se v nástroji považuje jako jedna licence. Jelikož tyto licence mohou být v jiném pořadí pro různé recepty, je poté těžší vytvoření seznamu použitých licencí a jejich seřazení od nejvíce používaných.

Nástroj vyžaduje připojení k internetu, jelikož je knihovna *D3.js* vyžadována v *JavaScript* souboru pomocí *URL*.

Mezi první možné rozšíření nástroje, které by usnadnilo práci s ním a rozšířilo možnosti vývojářů, patří umožnění použití nástroje ve více projektovém pracovním prostoru. Uživatel by například mohl specifikovat cestu, nebo název složky v pracovním prostředí, pro které potřebuje vygenerovat vizualizaci.

Dalším rozšířením je podpora práce s více grafy najednou. Grafy by mohly být zobrazeny ve více záložkách pod různými jmény a seznamy receptů by mohly být rozšířeny tak, že například ve stromovém zobrazení receptů odstraněných z vizualizace by pod kořenovým prvkem ještě existovaly prvky s názvy záložek, pod kterými by poté byly jednotlivé seznamy odstraněných receptů.

Nástroj by mohl být rozšířen funkcionalitou automatického generování *.dot* souborů. Během implementace byly snahy takovouto funkcionalitu do nástroje přidat, ale jelikož to nebylo hlavním cílem nástroje, bylo od toho prozatím upuštěno.

Rozšířením, které by mohlo uživateli usnadnit práci s nástrojem, je přidání možnosti ukládání aktuálního stavu do databáze, nebo minimálně do nějakého *JSON* souboru.

8 Závěr

V této práci byly popsány základy práce s *Yocto Project*, který je určený pro vývoj vestavěných linuxových systémů, a jeho jednotlivé komponenty. Dále byly popsány různé způsoby odstranění vizuálního šumu z rozsáhlých vizualizací a další způsoby vizualizace závislostí, které jsou použitelné pro softwarové systémy. Dále byl v práci navržen a implementován prototyp nástroje formou rozšíření do *Visual Studio Code*, který uživatelům umožňuje zobrazení závislostí receptů v *Yocto Project* a tím pomáhá analyzovat sestavení systému. Z testování nástroje vyplývá, že pracuje správně a vizualizace jsou čitelnější a jednodušší (oproti grafu vygenerovaném samotným *Yocto Project*). Jelikož se ale jedná o prototyp, neobsahuje některé možnosti, které by uživateli usnadnily práci se samotným rozšířením. Vytvořený prototyp nástroje byl publikován ve *Visual Studio Marketplace* a je tak dostupný ke stažení uživateli *Visual Studio Code*.

Samotná vizualizace se inspiruje vizualizací popsané v článku An Interactive UML-like Visualization for Large Software Diagrams [38] od Lukáše Holého, Iva Malého, Ladislava Čmolíka, Kamila Ježka a Přemka Brady, ale zjednodušuje jí pro použití s *Yocto Project* a přidává integraci do *Visual Studio Code*, což uživatelům zjednodušuje práci.

Použité zkratky

BSP	Board Support Package - Yocto Project vrstva dodávaná výrobcem hardware
OSGi	Specifikace modulárního systému pro jazyk Java
WSLv2	Windows Subsystem for Linux v2 - vrstva pro běh Linuxu pod Windows
DOT	Jazyk pro popis grafů
UML	Unified Modeling Language - jazyk pro vizualizaci software
CROPS	cross-platform framework, sloužící k vytvoření prostředí pro sestavování binárních souborů
eSDK	Extensible Software Development Kit - slouží pro vývoj a přidávání aplikací a knihoven do distribucí
DOM	Document Object Model - objektová reprezentace XML nebo HTML
ELF	Executable and Linkable Format - formát binárního souboru

Bibliografie

[1] *Yocto Project*. URL: https://www.yoctoproject.org (cit. 12. 04. 2022).

[2] *Yocto Project - About*. URL: https://www.yoctoproject.org/about/ (cit. 28. 01. 2022).

[3] *Yocto Project - The Ecosystem:Members*. URL: https://www.yoctoproject.org/ecosystem/members/ (cit. 28. 01. 2022).

[4] *Yocto Project - Overview and Concepts Manual - 2 Introducing the Yocto Project*. URL: https://docs.yoctoproject.org/overview-manual/yp-intro.html#introducing-the-yocto-project.

[5] *BitBake - GitHub*. URL: https://github.com/openembedded/bitbake (cit. 12. 04. 2022).

[6] *Yocto Project - BitBake - 1 Overview*. URL: https://docs.yoctoproject.org/bitbake/bitbake-user-manual/bitbake-user-manual-intro.html (cit. 28. 01. 2022).

[7] *Yocto Project - Overview and Concepts Manual - 2 Introducing the Yocto Project*. URL: https://docs.yoctoproject.org/overview-manual/yp-intro.html#reference-distribution-poky (cit. 11. 05. 2022).

[8] *Yocto Project - Software:Reference Distribution*. URL: https://www.yoctoproject.org/software-overview/reference-distribution/ (cit. 28. 01. 2022).

[9] *QEMU*. URL: https://www.qemu.org (cit. 12. 04. 2022).

[10] *QEMU*. URL: https://www.qemu.org/docs/master/about/index.html (cit. 12. 04. 2022).

[11] *Yocto Project - Software overview*. URL: https://www.yoctoproject.org/software-overview/ (cit. 28. 01. 2022).

[12] *Yocto Project - Reference manual - 2 Yocto Project Terms*. URL: https://docs.yoctoproject.org/ref-manual/terms.html (cit. 12. 04. 2022).

[13] *Yocto Project - Reference manual - 4 Source Directory Structure*. URL: https://docs.yoctoproject.org/ref-manual/structure.html (cit. 28. 01. 2022).

[14] *Yocto Project - Reference manual - 6 Tasks*. URL: https://docs.yoctoproject.org/ref-manual/tasks.html#do-configure (cit. 10. 05. 2022).

