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

Diplomová práce

Komponentový systém pro návrh a dělení dopravních sítí pro distribuovanou simulaci dopravy

Plzeň, 2012 Štěpán Cais

Prohlášení

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

V Plzni dne 10. 5. 2012 .....

Štěpán Cais

Poděkování

Rád bych poděkoval svému vedoucímu ing. Tomáši Potužákovi Ph.D. za cenné rady, věcné připomínky a metodické vedení diplomové práce. Dále bych rád poděkoval své rodině za velkou podporu během studia.

Abstract

This work deals with design and implementation of modular graphical editor for traffic network formation and division for distributed simulation of road traffic. The application should be user-friendly and allow adding new algorithms for division of traffic networks. Modularity of application is achieved by using OSGi framework. New division algorithms can be added as plugins. Inputs and outputs of application are XML files. These files contain description of traffic network and topology of interconnection of resulting traffic subnetworks.

Obsah

1 Úvod

S dopravou a dopravními prostředky se setkáváme všichni na každém kroku. Rozvoj dopravy, její nárůst a současný tlak na ekologii nás staví před otázky zefektivnění dopravy a procesů s ní souvisejících. Pro lepší pochopení dopravy a její zlepšení můžeme použít dopravní simulace. Pomocí těchto simulací dokážeme zkoumat dopravní situace a z dosažených výsledků se můžeme pokusit předcházet negativním jevům na komunikacích.

Dopravní simulace mohou být výpočetně velmi náročné a proto je výhodné využít k jejich realizaci distribuované simulace. Užitím distribuované simulace se simulovaný objekt rozdělí na několik částí a každá tato část je zpracovávána na vlastním výpočetním uzlu. Díky tomu se výpočetní složitost rozloží mezi několik výpočetních jednotek a výsledku je dosaženo v lepším výpočetním čase.

Rozdělení simulace na více částí s sebou přináší otázku, jak dělení provést co nejlépe. Je mnoho přístupů a algoritmů, které tuto problematiku řeší, některé jsou známé již relativně dlouhou dobu a další zřejmě ještě čekají na své objevení. Nejběžněji se setkáme s rozdělením dopravní sítě do několika částí, které se pak simulují na jednotlivých uzlech.

Pro programátora aplikace realizující rozdělení simulace vyvstává otázka, jak co nejlépe připravit program pro přidání dalších algoritmů v budoucnosti. Jako jedna z možných odpovědí se nabízí vývoj aplikace založené na komponentách.

Komponentové programování nahlíží na aplikaci jako na softwarový balík složený z různých komponent, které jsou samy na sobě nezávislé. Aplikace tak může být postupně skládána jako stavebnice z různých komponent od různých výrobců, které mohou být navíc opakovaně využitelné v dalších programech. Pomocí tohoto přístupu může programátor vytvořit software tak, že kdykoliv v budoucnu bude možné přidat další funkčnost aplikaci, či upravit její části bez širokých zásahů do zdrojového kódu.

Úkolem diplomové práce je vytvořit modulární grafický editor s příjemným uživatelským rozhraním, který umožní vytvářet a editovat modely dopravních sítí ve formátu XML, a zároveň bude umožňovat dělení těchto modelů. Jednotlivé

algoritmy pro dělení modelů dopravní sítě budou realizovány pomocí externích modulů, které bude možné kdykoliv do programu přidat.

2 Diskrétní distribuovaná simulace dopravy

Abychom mohli popsat diskrétní distribuovanou simulaci dopravy, je třeba probrat nezbytné pojmy o simulaci dopravy.

2.1 Simulace

Počítačová simulace modeluje určitý systém v průběhu času a jejím cílem je realizovat fungující model tohoto systému s co nejmenšími odchylkami od reality. Hlavním kritériem na simulaci je správnost výsledků, které – až na povolené statistické odchylky – musí být stejné, jako výsledky původního systému [Ham97]. Počítačové simulace můžeme klasifikovat podle mnoha kritérií [Fuj00], jednou z možností klasifikace je rozdělení [Cai09] simulací podle:

- Účelu simulace
- Reprezentace času

2.1.1 Účel simulace

Určuje k jakému konkrétnímu účelu bude simulace použita. Můžeme dále dělit [Cai09] na *simulace prostředí* a *analytické simulace*

Simulace prostředí je nejjednodušší si představit jako počítačovou hru, která simuluje reálné prostředí. Uživatel – hráč – může s tímto prostředím interagovat, komunikovat s dalšími účastníky simulace a ovlivňovat další vývoj v simulaci. Velkou mírou je zde nahlíženo na faktor času. Aby byla simulace co nejvěrnější své reálné podobě, je třeba, aby čas v simulaci utíkal pro uživatele alespoň přibližně stejnou rychlostí, jako čas v reálném prostředí. Můžeme říci, že v tomto typu simulace není kladen velký důraz na správné pořadí průběhu událostí – předpokládá se, že když uživatel nemůže zaregistrovat přesné pořadí událostí, není nutné přesné pořadí dodržet [Cai09]. Tyto druhy simulací můžeme nalézt ve vojenských simulátorech, v simulátorech dopravních prostředků nebo v již zmíněných počítačových hrách [Fuj00].

Analytická simulace slouží k simulování různých přírodních a vědecko-technologických jevů. Tyto simulace se využívají již velmi dlouhou dobu a jejich výsledkem může být například simulovaná dráha letu SCUD rakety nebo předpověď počasí [Fuj00]. Na rozdíl od simulace prostředí se čas v analytické simulaci nemusí chovat reálně, tj. jeho plynutí nemusí odpovídat reálnému prostředí [Cai09] (dráhu rakety nemusíme sledovat v reálném čase, stačí nám pouze znát výslednou trajektorii).

2.1.2 Reprezentace času

Reprezentace času v simulaci určuje, jakým způsobem bude čas v simulaci ubíhat. Simulace můžeme podle užitého časového mechanismu dělit na simulace spojité a diskrétní [Cai09].

Spojité simulace využívají spojitý čas. Model simulovaného systému bývá popsán pomocí diferenciálních rovnic, na jejichž základě je dopočítáván aktuální stav systému – ten můžeme díky spojitosti modelu dopočítat pro každý bod v čase. Spojité modely najdeme často u simulací dynamických systémů, například při modelování změn planetárního klimatu [Cai09].

Stav a vlastnosti se v *diskrétních modelech* mění v konkrétně definované množině bodů. Pouze v těchto bodech se přepočítává stav simulace. Množina bodů může být určena buď jako množina časových údajů nebo jako množina událostí. Podle typu množiny dále dělíme diskrétní modely na dva podtypy [Cai09] – *časově krokovaný model* a *událostně řízený model*

V *časově krokovaném modelu* je množina bodů určena časovými intervaly, které se získají rozdělením simulace na několik stejně dlouhých částí. Při přechodu od jednoho bodu k druhému je vždy znovu přepočten celý stav a vlastnosti modelu [Cai09]. Další informace o tomto způsobu simulace se lze dočíst v [Gor04], [Kle05], [Cet03].

Událostně řízený model má množinu bodů reprezentovanou jako množinu událostí s danou časovou známkou. Simulovaný systém jde od jedné události k druhé a v daném čase události provede akci, která je v události definována (změna stavu modelu). V době mezi dvěma událostmi se bere stav modelu jako neměnný [Cai09].

2.2 Dopravní simulace

Dopravní simulace řadíme z velké části do simulací s diskrétním modelem. Modely dopravních simulací kategorizujeme nejčastěji podle úrovně detailů, které nabízejí. Podle úrovně detailů můžeme modely rozdělit na několik typů [Pot06]: *makroskopické modely*, *mezoskopické modely* a *mikroskopické modely*.

2.2.1 Makroskopické modely

Nejjednodušší a nejstarší [Lig55] reprezentace modelu dopravní simulace jsou *makroskopické modely*, které místo simulace jednotlivých vozidel pracují s veličinami definující dopravní toky jednotlivými ulicemi (hustota, rychlost provozu atd.) [Cai09]. Makroskopické modely nemusí dosahovat tak přesných výsledků jako ostatní modely, ale díky své jednoduchosti mohou sloužit k odhadování náročnosti simulace nebo testovacím účelům.

2.2.2 Mezoskopické modely

Mezoskopické modely reprezentují přechod mezi makroskopickými a mikroskopickými modely. Mezoskopické modely se již zaměřují na detailnější reprezentaci dopravy – simulace pracuje se skupinami vozidel, kde každá skupina vozidel má specifické vlastnosti (rychlost, směr) [Cai09], více lze najít v [Nag96] a [Niz02]. Tyto modely umožňují většinou získat přesnějšími výsledky než makroskopické modely, ovšem na úkor zvýšení náročnosti simulace.

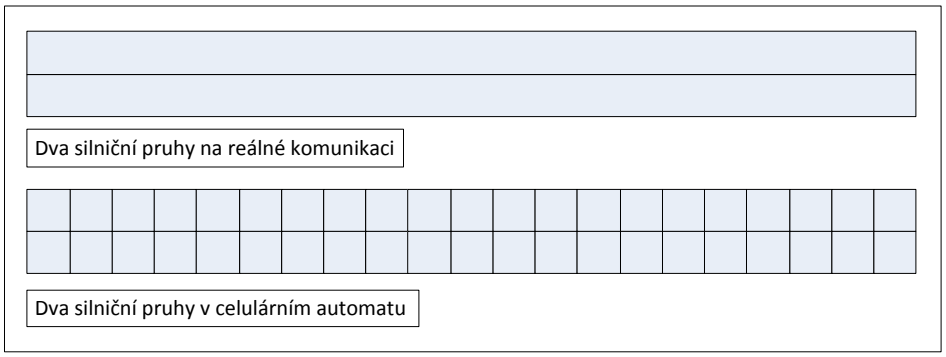
2.2.3 Mikroskopické modely

V *mikroskopických modelech* je simulována doprava již samostatnými dopravními prostředky, kde každý dopravní prostředek má své individuální vlastnosti jako rychlost, směr atd. Simulace využívající mikroskopické modely mají naproti předchozím dvěma typům modelů přesnější výsledky. Lepší výsledky jsou důsledkem detailnější reprezentace modelu, avšak daní za zpřesnění je vysoká výpočetní složitost mikroskopických modelů. Jedním z možných řešení, jak výpočet simulace urychlit, je využití distribuované simulace [Cai09].

Mikroskopické modely mohou být reprezentovány různě, například *celulárním automatem* nebo *metodou následování vozidel* [Cai09].

Metoda následování vozidel využívá diferenciálních rovnic k popisu pohybu vozidel. Každé vozidlo je definováno svojí rychlostí a polohou, a může být v libovolné části komunikace [Cai09]. Více o metodě následování vozidel lze najít v [Wag04].

Celulární automaty mohou být výhodně použity v dopravní simulaci díky možnosti buněčné reprezentace komunikací [Cai09]. Každá komunikace – jak je vidět na obr. 1 – je v těchto automatech rozdělena na části o konkrétní délce, kterým se říká buňky. Každá buňka je buď prázdná, nebo obsazená vozidlem.



Obrázek 1: Porovnání reálných silničních pruhů a pruhů v celulárním automatu

Program *MME – Modular Map Editor* pracuje s mapami pro distribuovanou simulaci dopravy využívané v programu *DUTS – Distributed Urban Traffic Simulator*. Program DUTS pracuje jak s modely celulárního automatu, tak s metodou následování vozidel.

2.3 Distribuovaná simulace

V případě náročné simulace je vhodné simulaci rozdělit do několika menších částí a ty pak simulovat zvlášť [Cai09]. Pokud simulaci dělíme na části, narazíme na tyto otázky [Pot07]:

- Jak simulaci rozdělit na části
- Jak rozdělené části synchronizovat

2.3.1 Rozdělení simulace na části

Způsob, kterým je simulace rozdělena (dekomponována), závisí na typu simulace. Simulaci můžeme rozdělit na části několika způsoby [Cai09]. U každého způsobu dekompozice je důležité klást důraz na rovnoměrné rozdělení simulace, jinak může docházet k přetěžování jednotlivých uzlů [Pot07]. Zde jsou tři nejobvyklejší způsoby dekompozice simulace:

- Modulová dekompozice
- Dekompozice času
- Dekompozice prostoru

Modulová dekompozice spočívá v rozdělení simulace do několika samostatných modulů, kde každý modul provádí v rámci simulace specifickou práci [Cai09]. Příkladem může být simulace supermarketu, kde jedním modulem je simulace práce kas a druhým například pohyb zákazníků po supermarketu. Více o modulové dekompozici lze najít v [Kle98].

Při použití *časové dekompozice* je simulace rozdělena na několik časových intervalů a každý interval je simulován zvlášť [Cai09]. Příkladem může být opět simulace supermarketu, kde na jednom výpočetním uzlu je simulován ranní provoz a na druhém večerní. Spojením výsledků pak získáme simulaci jednoho dne.

Prostorová dekompozice rozděluje simulaci prostorově. Celý model, se kterým pracuje simulace, se prostorově rozdělí na několik částí a tyto části jsou pak simulovány na jednotlivých uzlech. V našem příkladu se supermarketem by to znamenalo, že například první uzel simuluje první půlku pokladen, druhý uzel druhou půlku pokladen, třetí uzel oddělení zeleniny atd. Prostorová dekompozice bývá často využívána k simulace dopravy, kdy každému uzlu přiřadíme určitou část dopravních komunikací z celé dopravní sítě.

2.3.2 Synchronizování dekomponovaných částí

Při distribuované simulaci je nutné udržovat uzly synchronizované (pokud se nejedná například o časově dekomponovanou simulaci), protože jednotlivé části si-

mulace spolu potřebují komunikovat a vyměňovat si navzájem informace. Pokud budeme uvažovat, že každý uzel může pracovat různou rychlostí, musíme řešit problém časové souslednosti – k uzlu může doputovat informace, která je již zastaralá [Cai09]. Obecně lze tento problém řešit dvěma způsoby [Pot06]:

- Konzervativní přístup k synchronizaci
- Optimistický přístup k synchronizaci

Při použití *konzervativního přístupu k synchronizaci* komunikují uzly pod takovým komunikačním protokolem, že chyba v časové souslednosti nemůže nastat. Pro zabezpečení tohoto typu komunikace se mohou použít časové známky. Příkladem může být množina komunikujících uzlů, kdy každý uzel odesílá zprávu pouze tehdy, když si je jistý, že nikdo ze sousedních uzlů nemůže odeslat zprávu s menší časovou známkou [Cai09].

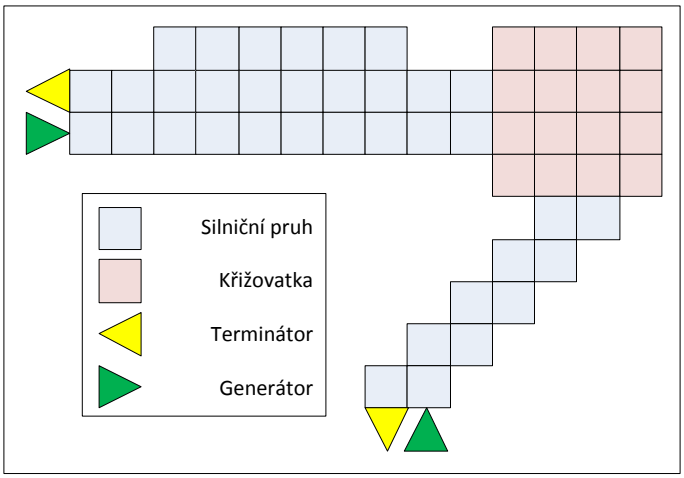
Optimistický přístup k synchronizaci je opakem konzervativního přístupu. Uzly spolu komunikují bez zabezpečení časové souslednosti a v případě kolize se o nápravu postará kontrolní algoritmus [Cai09] například zasláním speciálního typu zprávy [Fuj00].

2.4 DUTS (Distributed Urban Traffic Simulator)

Program DUTS patří mezi simulace s mikroskopickou úrovní detailů, řadíme ho k simulacím s diskrétním, časově krokovaným modelem. Program DUTS se využívá k simulování dopravy v městských částích, a to buď za použití metody následování vozidel, nebo celulárního automatu [Pot09]. Program MME je vytvářen pro práci s mapami celulárního automatu, kdy jedna buňka má 2,5 metru. Tyto mapy lze však bez úpravy využít i pro model následování vozidel.

Dopravními prostředky jsou v DUTS vozidla s různou délku, rychlostí a směrem pohybu. Rychlost vozidla je definována jako počet uražených buněk za sekundu [Cai09]. Pohyb vozidla se v simulaci vypočítává podle pravidel *zrychlení*, *zpomalení*, *začlenění náhody* a *přesun vozidla* [Pot07].

Všechna vozidla se nejprve snaží zrychlit na svou maximální rychlost. Vozidla zrychlují do doby, než před sebou objeví překážku, nebo než dosáhnou maximální rychlosti buněk za sekundu (tj. 2,5 * 6 m/s = 15 m/s = 54 km/h). Srážky nejsou v simulaci povoleny. Při zaregistrování překážky vozidlo zpomalí tak, aby ke srážce nedošlo. Zároveň může pohyb vozidla ovlivnit náhoda (chování řidiče, kvalita komunikace). Jakmile jsou všechny předchozí kroky dokončeny, vozidlo se přesune podle dopočtených parametrů na konkrétní pozici v simulaci.



Obrázek 2: Objekty v systému DUTS

V DUTS najdeme několik základních objektů (zobrazené na obr. 2), kde každý objekt je složen ze čtvercových buněk stejné délky. Tato velikost odpovídá v reálu 2,5 metru (viz výše). Mezi základní objekty patří:

- Dopravní pruhy
- Křižovatky
- Terminátory
- Generátory

2.4.1 Dopravní pruhy

Reprezentují silniční pruhy. Každá silnice je složena z jednoho a více pruhů. Pruhy propojují další objekty na mapě a probíhá na nich většina simulace – na pruzích

je simulován pohyb vozidel.

2.4.2 Křižovatky

Odpovídají reálným křižovatkám, ale jsou omezeny na čtyři ramena. Program MME také pro zjednodušení předpokládá, že do každého ramena vedou maximálně dva vjezdy a dva výjezdy. Pohyb vozidel křižovatkou je rovněž simulován.

2.4.3 Generátory

Generují s danou frekvencí vozidla do konkrétního pruhu. Jsou vždy napojeny na pruh, do kterého posílají vozidla. Slouží jak ke generování „nových“ vozidel do simulace, tak k přeposílání vozidel mezi jednotlivými výpočetními uzly v případě distribuované simulace.

2.4.4 Terminátory

Jsou opakem generátorů. Odebírají vozidla z pruhu, který je na ně napojen. Terminátory se používají buď k ukončování pohybu vozidla v simulaci, nebo jako kolektory vozidel, které budou přeneseny do dalšího výpočetního uzlu v případě distribuované simulace.

3 Dělení distribuované simulace dopravy

Jak bylo uvedeno v předchozí kapitole, simulaci lze dekomponovat několika způsoby. Při dělení distribuované simulace musíme zvážit nejprve všechny možnosti dekompozice.

Modulová dekompozice by byla vhodná v případě, kdyby se program DUTS skládal z několika výpočetně náročných částí. Potom by každá část mohla být převedena do samostatného modulu a tam simulována zvlášť. V programu DUTS je největším spotřebitelem výkonu pohyb vozidel, z tohoto důvodu není modulová dekompozice sítě vhodná [Cai09].

Při použití časové dekompozice by bylo třeba počítat stavy simulace na mezích intervalů dekompozice. Pro simulaci dopravy jsou však tyto stavy velmi komplexní, proto ani tato dekompozice není vhodná [Cai09].

Program DUTS pracuje nad reprezentací reálných prostorů – s mapami [Cai09]. Objekt mapy může být rozdělen na několik částí mnoha způsoby, a proto je prostorová dekompozice vhodným kandidátem na použití. Dalším důvodem může být fakt, že silniční síť lze reprezentovat grafem. Dělení grafu je známý problém a existuje několik osvědčených řešení vedoucích k vyřešení tohoto problému. Problematice dělení dopravní sítě bude věnována další kapitola.

3.1 Možnosti dělení dopravní sítě distribuované simulace dopravy

Existuje mnoho způsobů, jak dělit dopravní sítě. Nejintuitivnější možností, jak dělit dopravní síť, je její rozdělení do několika menších stejně velkých, pravoúhlých částí. Tento typ dělení je jednoduchý a přímočarý, nicméně postrádá jakoukoliv optimalizaci, a tak většinou získáme několik podsítí, jejichž výpočetní náročnost je velmi rozdílná. Kvůli rozdílné náročnosti simulování podsítí může docházet k zpomalování celé simulace (jeden uzel pracuje pomaleji než ostatní a ty na něj musí čekat). Z tohoto důvodu je tento způsob dělení vhodný pro sítě, kde jsou komunikace i hustota provozu rozloženy rovnoměrně [Pot07].