[15] *Yocto Project - Referefnce manual - 5 Classes*. URL: https://docs.yoctoproject.org/ref-manual/classes.html#insane-bbclass (cit. 09. 04. 2022).

[16] *Yocto Project - BitBake - 4 File Download Support*. URL: https://docs.yoctoproject.org/bitbake/bitbake-user-manual/bitbake-user-manual-fetching.html (cit. 28. 01. 2022).

[17] *Yocto Project - Overview and Concepts Manual - 4 Yocto Project Concepts*. URL: https://docs.yoctoproject.org/overview-manual/concepts.html (cit. 28. 01. 2022).

[18] *Yocto Project - BitBake - 3 Syntax and Operators*. URL: https://docs.yoctoproject.org/bitbake/bitbake-user-manual/bitbake-user-manual-metadata.html (cit. 28. 01. 2022).

[19] *Yocto Project - BitBake - Software:Project Components:devtool*. URL: https://www.yoctoproject.org/software-item/devtool/ (cit. 28. 01. 2022).

[20] *Yocto Project - Reference manual - 1 System Requirements*. URL: https://docs.yoctoproject.org/ref-manual/system-requirements.html#supported-linux-distributions (cit. 13. 04. 2022).

[21] *Yocto Project - Release 2.7 (warrior)*. URL: https://docs.yoctoproject.org/migration-guides/migration-2.7.html#eclipse-support-removed (cit. 13. 04. 2022).

[22] *Visual Studio Code - Bitbake extension*. URL: https://marketplace.visualstudio.com/items?itemName=EugenWiens.bitbake (cit. 13. 04. 2022).

[23] *BitBake - 1.5.2.4 Generating Dependency Graphs*. URL: https://docs.yoctoproject.org/bitbake/2.0/bitbake-user-manual/bitbake-user-manual-intro.html#generating-dependency-graphs (cit. 17. 05. 2022).

[24] Stefan Bieliauskas a Andreas Schreiber. „A Conversational User Interface for Software Visualization“. In: *2017 IEEE Working Conference on Software Visualization (VISSOFT)*. 2017, s. 139–143. DOI: 10.1109/VISSOFT.2017.21.

[25] *Graph visualization at scale: strategies that work*. URL: https://cambridge-intelligence.com/visualize-large-networks/ (cit. 30. 01. 2022).

[26] Ján Mojžiš a Michal Laclavík. „Graph clutter filtering based on connectivity distance and visibility“. In: *2014 Science and Information Conference*. 2014, s. 153–158. DOI: 10.1109/SAI.2014.6918184.

[27] S.G. Roy a A. Chakrabarti. „Chapter 11 - A novel graph clustering algorithm based on discrete-time quantum random walk“. In: *Quantum Inspired Computational Intelligence*. Ed. Siddhartha Bhattacharyya, Ujjwal Maulik a Paramartha Dutta. Boston: Morgan Kaufmann, 2017, s. 361–389. ISBN: 978-0-12-804409-4. DOI: https://doi.org/10.1016/B978-0-12-804409-4.00011-5. URL: https://www.sciencedirect.com/science/article/pii/B9780128044094000115.

[28] Patrizio Angelini et al. „1-Fan-Bundle-Planar Drawings of Graphs“. In: *Theoretical Computer Science* 723 (břez. 2018). DOI: 10.1016/j.tcs.2018.03.005.

[29] André Spritzer a Carla Freitas. „Navigation and Interaction in Graph Visualizations“. In: *Revista de Informática Teórica e Aplicada; Vol. 15, No 1 (2008); 111-136* 15 (zář. 2008). DOI: 10.22456/2175-2745.6015.

[30] *Force-directed graph layouts explained*. URL: https://cambridge-intelligence.com/keylines-faq-force-directed-layouts/ (cit. 09. 04. 2022).

[31] *Force Directed Graph - What is it ?* URL: https://www.toucantoco.com/en/glossary/force-directed-graph.html (cit. 09. 04. 2022).

[32] Michael Burch et al. „Evaluating Partially Drawn Links for Directed Graph Edges“. In: sv. 7034. Zář. 2011, s. 226–237. ISBN: 978-3-642-25877-0. DOI: 10.1007/978-3-642-25878-7_22.

[33] Markus Scheidgen, Nils Goldammer a Joachim Fischer. „Interactive Visualization of Software“. In: *SDL 2017: Model-Driven Engineering for Future Internet*. Ed. Tibor Csöndes, Gábor Kovács a György Réthy. Cham: Springer International Publishing, 2017, s. 1–17. ISBN: 978-3-319-68015-6.

[34] Camilo Arango Moreno, Walter F. Bischof a H. James Hoover. „Interactive visualization of dependencies“. In: *Computers Education* 58.4 (2012), s. 1296–1307. ISSN: 0360-1315. DOI: https://doi.org/10.1016/j.compedu.2011.12.027. URL: https://www.sciencedirect.com/science/article/pii/S0360131511003447.

[35] Andreas Schreiber a Marlene Brüggemann. „Interactive Visualization of Software Components with Virtual Reality Headsets“. In: *2017 IEEE Working Conference on Software Visualization (VISSOFT)*. 2017, s. 119–123. DOI: 10.1109/VISSOFT.2017.20.

[36] Ugo Erra et al. „Visualising a Software System as a City Through Virtual Reality“. In: čvn. 2017. ISBN: 978-3-319-60927-0. DOI: 10.1007/978-3-319-60928-7_28.

[37] Simone Romano et al. „On the use of virtual reality in software visualization: The case of the city metaphor“. In: *Information and Software Technology* 114 (2019), s. 92–106. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2019.06.007. URL: https://www.sciencedirect.com/science/article/pii/S0950584919301405.

[38] Lukas Holy et al. „An Interactive UML-like Visualization for Large Software Diagrams“. In: *Research Journal of Applied Sciences, Engineering and Technology* 11 (říj. 2015), s. 355–371. DOI: 10.19026/rjaset.11.1789.

[39] Lukas Holy et al. „Lowering Visual Clutter in Large Component Diagrams“. In: *2012 16th International Conference on Information Visualisation*. 2012, s. 36–41. DOI: 10.1109/IV.2012.17.