Mezi další způsoby, jak dělit síť pro distribuovanou simulaci dopravy, patří množina algoritmů, které pracují s ohodnocením komunikací v dopravní síti. Po-

stup dělení se u těchto algoritmů skládá se ze dvou kroků:

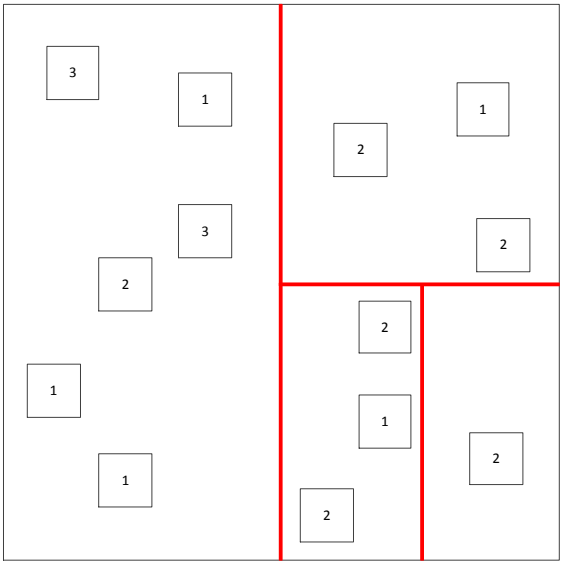
- Ohodnocení dopravní sítě
- Dělení ohodnocené sítě do podsítí

V prvním kroku jsou ohodnoceny všechny komunikace, které se nacházejí v dělené síti. Ohodnocení znamená, že každé komunikaci je přiřazena hodnota, která definuje její váhu při dělení. Ohodnocení komunikací je většinou vztaženo k hustotě provozu vozidel, které v průběhu simulace projedou skrz danou komunikaci, ale je možné zvolit i jiný druh ohodnocení. Ohodnocení se provádí před samotným rozdělením sítě, a to většinou sekvenčním provedením celé simulace (simulace se provede na jednom uzlu). Z toho důvodu mohou být kladeny vyšší nároky na hardware. Pro zjednodušení ohodnocení se první simulace může provést v menším simulačním rozlišením, tj. místo mikroskopické simulace můžeme použít mezoskopickou nebo makroskopickou simulaci [Pot10].

Ve druhém kroku je silniční síť rozdělena podle získaného ohodnocení z kroku prvního. Existuje mnoho algoritmů, které řeší rozdělení ohodnocené sítě. V následujících kapitolách jich bude několik popsáno.

3.1.1 Rozdělení ohodnocené sítě pomocí ortogonální rekurzivní bisekce

Algoritmus se dívá na silniční síť jako na neorientovaný ohodnocený graf. Vstupem algoritmu je dělená podsíť a požadovaný počet podsítí. Při použití této metody je nejprve každému uzlu sítě přiřazena hodnota součtu všech komunikací, které jsou na uzel napojené. Uzly jsou reprezentovány svými souřadnicemi v síti, je tedy známé jejich topologické umístnění. Po přiřazení hodnot uzlům jsou uzly rozděleny na dvě části tak, že je vedena vertikální dělící čára, která uzly rozdělí na dvě co nejvyváženější půlky s ohledem na ohodnocení uzlů. V dalším kroku je vybrána půlka s větším ohodnocením a tato půlka je opět rozdělena, tentokrát horizontálně. Algoritmus pak pokračuje rekurzivně, dokud není dosaženo požadovaného počtu podsítí [Nag01]. Princip algoritmu je zachycen na obr. 3.



Obrázek 3: Rozdělení sítě do několika podsítí pomocí ortogonální rekurzivní bisekce

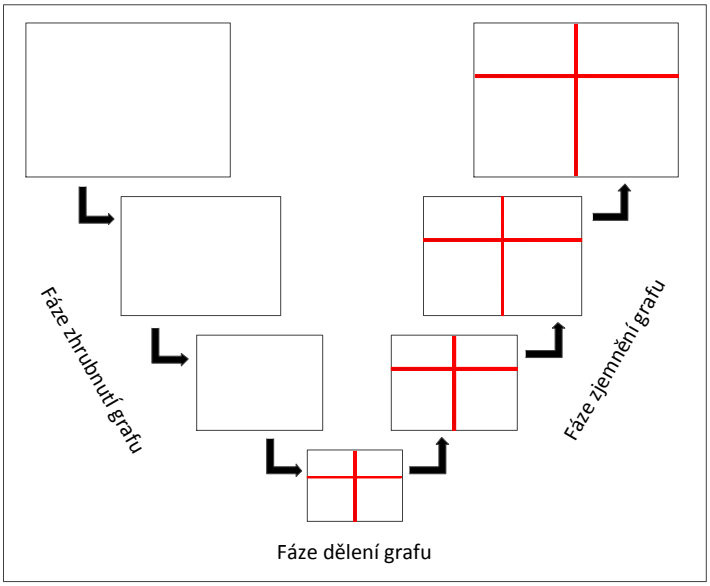
3.1.2 Multilevelové dělení ohodnocené sítě

Algoritmus má stejné vstupy jako ortogonální rekurzivní bisekce (ORB) a také se na síť dívá jako na graf, ale na rozdíl od ORB nedělí celý graf hned. Dělení se skládá ze tří fází: *zhrubnutí grafu*, *dělení grafu*, *zjemnění grafu*. Fáze dělení jsou zobrazeny na obr. 4.

V první fázi se z původního grafu získává posloupnost po sobě jdoucích aproximací. Každá aproximace je reprezentací původního grafu ve zmenšené podobě. Obvyklým způsobem k získání aproximací je spojování párování grafu (párování grafu je množina hran grafu, kde se žádná hrana nevztahuje ke stejným uzlům). Párování grafu mohou být vybrána náhodně nebo například výběrem párování s nejvyšším ohodnocením. Spojováním párování grafu postupně rozkládáme aproximace grafu na menší grafy [Abo06].

Po definovaném počtu aproximací obsahuje graf mnohem méně hran a vrcholů než na počátku. V tuto chvíli začíná fáze *dělení grafu*. Dělení se provádí například zmiňovanou metodou ORB, ale je možné zvolit i jiné algoritmy, viz [Abo06]. Dělením získáme rozdělení nejmenší aproximace grafu.

V poslední fázi dochází k rekonstrukci původního grafu postupnou projekcí



Obrázek 4: Fáze multilevelového dělení ohodnocené sítě

rozdělení nejmenší aproximace. Při rekonstrukci je zároveň použit speciální algoritmus na redukci množství dělených hran [Con03]. Opakováním tohoto postupu můžeme zrekonstruovat původní graf, nyní již rozdělený podle rozdělení nejmenší aproximace.

3.1.3 Rozdělení ohodnocené sítě prohledáváním do šířky

Vstupem algoritmu je graf silničních sítí a počet podsítí, do kterých má být rozdělen. Algoritmus nejprve sečte celkové ohodnocení sítě a tuto hodnotu vydělí počtem podsítí – získá tak maximální ohodnocení komunikací pro jednu podsíť.

Ve druhém kroku začíná prohledávání do šířky. Algoritmus vybere jeden uzel, přiřadí ho do první podsítě a přičte ohodnocení hran, které do něj vedou, do celkového ohodnocení. Dále pokračuje s prohledáváním do šířky se sousedy uzlu. U každého nově objeveného uzlu je jeho ohodnocení hran započteno do celkového ohodnocení a uzel je přiřazen do aktuální podsítě. Jakmile by mělo ohodnocení podsítě přesáhnout přidáním objeveného uzlu dané maximum, je uzel přidán do další podsítě a tato podsíť je označena jako aktuální podsíť. V rámci přechodu do nové podsítě je vynulováno celkové ohodnocení. Takto algoritmus pokračuje,

```
maxOhodnoceniPodsite = celkoveOhodnoceniSite / pocetPodsiti;
ohodnoceniUzlu = 0;
aktualniIDPodsite = 0;
aktualniUzel = uzly.get(0);
frontaUzlu.push(aktualniUzel);
dokud není fronta uzlů prázdná {
   aktualniUzel = frontaUzlu.pop();
   sousedeUzlu = aktualniUzel.vratSousedy()
   pro všechny sousedy v sousedeUzlu {
      if (nebyl soused ještě objeven) {
         přidej souseda do frontaUzlu;
         ohodnoceniUzlu += soused.getOhodnoceni();
      }
   }
   aktualniUzel.setPodsit(aktualniIDPodsite);
   if (ohodnoceniUzlu > maxOhodnoceniPodsite) {
            aktualniIDPodsite++;
            ohodnoceniUzlu = 0;
   }
}
```

Obrázek 5: Algoritmus rozdělení ohodnocené sítě prohledáváním do šířky v pseudokódu.

dokud nejsou objeveny všechny uzly [Pot10]. Algoritmus je popsán na obr. 5.

Postup lze ještě optimalizovat tím, že se provádí nad dělenou sítí několikrát, vždy s jiným uzlem daným v počátečním vstupu. Tím je možné vyzkoušet všechny možnosti průchodu grafem. Na závěr se vybere možnost, ve které je nejmenší počet rozdělených pruhů.

3.1.4 Genetický algoritmus dělící ohodnocené sítě

Genetické algoritmy jsou založené na principech přírodního výběru a genetiky. Jejich postup je jednoduchý a spočívá v postupném vytváření nových generací řešení problému. Z každé nové generace se mohou vybrat nejlepší jedinci a z těchto jedinců je vytvořena generace další. Každý jedinec představuje jedno konkrétní řešení problému [Pot11].

Generace jedinců bývá reprezentována polem hodnot, kde hodnota může být boolean (PRAVDA / NEPRAVDA) nebo číslo. Prvním krokem genetického algoritmu je vytvoření první generace, kdy se vytvoří několik polí s hodnotami. První generace bývá vytvořena náhodně, vygenerováním nebo vybráním náhodných jedinců. V následujícím kroku je vytvořena generace další, a to pomocí *křížení*,

mutace nebo *reprodukce* jedinců. Každý jedinec je v tomto kroku křížen s jiným jedincem, tj. část vlastností jedince přejde do vlastností druhého a vice versa. Dále jsou jedinci mutováni tj. jedincovi jsou změněny jeho vlastnosti. Poslední možností je, že dojde k reprodukci, kdy jedinec nezměněn přechází do další generace. Tento postup se opakuje, dokud není dosaženo zastavovací podmínky, nebo dokud není dosaženo maximálního počtu generací. K ohodnocování jedinců slouží takzvaná *fitness funkce*, která udává úspěšnost jedince [Pot11].

Pro dělení silniční sítě můžeme využít genetický algoritmus tak, že jedinec bude reprezentován polem délky rovné celkovému počtu uzlů. Každý index pole je jeden uzel a konkrétní hodnota v poli určuje podsíť, do které je uzel přiřazen. Po vytvoření druhé generace (aplikováním mutace, křížením či reprodukcí), použijeme fitness funkci na ohodnocení získaných jedinců a z nejlepších jedinců vytvoříme generaci další [Pot11].

Jednou z nejdůležitějších částí genetického algoritmu je zvolení vhodné *fitness funkce*, která ohodnocuje kvalitu jedinců. V případě dělení silniční sítě pro distribuovanou dopravu je možné *fitness funkci* upravit tak, aby obsáhla nejdůležitější hodnotící parametry dělení distribuované simulace dopravy tj. nejmenší počet dělených pruhů v rámci jedince a co nejvyváženější vytvářené podsítě. Pro výpočet vyváženosti podsítě můžeme použít vzorec [Pot11] :

$$E = 1 - \frac{\sum \frac{|w_{s_i} - \overline{w_s}|}{\overline{w_s}}}{M}$$

E je vyváženost jedince, w_{s_i} je celkové ohodnocení í-té podsítě, $\overline{w_s}$ je průměrná hodnota ohodnocení všech podsítí a M je počet podsítí.

Pro ohodnocení nejmenšího počtu dělených pruhů můžeme použít výpočet [Pot11]:

$$C = \frac{L - L_d}{L}$$

L označuje celistvost jedince, L je celkový počet komunikací a L_d je počet rozdělených komunikací.

Fitness funkci pak vyjádříme jako vztah

$$F = E + C$$

F je *fitness funkce* a E a C dosazené parametry. V závislosti na požadovaných výsledcích je možné *fitness funkci* dále upravovat [Pot11].

3.2 Rozdělení silniční sítě pro systém DUTS

Během dělení dopravní sítě pro systém DUTS musíme vyřešit několik otázek. Mezi ně patří:

- Rozdělení silniční sítě do podsítí
- Zasílání vozidel mezi podsítěmi
- Synchronizace systému

3.2.1 Rozdělení sítě pro systém DUTS

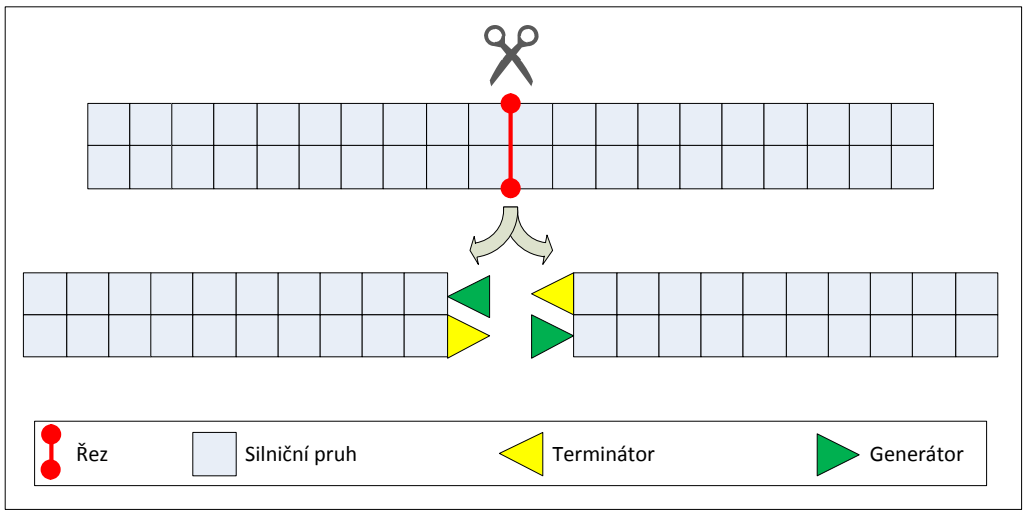
Každá mapa je jednoznačně definována svým jménem a je složena z množiny křižovatek, pruhů, terminátorů a generátorů. Všechny objekty na mapě jsou propojené a utvářejí tak graf.

> Graf je G dvojice $G = (V, E)$, kde V je konečná množina a $E \in \binom{V}{2}$ přičemž:
>
> $$\binom{V}{2} = \{\{x, y\} : x, y \in V, x \neq y\}$$
>
> je množina všech dvouprvkových množin *(neuspořádaných dvojic)* prvků množiny V. Prvky množiny V nazýváme *vrcholy*, prvky množiny E pak *hrany grafu* G. Vrcholy $x, y \in V$ jsou sousední, pokud $\{x, y\} \in E$ [Cad04].

Touto definicí je určen neorientovaný graf bez násobných hran (z jednoho vrcholu vede do druhého pouze jedna hrana). Vrcholy V grafu G jsou terminátory, generátory a křižovatky, hrany E jsou silniční pruhy. Komunikační síť v DUTS

může obsahovat několik opačně orientovaných silničních pruhů propojujících dva vrcholy, ale pro dekompozici se všechny silniční pruhy mezi dvěma vrcholy berou jako jedna neorientovaná hrana.

Terminátory, generátory a křižovatky jsou nedělitelnými objekty na mapě. Jediným objektem, který lze na mapě dělit, je silniční pruh. Dělení silničního pruhu je znázorněno na obr. 6. Silniční pruh propojuje křižovatky mezi sebou, nebo propojuje křižovatky s terminátory a generátory.



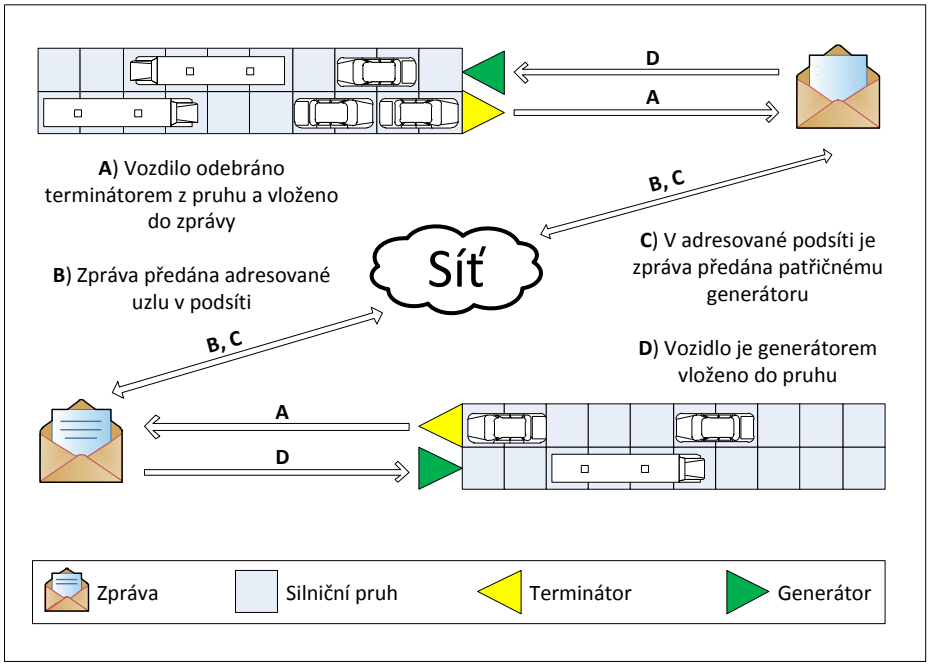
Obrázek 6: Dělení silničního pruhu

Jako vhodný dělící algoritmus pro mapy systému DUTS byl vybrán algoritmus dělení silniční sítě pomocí průchodu grafu do šířky. Tento algoritmus byl vybrán z důvodu jednoduchosti své implementace. Více o konkrétní implementaci se lze dočíst v kapitole 7.

3.2.2 Zasílání vozidel mezi podsítěmi

Protože všechny podsítě vznikly rozdělením původní konzistentní sítě, je potřeba zajistit komunikaci tj. zasílání vozidel z jedné podsítě do druhé. Vozidla jsou zasílána mezi podsítěmi pomocí speciálních zpráv [Pot07]. V situaci, kdy vozidlo dojede na „konec“ podsítě, je nutné ho buď úplně odstranit ze simulace (podsíť nemá žádného souseda, kam auto odeslat) nebo se vozidlo pošle na sou-

sední podsíť. Toto odebrání vozidla a jeho následné vygenerování v další podsíti je umožněno dvojicí párových generátorů a terminátorů, kde terminátor se stará o odebrání vozidla a generátor o jeho vygenerování [Cai09]. Princip zasílání je ukázán na obr. 7.



Obrázek 7: Zasílání vozidel mezi podsítěmi v systému DUTS.

V programu MME je tato problematika řešena pouze v rámci vygenerování párových generátorů a terminátorů v průběhu dekompozice sítě, samotné přeposílání vozidel řeší systém DUTS.

3.2.3 Synchronizace systému podsítí

V systému DUTS je využíván princip synchronizace *master – slave*. Master uzel řídí synchronizaci systému tím, že posílá slave uzlům řídící zprávy. Komunikace pak vypadá tak, že slave uzel provede simulaci podsítě pro daný krok a pošle masteru zprávu, že je hotov. Master tyto zprávy shromažďuje a jakmile je dostane od všech uzlů, rozešle všem příkaz k pokračování simulace dalším krokem [Cai09].

4 Komponentové programování

V průmyslu je již velmi dlouhou dobu běžné setkat se s procesem výroby, kdy jednotlivé části výrobku jsou vytvářeny nezávisle na sobě. Každá část může být vyrobena v jiné části zeměkoule, jinou firmou a jinými technologickými postupy. Jako příklad nám může posloužit automobilový průmysl. V automobilovém průmyslu jsou jednotlivé součástky – ale i celé partie vozu – vyráběny a dodávány různými firmami. V cílové automobilce se pak všechny části vozidla pouze upraví na míru, smontují dohromady a sestaví se z nich požadované vozidlo.

Při použití tohoto postupu se automobilka nemusí starat o výrobu každé součástky a díky tomu dokáže ušetřit mnoho času a peněz. Nejdůležitějším kritériem na tento *outsourcing* dílů je dodržení rozhraní výrobku – například při výrobě dveří je nutné zajistit, aby dveře šly správně usadit do auta, tj. dveře musí mít předem daný počet pantů a tvar. Druhým kritériem je kvalita dodávaných dílů. Je nutné, aby každé vozidlo bylo bezpečné. Firmy, které dodávají součástky, tak musí dokázat vyrábět díly bezpečné a odpovídající daným normám.

V programování je myšlenka komponentové výroby – tedy skládání cílového softwaru z menších částí – známá již delší dobu. Realizace této myšlenky se dostávala mezi širší pole uživatelů relativně pomalu a až v poslední dekádě se tento přístup k výrobě softwaru rozšířil. Abychom mohli pokračovat dál, je nutné nejprve definovat pojem *softwarová komponenta*.

4.1 Softwarová komponenta

V literatuře můžeme nalézt několik definic *softwarové komponenty*. Zřejmě nejpřesnější definici softwarové komponenty obsahuje [Szy02]:

> Softwarová komponenta je nezávisle vyvíjená znovu použitelná a znovu spustitelná binární jednotka, která může být skládána spolu s dalšími komponentami tak, že vytvářejí větší funkční systém.

Tato definice nám představuje softwarovou komponentu v podobném duchu, jako jsme si v předchozí kapitole ukázali outsourcovanou výrobu automobilových

dílů. Softwarová komponenta je v tomto případě samostatně dodávaným dílem vozidla a společně s ostatními díly (komponentami) vytváří větší celek, tedy celé vozidlo (softwarový systém).

Stejně jako automobilka nemusí znát přesnou výrobu dílu, ani softwarový systém složený z komponent nemusí znát vnitřní strukturu komponenty. Zároveň jako u zmiňovaných dveří musí být přesně dány rozměry a počet pantů, každá komponenta musí splňovat požadavky na služby, které má poskytovat. Na komponentu je tak nahlíženo jako na tzv. „black-box“, kdy cílový systém používá komponentu pro realizaci předem definovaných služeb, ale už se nezajímá o to, jakým způsobem komponenta služby zajišťuje.

Podobně je to i se zajištěným bezpečnosti komponent. Každý výrobce komponent se stará o bezpečnost svých komponent, a to jak splněním předem daných podmínek výroby komponenty, tak i například certifikací. Tímto přístupem se zajišťuje správný chod komponent v různých systémech.

4.2 Proč vytvářet software s použitím komponent

Existují dva základní přístupy, jak je dnes vytvářen software. První je vytvoření kompletně nového softwaru pro zákazníka neboli vytvoření „řešení na zelené louce“. Druhým je zakoupení již vytvořeného řešení podobného (nebo stejného) problému a následné svázání tohoto řešení se zákazníkovými procesy. Oba dva přístupy jsou dnes hojně využívané, přičemž například v podnikových informačních systémech je v posledních letech zaznamenáno zvyšování použití druhého přístupu na úkor prvního [Bas08].

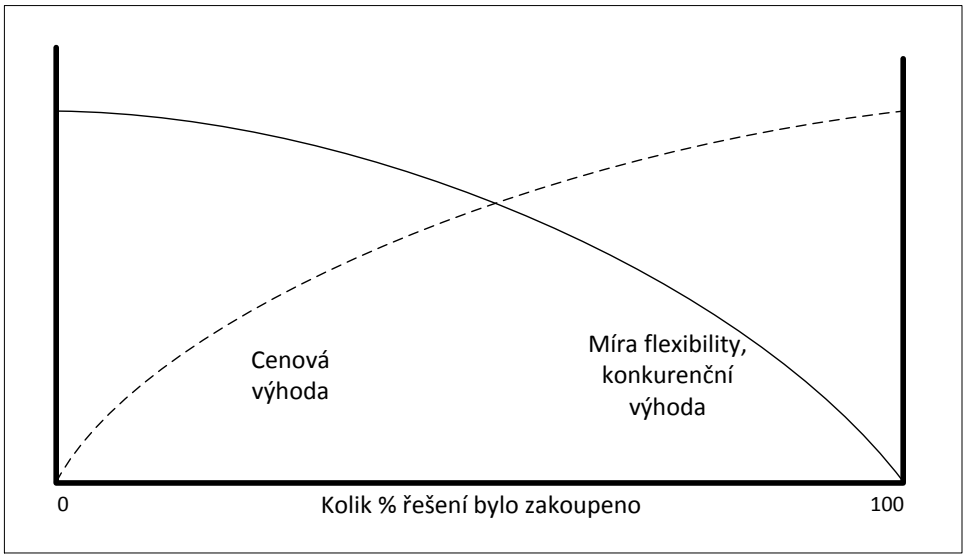
Pokud je software vytvářen na zakázku přímo pro zákazníka, je řešení „ušito“ na míru, aby přesně vystihovalo a řešilo zákazníkův problém. Toto je základní výhoda použití principu na zelené louce, protože řešení přímo postihuje zákazníkův problém. V ideálním případě je řešení zákazníkem objednáno, zpracovatelskou firmou v daném čase vytvořeno a následně dodáno zákazníkovi, který program bez problémů používá. Bohužel v reálném světě dochází k mnoha komplikacím, které doprovázejí celý cyklus vytváření software na zelené louce. Mezi tyto komplikace

patří například nedodržení termínu dodání softwaru, překročení finančního limitu či nedostatečné otestování řešení. Všechny tyto komplikace lze sice předpokládat a řádně se na ně připravit, ale nikdy je nelze zcela eliminovat. Například finální odladění nově vytvořeného programu je téměř vždy provedeno až po nasazení řešení u zákazníka, protože dodavatelská firma není schopna vyvinout takové množství testů, které by plně simulovaly zákazníkovy procesy. Zároveň na dodavatelskou firmu padá zodpovědnost údržby, kdy musí zákazníkovi poskytovat podporu pro svůj produkt ještě po jeho dodání [Szy02].

Mezi další nevýhody řešení na zelené louce patří bezesporu finanční náročnost. Vytváření řešení totiž zabere mnohem více času (a tím pádem i peněz), než kdybychom software získali od jiné firmy. Zároveň následná údržba softwaru je náročná a dodavatele stojí další náklady. Tím se dostáváme k druhému přístupu vytváření software – zakoupení již hotového řešení [Szy02].

Pokud zakoupíme již hotové řešení, získáme oproti vytváření vlastního několik výhod, ale i zde můžeme narazit na komplikace. Mezi základní plusy zakoupení již hotového softwaru patří finanční a časové hledisko. Protože kupujeme řešení, které je již hotové, je doba od podepsání smlouvy do nasazení řešení u zákazníka minimalizována. Zároveň je zakoupeno řešení, které bylo s velkou pravděpodobností již nasazeno u jiných zákazníků a tím pádem je jeho chybovost mnohem nižší. Pokud se i přesto v průběhu používání objeví nějaké potíže, problém je předán dodavateli řešení a my se jím nemusíme zabývat.

Tím, že zakoupíme již hotové řešení, můžeme ušetřit čas a peníze (ušetřené peníze za vývoj, podpora softwaru spadá na dodavatelskou firmu), ale dostáváme se do situace, kdy dodané řešení nepokrývá přesně zákazníkův problém. Toto je největší nevýhoda přístupu vytvoření softwaru zakoupením již hotového řešení. Protože dodané řešení nemusí plně pokrývat zákazníkův problém, musíme zajistit upravení dodaného software. V některých případech není dokonce ani toto upravení možné a je nutné, aby zákazník podle dodaného softwaru měnil své procesy. Zároveň se dostáváme do situace, kdy naše řešení zákazníkova problému je přímo závislé na cizím řešení. Tím máme omezené možnosti, jak reagovat na přicházející



Obrázek 8: Cenová výhodnost a míra flexibility při různé výrobě softwaru [Szy02].

změny či požadavky od zákazníka [Szy02].

K dokreslení celé situace nám může posloužit graf na obr. 8. Na grafu vidíme dvě křivky, kde jedna určuje cenovou výhodnost a druhá míru flexibility a konkurenční výhodu (výhoda získaná dodaným řešením oproti konkurenci). Pokud vytváříme software na zelené louce, je cenová výhoda minimální, ale konkurenční výhoda a míra flexibility (tj. schopnosti reagovat na změny) je vysoká. Čím více procent našeho řešení je outsourcováno, tím je nižší cena vytvářeného řešení. Zároveň s posouváním do pravé části grafu klesá míra flexibility a konkurenční výhody. Můžeme říci, že ideálním místem v grafu je protnutí obou křivek uprostřed. A právě toto protnutí může být chápáno jako bod, který splňuje komponentové programování.

Pokud řešení skládáme z komponent, je cena výsledného produktu přímo úměrná ceně použitých komponent. Trh s komponentami může obsahovat komponenty řešící stejné problémy, ale tyto komponenty mohou mít jinou cenu a vlastnosti (efektivitu řešení, stupeň certifikace atd.) [Szy02]. Příklad můžeme dát opět do souvislosti s automobilovým průmyslem, kdy dvě firmy mohou dodávat dveře do auta, ale každá firma dodá dveře s jinou cenou. Stejná situace může nastat na trhu

s komponentami, kde jedna firma nabízí certifikovanou komponentu za vyšší cenu, zatímco druhá nabízí komponentu za nižší cenu, ale bez certifikace. Záleží jen na zákazníkovi, zda je ochoten si připlatit za certifikaci, či zda si vystačí s komponentou bez certifikace. Tato cenová modularita je jednou z výhod komponentového programování.

Další výhodou komponentového programování je, že odstraňuje bolestivý proces masivní aktualizace softwaru [Szy02]. Díky tomu, že je software rozdělen na komponenty, stačí pouze upravit ty jeho části, jichž se změna týká. Odpadá tak nutnost aktualizace celého systému, pokud potřebujeme opravit pouze malou chybu, či reagovat na požadovanou změnu od zákazníka. Zákazník rovněž může po čase požadovat, aby část systému pracovala rychleji, nebo aby byla zpracovávána certifikovanými komponentami. Díky tomu, že každou komponentu lze jednoduše nahradit za jinou, stačí pouze dokoupit požadovanou komponentu a nahradit s ní komponentu původní. Je jedno, že původní komponenta byla vyrobena jiným výrobcem, či že je novější než původní. Pokud je nová komponenta napsána podle konvencí, nic nebrání jejímu nasazení v systému.

Nahrazením původní komponenty jinou komponentou jsme se dostali až k výhodě znovupoužitelnosti komponent. Přestože komponenta reprezentuje jedno konkrétní řešení, lze jí úspěšně využít v mnoha programech bez potřeby sebemenších úprav. Zbavujeme se tak nutnosti vyvíjet vlastní řešení a zároveň získáváme možnost využít již funkčních a otestovaných komponent. Díky tomu šetříme čas a náklady, které by byly spojené s vývojem a testováním [Szy02].

4.3 Vývoj softwarových komponent

V předchozí kapitole bylo vysvětleno, proč může být výhodnější vytvářet software komponentovým programováním. Jak bylo naznačeno, aby komponenty spolu mohly spolupracovat, a aby mohly být vyvíjeny nezávisle na sobě, je potřeba mít určitou sadu pravidel, které budou jakýmsi standardem při vývoji komponent. Takový standard se nazývá komponentový model.

4.4 Komponentový model

Komponentový model je soubor pravidel a konvencí, které musí vývojář při vytváření komponent dodržet. Tato pravidla zajišťují, že komponenty mohou být snadno vytvářeny a spravovány. Neexistuje přesná specifikace, co by měl každý komponentový model obsahovat, nicméně podle [Bac00] se nabízí tyto typy pravidel:

- Definice typů komponent, které budou podporovány, s popisem jejich vlastností a vzájemného postavení. Zde by mělo být definováno, jak bude každá komponenta tvořena, z jakých částí se bude skládat atd. [Bac00]
- Definice komunikačního schématu. Zde je specifikováno, jakým protokolem komponenty komunikují, jak spolu komunikují, jak je zajištěna bezpečnost a spolehlivost komunikace [Bac00].
- Definice zdrojů a přístupů ke zdrojům. V tomto pravidle je popsáno, které zdroje a kdy jsou komponentám přístupné a jak jsou komponenty se zdroji svázány. Zdroj je v tomto případě komponentový framework, nebo jiná komponenta [Bac00].

Komponentových modelů existuje více, vzájemně se odlišují realizací výše uvedených pravidel.

4.5 Komponentový framework

Komponentový framework je již konkrétní implementací komponentového modelu a slouží jako prostředí pro běh komponent. Na komponentový framework se můžeme dívat jako na malý operační systém, jehož procesy jsou komponenty. Podobně jako v operačním systému, framework pracuje nad komponentami, usměrňuje a řídí jejich životní cyklus. Dále framework umožňuje komponentám přistupovat ke zdrojům a zajišťuje prostředky komunikace mezi komponentami [Bac00].

Tak jako existuje více komponentových modelů, existuje i více komponentových frameworků. Velmi často existuje několik implementací (tj. frameworků) jednoho komponentového modelu, jako například u OSGi.

4.6 OSGi

OSGi je specifikace modulárního systému pro jazyk Java. Vznik OSGi technologie spadá do roku 1999, kdy byla založena nezisková společnost OSGi Alliance. OSGi Alliance definuje OSGi specifikaci, která popisuje chování OSGi frameworku a standardní API. Pokud budeme mluvit o OSGi frameworku, již máme na mysli konkrétní implementaci OSGi specifikace. Původně bylo OSGi akronymem pro *Open Services Gateway Initiative*, v současnosti je již jméno OSGi obchodní známkou OSGi technologie [Hal11].

Stejně jako OSGi specifikace určuje vlastnosti OSGi frameworku, definuje také standardní služby. Mezi standardní služby se řadí systémové služby (logovací služba, konfigurační služba atd.), protokolové služby (Http služba atd.) a další [Hal11].

V dnešní době existuje relativně velké množství OSGi frameworků. Díky vlastnostem OSGi specifikace je možné, aby aplikace napsaná pro jeden OSGi framework pracovala pod druhým OSGi frameworkem [Hal11]. Nyní budou popsány nejpoužívanější OSGi frameworky.

4.6.1 OSGi frameworky

Equinox Zřejmě nejznámějším OSGi frameworkem je Equinox. Současná verze implementuje nejnovější OSGi R4 specifikaci a celý framework je open-source. Equinox dosáhl své popularity především proto, že je standardně součástí vývojového prostředí Eclipse, které je na něm postaveno. Pluginy do Eclipse jsou vytvářeny jako OSGi komponenty [Equ12].

Apache Felix Tento OSGi framework je produktem společnosti Apache, vznikl z projektu Oscar od open-source komunity ObjectWeb. Vyznačuje se svou kompaktností a je vyvíjen jako open-source. Aktuální verze implementuje nejnovější OSGi R4 specifikaci [Fel12].

Knopflerfish Jeden z nejrozšířenějších OSGi frameworků od společnosti MakeWave, který je distribuován pod BSD licencí. Nejnovější verze implementují OSGi R4 specifikaci [Kno12].

Concierge Velmi malý a optimalizovaný framework vyvíjený pro potřeby uplatnění OSGi v mobilních telefonech a embeded zařízeních. Concierge je distribuován pod BSD licencí [Con12].

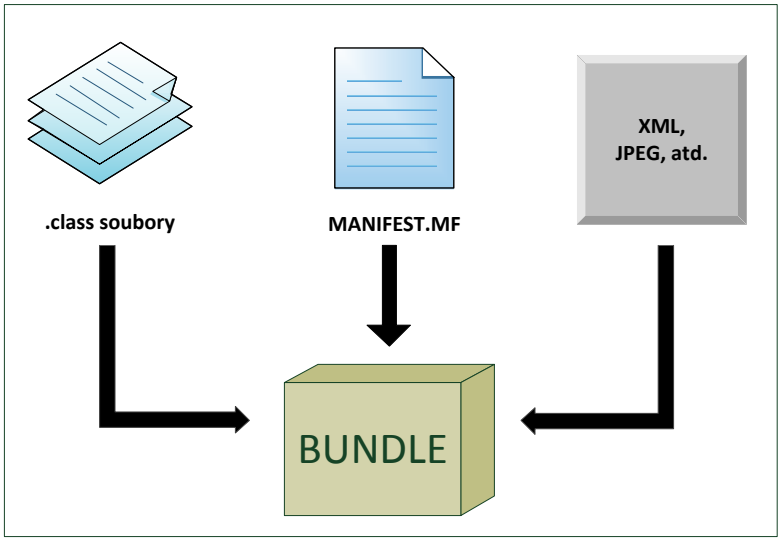
OSGi specifikace definuje tři základní architektonické vrstvy OSGi frameworků. Tyto vrstvy jsou: *vrstva modulů*, *vrstva životního cyklu* a *servisní vrstva*. Každá vrstva definuje konkrétní vlastnosti OSGi frameworku. *Vrstva modulů* přesně určuje pohled na komponenty v OSGi, *vrstva životního cyklu* popisuje životní cyklus komponent a *servisní vrstva* definuje komunikaci mezi komponentami [Hal11]. V následujícím textu budou vrstvy popsány podrobněji.

4.6.2 Vrstva modulů

Ve vrstvě modulů je přesně nadefinováno, jak vypadá jednotka modularizace v OSGi – *bundle*. Bundle je konkrétní realizací abstraktní myšlenky komponenty v OSGi specifikaci. Bundle je tvořen JAR archivem, jehož obvyklým obsahem jsou class soubory v balících (Java packages), potřebné zdroje (obrázky, xml...) a manifest bundlu. Struktura obsahu bundlu je vidět na obr. 9.

Vše, co je obsahem JAR archivu, je bráno jako nedělitelná součást konkrétního bundlu. Na rozdíl od standardního JAR archivu má bundle výhodu v tom, že dokáže specifikovat, jaké balíky jsou z bundlu viditelné do okolního světa, a zároveň umožňuje definovat, jaké balíky bundle potřebuje od bundlů v okolním světě. Tímto je realizováno propojení s ostatními bundly. Provázanosti bundlů se dosahuje importováním či exportováním konkrétních balíků v manifestu. Pokud mluvíme o provázanosti bundlů pomocí importů a exportů balíků, mluvíme o takzvané statické závislosti. V OSGi existuje ještě dynamická závislost, která bude probrána v podkapitole 4.6.4.

Použitím exportů a importů balíků se rozšiřují možnosti Javy s viditelností



Obrázek 9: Struktura bundlu.

kódu [Hal11]. V Javě existují čtyři druhy viditelnosti: *implicitní*, *private*, *protected* a *public*. Pokud chceme vidět kód z jednoho balíku v jiném balíku, musíme nastavit jeho přístupový modifikátor na public. Tím ale dosáhneme toho, že kód bude viditelný nejenom pro požadovaný balík, ale bude viditelný úplně všude [Hal11]. Tím může v budoucnosti vzniknout závislost kódu na veřejném API. V případě OSGi se lze tomuto problému úspěšně vyhnout právě použitím importů a exportů balíků.

Nejdůležitější součástí bundlu je manifest. Manifest je jeden soubor uvnitř bundlu, který definuje závislosti a vlastnosti bundlu. Tyto informace jsou důležité jak pro samotný bundle, tak především pro ostatní bundly ve frameworku. Manifest je soubor se jménem MANIFEST.MF, jeho ukázku lze vidět na obr. 10. Manifest standardně obsahuje tyto informace:

- Bundle-ManifestVersion – Určuje verzi manifestu bundlu.
- Bundle-Name – Jméno bundlu.
- Bundle-SymbolicName – Symbolické označení bundlu sloužící k jednoznačné identifikaci bundlu (spolu s Bundle-Version) ve frameworku.
- Bundle-Version – Definuje konkrétní verzi bundlu. Spolu s *Bundle-SymbolicName*

```
Bundle-ManifestVersion: 2
Bundle-Name: Test
Bundle-SymbolicName: cz.zcu.kiv.testing
Bundle-Version: 1.0.0
Bundle-Description: Testing example
Bundle-Activator: cz.zcu.kiv.testing.Activator
Bundle-Vendor: Ordinary Joe
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
Export-Package: cz.zcu.kiv.testing.helloworld;version="1.0.0"
Import-Package: org.osgi.framework;version="1.3.0"
```

Obrázek 10: Ukázka manifestu OSGi bundlu.

jednoznačně definuje jeden bundle.

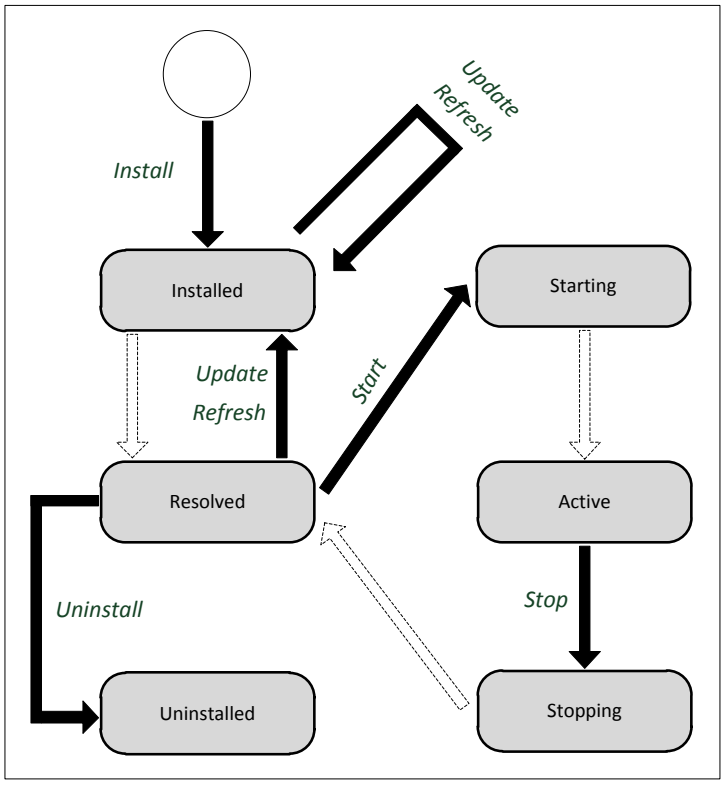
- Bundle-Vendor – Vydavatel bundlu.
- Bundle-Description – Stručný popis funkcionality bundlu.
- Bundle-Activator – Definuje třídu, která bude použitá jako aktivátor bundlu. Aktivátor je nezbytný pouze pokud bundle bude komunikovat s OSGi API nebo pokud v něm chceme provést vlastní inicializaci.
- Bundle-RequiredExecutionEnvironment – Požadovaná verze vývojového prostředí nutná ke správnému provozu bundlu.
- Export-Package – Seznam balíků, které jsou bundlem exportovány do okolního světa. Balíky uvedené v tomto seznamu mohou být viděny ostatními bundly.
- Import-Package – Seznam balíků, které bundle potřebuje ke svému provozu od ostatních bundlů ve frameworku.

4.6.3 Vrstva životního cyklu

Druhá z vrstev OSGi specifikace definuje životní cyklus bundlu v OSGi frameworku. Bundle má podobné chování jako program, který nám může běžet na našem operačním systému – můžeme ho nainstalovat, odinstalovat či aktualizovat.

Stejně jako program má svůj stav v operačním systému, má bundle svůj stav ve frameworku. Každý bundle může procházet během svého životního cyklu těmito stavy:

- *installed* – Bundle je nainstalován ve frameworku.
- *resolved* – Závislosti bundlu byly vyřešeny a bundle je připraven k použití.
- *starting* – Bundle je právě startován.
- *active* – Bundle běží ve frameworku.
- *stopped* – Bundle byl zastaven.
- *uninstalled* – Bundle byl odinstalován a již není k dispozici.



Obrázek 11: Životní cyklus bundlu. Čárkované čáry značí přechody, které provádí pouze framework.

Každý bundle může těmito stavy procházet buď externím přístupem, nebo pomocí vlastního kódu. Externím přístupem se rozumí použití příkazů `install`, `update`, `unistall`, `start` a `stop` ve frameworku. Bundle dokáže změnit životní cyklus i programově ve vlastním kódu, tím může sám ovlivňovat svůj chod. Díky praktické OSGi specifikaci není při žádné změně v životním cyklu bundle třeba restartovat framework. Díky tomu může bundle za svojí existenci ve frameworku libovolněkrát změnit svůj stav [Hal11]. Životní cyklus bundlu je zobrazen na obr. 11.