[40] Mireille Samia a Michael Leuschel. „Towards pie tree visualization of graphs and large software architectures“. In: *2009 IEEE 17th International Conference on Program Comprehension*. 2009, s. 301–302. DOI: 10.1109/ICPC.2009.5090068.

[41] *VS Code API*. URL: https://code.visualstudio.com/api/references/vscode-api (cit. 30. 04. 2022).

[42] *VS Code Extension Samples*. URL: https://github.com/microsoft/vscode-extension-samples (cit. 05. 05. 2022).

[43] *YouTube - How to Code a VSCode Extensions*. URL: https://youtu.be/a5DX5pQ9p5M (cit. 05. 05. 2022).

[44] *The web's scaffolding tool for modern webapps | Yeoman*. URL: https://yeoman.io (cit. 05. 05. 2022).

[45] *Mocha - the fun, simple, flexible JavaScript test framework*. URL: https://mochajs.org (cit. 01. 05. 2022).

A Uživatelská dokumentace

A.1 Instalace rozšíření

Visual Studio Code umožňuje instalaci rozšíření dvěma způsoby. První je instalace rozšíření pomocí *Visual Studio Marketplace*, které lze otevřít přímo ve *Visual Studio Code*. Druhou možností je ruční instalace.

Ruční instalaci rozšíření lze provést zavoláním příkazu:

```
code -install-extension <extension-name>
```

ze složky, která daný soubor obsahuje. Parametr *extension-name* je celý název souboru s rozšířením, tedy *yocto-project-dependency-visualizer-0.0.3.vsix*.

Pro nainstalování rozšíření z *Visual Studio Marketplace* je nutné vyhledat *Yocto Project Dependency Visualizer* v nabídce rozšíření a zvolit možnost *Install*.

A.2 Předpoklady pro spuštění rozšíření

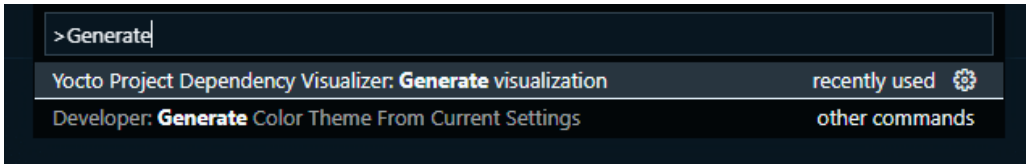
Pro správnou funkcionalitu je nutné otevřít složku s *Yocto Project* projektem. Složka musí obsahovat složku *build* se souborem *task-depends.dot*. Je možné, aby otevřená složka s *Yocto Project* projektem byla součástí pracovního prostoru s více složkami, ale v tomto pracovním prostoru musí být uvedena na prvním místě.

Dále je vhodné, aby v počítači existovaly soubory receptů na takových umístěních, které jsou uvedeny v souboru *task-depends.dot*. Není nutné aby tyto soubory byly součástí složky s *Yocto Project* projektem, ale je to obvyklé. Pokud v počítači soubory s recepty nejsou, nebude možné využívat funkcionality otevírání těchto souborů přímo z rozšíření a zároveň nebude možné provádět analýzu licencí, protože je načtena z těchto souborů.

A.3 Spuštění rozšíření

Rozšíření lze v rámci *Visual Studio Code* spustit dvěma způsoby. Prvním způsobem je spuštění pomocí palety příkazů. Tento způsob zobrazí základní vizualizaci bez uživatelského nastavení a je vhodný pro rychlou analýzu závislostí. Paletu příkazů lze otevřít klávesou *F1* nebo nebo kombinací kláves

CTRL + SHIFT + P, pokud nebylo uživatelem definováno jinak. V paletě je poté vhodné hledat klíčové slovo *"yocto"* nebo *"generate"*. Celý název příkazu je *Yocto Project Dependency Visualizer: Generate visualization*. Tento způsob lze vidět na obrázku A.1 Druhým způsobem spuštění je otevření hlavního menu rozšíření a stromových seznamů uzlů z bočního seznamu, kde se nachází například správce souborů nebo správce rozšíření. V tomto zobrazení je možné konfigurovat generování grafu a například prohlížet seznam uzlů odstraněných z grafu.



Obrázek A.1: Ukázka spuštění příkazu Yocto Project Dependency Visualizer: Generate visualization.

A.4 Hlavní menu rozšíření

Jak již bylo zmíněno v předchozí kapitole, hlavní menu lze otevřít pomocí bočního seznamu. Hlavní menu poté nabízí několik možností konfigurace. Lze ho vidět na obrázku A.2

A.4.1 Nastavení typu *BitBake* úkolu

První možností konfigurace je nastavení typu *BitBake* úkolu, pro který se bude graf závislostí generovat, s názvem *Task type*. Po kliknutí na rozbalovací seznam se zobrazí jednotlivé možnosti, kde na prvním místě je možnost *"DEFAULT"*, která vygeneruje graf pro typ *BitBake* úkolu *"do_prepare_recipe_sysroot"* na levé straně závislosti v *.dot* souboru a typ úkolu *"do_populate_sysroot"* na pravé straně závislosti v *.dot* souboru. Pro ostatní zvolené možnosti se vygeneruje graf pro zvolený typ úkolu na levé straně závislosti v *.dot* a jakýkoliv typ úkolu na pravé straně závislosti.

A.4.2 Nastavení režimu analýzy

Druhou možností konfigurace je zvolení režimu analýzy grafu, kterou je možné nalézt pod jménem *Mode*. Uživateli nabízí základní režim analýzy (*"DEFAULT"*), při kterém dochází ke zvýraznění přímo propojených uzlů se

zvoleným uzlem, režim analýzy licencí (*"Licenses"*), který zvýrazní jednotlivé uzly podle použitých licencí, a režim analýzy ovlivnění uzlů po odstranění zvoleného uzlu (*"Affected nodes"*), který rekurzivně zvýrazní všechny uzly, které přímo i nepřímo vyžadují zvolený uzel.

A.4.3 Nastavení parametrů *force-firected* algoritmu

Třetí část hlavního menu nabízí možnost nastavení parametrů *force-directed* algoritmu pro rozvržení vizualizace. Možnost *Force link distance* umožňuje uživateli nastavit délku hran, možnost *Force link iterations* poté počet iterací pro výpočet a možnost *Force node strength (repulsion)* slouží pro nastavení síly mezi jednotlivými uzly, kde záporné hodnoty znamenají, že se uzly odpuzují a kladné hodnoty znamenají, že se uzly přitahují (nastavení kladných hodnot se nedoporučuje).

A.4.4 Vygenerování vizualizace a export do *SVG*

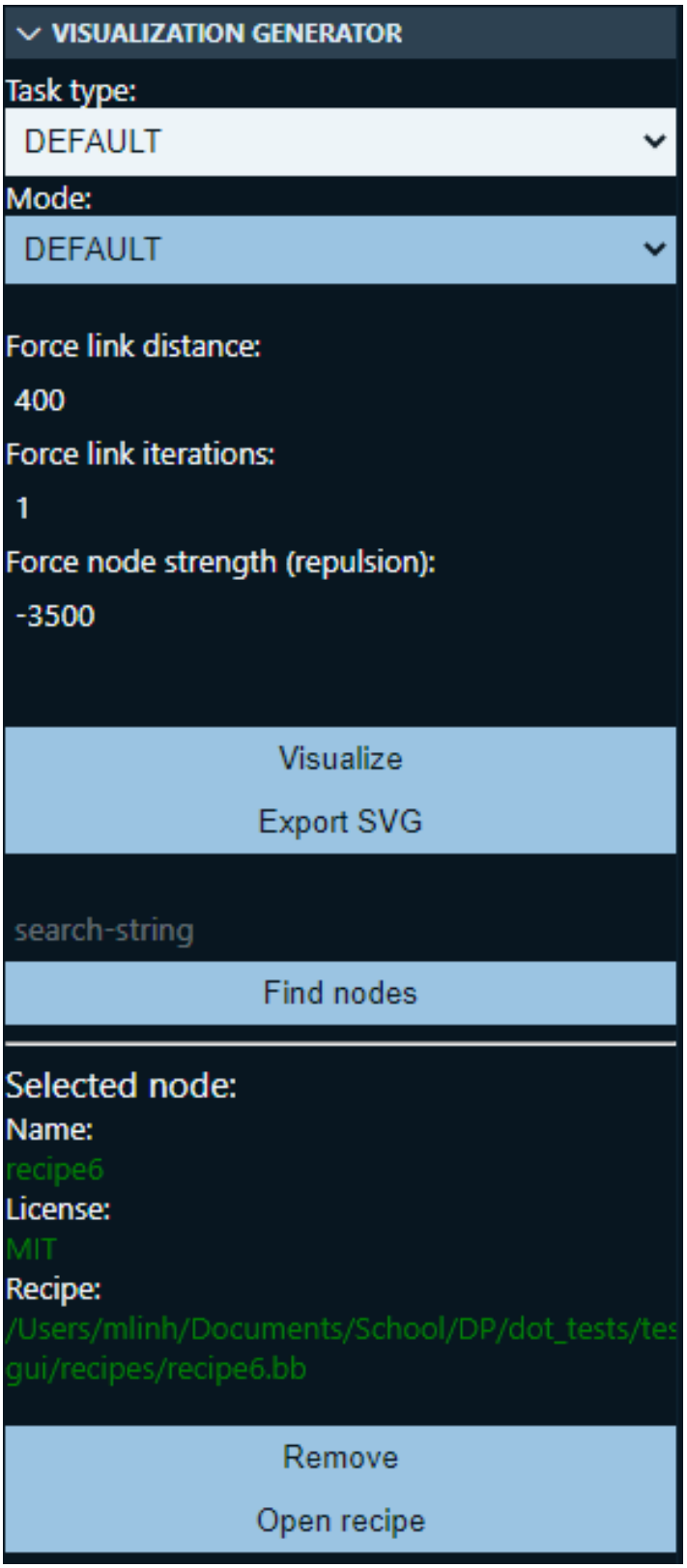
Dále hlavní menu nabízí tlačítko pro export vizualizace do *SVG* souboru (*Export SVG*) a tlačítko *Visualize* pro vygenerování samotného grafu.

A.4.5 Vyhledávání uzlů ve vizualizaci

Pod tlačítky pro vizualizaci a export do *SVG* je box pro zadávání textu pro vyhledávání jednotlivých uzlů ve vizualizaci, přičemž jsou vždy zelenou barvou zvýrazněny uzly takové, jejichž názvy receptů obsahují zadaný řetězec. Při vyhledávání dojde ve vizualizaci k odstranění veškerých zvýraznění (i pro licence). Pro zrušení zvýraznění pro vyhledávání je nutné kliknout na jakýkoliv z uzlů a tak ho zvolit. Pod boxem pro zadávání vyhledávaného řetězce je tlačítko *Find nodes* pro spuštění vyhledávání.

A.4.6 Zvolený uzel a jeho možnosti

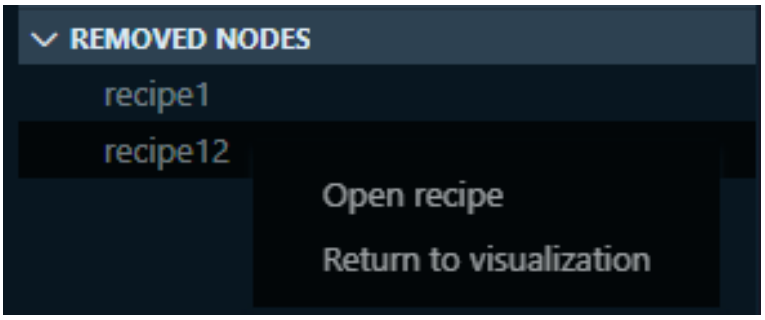
V poslední sekci lze nalézt informace o zvoleném uzlu. Mezi zobrazené informace patří název receptu, použitá licence a cesta k receptu. Hodnoty *"–none–"* ve všech polích znamenají, že žádný uzel nebyl zvolen. Dále sekce obsahuje tlačítko *Remove*, které odstraní zvolený uzel z vizualizace a tlačítko *Open recipe*, které slouží pro otevření souboru receptu v rámci *Visual Studio Code*.