4.6.4 Služební vrstva

Třetí vrstvou OSGi architektury je vrstva definující komunikaci mezi bundly. Tato vrstva je jedinou možností, jak spolu mohou bundly komunikovat navzájem. Vrstva se nazývá služební, protože komunikace je řešena pomocí služeb. Služby fungují na principu poskytovatel – příjemce, kdy poskytovatel zaregistruje službu do registru služeb, příjemce službu najde a přihlásí se k jejímu odběru. V tomto případě mluvíme o takzvané dynamické závislosti, protože každý bundle může svoje služby libovolně registrovat a příjemce se od nich může zase libovolně odregistrovat [Hal11].

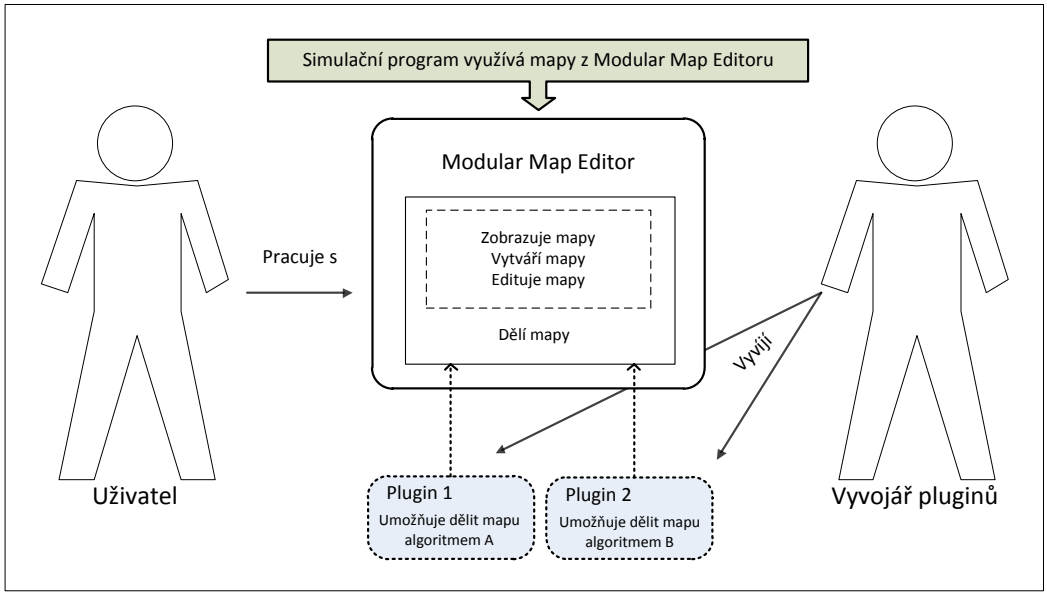
Služby jsou definované jako Java objekty, jejich registrací dává poskytovatel najevo, že přes dané služby je s ním možné komunikovat. Jako body komunikace slouží pak veřejné metody zaregistrovaného Java objektu. Protože se služby mohou ve frameworku měnit za běhu, je nutné dostupnost služeb kontrolovat. Pokud chce příjemce využít určitý druh služby, nejprve zjistí podle registru služeb, jestli je služba ve frameworku dostupná. Pokud služba je dostupná, může se na ní registrovat, v opačném případě je nutné čekat, dokud služba nebude dostupná. Pokud existuje ve frameworku více služeb zaregistrovaných pod stejným jménem, je použit speciální filtr k rozlišení požadované služby [Hal11].

Programové využívání služeb vypadá tak, že při registraci služby programátor získá referenci na framework přes tzv. *bundle kontext*. Bundle kontext reprezentuje styčný bod mezi bundlem a OSGi frameworkem. V dalším kroku se přes

bundle kontext zaregistruje požadovaná služba pomocí volání registrační metody. Příjemce obvykle zachytí službu tak, že má na své referenci bundle kontextu nastaveného posluchače, který odposlouchává změny služeb ve frameworku. Jakmile je hledaná služba zaregistrována, posluchač upozorní svůj bundle o změně.

5 Analýza řešení

Program MME má sloužit jako grafický nástroj pro zpracování map silničních sítí pro simulační systém DUTS. Protože mapy systému DUTS jsou reprezentovány XML soubory, musí MME umět pracovat s XML dokumenty. Mapové soubory XML mají pevně danou strukturu a program MME musí umět s touto strukturu pracovat (při čtení a editaci), a zároveň musí umět mapy s požadovanou strukturou vytvořit. Program MME musí dále umět dělit mapu silničních sítí a umožnit přidání dalších algoritmů dělení. Kontext systému je zobrazen na obr. 12.



Obrázek 12: Kontext systému.

5.1 Programátorské prostředky

Program MME bude naprogramován v jazyce Java ve verzi Java SE 7u3. Jazyk Java byl vybrán pro své objektové založení a s ohledem na zkušenosti s vývojem softwaru tímto jazykem. Grafické prostředí programu bude vytvořeno pomocí knihovny *Swing*.

Protože program MME má umožnit přidání dalších algoritmů dělení, celá architektura programu musí být navržena s ohledem na tento požadavek. Nej-

vhodnějším prostředkem pro zpracování tohoto požadavku se jeví využití principu komponentového programování. Protože existuje množství OSGi frameworků, kde většina z nich dokáže nabídnout dostačující prostředí pro realizaci našeho řešení, bylo OSGi zvoleno jako vhodná platforma. Mezi OSGi frameworky byl následně vybrán framework Equinox k implementaci programu. Framework Equinox byl vybrán hlavně z toho důvodu, že je přímou součástí vývojového prostředí Eclipse, pod kterým je program vytvářen.

5.2 Architektura řešení

Program MME se bude skládat z jednoho hlavního bundlu, který bude realizací samotného editoru map silničních sítí. Implementace algoritmů dělení budou vytvořeny jako další bundly, které budou využívat balíky z hlavního bundlu. Hlavní bundle bude exportovat šest balíků. Každý dělící algoritmus, který bude implementován jako bundle, musí těchto šest balíků importovat. Hlavní bundle bude exportovat tyto balíky:

- Balíček definující prostředek komunikace mezi hlavním bundlem a bundlem s implementací dělícího algoritmu. Prostředkem komunikace se myslí konkrétní datová struktura, která bude sloužit jako šablona pro každou implementaci dělícího algoritmu.
- Balíček obsahující definice objektů, které se mohou vyskytnout na mapě.
- Balíček obsahující implementaci objektů, které se mohou vyskytnout na mapě.
- Balíček umožňující pracovat s ID objektů na mapě.
- Balíček obsahující definici a implementaci mapy a umožňující práci s mapou.
- Balíček s implementací struktur, které vznikají při dělení mapy na více částí.

Hlavní bundle bude vytvořen v duchu třívrstvé architektury, kde každá vrstva má své speciální určení.

5.2.1 Grafická vrstva

Zprostředkovává komunikaci mezi uživatelem a programem a zobrazuje mapy silničních sítí. Dále poskytuje vizuální informace o mapách a dělících algoritmech, které jsou k dispozici.

5.2.2 Aplikační vrstva

Obsahuje definice a implementace datových struktur a poskytuje prostředí pro práci s těmito strukturami. Zároveň je přímým prostředníkem mezi grafickou vrstvou a datovou vrstvou.

5.2.3 Datová vrstva

Obsahuje implementaci postupů pro práci s daty. Umožňuje zpracovávat XML soubory (načítání, vytváření).

5.3 Požadavky na systém

Program MME musí splňovat tyto funkční požadavky:

- Vytvářet mapy silničních sítí v podobě XML souborů s danou strukturou.
- Zobrazovat mapy silničních sítí uložených jako XML soubory s danou strukturou.
- Umožnit přiblížení mapy při jejím zobrazení (zoom).
- Editovat načtené mapy silničních sítí.
- Pracovat s více mapami silničních sítí v jeden čas.
- Umožnit vkládat, odebírat a nastavovat objekty na mapě.
- Umožnit dělení mapy silničních sítí pomocí pluginů.
- Umožnit kdykoliv přidat nový algoritmus dělení mapy do programu jako plugin.

- Umožnit vytvoření XML souborů popisujících rozdělení mapy do částí při dělení mapy.
- Být přehledný a jednoduše ovladatelný.

5.4 Analýza a návrh grafického editoru

Protože hlavním účelem programu je zjednodušit práci s XML mapami silničních sítí, byl velký důraz kladen na návrh samotného grafického zobrazení aplikace. Aby byl grafický editor co nejpřehlednější, bylo zvoleno řešení s jedním velkým oknem (viz obr. 13).

Obrázek 13: Hlavní okno programu.

Hlavní okno se skládá z několika menších částí. Největší prostor zabírá plocha pro zobrazení mapy a jejích objektů. Pod plochou pro mapu se nachází místo pro konzoli, která slouží k informování uživatele o běhu programu. Nad plochou pro mapu najdeme nástrojový panel, který slouží pro práci s mapou a objekty na mapě. V levé části hlavního okna se nachází prostor vymezený pro zobrazení map, které jsou právě otevřené v programu. Celé hlavní okno lemuje v horní části

lišta s menu.

Program MME má pracovat se systémem DUTS, který využívá pro simulaci dopravy celulární automaty. Protože komunikace jsou v DUTS reprezentovány pomocí buněk, i program MME musí umožnit zobrazení mapy pomocí buněčných sítí. Hlavní okno s mapou bude vždy obsahovat síť čtvercových buněk, kde každá buňka odpovídá jedné buňce v systému DUTS.

5.4.1 Zobrazení otevřených map v projektu

Uživatel musí mít přehled o všech mapách, které si v programu otevřel, a musí mu být umožněno mapy editovat, přepínat mezi nimi a případně je odebírat z přehledu. Aby bylo dosaženo dostatečně intuitivního zobrazení map v projektu, budou mapy vykresleny ve stromové struktuře. Každá nově přidaná mapa se objeví jako potomek kořenového listu stromové struktury. Mapy budou reprezentovány svými názvy a budou seřazeny podle času přidání do přehledu.

Knihovna Swing obsahuje dostatečnou podporu pro zobrazení hierarchických struktur. Pro zobrazení map v přehledu bude použita Swing komponenta `JTree`.

5.4.2 Objekty na mapě silniční sítě

Program MME bude umět pracovat se všemi objekty, které se mohou vyskytnout na mapě v systému DUTS. Každý objekt bude mít specifický tvar a barvu. Nyní budou popsány všechny objekty a jejich zobrazení.

- *Silniční pruh.* Bude reprezentován jako řada buněk s pevně danou šířkou jedné buňky a s proměnnou délkou. Minimální délka silničního pruhu jsou v MME dvě buňky. Pruh může být položen jakýmkoliv směrem na mapě. Pruh bude vykreslován šedou barvou (odpovídající silničnímu pruhu v reálném světě). Aby bylo možné rozlišit začátek a konec pruhu, bude první buňka pruhu označena černou barvou.

- *Křižovatka.* Vyplňuje svým obsahem 12 buněk a má podobu kříže. Velikost křižovatky je neměnná. Každé rameno kříže odpovídá jednomu rameni

křižovatky a obsahuje právě jeden výjezd a právě jeden vjezd. Pro lepší rozlišení výjezdů a vjezdů bude mít vjezd červenou barvu a výjezd oranžovou barvu.

- *Terminátor*. Má délku 6 buněk a jeho velikost je neměnná. Může být umístěn ve 4 základních polohách – na sever, na jih, na západ a na východ. Terminátor bude mít žlutou barvu. K odlišení začátku a konce bude mít jeho první buňka černou barvu.

- *Generátor*. Má stejné vlastnosti jako terminátor, pouze bude mít zelenou barvu.

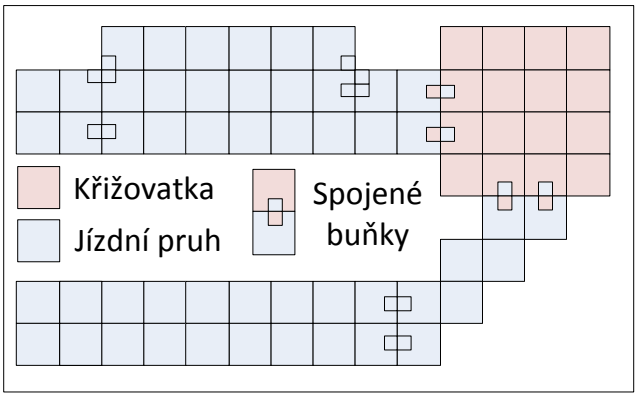
Všechny objekty ležící na mapě lze editovat. Editace bude probíhat označením objektu (kliknutím na objekt) a následným vyplněním hodnot v nastavovacím formuláři. Při vložení nového objektu na mapu se vytvoří objekt s implicitními hodnotami. Aby uživatel nemusel při vložení dalšího objektu klikat více než je nutné, nebude formulář pro nastavení vlastností při vytváření zobrazen.

5.4.3 Spojování objektů mapy

Všechny objekty na mapě utvářejí obsah mapy, ale až propojení objektů dělá z obsahu mapy celistvou silniční síť. Je proto důležité mít vhodně zajištěné propojení objektů na mapě.

Jako jedna z možností řešení bylo uvažováno vytvoření speciálního objektu, který by sloužil k označení dvou propojených objektů. V praxi by k standardním objektům mapy (silniční pruh, křižovatka ...) přibyl ještě další objekt, který by se dal umístit pouze přes dva jiné sousedící objekty. Tento přístup byl po úvaze zamítnut proto, že program MME má pracovat se standardními objekty systému DUTS a přidání dalšího objektu by mohlo být pro uživatele matoucí.

Propojení bude řešeno pomocí „prolnutí buněk“. Po požadavku na propojení dvou objektů budou jejich sousední buňky částečně „prolnuty“ dohromady. Tímto „prolnutím“ se myslí prostoupení prostorově odpovídajících částí sousedních buněk do sebe. K propojení objektů tímto způsobem nebude potřeba žádný další objekt.



Obrázek 14: Princip zapouštěcího mechanismu pro spojování buněk.

Propojení se provede zvolením volby propojení v panelu nástrojů a označením nejdříve prvního a následně druhého objektu. Ukázka způsobu propojení je vidět na obr. 14.

5.5 Zpracování XML souborů s mapami silničních sítí

Pro práci s XML soubory map musí být vybrán vhodný XML parser. Průzkumem dostupných možností vykrystalizovaly tyto parsery: SAX, STaX a DOM.

SAX parser umožňuje rychlé čtení XML dokumentu na bázi vyvolávání událostí. SAX čte celý dokument po částech sekvenčně a umožňuje validaci dokumentů [Cai09]. Bohužel tento parser neumožňuje zápis do XML dokumentu.

DOM pracuje s XML dokumentem pomocí stromové reprezentace a umožňuje jeho čtení, zápis i validaci. Při čtení XML je celý dokument uložen do paměti jako stromová struktura. DOM je oproti SAXu pomalejší a nehodí se ke zpracování velkých XML souborů [Her07]. Výhodou proti SAXu je možnost vytvoření XML dokumentu.

STaX umožňuje čtení a zápis XML dokumentů po částech. Oproti DOMu má nižší nároky na paměť a dokáže velké dokumenty zpracovat rychleji. Na rozdíl od předchozích dvou parserů neumí validovat XML [Her07].

Pro program MME je důležitá rychlost práce s XML a také oboustranná funkcionalita (zápis i čtení). Validace dokumentu není v rámci zadání diplomové

práce uvažována. Vzhledem k uvedenému porovnání byl jako vhodný XML parser vybrán STaX.

5.6 Pluginy s dělícími algoritmy

Dělení mapy bude prováděno pomocí pluginů, které se budou zobrazovat v záložce v hlavním menu aplikace. Výběr dělícího algoritmu bude provedeno vybráním z nabídky dělících algoritmů v této záložce.

Pro každý algoritmus dělení půjde nastavit volitelný počet parametrů. Počet parametrů bude záviset na konkrétním typu algoritmu. Nastavení parametrů dělení bude provedeno pomocí formuláře, který se zobrazí po vybrání algoritmu v záložce menu. Ve formuláři budou zobrazeny jak nutné parametry k nastavení, tak i stručný popis algoritmu. U každého parametru bude moci být provedena typová kontrola.

5.6.1 Přidání a odebrání dělících algoritmů v MME

Všechny pluginy s dělícími algoritmy půjdou do MME přidat v libovolný okamžik. Po nainstalování pluginu do frameworku OSGi zachytí MME přidání pluginu a postará se, aby byl plugin s algoritmem viditelný v menu MME.

Stejným způsobem bude probíhat i odebrání pluginu. Kdykoliv bude plugin z frameworku odebrán (odinstalován), bude aplikace informována o odebrání pluginu. Ten pak zmizí z nabídky dělících algoritmů v menu programu.

K realizaci přidání a odebrání algoritmů bude použito modelového přístupu. Všechny algoritmy načtené v MME budou uloženy do modelu. Tento model se bude měnit v závislosti na instalaci či odinstalaci pluginů s dělícími algoritmy v OSGi frameworku.

5.6.2 Předávaná struktura mapy silničních sítí

Všem algoritmům dělení budou jako mapa silničních sítí předány objekty, které reprezentují silniční síť v MME. Jako varianta tohoto přístupu se nabízela možnost předávání objektů pomocí externích nástrojů – například využitím knihovny Jung

[Jun12]. Knihovna Jung umožňuje pracovat s grafy či sítěmi, pro které obsahuje množství předpřipravených datových struktur a algoritmů z oblasti teorie grafů. Knihovna Jung nebude nakonec použita, protože samotná reprezentace silniční sítě je v MME řešena dostatečně přehledně.

5.6.3 Výběr algoritmu dělení

K implementaci byl vybrán algoritmus dělení průchodem grafu silniční sítě do šířky. Tento algoritmus byl zvolen, protože poskytuje dostačující výsledky dělení map, a protože mapu silničních sítí je možné díky svým vlastnostem snadno považovat za graf.

Pro lepší otestování pluginové funkčnosti programu MME bude navíc mimo rozsah diplomové práce vytvořen ještě jeden plugin realizující dělení dopravní sítě. Tento plugin bude používat algoritmus poměrového dělení sítě průchodem grafu do šířky. Více o konkrétním řešení se lze dočíst v kapitole 7.3.

6 Implementace Modular Map Editoru

V následujícím textu bude popsána implementace MME se zaměřením na implementaci grafické a aplikační vrstvy programu. Popis implementace modulů dělení je uveden v další kapitole.

6.1 Implementace grafické vrstvy aplikace

V implementaci grafické části budou popsány detaily implementace zobrazovacích metod, dále princip fungování vykreslení jednotlivých objektů mapy a jako poslední práce s modely.

Protože je celé grafické prostředí programu MME vytvářeno v Java knihovně Swing, jsou použity postupy a komponenty, které tato knihovna obsahuje.

Zobrazení hlavního okna je vytvořeno pomocí jednoho panelu, který má v sobě vložené další komponenty. Hlavní panel neobsahuje žádnou funkčnost, slouží pouze jako plocha pro umístění ostatních komponent. Do hlavního panelu jsou vloženy tyto komponenty:

- *Panel s mapou.* Obsahuje implementaci zobrazení mapy. Stará se o vykreslování mapy a objektů s ní spojených.
- *Komponenta s konzolí.* Umožňuje informovat uživatele o jeho provedených akcích a o činnostech programu.
- *Panel se stromovou komponentou.* Zobrazuje otevřené mapy v programu a umožňuje přepínaní mezi nimi.
- *Panel s nástroji.* Zobrazuje nástroje pro práci s programem. Umožňuje uživateli pracovat s mapou, vkládat na mapu objekty a provádět operace s mapou.

Protože umístění panelů a komponent pomocí layout managerů knihovny Swing bývá komplikované, je k rozmístění využit layout manager třetí strany – `MigLayout`. Díky této knihovně je umístění komponent mnohem snadnější a pohodlnější.

`MigLayout` umožňuje rozmístit komponenty přímo bez nutnosti prokládání panely navíc (`BorderLayout`), či nastavovat mnoho parametrů (`GridBagLayout`). Více se lze o tomto layout manageru dočíst v [Mig12].

6.1.1 Implementace vykreslování mapy

Mapa je v základu `JPanel` s přidanými vlastnostmi, umístěný do hlavního panelu. Kvůli množství specifických vlastností, které tato komponenta má, byl zvolen postup postupného oddědění od obecnějšího předka. Celá komponenta se tak skládá ze čtyř tříd, které budou popsány v následujících odstavcích. Princip implementace je možné vidět na UML diagramu v příloze B.

MapPanelBase Abstraktní třída a zároveň předek všech tříd mapové komponenty, oddděná od `JPanel`. Obsahuje reference na objekty, které mohou být použity napříč všemi třídami mapové komponenty. Dále obsahuje implementaci obecných metod, které se používají při práci s mapou. Jako příklad těchto metod můžeme uvést `transformMouseCoor(int x, int Y)`, která převádí souřadnice kurzoru na mapové souřadnice.

MapPanelFormManager Abstraktní třída dědící od `MapPanelBase`. Stará se o správné zobrazení formulářů, které se používají k nastavení vlastností objektů na mapě.

MapPanelActions Abstraktní třída dědící od `MapPanelFormManager`. Obsahuje implementaci reakcí na události vyvolaných uživatelem při práci s mapou.

MapPanel Třída dědící od `MapPanelActions`. Stará se o správné vykreslení mapy a objektů na mapě. Mezi metody, které obsahuje, patří například metoda pro vykreslování křižovatky. Metoda se jmenuje `paintCrossroad(Crossroad c, Graphics2D g)` a vykreslí na mapě právě jednu křižovatku.

Mapa se vykresluje v překryté metodě `paintComponent()`. Vždy při jejím vykreslení se bere ohled na aktuální zvětšení zobrazené plochy zoomem. Při zo-

omování je zvětšena konstanta udávající velikost jedné buňky v pixelech a celá mapa je znovu vykreslena. Vykreslování mapy s objekty pak funguje ve čtyřech fázích.