Obrázek A.2: Ukázka hlavního menu rozšíření.

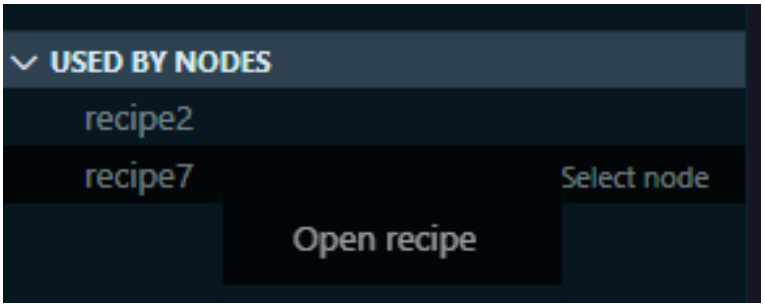
A.5 Seznam odstraněných uzlů

Pod hlavním menu lze nalézt několik seznamů uzlů. Mezi ně patří seznam uzlů, které byly odstraněny z vizualizace. Po odstranění uzlu z vizualizace se uzly zobrazí v tomto seznamu s názvem *REMOVED NODES*. Jednotlivé uzly v seznamu pak pomocí kontextového menu nabízí možnost vrátit uzel zpět do vizualizace pomocí možnosti *Return to visualization*, nebo možnost otevřít soubor receptu *Open recipe*. Seznam odstraněných uzlů lze vidět na obrázku A.3.



Obrázek A.3: Ukázka seznamu odstraněných uzlů s kontextovým menu.

A.6 Seznam vyžadovaných uzlů a seznam uzlů vyžadujících zvolený uzel



Obrázek A.4: Ukázka seznamu uzlů které vyžadují zvolený uzel s kontextovým menu a tlačítkem *Select node*.

Dalšími seznamy jsou seznam vyžadovaných uzlů a seznam uzlů vyžadujících zvolený uzel, které lze najít pod názvem *REQUESTED NODES* respektive *USED BY NODES*. Po najetí myší na některý z uzlů v těchto seznamech je uživateli nabídnuta možnost uzel zvolit ve vizualizaci (*Select*

node). Dále je pomocí kontextového menu a možnosti *Open recipe* možné otevřít soubor receptu. Seznam *USED BY NODES* lze vidět na obrázku A.4

A.7 Seznam ovlivněných receptů

Tento seznam se uživateli zobrazí po vygenerování grafu v režimu analýzy ovlivnění uzlů odstraněním zvoleného uzlu. Do seznamu se přidají všechny uzly, které přímo i nepřímo vyžadují zvolený uzel. U uzlů v seznamu je opět po najetí myší na některý z nich nabídnuta možnost uzel zvolit ve vizualizaci (*Select node*) a pomocí kontextového menu a možnosti *Open recipe* otevřít soubor receptu.

A.8 Legenda

Po vygenerování grafu v režimu analýzy použitých licencí, dojde k zobrazení legendy, kterou lze najít pod názvem *LEGEND*, pod hlavním menu. Legenda obsahuje názvy jednotlivých licencí, které jsou obarveny použitými barvami (viz obrázek A.5).

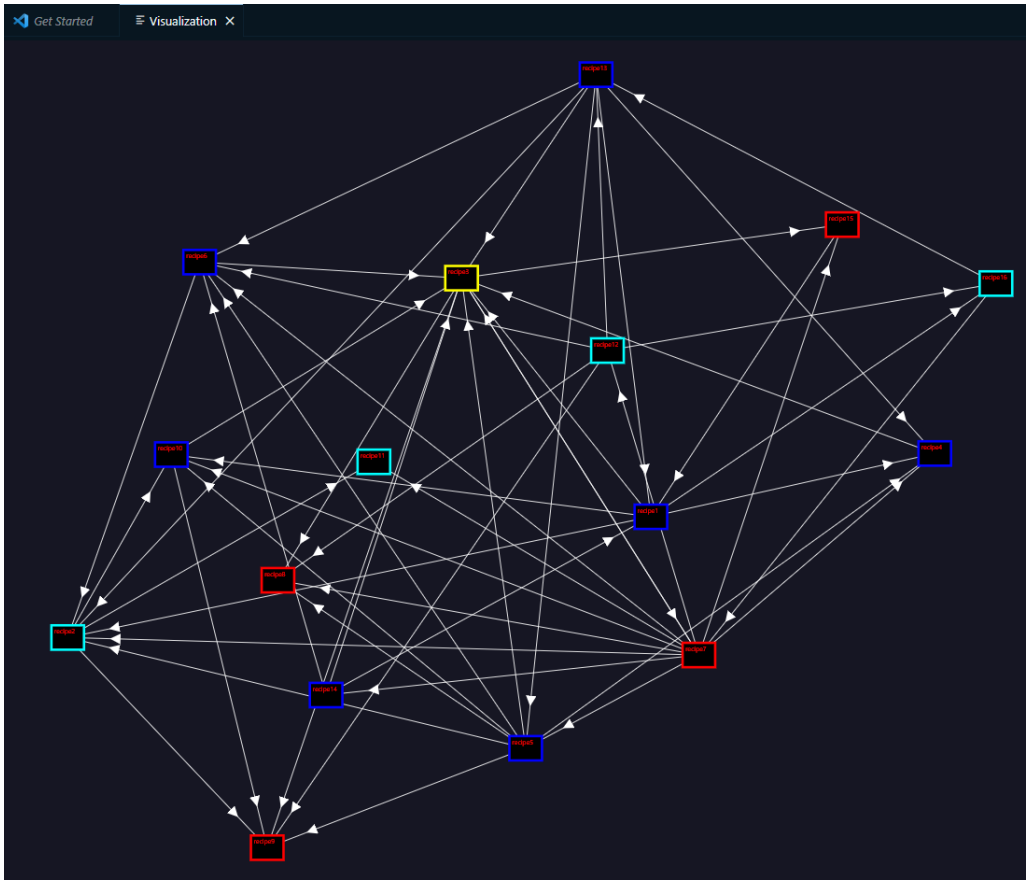
Obrázek A.5: Ukázka legendy.

A.9 Záložka s vizualizací

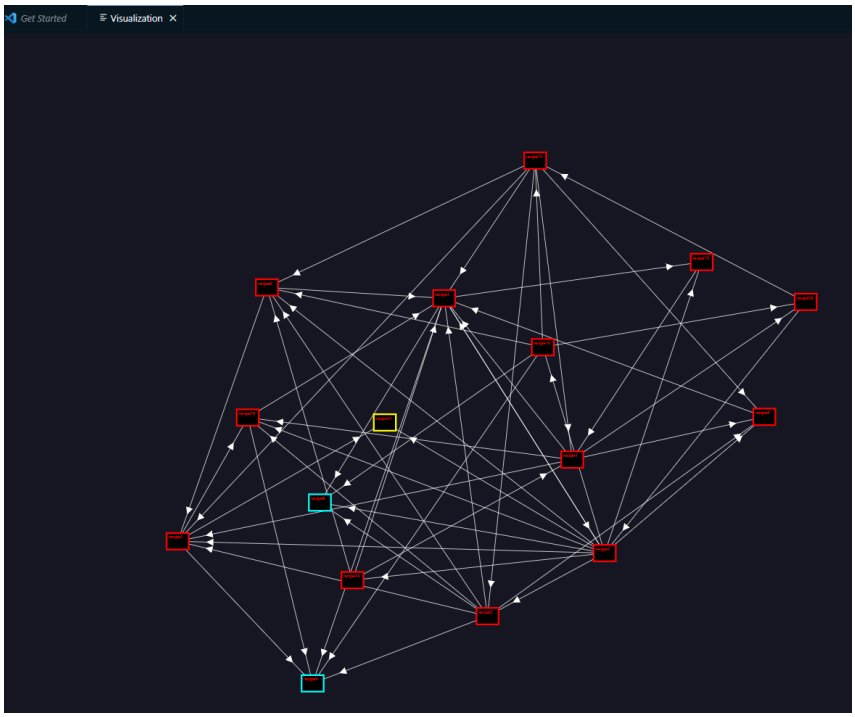
Po stisknutí tlačítka *Visualize* (nebo po spuštění příkazu z palety příkazů) dojde k zobrazení záložky s vizualizací. Ve vizualizaci je možné se posouvat pomocí podržení levého tlačítka myši mimo uzel a dále pohybem myší. Pomocí kolečka myši je možné vizualizaci přiblížit nebo oddálit.

Stisknutím levého tlačítka myši na uzlu (mimo oblast textu s názvem uzlu) dojde k jeho zvolení a vyplnění údajů o něm v hlavním menu. Pokud uživatel zvolil režim základní analýzy, dojde k zvýraznění zvoleného uzlu

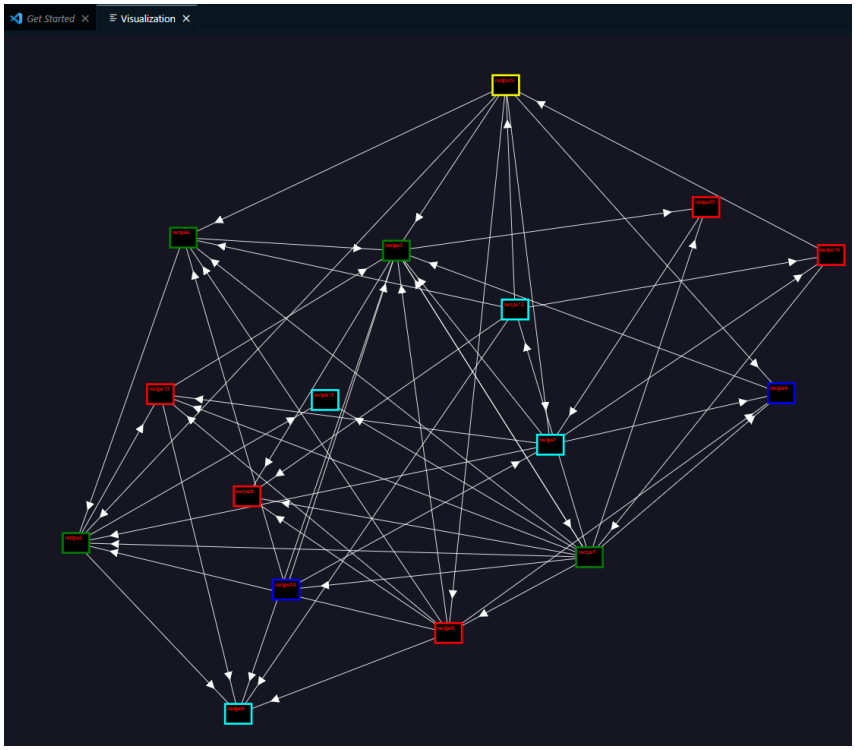
žlutou barvou a ke zvýraznění uzlů závisejících na zvoleném uzlu barvou modrou a uzlů, na kterých zvolený uzel závisí barvou červenou. V případě režimu analýzy ovlivnění uzlů odstraněním zvoleného uzlu dojde ke zvýraznění zvoleného uzlu žlutou barvou a k rekurzivnímu zvýraznění všech uzlů, které přímo i nepřímo vyžadují zvolený uzel, červenou barvou. V případě analýzy licencí nedojde k žádnému zvýraznění. Ukázky vizualizací lze vidět na obrázcích A.6, A.7 a A.8.



Obrázek A.6: Ukázka základní vizualizace.



Obrázek A.7: Ukázka vizualizace ovlivnění uzlů odstraněním zvoleného uzlu.