V *první fázi* se prochází seznamy objektů, které obsahuje model mapy, a pokud není seznam prázdný, každý objekt se vykreslí. K vykreslení objektu je vždy použita metoda `paintNAZEVOBJEKTU()`, kde NAZEVOBJEKTU se liší v závislosti na typu vykreslovaného objektu. Protože každý objekt je reprezentován množinou buněk, je k vykreslení každé buňky objektu volána metoda `paintCell(int x, int y, Graphics2D g, Color c)`, kde parametry jsou souřadnice buňky, odkaz na grafickou primitivu a barva vykreslované buňky.

V *druhé fázi* je vykreslován objekt, se kterým právě uživatel pracuje. Pokud například uživatel chce umístit na mapu generátor a posunuje s ním po mapě, je vykreslování generátoru prováděno právě v této fázi. K vykreslení se používá stejného postupu jako při vykreslování již umístěných objektů.

Ve *třetí fázi* jsou vykreslována spojení mezi objekty. Jsou znovu procházeny seznamy objektů mapy, a pokud se narazí na objekt, který je spojen s jiným objektem, je vykresleno spojení.

Ve *čtvrté fázi* se vykreslují linky tvořící vodící síť mapy.

6.1.2 Implementace konzole

Konzole slouží k informování uživatele o jeho práci s programem. Jedná se zejména o informování o otevření mapy, odebrání mapy, umístění či odebrání objektů na mapu. Komponenta s konzolí je umístěna ve spodní části hlavního okna programu.

Samotná konzole je tvořena Swing komponentou `JList`. Tato komponenta dokáže zobrazovat potřebné informace a zároveň obsahuje připravený jednoduchý způsob, jak informace zvýraznit (barva textu, obrázek u textu atd.). Komponentě `JList` se nejprve musí nastavit, jaký typ zpráv jí bude zasílán. U MME je vytvořena nová třída definující zprávu – `ConsoleMessage`. Instance třídy `ConsoleMessage` reprezentuje právě jednu zprávu, která může být odeslána do konzole. Každá zpráva má tyto atributy:

- *Text zprávy.* Definuje, jaký text bude mít zpráva v konzoli.
- *Barvu.* Definuje barvu, jakou bude zpráva zobrazena v konzoli.
- *Obrázek.* Definuje obrázek, který bude zobrazen spolu se zprávou v levém rohu konzole.
- *Pravdivostní hodnotu* udávající, jestli se má zpráva vložit na nový řádek, nebo jestli má být vložena na stejný řádek za předchozí zprávu.

Další důležitou součástí komponenty `JList` je jeho model. V MME je model konzole implementován třídou `ConsoleModel`. Model slouží jako komunikační prostředek s konzolí. Kdykoliv chce jakákoliv komponenta v MME komunikovat s konzolí, odesílá zprávu přes model. Model přímo definuje styl komunikace, například pomocí metody `addElement(ConsoleMessage m)`, která odešle zprávu předanou parametrem do konzole.

Poslední nezbytnou součástí konzole je třída `ConsoleCellRenderer`, která určuje, jak se bude každá jednotlivá zpráva zobrazovat. V praxi její funkčnost vypadá tak, že při každém poslání zprávy je zpráva předána třídě `ConsoleCellRenderer`. Tato třída se postará o správné zobrazení zprávy podle atributů objektu zprávy (text, barva, atd.).

Aby se neztrácely starší zprávy, je celá konzole „obalená“ komponentou `AutoScrollPane`, která se stará o zobrazení posuvníků v postraní liště konzole.

6.2 Implementace aplikační vrstvy programu

V následujícím textu bude popsáno, jakým způsobem je provedena implementace aplikační vrstvy programu. Konkrétně bude popsáno, jak jsou vytvořeny jednotlivé objekty mapy, jakým způsobem funguje generování ID objektů a jak pracuje vnitřní logika programu při přidávání objektů do mapy.

6.2.1 Implementace objektů mapy

Každá mapa silničních sítí může obsahovat různé množství objektů simulace DUTS. Jak již bylo uvedeno, mezi tyto objekty patří: silniční pruhy, generátory,

terminátory a křižovatky. Všechny objekty mapy mají podobné množiny vlastností, z toho důvodu bylo určeno několik rozhraní, které tyto vlastnosti popisují.

Mezi rozhraní definující vlastnosti objektů patří tyto:

- `Connectable`. Rozhraní sloužící pro spojování objektů. Každý spojitelný objekt by měl obsahovat dva seznamy – seznam objektů, do kterých je připojen a seznam objektů, které jsou připojeny k němu. Každý objekt, který může být spojen s jiným objektem, implementuje toto rozhraní.
- `Description`. Umožňuje získat a nastavit popis objektu. Všechny objekty, které mohou obsahovat vlastní popis (například důvod umístění objektu na mapě), implementují toto rozhraní.
- `Placeable`. Dovoluje objektu nastavit pozici na mapě. Objekty implementující toto rozhraní mohou být vloženy do mapy na konkrétní pozici.
- `Positionable`. Rozhraní slouží k práci s umístěním objektu na mapě. Definuje konkrétní typ umístění jednoho objektu – objekt může být umístěný pod jednou souřadnicí (křižovatka), nebo může být definován dvěma souřadnicemi (silniční pruh). Toto rozhraní je použito jako rozšíření rozhraní `Placeable`.
- `Sizeable`. Umožňuje zjistit buňky objektu na mapě, na kterých se objekt nachází. Toto rozhraní je použito, protože objekt je většinou rozložený na více buňkách.

Mimo tato rozhraní musí být ještě každý objekt zděděn od abstraktního předka, který umožňuje jednoznačnou identifikaci objektu na mapě. Tento abstraktní předek se jmenuje `Identifiable` a obsahuje informaci sloužící k jednoznačné identifikaci objektu.

Protože většina objektů se liší pouze v drobnostech, je mnoho z nich zděděno od jednoho společného předka, který v sobě udržuje všechny základní vlastnosti včetně identifikace. Každému objektu, který vznikne děděním od společného předka, jsou pak v jeho konkrétní implementaci přidány vlastnosti, kterými se liší od ostatních objektů.

Tímto společným předkem je třída `RoadStructure`. Třída implementuje všechna uvedená rozhraní a dědí od abstraktní třídy `Identifiable`. Třída překrývá všechny metody z uvedených rozhraní mimo metody `calculateCoors()` a `calculatePositionType()`, které jsou úzce spojené s typem objektu a musí být překryty až v potomkovi třídy. Pro vytvoření nového objektu stačí nový objekt oddědit od `RoadStructure` a překrýt tyto dvě metody. Potomky třídy `RoadStructure` jsou třídy: `Road`, `Generator`, `Terminator` a `CrossroadArmWay`. Princip implementace je možné vidět na UML diagramu v příloze B.

Jediným objektem, který lze umístit na mapě, a není oděděn od `RoadStructure`, je třída `Crossroad`. Křižovatka není oděděna od `RoadStructure`, protože nemůže být spojená s ostatními objekty na mapě. Tato vlastnost křižovatky je dána tím, že s ostatními objekty mapy se spojují až konkrétní výjezdy a vjezdy do křižovatky, ale nikoliv křižovatka samotná. Křižovatka je objekt mapy, který vznikl speciální konstrukcí. Každá křižovatka je tvořena několika rameny, které jsou reprezentovány třídou `CrossroadArm`. Každé rameno obsahuje seznam vjezdů a výjezdů, které z / do ramene vedou. Vjezdy i výjezdy jsou definovány třídou `CrossroadArmWay`. Při práci s objektem křižovatky na mapě je vždy pracováno s objektem křižovatky jako celkem. Pokud však dojde na spojování objektů, pracuje se pouze s vjezdy a výjezdy.

6.2.2 Generování ID objektů

Každý objekt, který lze umístit na mapě, musí obsahovat ID, podle něhož je ho možné jednoznačně identifikovat. ID je reprezentováno třídou `IDUniversal`, která obsahuje tyto atributy:

- *Typ objektu.* Určuje typ objektu, pro který je ID vytvořeno (silniční pruh, generátor, atd.).
- *Název mapy.* Určuje mapu, ve které je objekt umístěn.
- *Identifikační číslo objektu.* Jednoznačné identifikační číslo konkrétního typu objektu v konkrétní mapě. Atribut *typ objektu* i *název mapy* jsou obvykle

zastoupeny mezi objekty jedné mapy vícekrát, ale *identifikační číslo* jednoznačně definuje jeden konkrétní objekt jedné konkrétní mapy.

Objekty jsou tedy jednoznačně identifikovatelné pomocí třech atributů třídy `IDUniversal`. Existují i objekty, pro které by byla tato identifikace nedostatečná – konkrétně se jedná o objekt vjezdu a výjezdu do křižovatky a rameno křižovatky. Rameno křižovatky musí obsahovat ještě informaci o křižovatce, ve kterém je umístěno, a vjezd s výjezdem potřebují znát, v jakém rameni leží. Z tohoto důvodu jsou ID těchto objektů tvořena rozšířením třídy `IDUniversal`. Pro rameno křižovatky se jedná o třídu `IDArm` (zděděnou od třídy `IDUniversal`). Pro vjezd s výjezdem se jedná o třídu `IDWay` (zděděnou od třídy `IDArm`).

Generování ID objektů se provádí ve třídě `GeneratorID`, která nabízí veřejné statické metody pro vytvoření ID podle zadaných parametrů nebo pro získání ID z textového řetězce (využité při čtení map z XML).

6.2.3 Práce s mapou

Mapa udržuje informace o svých objektech a umožňuje práci s nimi. Mapa je reprezentována třídou `MapStructures`, která je posledním potomkem posloupnosti tříd, které obsahují implementaci práce s mapou. Každá třída v této posloupnosti implementuje jednu konkrétní funkčnost mapy. Tento postup, kdy je postupným děděním třída doplňována o další vlastnosti, byl zvolen z důvodu lepší přehlednosti zdrojového kódu. Nyní budou tyto třídy popsány.

MapStructuresLists Abstraktní předek všech tříd z posloupnosti tříd definujících mapu. Udržuje seznamy objektů mapy, do kterých jsou objekty vkládány při svém vytvoření, či ze kterých se odebírají při odstranění z mapy. Dále udržuje čítače objektů, které se využívají pro generování ID objektů. Při generování ID nelze přímo využít velikosti seznamů objektů, protože v průběhu práce s mapou mohou být objekty z mapy libovolně mazány a přidávány. Pokud bychom se spoléhali na velikost seznamu, mohli bychom získat pro více objektů stejné ID.

MapStructuresConnect Je abstraktní třída dědící od `MapStructuresLists`, která obsahuje implementaci metod pro spojování objektů. Mezi metody třídy patří například `connect(RoadStructure from, RoadStructure to)`, která spojí dva objekty na mapě.

MapStructuresAdd Abstraktní třída dědící od `MapStructuresConnect` a obsahující implementaci metod zodpovědných za přidávání objektů do mapy. Příkladem může být metoda `addRoad(Road road)`, která přidá do mapy jeden silniční pruh.

MapStructuresDelete Je opakem `MapStructuresAdd`. Taktéž abstraktní třída, ale obsahuje implementaci metod zodpovědných za odebírání objektů z mapy. Dědí od `MapStructuresAdd`.

MapStructures Třída dědí od `MapStructuresDelete`, a tím získává vlastnosti implementované ve všech předcích. Instance této třídy jsou reprezentací jednotlivých map.

7 Implementace algoritmů dělení silniční sítě

V následujícím textu budou popsány principy implementace algoritmů dělení silniční sítě. Celá kapitola je členěna do třech částí. V první části bude popsána struktura, kterou musí každý dělící algoritmus dodržet, pokud chce provádět dělení silniční sítě. Ve druhé části bude následovat popis prvního algoritmu dělení – dělení pomocí BFS algoritmu. Ve třetí a poslední části pak bude popsána implementace dělení pomocí BFS proporcionálního algoritmu.

7.1 Implementace struktury pro algoritmy dělení silniční sítě

Jak bylo uvedeno v kapitole analýzy programu, každý algoritmus dělení musí importovat celkem šest balíků z hlavního bundlu programu. V této kapitole budou popsány dva balíky, které obsahují implementaci struktury dělících algoritmů – tedy ty balíky, jež popisují způsob, jakým má být dělící algoritmus naprogramován. Zbylé čtyři balíky obsahují implementaci objektů mapy (silniční pruh, křižovatka atd.), implementaci mapy a implementaci práce s mapou.

Prvním balíkem, který bude popsán, je balík `cz.sc.mme.bl.dividing`. Tento balík obsahuje implementaci kostry algoritmu dělení. Kostra je reprezentována třídou `AbstractDivisionAlgorithm`. Tato třída je abstraktní reprezentací dělícího algoritmu. Každý dělící algoritmus musí dědit od této třídy. V `AbstractDivisionAlgorithm` jsou definovány členské proměnné: název algoritmu, popis algoritmu a seznam parametrů algoritmu. Navíc je zde abstraktní metoda `divideGraph(MapStructures m, List<Object> p)`, která provádí dělení mapy, a která musí být v konkrétní implementaci překrytá.

Název určuje jméno algoritmu dělení a popis slouží k uchování podrobnějšího popisu algoritmu. Parametry jsou reprezentovány seznamem objektů třídy `Parameter`. Tato třída reprezentuje právě jeden parametr. Obsahuje název parametru a typ parametru. Typ parametru je definován jako `Object` – díky tomu může být parametr libovolný neprimitivní datový typ. Tímto přístupem je možné v programu provádět typovou kontrolu hodnot parametrů algoritmu zadaných uživatelem.

Metoda `divideGraph(MapStructures m, List<Object> p)` vrací jako výsledek

dělení objekt typu `GraphDivision`. Třída `GraphDivision` obsahuje seznam map, které vznikly dělením (seznam objektů `MapStructures`), a objekt `DivisionDescriptor`.

Třída `DivisionDescripton` slouží jako popisná třída propojení nově vzniklých map. Uchovává v sobě seznam topologií sítě, tedy přesný popis propojení mezi mapami vzniklými dělením původní mapy. Dále obsahuje proměnnou typu `Object`, kterou lze využít jako případný přídavný parametr, který lze předat programu MME. V aktuální verzi je tento objekt používán jako textová zpráva, která se doručí MME při úspěšně provedeném dělení.

Druhým popisovaným balíkem je `cz.sc.mme.bl.topology`, který obsahuje objekty používané pro popis propojení map vzniklých dělením. Ke každé mapě, která vznikne dělením, je vygenerován objekt topologie a tento objekt je následně převeden do XML podoby. Objekt topologie se skládá ze seznamu sousedů mapy – instancí třídy `Neighbour`. Každý soused obsahuje seznam odchozích a příchozích párů (instance třídy `Couples`). Každý příchozí nebo odchozí pár obsahuje jeden terminátor a jeden generátor. Typ páru – tedy zda se jedná o pár příchozí nebo odchozí – se určí ze směru dopravy mezi mapami. Ukázku XML souboru lze najít v příloze C.

7.2 Implementace algoritmu dělení silniční sítě metodou BFS

V této kapitole bude popsán algoritmus dělení mapy silniční sítě metodou BFS – postupným prohledáváním grafu do šířky 3.1.3. Protože balíky z hlavního programu nutné k implementaci byly již popsány, text bude zaměřen především na konkrétní implementaci algoritmu. Vytvořená implementace neobsahuje optimalizaci dělení minimalizací dělených pruhů (tj. postupné zkoušení vstupních křižovatek), ale nebyl by problém algoritmus o tuto možnost rozšířit.

Vstupem dělícího algoritmu je dělená mapa silniční sítě a počet podsítí, do kterých má být mapa rozdělena. Výstupem je pak struktura popsaná v předchozí kapitole, tedy seznam nově vzniklých map a propojení mezi nimi. Celý algoritmus lze rozdělit do pěti kroků, které budou nyní popsány.

7.2.1 Algortimus dělení metodou BFS – 1. krok

Prvním krokem algoritmu je výpočet celkového ohodnocení pro každou novou mapu. Tento výpočet se provede tak, že se spočte celkové ohodnocení silničních pruhů v dělené mapě a vydělí se počtem map, které mají vzniknout dělením. Výsledná hodnota je pak maximální hranice ohodnocení pro každou nově vznikající mapu. Ohodnocení silničních pruhů se získá z parametrů pruhů – každý silniční pruh má informaci o svém ohodnocení (nastavené uživatelem).

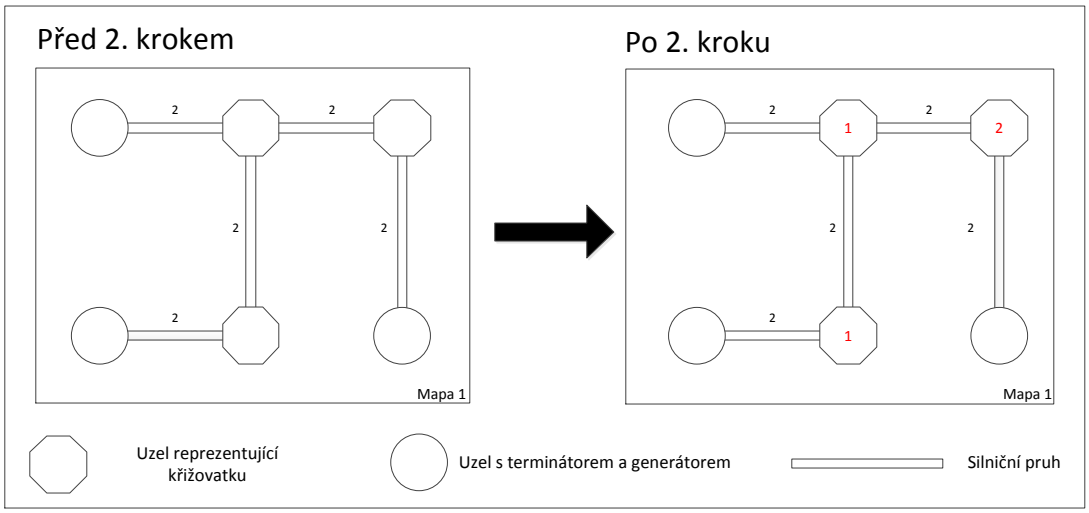
7.2.2 Algortimus dělení metodou BFS – 2. krok

Druhým krokem se zjistí rozložení křižovatek v nových mapách. Celý krok je prováděn ve třídě `AbstractBFSDistributor`. V tomto kroku se dostává poprvé ke slovu metoda BFS. Algoritmus vybere první křižovatku ze seznamu křižovatek a určí jí jako počáteční uzel prohledávání grafu. Křižovatka je vložena do fronty uzlů a zároveň je nastavena její hodnota v pomocné datové struktuře typu `Map<Crossroad, State>` na stav „čerstvá“. Aktuální ID mapy, do které se budou objekty přidávat, se nastaví na 0 a je vynulována celková váha. Algoritmus pak zjistí všechny bezprostřední sousedy křižovatky.

Každý soused je spojen s křižovatkou hranou, tedy posloupností silničních pruhů. Sousedé se zjistí průchodem těchto pruhů. Průchod je zajištěn tak, že každý silniční pruh obsahuje seznam objektů připojených k němu a seznam objektů, ke kterým je připojen (dědění od `RoadStructure`). Díky tomu dokážeme dojít přes posloupnost pruhů až k prvnímu objektu, který není pruh (a je tedy uzlem). Pro průchod pruhů se používá metoda DFS – prohledávání do hloubky. V tomto případě je použita metoda DFS, protože vede k efektivnějšímu získání souseda (postupujeme přímo do hloubky místo do šířky). Více o metodě DFS se lze dozvědět v [Alg12].

Získaní sousedé jsou ve tvaru seznamu objektů typu `EvaluatedNeighbour`. Každý objekt typu `EvaluatedNeighbour` obsahuje uzel a celkové ohodnocení silničních pruhů, které z křižovatky do tohoto souseda vedou. Ohodnocení se bere obousměrné, tedy součet ohodnocení silničních pruhů vedoucí z křižovatky do

souseda a zpět. Ze sousedů se vyberou ti sousedé, kteří nejsou typu křižovatka, a jejich ohodnocení je připočteno do celkové váhy. Následně jsou procházeni sousedé typu křižovatka, a pokud nebyli ještě objeveni (jejich stav je v `Map<Crossroad, State>` veden jako „čerstvý“), jsou vkládáni do fronty křižovatek. Zároveň je připočítáváno jejich ohodnocení k celkovému ohodnocení, ale pouze pokud je hrana spojující souseda s křižovatkou procházena poprvé. Jinak by docházelo k tomu, že jeden soused by mohl být započítán vícekrát.



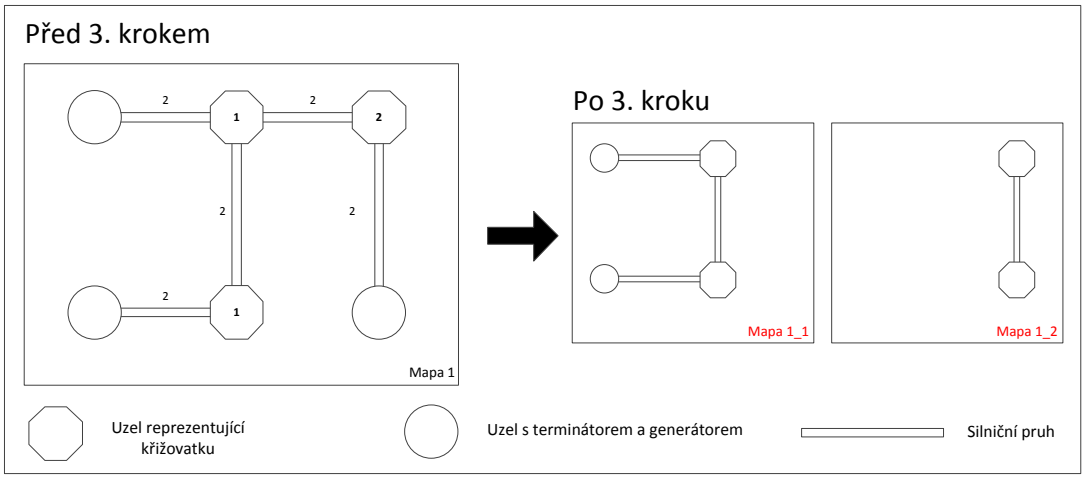
Obrázek 15: Znázornění dělení grafu silniční sítě před a po druhém kroku.