Obrázek A.8: Ukázka vizualizace licencí.

B Hlavní testovací scénáře grafického uživatelského rozhraní

B.1 Scénář 1 - Základní testy

Tento scénář byl použit pro otestování základní práce s vizualizací:

1. Uživatel spustí *Visual Studio Code*, otevře složku s testy a zvolí rozšíření v levé části aplikace.
2. Uživatel klikne na tlačítko *Visualize* -> zobrazí se graf s 16ti uzly.
3. Uživatel klikne na text s názvem libovolného uzlu -> nic by se nemělo stát.
4. Uživatel klikne na uzel s názvem *recipe1* mimo text s názvem uzlu -> v levé části aplikace se vyplní údaje o názvu uzlu, cestě k receptu (*<cesta>test-gui/recipes/recipe1.bb*) a použité licenci (*GPLv2*), kde *<cesta>* je cesta k receptu; ve stromovém zobrazení vyžadovaných uzlů se objeví uzly *recipe2*, *recipe3*, *recipe4*, *recipe10* a *recipe16* a zároveň se tyto uzly ve vizualizaci zvýrazní červenou barvou; ve stromovém zobrazení uzlů, které zvolený uzel vyžadují se objeví uzly *recipe13*, *recipe14* a *recipe15* a zároveň se tyto uzly ve vizualizaci zvýrazní modrou barvou; zvolený uzel se zvýrazní barvou žlutou
5. Uživatel klikne na tlačítko *Open recipe* -> ve *Visual Studio Code* se otevře soubor *<cesta>test-gui/recipes/recipe1.bb*.
6. Uživatel klikne na tlačítko *Remove* -> zvolený uzel se přidá do stromového zobrazení odstraněných uzlů, ostatní stromová zobrazení (kromě odstraněných uzlů) se vyčistí; uzel se odstraní z vizualizace.
7. Uživatel klikne na uzel se jménem *recipe2* -> opět dojde k vyplnění údajů.
8. Uživatel klikne na tlačítko *Remove* -> zvolený uzel se přidá do stromového zobrazení odstraněných uzlů, ostatní stromová zobrazení se vyčistí; uzel se odstraní z vizualizace.

9. Uživatel klikne pravým tlačítkem myši na uzel se jménem *recipe1* ve stromovém zobrazení odstraněných uzlů -> objeví se kontextové menu s možnostmi *Open recipe* a *Return to visualization*.

10. Uživatel klikne na možnost I -> ve *Visual Studio Code* se otevře soubor *<cesta>test-gui/recipes/recipe1.bb*.

11. Uživatel klikne pravým tlačítkem myši na uzel se jménem *recipe1* a zvolí možnost *Return to visualization* -> uzel se smaže ze stromového zobrazení odstraněných uzlů; uzel se opět zobrazí ve vizualizaci.

12. Uživatel klikne na uzel s názvem *recipe13* -> opět se v menu vyplní relevantní data.

13. Uživatel najede myší na uzel s názvem *recipe2* ve stromovém zobrazení požadovaných uzlů a stiskne tlačítko *Select node* -> vyskočí chybová hláška, že není možné uzel zvolit, protože je odstraněný.

14. Uživatel klikne pravým tlačítkem myši na uzel se jménem *recipe2* ve stromovém zobrazení požadovaných uzlů -> objeví se kontextové menu s možností *Open recipe*.

15. Uživatel klikne na možnost *Open recipe* -> ve *Visual Studio Code* se otevře soubor *<cesta>test-gui/recipes/recipe2.bb*.

16. Uživatel klikne pravým tlačítkem myši na uzel se jménem *recipe12* ve stromovém zobrazení uzlů vyžadujících zvolený uzel -> objeví se kontextové menu s možností *Open recipe*.

17. Uživatel klikne na možnost *Open recipe* -> ve *Visual Studio Code* se otevře soubor *<cesta>test-gui/recipes/recipe12.bb*.

18. Uživatel najede myší na uzel s názvem *recipe3* ve stromovém zobrazení požadovaných uzlů a stiskne tlačítko *Select node* -> ve vizualizaci dojde ke zvýraznění uzlu s názvem *recipe3* žlutou barvou a dalších uzlů relevantními barvami; stromová zobrazení (kromě odstraněných uzlů) se vyčistí a vyplní novými údaji; dojde k vyplnění údajů o uzlu.

B.2 Scénář 2 - Testy vizualizace ovlivněných uzlů odstraněním zvoleného

Tímto testem byla otestována funkcionalita zvýrazňování uzlů, které by byly ovlivněny odstraněním zvoleného uzlu ve vizualizaci:

1. Uživatel spustí *Visual Studio Code*, otevře složku s testy a zvolí rozšíření v levé části aplikace.

2. Uživatel změní hodnotu výběrového boxu s názvem *Mode* z *DEFAULT* na *Affected nodes*

3. Uživatel klikne na tlačítko *Visualize* -> zobrazí se graf s 16ti uzly.

4. Uživatel klikne na uzel s názvem *recipe1* mimo text s názvem uzlu -> uzly *recipe13*, *recipe12*, *recipe7*, *recipe16*, *recipe3*, *recipe4*, *recipe5*, *recipe6*, *recipe10*, *recipe13*, *recipe14* a *recipe2* se zvýrazní červenou barvou a uzel *recipe1* se zvýrazní barvou žlutou; do stromového seznamu ovlivněných uzlů se přidají červeně zvýrazněné uzly.

5. Uživatel klikne pravým tlačítkem myši na uzel se jménem *recipe5* v seznamu ovlivněných uzlů -> objeví se kontextové menu s možností *Open recipe*.

6. Uživatel klikne na možnost *Open recipe* -> ve *Visual Studio Code* se otevře soubor *<cesta>test-gui/recipes/recipe5.bb*.