Po průchodu sousedů a přičtení všech ohodnocení se přiřadí křižovatka k mapě, která odpovídá aktuálnímu ID mapy. Následně se otestuje, zda je celková váha větší, než vypočítané ohodnocení pro jednu novou mapu (získané v prvním kroku). Pokud je váha větší (či rovna), zvýší se proměnná aktuálního ID mapy. Aktuální váze je přiřazen rozdíl mezi aktuální váhou a vypočítaným ohodnocením pro jednu mapu. Pokud je váha menší, ani aktuální ID mapy se měnit nebude, protože ještě nebylo dosaženo potřebné maximální hodnoty ohodnocení pro jednu mapu.

V poslední řadě se křižovatka nastaví v `Map<Crossroad, State>` jako „uzavřená“ a je vybrána další křižovatka z fronty křižovatek, jejíž sousedé se budou procházet. Dělení grafu před a po druhém kroku je zobrazeno na obr. 15.

7.2.3 Algortimus dělení metodou BFS – 3. krok

V třetím kroku již víme, v jakých mapách budou jednotlivé křižovatky ležet. Díky této znalosti rozdělení křižovatek mezi mapami můžeme provést kopírování objektů do nových map.



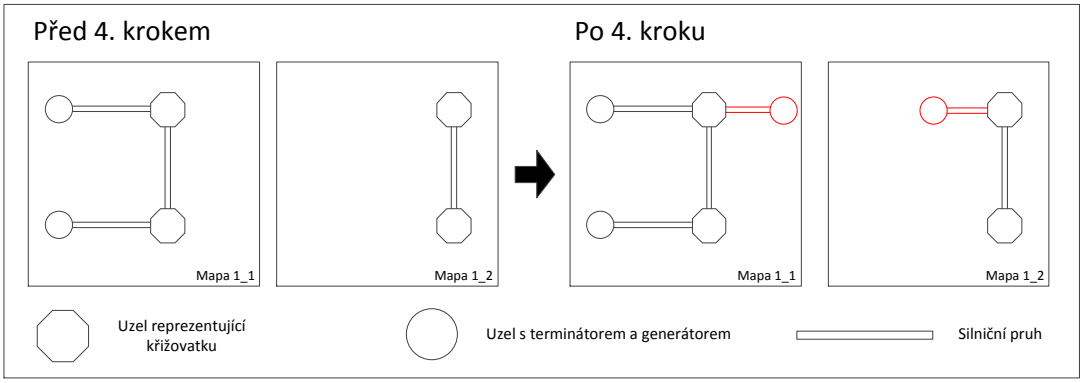
Obrázek 16: Znázornění dělení grafu silniční sítě před a po třetím kroku.

Nejprve se vytvoří prázdné mapy, do kterých se budou postupně přidávat objekty z mapy původní. Poté se provede nakopírování křižovatek do nových map. Zároveň s křižovatkami jsou do map nakopírováni ti sousedé křižovatek, kteří nejsou typu křižovatka (předpokládá se, že tyto sousedy nedělíme), a silniční pruhy, které do těchto sousedů vedou (sousedé typu terminátor a generátor). Dělení grafu před a po třetím kroku je zobrazeno na obr. 16.

7.2.4 Algortimus dělení metodou BFS – 4. krok

Ve čtvrtém kroku se provede kopírování silničních pruhů mezi křižovatkami a sousedy, kteří jsou typu křižovatka. Znovu je procházen celý graf metodou do šířky, a pokud se narazí na hranu, která ještě nebyla při průchodu objevena, provede se vložení silničních pruhů z této hrany do mapy. Protože hrana může být již hranou vedoucí do jiné mapy, je možné, že se provede dělení hrany (respektive silničních pruhů obsažených v hraně).

Dělení hrany proběhne tak, že se vezmou silniční pruhy nejbližší ke zpracovávané křižovatce, a ty jsou rozděleny na dvě půlky. Každá půlka silničního pruhu je vložena do příslušné mapy. Zároveň s dělením se provede vygenerování páru generátor a terminátor. Ty jsou také vloženy do odpovídajících map jako předchozí pruhy. Spolu s vygenerovaným párem je vytvořen i objekt topologie, popisující toto nově vzniklé propojení dvou map. Dělení grafu před a po čtvrtém kroku je zobrazeno na obr. 17.



Obrázek 17: Znázornění dělení grafu silniční sítě před a po čtvrtém kroku.

7.2.5 Algortimus dělení metodou BFS – 5. krok

Poté, co je celý graf již zpracovaný, a všechny objekty z původní mapy jsou nakopírované do map nových, se provede přejmenování objektů v nových mapách. K přejmenování musí dojít, protože všechny objekty byly vkládány na mapu s ID z původní mapy. Jakmile jsou všechny objekty přejmenovány, algoritmus je hotov.

7.3 Implementace algoritmu dělení silniční sítě proporcionální metodou BFS

Algoritmus dělení sítě proporcionální metodou BFS byl naimplementován navíc, jako vhodné rozšíření diplomové práce. Toto řešení bylo naimplementováno hlavně z důvodu otestování funkčnosti modularity programu. Dělení sítě proporcionálně se může využít například pro počítačové sítě s nestejným výkonem výpočetních

uzlů, kde proporcionální dělení umožní rovnoměrné zatížení jednotlivých uzlů.

Vstupem algoritmu je stejně jako u předchozí metody mapa silničních sítí, tentokrát však druhým parametrem není počet podsítí, ale poměrové vyjádření dělení mapy. Toto je nejmarkantnější rozdíl oproti původní metodě. Proporcionální algoritmus jinak funguje stejným způsobem, jako předchozí (tj. obsahuje pět fází dělení grafu).

Poměrové vyjádření je zadané parametrem a udává, s jakým poměrem se bude mapa dělit do menších map. Nejlépe si princip vysvětlíme na příkladu. Mějme zadáno poměrové vyjádření jako hodnotu 1:5:10 a celkové ohodnocení silničních pruhů na mapě ať je 80. Algoritmus nejprve sečte poměry (výsledek 16) a vydělí celkové ohodnocení sítě touto hodnotou. Výsledek pak vynásobí s každou částí poměru – dostaneme 5:25:50. Tímto jsme získali maximální ohodnocení jednotlivých map, do kterých se bude naše hlavní mapa dělit. V dalším kroku pak probíhá dělení stejným způsobem, jako u předchozího algoritmu BFS, ale jako maximální hodnota ohodnocení pruhů je zvoleno dopočtené ohodnocení pro každou mapu. Pro první mapu bude toto maximální ohodnocení 5, pro druhou 25 a pro třetí 50.

8 Testování

Pro ověření správně funkčnosti programu MME bylo vytvořeno několik sérií testů. Celkem se prováděly dva druhy testů – testy na správnou funkčnost grafického editoru a testy na ověření správné činnosti pluginů s dělícími algoritmy. Tyto testy budou popsány v následujím textu.

8.1 Testování grafického editoru

Na grafickém editoru byla provedena série testů, které budou popsány v dalším textu. Shrnutí výsledků testování grafického editoru je na obr. 20. V grafickém editoru byla testována tato funkčnost:

- Umístění objektů na mapě.
- Propojování objektů na mapě.
- Nastavení vlastností objektů.
- Uložení mapy.
- Načtení mapy.

8.2 Testování umístitelnosti objektů na mapě

Každý objekt mapy má svůj specifický tvar. Pro správnou funkčnost programu je třeba otestovat, zda objekty nejde umístit mimo mapu dopravní sítě.

Testování umístění objektu mimo mapu bylo provedeno tak, že jednotlivé objekty byly postupně vybírány z panelu nástrojů a byl s nimi prováděn pohyb po mapě. Pohyb po mapě byl veden po okraji mapy se snahou o přechod mimo mapu. Všechny objekty byly tímto postupem otestovány a ani u jednoho nebylo zjištěno, že by objekt přesahoval přes okraj mapy. Pokud uživatel nemůže pohybem s objektem přesáhnout okraj mapy, nemůže objekt ani přes okraj mapy umístit. Tím bylo otestováno, že objekty nejdou umístit mimo mapu.

8.3 Testování propojitelnosti objektů na mapě

Silniční síť tvoří propojené objekty. Propojení objektů musí splňovat určité logické požadavky, jejichž splnění je ověřováno v programu při spojování objektů. Mezi tyto požadavky patří:

- Terminátor a generátor nejdou napojit přímo do křižovatky.
- Silniční pruh nesmí vést do generátoru.
- Silniční pruh nesmí vést z terminátoru.
- Do vjezdu do křižovatky může vést pruh, ale nemůže vést opačně.
- Z výjezdu z křižovatky může vést pruh, ale nemůže vést opačně.

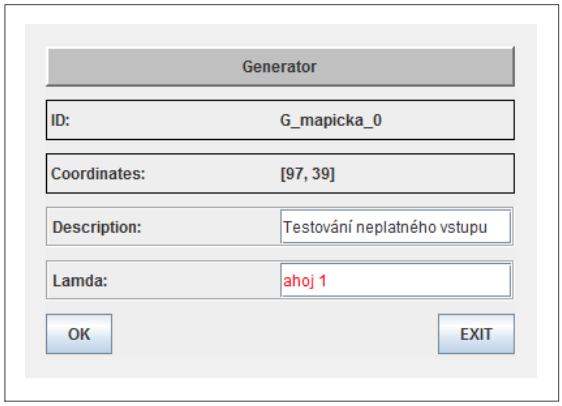
Pro otestování těchto kritérií byla vytvořena mapa silničních sítí. Do mapy byly vloženy objekty tak, aby při případném spojení došlo k nesplnění některého z uvedených požadavků. Následně bylo testováno, zda lze objekty spojit.

Nejprve bylo testováno spojení terminátoru a generátoru přímo na vjezd a výjezd křižovatky – spojení nešlo provést. V dalším kroku bylo testováno napojení silničního pruhu do generátoru a napojení terminátoru na silniční pruh. Opět bylo otestováno, že spojení nelze provést. Jako poslední bylo prověřeno, že ani jízdní pruh nelze napojit do výjezdu z křižovatky a zároveň, že vjezd do křižovatky nemůže být napojen do jízdního pruhu. Všechny požadavky tak byly s úspěchem otestovány.

8.4 Nastavení vlastností objektů na mapě

Každý objekt umístěný na mapu může mít nastavené různé vlastnosti. Standardní vlastností objektu je popis objektu. Popis objektu slouží k přidání textového popisu k objektu. Každý objekt obsahuje také vlastnosti, které jsou specifické právě pro tento objekt. Bylo třeba otestovat, zda vlastnosti všech objektů jdou nastavit, a zda se u číselných vlastností provádí typová kontrola.

Nastavení vlastností objektu se provede kliknutím na objekt umístěný na mapě. Poté se objeví formulář, který toto nastavení umožňuje. Byly testovány



Obrázek 18: Testování vložení neplatného vstupu ve formuláři generátoru.

nastavovací formuláře všech objektů mapy. Shrnutí testování všech formulářů je na obr. 19. Nyní bude popsáno testování jednotlivých formulářů.

8.4.1 Formulář terminátoru

Nejjednodušší formulář, jediná volba, co jde nastavit, je popisek terminátoru. Jako text lze nastavit libovolný textový řetězec včetně prázdného řetězce. Formulář byl otestován vložením textu „ahoj 1“ a následně vložením textu s prázdným textovým řetězcem. Pro oba dva textové řetězce formulář pracoval správně.

8.4.2 Formulář generátoru

Obsahuje proti formuláři terminátoru navíc pole pro nastavení hodnoty lambda. Hodnota pole je kladné reálné číslo, v případě zadání jiného než kladného reálného čísla bude text v políčku zvýrazněn. Do pole lamda byly postupně zadány tyto hodnoty: 0, 1.5, 1000, -1000.9999, „ahoj 1“. Pro všechny číselné nezáporné hodnoty formulář nastavil lambdu správně, pro textový řetězec informoval o špatném formátu vstupu zvýrazněným textem (viz obr. 18). Pole popisku bylo testováno podle stejného scénáře jako u formuláře terminátoru. Test dopadl také bez problémů.

8.4.3 Formulář jízdního pruhu

Mimo nastavení svého popisku obsahuje ještě kolonku na vložení svého ohodnocení. Pole ohodnocení může být pouze kladné reálné číslo. Ohodnocení bylo testováno stejnými hodnotami jako hodnota lambda u formuláře generátoru. Pro hodnoty „ahoj 1“ a -1000.9999 byl uživatel upozorněn na neplatnost vstupu, všechny ostatní hodnoty formulář vyhodnotil jako správné. Pole popisku bylo testováno stejně jako u předchozích formulářů a nevyskytly se žádné problémy.

8.4.4 Formulář křižovatky

Formulář křižovatky se skládá ze čtyř různých podformulářů. Každý podformulář slouží k nastavení různých atributů křižovatky. Všechny podformuláře byly jednotlivě otestovány.

Nejprve byl otestován samotný formulář křižovatky. V tomto formuláři se dá nastavit popisek křižovatky a hlavní silnice v křižovatce. Popisek byl otestován stejným způsobem jako u předchozích formulářů a nebyla nalezena žádná chyba. Nastavení atributu hlavní silnice bylo testováno postupným ověřením, zda křižovatka může mít jako hlavní silnici nastavené libovolné své rameno, či zda nemusí obsahovat žádnou hlavní silnici. Testování prověřilo, že všech pět variant je možných.

Formulář nastavení ramen křižovatky obsahuje pouze možnost nastavení popisku konkrétního ramene. Nastavení popisku bylo otestováno stejným způsobem jako u předchozích formulářů a nebyl objeven žádný problém.

Dalším testovaným podformulářem byl formulář nastavení vjezdů a výjezdů z křižovatky pro konkrétní rameno. Tento formulář obsahuje opět pouze možnost nastavení popisku vjezdu nebo výjezdu ramene. Popisek byl testován stejným způsobem jako u předchozích formulářů a nebyl objeven žádný problém.

Posledním testovaným podformulářem byl formulář nastavení vnitřního provozu křižovatky. V tomto formuláři se nastavuje, jakým směrem může vozidlo jet z jednoho ramen do druhého, a s jakou pravděpodobností se to provede. Pravděpodobnost může být kladné reálné číslo, které je menší nebo rovno jedné.

Všem ramenům byly postupně nastaveny všechny tři možnosti směru jízdy (vpravo, vlevo, středem). Pro každou možnost byly testovány tyto hodnoty pravděpodobnosti: 1, 1.0001, -1 a „ahoj 1“. Pouze pro první hodnotu program neupozornil na chybu, u všech ostatních byl uživatel informován o neplatnosti vstupu. Zároveň formulář umožnil bezproblémové nastavení libovolného směru spojení z ramene křižovatky. Tímto bylo otestováno, že formulář pracuje správně.

Testování formulářů		
Formulář terminátoru	Testování popisku	
	OK	
Formulář generátoru	Testování popisku	Testování vstupu pro hodnotu *lambda*
	OK	**OK**
Formulář jízdního pruhu	Testování popisku	Testování vstupu ohodnocení
	OK	**OK**
Formulář křižovatky	Testování popisku	Testování hlavní silnice
	OK	**OK**
Formulář ramen křižovatky	Testování popisku	
	OK	
Formulář vjezdů a výjezdů	Testování popisku	
	OK	
Formulář vnitřního provozu	Testování směru jízdy	Testování vstupu pro pravděpodobnost
	OK	**OK**

Obrázek 19: Shrnutí testování formulářů.

8.5 Uložení mapy a načtení mapy

Mapa silniční sítě se musí dát uložit ve formátu XML. V tom samém formátu se musí dát také načíst do grafického editoru. Spolu s uložením mapy se musí uložit i všechny objekty mapy (se svými vlastnostmi), které byly na mapu umístěny. Po opětovném načtení mapy musí načtená mapa odpovídat předchozí vytvořené mapě.

Otestování uložení a načtení mapy bylo provedeno tak, že se vytvořila jednoduchá mapa, na které se vyskytovaly všechny objekty, které mapa může obsahovat. Tyto objekty byly v mapě spojené a každý objekt měl nastaven svoje atributy na předem dané hodnoty. Všem objektům byl nastaven popisek na jméno typu objektu (tj. jízdní pruh měl v popisku text: „jízdní pruh“). Generátoru byla lambda nastavena na 1234.567, dvěma silničním pruhům bylo nastaveno ohodnocení na 2

a 200. Křižovatka neměla hlavní silnici a vnitřní provoz byl nastaven takto:

- Ze severního ramene se lze dostat do jižního s pravděpodobností 0.1 středem, do východního s pravděpodobností 0.2 po levé straně a do západního s pravděpodobností 0.7 po pravé straně.
- Z východního ramene se lze dostat do západního s pravděpodobností 0.5 středem, do jižního s pravděpodobností 0.2 po levé straně a do severního s pravděpodobností 0.3 po pravé straně.
- Z jižního ramene se lze dostat do západního s pravděpodobností 0.3333 po levé straně, do severního s pravděpodobností 0.3333 středem a do východního s pravděpodobností 0.333 po pravé straně.
- Ze západního ramene se lze dostat do severního s pravděpodobností 0.45 po levé straně, do východního s pravděpodobností 0.11 středem a do jižního s pravděpodobností 0.44 po pravé straně.

Výsledná mapa byla uložena a poté byla odebrána ze seznamu map otevřených v MME. Následně byla mapa v programu otevřena a bylo zkontrolováno, zda mapa obsahuje všechny objekty, které do ní byly umístěny. Dále bylo zkontrolováno, zda objekty obsahují nastavené atributy. Obě kontroly nenalezly žádnou odlišnost od původní mapy, ukládání a načítání mapy tedy funguje bez potíží.

Testování grafického editoru	
Test umístění objektů na mapě	Testování pohybem po krajních mezích mapy
	OK
Test propojení objektů na mapě	Testování variant napojení
	OK
Test nastavení vlastností objektům	Testování formulářů
	OK
Test uložení mapy	Testování uložení mapy s umístěnými objekty
	OK
Test načtení mapy	Testování načtení mapy s umístěnými objekty
	OK

Obrázek 20: Shrnutí testování grafického editoru.

8.6 Testování správné činnosti pluginů

Testování pluginů můžeme rozdělit do tří částí. V první části bylo nejprve třeba otestovat, zda program MME dokáže reagovat na přidání či odebrání bundlu s implementací dělícího algoritmu z OSGi frameworku. V druhé a třetí části byly poté otestovány samotné pluginy s algoritmy dělení.

8.7 Testování přidávání a odebíraní bundlů s dělícími algoritmy

Pro tento test byly využity již naimplementované bundly obsahující algoritmy dělení. V prvním kroku testu byl program MME spuštěn v Equinoxu zároveň s oběma bundly obsahujícími algoritmy dělení. Oba dva algoritmy byly spuštěny s menší prodlevou po hlavním bundlu programu. Program zaregistroval oba dva bundly bez problémů a oba zobrazil v menu programu.

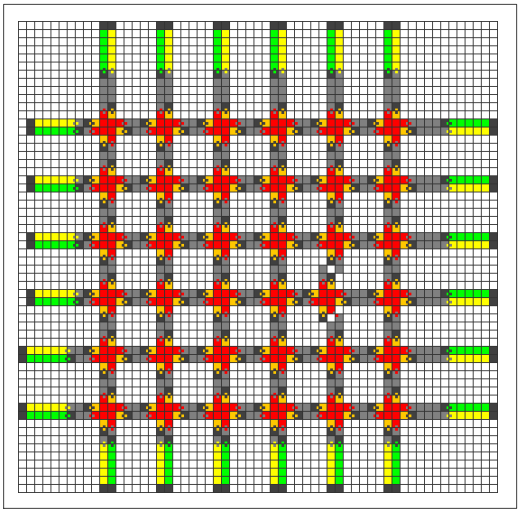
Ve druhém kroku testu bylo prověřeno odebrání bundlů z OSGi frameworku. Nejprve byl pomocí příkazu `stop` zastaven jeden bundle a poté byl tím samým příkazem zastaven i bundle druhý. Program MME obě dvě změny zachytil a ve výpisu dostupných algoritmů již nebyly algoritmy uvedeny.

Ve třetím kroku byly oba dva bundly pomocí příkazu `start` opět aktivovány ve frameworku. Program MME dokázal tuto opětovnou aktivaci zachytit a zobrazil oba dostupné bundly v nabídce algoritmů dělení.

Těmito třemi kroky bylo ověřeno, že program MME dokáže reagovat na přidání a odebrání bundlů s dělícími algoritmy z OSGi frameworku.

8.8 Testování BFS algoritmu dělení

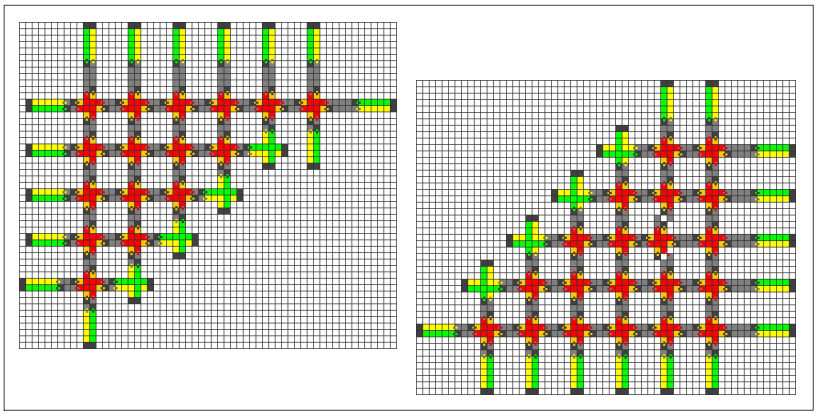
Pro otestování BFS algoritmu dělení byla vytvořená speciální mapa silničních sítí. Křižovatky byly umístěny do mapy v mřížce o rozměru 6 x 6 křižovatek. V mřížce byly křižovatky umístěny po sloupcích od shora dolů, s přechodem směrem zleva doprava. První křižovatka byla v horním levém rohu a poslední umístěná křižovatka byla ve spodním pravém rohu. Mřížka je vidět na obr. 21. Všechny pruhy mezi křižovatkami měly nastavené stejné ohodnocení na hodnotu jedna. Krajní křižovatky dále obsahovaly napojení na terminátory a generátory. Celý



Obrázek 21: Původní mapa.

tento tvar mapy byl zvolen, aby bylo dobře viditelné rozdělení mřížky metodou BFS. Dělení pomocí BFS této mřížky by mělo mapu rozdělit tak, že mapa bude rozdělena šikmými řezy kolmými na diagonálu mapy vedoucí z hora dolů.

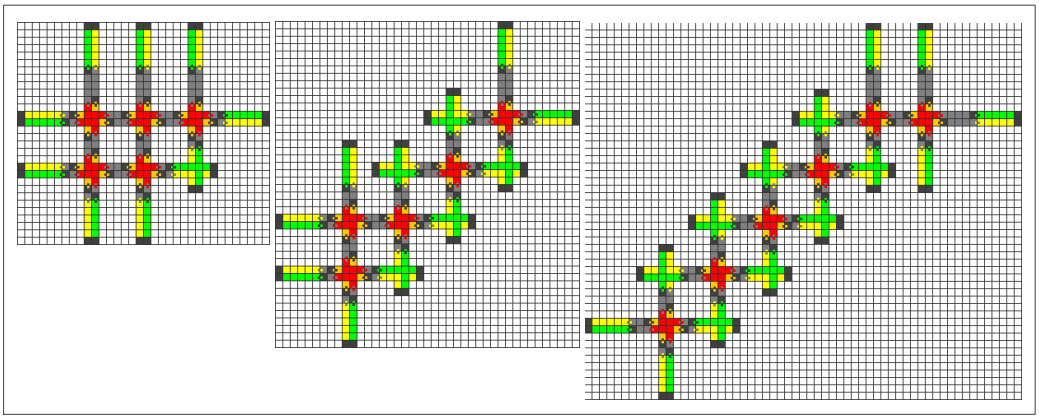
Nejprve bylo testováno rozdělení mřížky na dvě části. Výsledek dělení je vidět na obr. 22. Mřížka byla rozdělena správně, vznikly dvě nové mapy, které mají tvar původní mapy rozdělené šikmými řezy.



Obrázek 22: Mapa rozdělená na dvě části algoritmem BFS.

V dalším kroku byl na nově vzniklou mapu z původního řezu aplikován opět algoritmus BFS, tentokrát ale s rozdělením mapy na části tři. Výsledek dělení je vidět na obr. 23. Opět byla mapa rozdělena správně, nově vzniklé mapy mají

tvar odpovídající vedeným šikmým řezům. Pomocí testu bylo zjištěno, že BFS algoritmus dělení funguje správně.

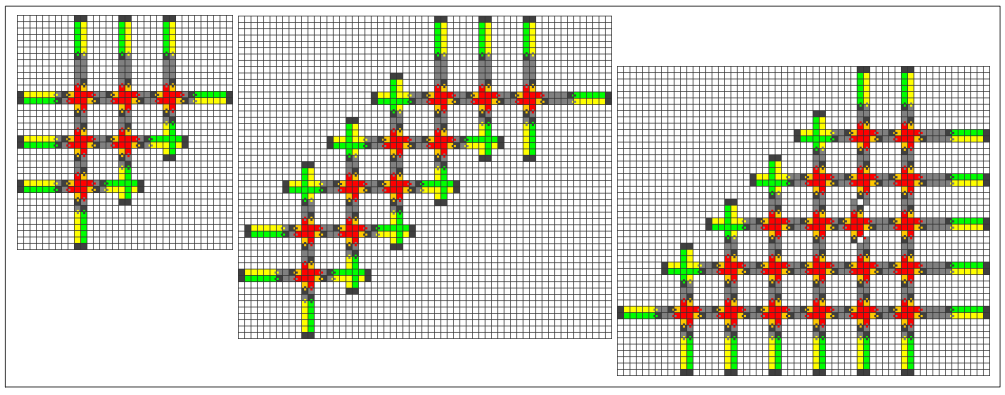


Obrázek 23: Rozdělení mapy (vzniklé dělením původní mapy) na tři části.

8.9 Testování proporcionálního BFS algoritmu dělení

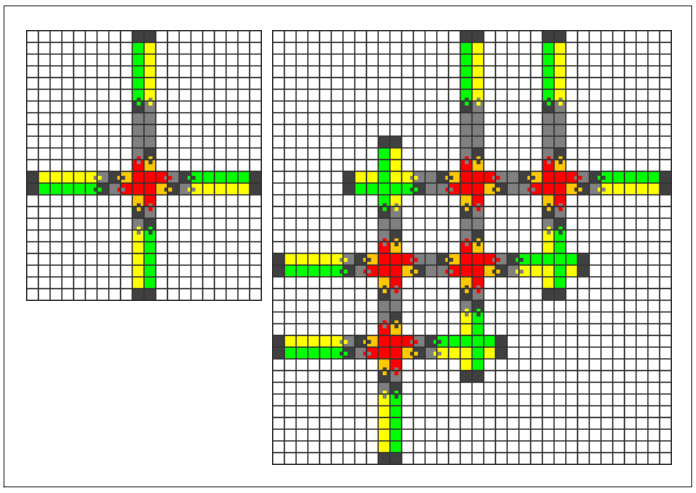
Při testování proporcionálního BFS algoritmu byla použita stejná mapa s mřížkou, jako u testování normálního BFS algoritmu dělení. Nejprve byla mapa rozdělena v poměru 1:3:5. Výsledek dělení je vidět na obr. 24. Rozdělené mapy mají podobu řezů kolmých na diagonálu a jejich velikost odpovídá přibližně poměru, se kterým byly děleny.

V dalším kroku byla nově vzniklá mapa (horní část původní mapy) rozdělena



Obrázek 24: Rozdělení původní mapy proporcionálním BFS algoritmem v poměru 1:3:5.

v poměru 1:4. Výsledek dělení je vidět na obr. 25. Nově vzniklé mapy odpovídají svou velikostí poměru, se kterým byly rozděleny.



Obrázek 25: Rozdělení mapy (vzniklé dělením původní mapy) v poměru 1:4.

U obou dvou testů byly mapy rozděleny odpovídajícím poměrem a tím byla otestována správná funkčnost proporcionálního BFS algoritmu dělení.

Testování správné činnosti pluginů			
Test přidání a odebrání pluginů	Testování spuštění programu s pluginy	Testování odebrání pluginů	Testování přidání pluginů
	OK	**OK**	**OK**
Test BFS algoritmu dělení	1. Rozdělení mapy	2. Rozdělení mapy	
	OK	**OK**	
Test proporcionálního BFS algoritmu	1. Rozdělení mapy	2. Rozdělení mapy	
	OK	**OK**	

Obrázek 26: Shrnutí testování pluginů.

9 Závěr

Předložená práce splňuje zadání ve všech bodech a v některých částech ho i rozšiřuje. Program MME umožňuje uživatelský příjemným způsobem pracovat s XML mapami silničních sítí. Mapy silničních sítí dokáže program vytvářet, ukládat, editovat a dělit. Díky použití komponentového frameworku OSGi je možné do programu snadno přidat další pluginy s novými dělícími algoritmy.

Do mapy lze vložit všechny typy objektů, které můžeme nalézt v simulačním systému DUTS. Tyto objekty jdou do mapy libovolně vkládat, zároveň jdou z mapy odebírat, či se dají nastavovat jejich vlastnosti. Všechny tyto operace jsou díky grafickému editoru pro uživatele intuitivní a přehledné.

Spolu s grafickým editorem byly naimplementovány také dva algoritmy dělení. Druhý algoritmus dělení byl naimplementován jako rozšíření diplomové práce, které nebylo uvedeno v zadání. Oba dva algoritmy byly vytvořeny jako pluginy, na kterých byla otestována funkčnost modularity programu. Jak bylo s úspěchem otestováno v kapitole 8.6, oba algoritmy dokážou správně dělit mapu silniční sítě do menších map.

Modularita programu umožňuje jednoduše přidat nový algoritmus dělení do programu, a tak vylepšit jeho stávající funkčnost. Do budoucna je tak možné program snadno rozšiřovat přidáním nových algoritmů dělení. Jako alternativa rozšíření programu k přidávání nových algoritmů dělení se jeví například validace vstupních map silničních sítí.

Seznam zkratek

MME Modular Map Editor

DUTS Distributed Urban Traffic Simulator

ORB Ortogonální rekurzivní bisekce

OSGi Open Services Gateway Initiative

BSD Berkeley Software Distribution

JAR Java ARchive

XML Extensible Markup Language

API Application Programming Interface

XSD XML Schema Definition

SAX Simple API for XML

StAX Streaming API for XML

DOM Document Object Model

BFS Breadth First Search

DFS Deep First Search

Reference

[Fuj00] FUJIMOTO, R. M. *Parallel and Distributed Simulation Systems.* New York: J.Wiley, c2000, 300 s. Wiley series on parallel and distributed computing. ISBN 04-711-8383-0.

[Ham97] HAMILTON, J. A. - POOCH, U. W. - NASH, D. A. *Distributed Simulation.* CRC Press 1997; 1 edition, 416 s. ISBN 978-0849325908.

[Cai09] CAIS, Š. *Editor pro tvorbu dopravní sítě a její dělení pro distribuovanou simulaci silniční dopravy.* Plzeň 2009. 67 s. Diplomová práce. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky.

[Pot07] POTUŽÁK, T. - HEROUT, P. *Use of Distributed Traffic Simulation in the JUTS Project* Proceedings of EUROCON 2007 The International Conference on „Computer as a Tool“ ISBN 1-4244-0813-X.

[Pot06] POTUŽÁK, T. *Current Trends in Distributed Traffic Simulation.* Proceedings of 41th Spring International Conference MOSIS '07 - Modelling and Simulation of Systems 2006 ISBN 978-80-86840-30-7.

[Pot09] POTUŽÁK, T. *Methods for reduction of inter-process communication in distributed simulation of road traffic.* Plzeň, 2009. 152 s. Dizertační práce. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky.

[Pot11] POTUŽÁK, T. *Utilization of a Genetic Algorithm in Division of Road Traffic Network for Distributed Simulation* ECBS-EERC 2011 - 2011 Second Eastern European Regional Conference on the Engineering of Computer Based Systems Bratislava, září 2011, str. 151-152, ISBN 978-0-7695-4418-2.

[Pot10] POTUŽÁK, T. *Division of Traffic Network for Distributed Microscopic Traffic Simulation Based on Macroscopic Simulation.* Proceedings of the

7th EUROSIM Congress on Modelling and Simulation, Vol. 2, Prague (2010).

[Lig55] LIGHILL, M. H. - WHITMAN, G. B. *On Kinematic Waves II: A Theory of Traffic Flow on Long Crowded Roads.* Proceeding of the Royal Society of London s. A, 229, pp. 317-345, 1955.

[Gor04] GORGOULIS - TERSTYANSKY, G. - KACSUK, P. - WINTER, S. *Creating Scalable Traffic Simulation on Clusters.* Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network- Based Processing (EUROMICRO-PDP'04), 2004.

[Kle05] KLEFSTAD, R. - ZHANG, Y. - LAI, M. - JAYAKRISHNAN, R. - LAVANYA, R. *A Scalable Synchronized, and Distributed Framework for Large-Scale Microscopic Traffic Simulation.* Proc. of Intelligent Trans. Syst, Vienna, 2005.

[Cet03] CETIN - BURRI, A. - NAGEL, K. *A Large-Scale Agent-Based Traffic Microsimulation Based on Queue Model.* Proceedings of 3rd Swiss Transport Research Conference, 2003.

[Nag96] NAGATANI, T. *Gas Kinetic Approach to Two-Dimensional Traffic Flow.* J. Phys Soc Jap 65(10), pp. 3150-3152, 1996.

[Niz02] NIZARD, L. *Combining Microscopic and Mesoscopic Traffic Simulators.* Raport de stage d'option scientifique, Ecole Polytechnique, Paris, 2002.

[Wag04] WAGNER, P. - LUBASHEVSKY, L. *Empirical Basis for Car Following Theory Development.* Eprint arXiv:condmat/0311192, 2004.

[Kle98] KLEIN, U. - SCHULZE, T. - STRASSBURGER, S. - MENZLER, H. *Distributed Traffic Simulation based on the High Level Architecture.* Proceedings of Simulation Interoperability Workshop, USA, 1998.

[Cad04] ČADA, R. - KAISER, T. - RYJÁČEK, Z. *Diskrétní matematika.* Plzeň: Západočeská univerzita, 2004. ISBN 80-7082-939-7.

[Nag01] NAGEL, K. - RICKERT, M. *Parallel Implementation of the TRANSIMS Micro-Simulation.* Parallel Computing, 27(12):1611-1639, 2001.

[Abo06] ABOU-RJEILI, A. - KARYPIS, G. *Multilevel Algorithms for Partitioning Power-Law Graphs.* IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2006.

[Con03] CONG, J. - HINNERL, J. R. *Multilevel Optimization in Vlsicad.* Springer; 1 edition (March 31, 2003).

[Szy02] SZYPERSKI, C. *Component software: beyond object-oriented programming.* 2nd ed. Addison-Wesley: London, 2002. xxxii, 589 s. ISBN 0-201-74572-6.

[Bac00] BACHMANN, F., et al. *Volume II: Technical Concepts of Components-Based Software Engineering* 2nd ed. Carnegie Mellon University: Pittsburgh, 2000.

[Bas08] BASL, J. *Podnikové informační systémy: podnik v informační společnosti.* 2., výrazně přepracované a rozšířené vydání. Grada: Praha, 2008. 283 s. ISBN 978-80-247-2279-5.

[Hal11] HALL, RICHARD S., et al. *OSGi in Action: Creating Modular Applications in Java.* Manning Publications Co. 2011, ISBN 9781933988917.

[Jun12] *Java Universal Network / Graph framework* [online]. 2012 [cit. 2012-04-26] Dostupné z WWW: http://jung.sourceforge.net/.

[Equ12] *Equinox* [online]. 2012 [cit. 2012-04-9] Dostupné z WWW: http://www.eclipse.org/equinox/.

[Fel12] *Apache felix - index* [online]. 2012 [cit. 2012-04-9] Dostupné z WWW: http://felix.apache.org/site/index.html.

[Kno12] *Knopflerfish OSGI - open source OSGi service platform* [online]. 2012 [cit. 2012-04-9] Dostupné z WWW: http://www.knopflerfish.org/.

[Con12] *Maven - Concierge OSGi - An optimized OSGi R3 implementation for mobile and embeded systems - overview* [online]. 2012 [cit. 2012-04-9] Dostupné z WWW: http://concierge.sourceforge.net/.

[Her07] HEROUT, P. *Java a XML*. 1. vyd. České Budějovice: Kopp, 2007, 313 s. ISBN 978-80-7232-307-4.

[Mig12] *MiG Layout Java Layout Manager for Swing and SWT* [online]. 2012 [cit. 2012-04-28] Dostupné z WWW: http://www.miglayout.com/.

[Alg12] *Algorithm Design* [online]. 2012 [cit. 2012-05-05] Dostupné z WWW: ww3.algorithmdesign.net/handouts/DFS.pdf.

Seznam obrázků

Přílohy dokumentu

A Uživatelská příručka

Uživatelská příručka slouží jako manuál popisující postupy, jak pracovat s programem MME. Příručka je rozdělena do několika kapitol, ve kterých jsou tyto postupy popsány.

A.1 Prostředí pro běh programu MME

Program MME byl vyvíjen pro běh v prostředí OSGi frameworku Equinox. Pro správnou funkčnost programu je doporučeno používat také tento framework.

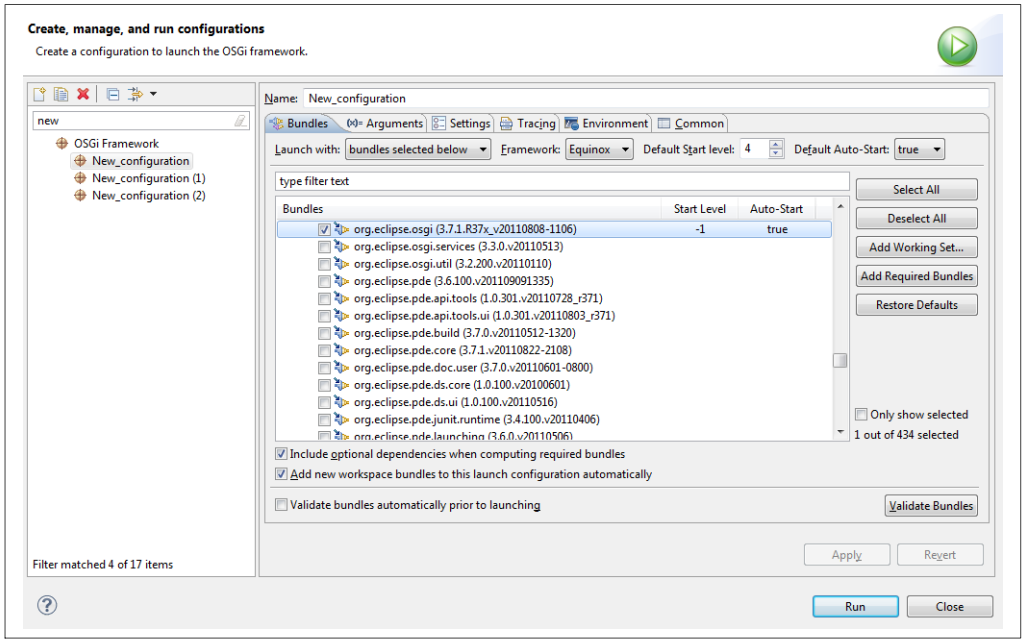
Program MME byl vytvořen jazykem Java ve verzi Java SE 1.7. Je doporučeno používat tuto (či vyšší) verzi Javy. Program nejde spustit pod Java verze 1.6 či nižší.

A.1.1 Instalace prostředí

Framework Equinox je součástí vývojového prostředí Eclipse, proto je třeba nejprve nainstalovat Eclipse SDK. Program byl vytvořen v Eclipse SDK verze 3.7.2.

Po instalaci Eclipse je postup následující. Nejprve je potřeba vytvořit spouštěcí konfiguraci OSGi – tu vytvoříme v Elipse pomocí `Run -> Run Configurations`. V levém seznamu vybereme položku `OSGi Framework` a dvakrát na ní poklikáme levým tlačítkem myši – vytvoří se nová konfigurace, která se otevře v pravé části okna. V položkách `Target Platform` necháme zaškrtnutý pouze bundle se jménem `org.eclipse.osgi`. Poté spustíme konfiguraci tlačítkem `Run`. Ukázka nastavení konfigurace je vidět na obr. 27.

OSGi framework se spustí a můžeme ho ovládat pomocí standardních příkazů (více lze nalézt v [Equ12]). Nejprve nainstalujeme bundle obsahující implementaci grafického editoru. Bundle se nainstaluje příkazem `install file:CESTA/NAZEV`, kde `CESTA` je cesta k adresáři, ve kterém je bundle umístěný a `NAZEV` je název bundlu. Bundle s grafickým editorem má název `cz.sc.mme-1.0.0.jar`. Příkaz může vypadat například takto: `install file:d:/plugins/cz.sc.mme-1.0.0.jar`. Po instalaci bundlu s grafickým editorem nainstalujeme také pluginy obsahující algoritmy dělení. Nainstalování provedeme stejným způsobem, jako při instalaci



Obrázek 27: Nastavení konfigurace v prostředí Eclipse.

bundlu s grafickým editorem, pouze se změní název bundlu. Bundly obsahující dělící algoritmy mají název: `cz.sc.bfsmapdivider-1.0.0.jar` a `cz.sc.propbfs-mapdivider-1.0.0.jar`. To, že jsou bundly správně nainstalované, můžeme zkontrolovat příkazem `ss`, který vypíše všechny bundly v OSGi frameworku.

Po instalaci je třeba bundly spustit. Bundly můžeme spouštět v libovolném pořadí. Spuštění se provede příkazem `start CISLOBUNDLU`, kde `CISLOBUNDLU` je číselné označení bundlu (id) v OSGi frameworku. Toto číslo je vypsáno u bundlu při použití příkazu `ss`. Pokud chceme spustit všechny tři bundly a mít tak k dispozici grafický editor i oba dva algoritmy dělení, provedeme například tuto sérii příkazů: `start 2`, `start 3`, `start 4` (předpokládáme, že bundle s grafickým editorem má id `2`, dělení metodou BFS `3` a proporcionální dělení `4`).