7. Uživatel najede myší na uzel s názvem *recipe7* ve stromovém zobrazení ovlivněných uzlů a stiskne tlačítko *Select node* -> uzly *recipe13*, *recipe12*, *recipe1*, *recipe16*, *recipe3*, *recipe4*, *recipe5*, *recipe6*, *recipe10*, *recipe13*, *recipe14* a *recipe2* se zvýrazní červenou barvou a uzel *recipe7* se zvýrazní barvou žlutou; stromový seznam ovlivněných uzlů se vyčistí a přidají se do něj červeně zvýrazněné uzly.

B.3 Scénář 3 - Testy vizualizace použitých licencí

Pomocí tohoto scénáře byla otestována funkcionalita zobrazení použitých licencí:

1. Uživatel spustí *Visual Studio Code*, otevře složku s testy a zvolí rozšíření v levé části aplikace.

2. Uživatel změní hodnotu výběrového boxu s názvem *Mode* z *DEFAULT* na *Licenses*.

3. Uživatel klikne na tlačítko *Visualize* -> zobrazí se graf s 16ti uzly, kde uzly *recipe1*, *recipe5*, *recipe10*, *recipe8*, *recipe15* a *recipe16* budou zvýrazněny červenou barvou, uzly *recipe2*, *recipe7*, *recipe3* a *recipe6* budou

zvýrazněny barvou zelenou, uzly *recipe14* a *recipe4* barvou modrou uzel *recipe13* barvou žlutou a zbylé uzly klasickou azurovou; v bočním menu se zobrazí legenda se seznamem, kde červenou barvou bude napsáno *GPLv2*, zelenou barvou *MIT*, modrou barvou *BSD* a žlutou barvou *MPL*.

B.4 Scénář 4 - Obecné testu prvků grafického uživatelského rozhraní

V tomto scénáři je popsán postup otestování správné funkcionality tlačítek, textových polí a chybových hlášek:

1. Uživatel spustí *Visual Studio Code*, otevře složku s testy a zvolí rozšíření v levé části aplikace.
2. Uživatel stiskne tlačítko *Visualize* -> zobrazí se graf s 16ti uzly.
3. Uživatel stiskne tlačítko *Export SVG* -> zobrazí se dialogové okno pro zadání názvu souboru a jeho uložení.
4. Uživatel nastaví jméno souboru a uloží ho.
5. Uživatel otevře soubor v internetovém prohlížeči -> soubor obsahuje vygenerovaný graf.
6. Uživatel zavře vygenerovaný soubor.
7. Uživatel nastaví hodnotu pole pro *Force link distance* na *1000*.
8. Uživatel stiskne tlačítko *Visualize* -> zobrazí se graf s 16ti uzly, ale oproti původnímu jsou propojené uzly dále od sebe.
9. Uživatel nastaví hodnotu pole pro *Force link distance* zpět na *400*.
10. Uživatel nastaví hodnotu *Force node strength (repulsion)* na *-6000*.
11. Uživatel stiskne tlačítko *Visualize* -> zobrazí se graf s 16ti uzly, ale oproti původnímu jsou uzly, které nejsou propojené dále od sebe.
12. Uživatel nastaví hodnotu *Force node strength (repulsion)* zpět na *-3500*.
13. Uživatel nastaví hodnotu pole pro *Iterations* na *500*

14. Uživatel stiskne tlačítko *Visualize* -> zobrazí se graf s 16ti uzly, ale oproti původnímu jsou uzly jinak uspořádané.
15. Uživatel nastaví hodnotu pole *Iterations* zpět na hodnotu *1*.
16. Uživatel vymaže hodnotu pole *Force link distance*.
17. Uživatel stiskne tlačítko *Visualize* -> zobrazí se chybová hláška, že je algoritmus špatně nastaven.
18. Uživatel nastaví hodnotu pole pro *Force link distance* zpět na 400.
19. Uživatel vymaže hodnotu pole *Force node strength (repulsion)*.
20. Uživatel stiskne tlačítko *Visualize* -> zobrazí se chybová hláška, že je algoritmus špatně nastaven.
21. Uživatel nastaví hodnotu *Force node strength (repulsion)* zpět na *-3500*
22. Uživatel vymaže hodnotu pole *Iterations*.
23. Uživatel stiskne tlačítko *Visualize* -> zobrazí se chybová hláška, že je algoritmus špatně nastaven.
24. Uživatel nastaví hodnotu pole *Iterations* zpět na hodnotu *1*.
25. Uživatel stiskne tlačítko *Visualize* -> zobrazí se graf s 16ti uzly.
26. Uživatel vyplní pole pro vyhledávání hodnotou *5*.
27. Uživatel stiskne tlačítko *Find nodes* -> zelenou barvou se zvýrazní uzly *recipe5* a *recipe15*.
28. Uživatel klikne na uzel *recipe4* -> dojde vybrání uzlu a *recipe4*, zrušení označení zelenou barvou a dojde ke zvýraznění zvoleného uzlu a uzlů s ním spojených.
29. Uživatel vymaže pole pro vyhledávání.
30. Uživatel vyplní pole pro vyhledávání hodnotou *xxx*.
31. Uživatel stiskne tlačítko *Find nodes* -> objeví se hláška, že žádný uzel nebyl nalezen.
32. Uživatel stiskne tlačítko *Find nodes* -> objeví se chybová hláška, že pole nemůže být prázdné.

33. Uživatel zavře záložku s vizualizací, poté stiskne klávesu *F1* nebo kombinaci kláves *CTRL + SHIFT + P* a vyhledá příkaz *Yocto Project Dependency Visualizer: Generate Visualization*, který zavolá -> zobrazí se graf s 16ti uzly.

34. Uživatel přejmenuje soubor receptu *<cesta>test-gui/recipes/recipe1.bb* na *rec1.bb*.

35. Uživatel zvolí uzel *recipe1* ve vizualizaci.

36. Uživatel v bočním menu stiskne tlačítko *Open recipe* -> objeví se chybová hláška, že soubor receptu nemůže být otevřen.

37. Uživatel ve *Visual Studio Code* otevře složku, která neobsahuje složku *build* se souborem *task-depends.dot*.

38. Uživatel stiskne tlačítko *Visualize* -> objeví se chybová hláška, že nelze vytvořit vizualizaci.