A.2 Hlavní okno programu

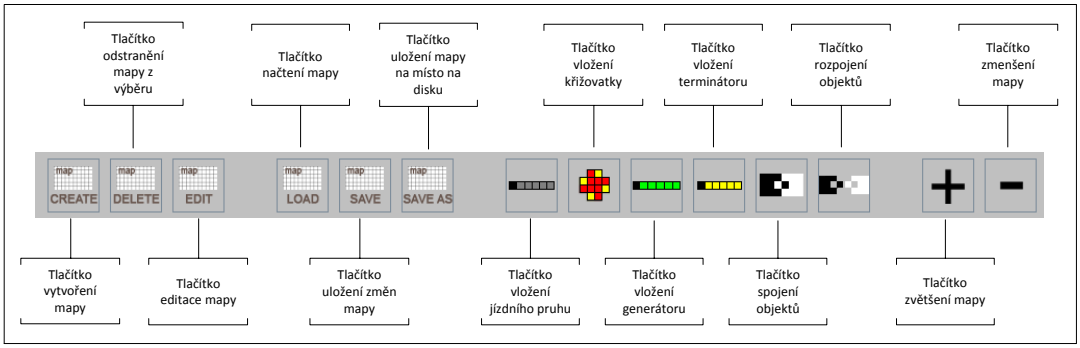
Hlavní okno programu se skládá z pěti částí, které jsou vidět na obr. 28. Panel s mapou sloužící k zobrazení aktuální mapy je označen číslem `1`. Pod číslem `2` najdeme panel, ve kterém se zobrazují všechny mapy, které byly načteny nebo

Obrázek 28: Hlavní okno programu.

vytvořeny. Panel s číslem 3 pokrývá konzoli, která slouží k zobrazování informací pro uživatele. Panel číslo 4 obsahuje nástroje na práci s mapou a dále objekty, které lze do mapy vložit. Poslední panel číslo 5 zobrazuje menu programu.

A.3 Panel nástrojů

Panel nástrojů se používá pro práci s mapou a pro vkládání objektů na mapu. Na obr. 29 jsou popsána jednotlivá tlačítka panelu nástrojů.



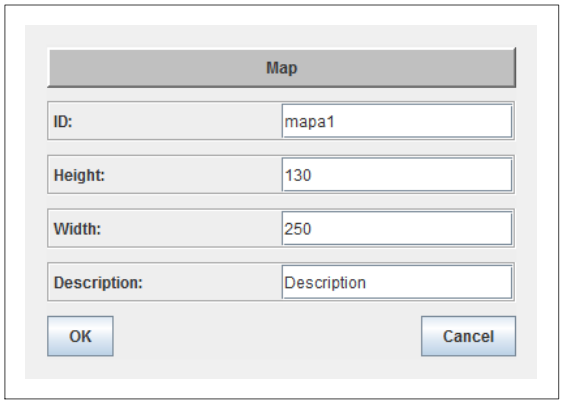
Obrázek 29: Panelu nástrojů.

A.4 Práce s mapou

Mapa silničních sítí se dá v programu MME vytvořit, uložit a načíst. Tyto funkce budou popsány v následujícím textu.

A.4.1 Vytvoření mapy

Nová mapa se vytvoří buď přes nástrojový panel kliknutím na tlačítko `CREATE MAP`, nebo přes `Menu -> New map`, nebo klávesovou zkratkou `CTRL + N`.



Obrázek 30: Formulář pro vytvoření mapy.

Mapa se vytvoří přes formulář, který je vidět na obr. 30. Formulář mapy obsahuje tyto položky:

- `ID` – název mapy, pod kterým bude mapa uložena ve formátu XML.
- `Height` – výška mapy udaná v buňkách.
- `Width` – šířka mapy udaná v buňkách
- `Description` – popis mapy.

Po stisknutí tlačítka `OK` se vypočte ideální rozlišení panelu pro zadanou výšku a šířku mapy a vytvoří se nová mapa. Nová mapa je zatím vytvořená pouze v editoru, pro vytvoření jejího XML souboru je třeba mapu uložit.

A.4.2 Uložení mapy

Uložení mapy se dá provést dvěma způsoby. Prvním způsobem je uložení změn do mapy a druhým uložením mapy do nové pozice na disku.

A.4.3 Uložení změn mapy

Uložení změn mapy se provede buď kliknutím na tlačítko `SAVE` v panelu nástrojů, nebo přes `Menu -> Save map`, nebo přes klávesovou zkratku `CTRL + S`. Při uložení změn mapy se všechny neuložené změny uloží do XML souboru mapy. Pokud ještě nebyla mapa uložena, vytvoří se XML soubor s mapou. Soubor bude mít název stejný jako je ID mapy.

A.4.4 Uložení mapy do nové pozice v souborovém systému

Uložení mapy do nové pozice v souborovém systému se provede buď kliknutím na tlačítko `SAVE AS` v panelu nástrojů, nebo přes `Menu -> Save map as`, nebo klávesovou zkratkou `CTL + A`. Uživateli je nabídnut dialog, kde vybere složku, kam má být mapa uložena. Po vybrání složky je mapa do této složky uložena pod svým ID jako XML soubor.

A.4.5 Odstranění mapy

Odstranění mapy odebere mapu z výběru map, ale nesmaže její soubor na disku. Odstranění mapy z výběru se provede kliknutím na tlačítko `DELETE` v panelu nástrojů, nebo přes `Menu -> Remove map`, nebo přes klávesovou zkratku `CTRL + R`. Vybraná mapa v panelu s mapami je poté odebrána.

A.4.6 Načtení mapy

Načtení mapy se provede buď kliknutím na tlačítko `LOAD` v panelu nástrojů, nebo přes `Menu -> Open map`, nebo přes klávesovou zkratku `CTRL + O`. Uživateli je zobrazen dialog, přes který vybere XML soubor s mapou, kterou chce otevřít. Mapa se po načtení zobrazí v panelu s mapami.

A.4.7 Zvětšení / zmenšení mapy

Mapy načtené v programu lze zvětšovat nebo zmenšovat pomocí tlačítek s obrázky plus a mínus v panelu nástrojů. Aktuální vybrané mapě je po stisknutí tlačítka plus zvětšena velikost všech buněk, a tím je dosaženo efektu zvětšení mapy. Obráceně stisknutím tlačítka mínus je velikost buněk zmenšena.

A.5 Práce s objekty

Mapa silničních sítí může obsahovat čtyři základní objekty – jízdní pruh, křižovatku, terminátor a generátor. Každý objekt lze do mapy vložit, spojit s jiným objektem, nastavit mu vlastnosti, či odebrat z mapy. V následujícím textu bude práce s objekty popsána.

A.5.1 Vložení objektů na mapu

Vložení objektu se provede přes panel nástrojů kliknutím na objekt a následně jeho umístěním na místo v mapě. Umístění se provede kliknutím levého tlačítka na pozici na mapě. Objekty nejdou umístit mimo mapu a nejdou ani umístit přes sebe. Pro umístění dalšího objektu stačí kliknout na další objekt v panelu nástrojů. Pokud uživatel nechce umístit objekt na mapě, stačí kliknout pravým tlačítkem myši kamkoliv na mapu.

A.5.2 Spojení a odpojení objektů

Spojení objektů se provede kliknutím na tlačítko spojení objektů. Poté se očekává, že uživatel spojí dva objekty, které se spojit dají, způsobem, který neodporuje logice programu. Program umožňuje vytvořit tato spojení:

- Generátor na jízdní pruh.
- Jízdní pruh na terminátor.
- Jízdní pruh na jízdní pruh.
- Výjezd z křižovatky na jízdní pruh.

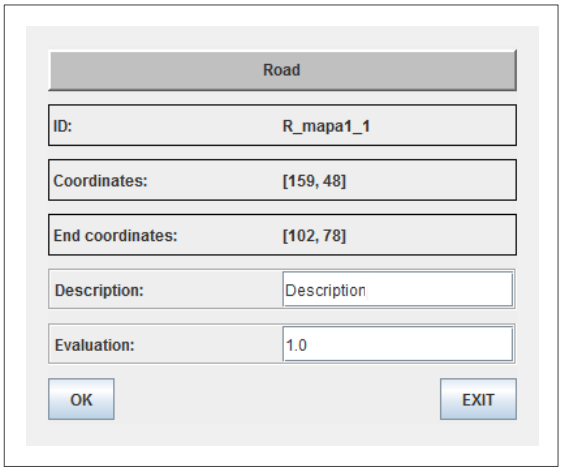
- Jízdní pruh na vjezd do křižovatky.

Jiná spojení nejsou možná. Pokud se uživatel pokusí spojit například generátor s terminátorem, program ho upozorní varovnou hláškou a spojení neprovede. Jízdní pruh je možné napojit na jízdní pruh, ale pouze na jeho první buňky (buňka zvýrazněná tmavě šedou barvou).

Rozpojení objektů se provede kliknutím na tlačítko rozpojení objektů a následným zvolením dvojice objektů, které mají být rozpojeny. Pokud uživatel zvolí objekty, které nejsou spojené, program ho upozorní varovnou hláškou, jinak odstraní spojení mezi objekty.

A.5.3 Odebrání objektů z mapy

Objekty se odebírají z mapy kliknutím pravého tlačítka na objekt na mapě. Pokud uživatel odebere objekt, který byl spojený s jinými objekty, budou odpovídající spojení také odebrána z mapy.



Obrázek 31: Formulář pro nastavení vlastností jízdního pruhu.

A.5.4 Nastavení vlastností objektů

Každému objektu mapy lze nastavit vlastnosti. Vlastnosti se nastavují kliknutím levým tlačítkem na objekt umístěný na mapě. Po kliknutí na objekt se zobrazí

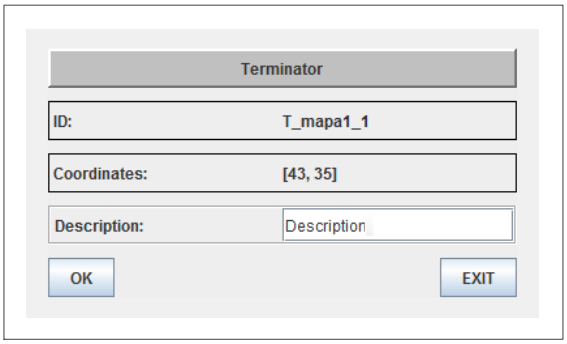
formulář s informace o objektu, ve kterém se dají nastavit i vlastnosti objektu. V následujícím textu budou popsány formuláře všech objektů.

A.5.5 Formulář jízdního pruhu

Formulář jízdního pruhu lze vidět na obr. 31. Formulář obsahuje tyto položky:

- `ID` – identifikační číslo jízdního pruhu.
- `Coordinates` – počáteční souřadnice jízdního pruhu.
- `End coordinates` – konečná souřadnice jízdního pruhu.
- `Description` – popisek jízdního pruhu.
- `Evaluation` – ohodnocení jízdního pruhu.

Lze nastavit vlastnosti popisku a ohodnocení jízdního pruhu. Ohodnocení jízdního pruhu určuje váhu jízdního pruhu při dělení mapy. Ostatní vlastnosti jízdního pruhu jsou neměnné.



Obrázek 32: Formulář pro nastavení vlastností terminátoru.

A.5.6 Formulář terminátoru

Formulář terminátoru lze vidět na obr. 32. Formulář obsahuje tyto položky:

- `ID` – identifikační číslo terminátoru.
- `Coordinates` – počáteční souřadnice terminátoru.

- `Description` – popisek terminátoru.

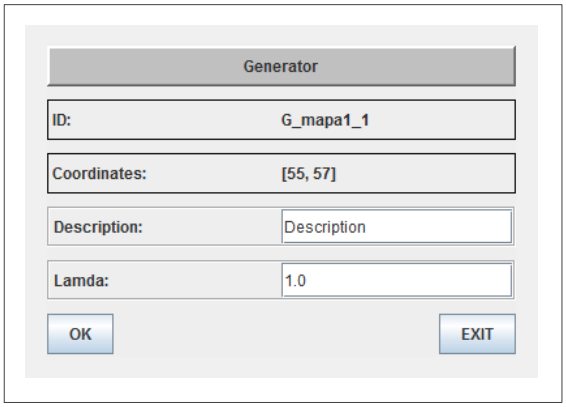
Lze nastavit pouze popisek, ostatní vlastnosti jsou neměnné.

A.6 Formulář generátoru

Formulář generátoru lze vidět na obr. 33. Formulář obsahuje tyto položky:

- `ID` – identifikační číslo generátoru.
- `Coordinates` – počáteční souřadnice generátoru.
- `Description` – popisek generátoru.
- `Lambda` – frekvence generování vozidel.

Lze nastavit popisek a hodnotu lambda. Lambda udává frekvenci generování vozidel v simulaci. Ostatní vlastnosti generátoru jsou neměnné.



Obrázek 33: Formulář pro nastavení vlastností generátoru.

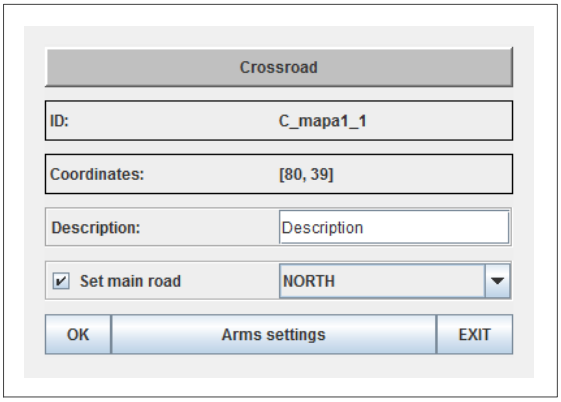
A.6.1 Formulář křižovatky

Skládá se celkem ze čtyř formulářů – formuláře křižovatky, formuláře ramen křižovatky, formuláře pro nastavení vjezdů a výjezdů křižovatky a formuláře pro nastavení provozu uvnitř křižovatky.

Formulář křižovatky lze vidět na obr. 34. Formulář obsahuje tyto položky:

- `ID` – identifikační číslo generátoru.
- `Coordinates` – počáteční souřadnice generátoru.
- `Description` – popisek generátoru.
- `Set main road` – nastavení hlavního ramene křižovatky.
- `Arms settings` – tlačítko pro přechod do formuláře ramen křižovatky.

Ve formuláři křižovatky lze nastavit popis křižovatky a hlavní silnici křižovatky. Dále se dá přepnout do formuláře pro nastavení ramen křižovatky.



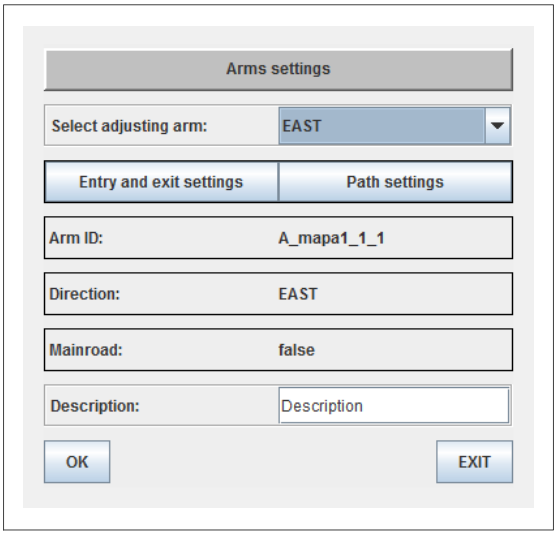
Obrázek 34: Formulář pro nastavení vlastností křižovatky.

Formulář ramen křižovatky lze vidět na obr. 35. Formulář obsahuje tyto položky:

- `Select adjusting arm` – výběr ramene, které se bude nastavovat.
- `Entry and exit settings` – tlačítko pro přepnutí do formuláře nastavení vjezdů a výjezdů ramene.
- `Path settings` – tlačítko pro přepnutí do formuláře nastavení provozu z vybraného ramene do dalších ramen.
- `Arm ID` – identifikační číslo ramene křižovatky.
- `Direction` – umístění ramene křižovatky v rámci křižovatky.
- `Main road` – hodnota udává, zda je rameno hlavním ramenem křižovatky.

- `Description` – nastavení popisku ramene křižovatky.

Ve formuláři ramen křižovatky se nastavuje popisek jednotlivých ramen křižovatky. Dále se dá z formuláře ramen přepnout do formuláře nastavení vjezdů a výjezdů vybraného ramene, nebo do nastavení provozu z vybraného ramene do dalších ramen. Ostatní vlastnosti jsou neměnné.



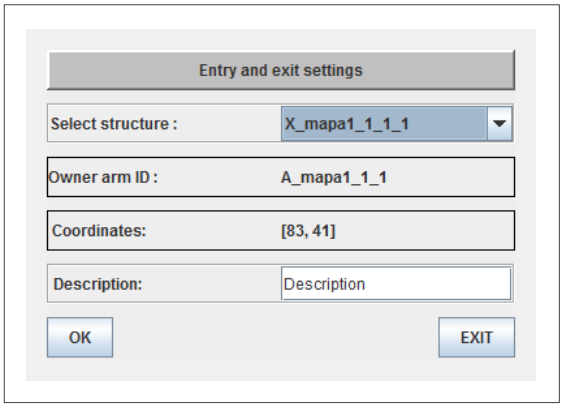
Obrázek 35: Formulář pro nastavení vlastností ramen křižovatky.

Formulář vjezdů a výjezdů ramene křižovatky lze vidět na obr. 36. Formulář obsahuje tyto položky:

- `Select structure` – výběr vjezdu nebo výjezdů ramene, který se bude nastavovat.
- `Owner arm ID` – název ramene, které obsahuje vybraný vjezd nebo výjezd.
- `Coordinates` – souřadnice vybraného vjezdu nebo výjezdu na mapě.
- `Description` – popisek vybraného vjezdu nebo výjezdu.

Ve formuláři vjezdů a výjezdů lze nastavit popisek u konkrétních vjezdů a výjezdů vybraného ramene křižovatky. Ostatní vlastnosti jsou neměnné.

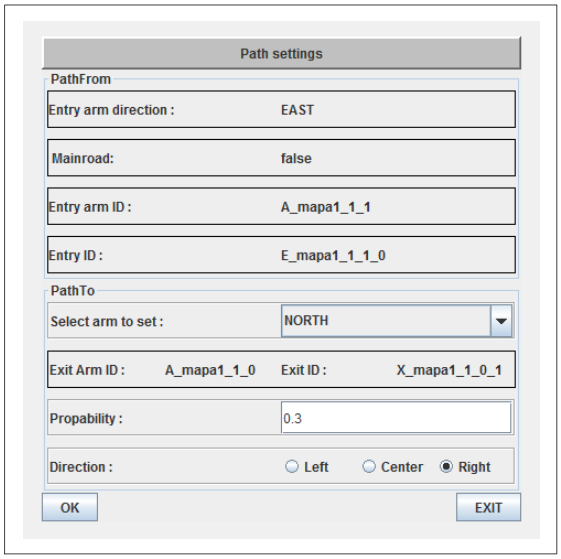
Formulář nastavení provozu uvnitř křižovatky lze vidět na obr. 37. Formulář obsahuje tyto položky:



Obrázek 36: Formulář pro nastavení vlastností vjezdu a výjezdu ramene křižovatky.

- `Entry arm direction` – umístění ramene křižovatky, jehož provoz se nastavuje.
- `Mainroad` – hodnota určuje, zda je nastavované rameno hlavním ramenem křižovatky.
- `Entry arm ID` – identifikační číslo nastavovaného ramene.
- `Entry ID` – identifikační číslo vjezdu do nastavovaného ramene.
- `Select arm to set` – výběr ramene, kam se bude nastavovat provoz z nastavovaného ramene.
- `Exit arm ID` – identifikační číslo vybraného ramene.
- `Exit ID` – identifikační číslo výjezdu z vybraného ramene.
- `Propability` – pravděpodobnost, že vozidlo pojede z nastavovaného ramene do vybraného ramene.
- `Direction` – nastavení, po jaké straně vozovky vozidlo pojede.

Ve formuláři nastavení provozu uvnitř křižovatky lze nastavit, z jakého ramene může vozidlo pokračovat do dalšího ramene, s jakou pravděpodobností to provede a po jaké straně vozovky pojede.



Obrázek 37: Formulář pro nastavení vlastností provozu uvnitř křižovatky.

A.7 Pluginy s algoritmy dělení

Další funkcí programu MME je dělení mapy silničních sítí. Dělení se provádí pomocí pluginů, které obsahují implementaci konkrétních algoritmů dělení silniční sítě. V této kapitole bude popsáno přidání nového algoritmu dělení do programu a dále práce s již naimplementovanými algoritmy dělení.

Dostupné algoritmy dělení lze najít v menu programu pod záložkou `Dividing`. Po kliknutí na název algoritmu se objeví okno, ve kterém je uveden stručný popis algoritmu a pole pro zadání parametrů algoritmu.

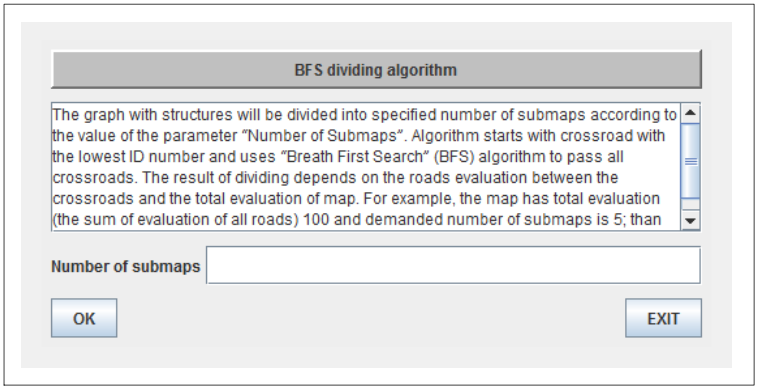
A.7.1 Přidání nového pluginu s algoritmem dělení

Plugin, který obsahuje algoritmus dělení, se musí nejprve nainstalovat v OSGi frameworku, aby byl vidět v programu MME. Po instalaci se plugin ve frameworku spustí a program MME již automaticky zajistí jeho zobrazení v menu programu.

Aby byl plugin viditelný v programu MME, musí mít přesnou strukturu algoritmu dělení. Více se o této struktuře lze dozvědět v kapitolách 7 a 5.2. Pokud nebude plugin splňovat požadavek na strukturu, nebude pracovat s programem MME správně.

A.7.2 Práce s pluginem obsahující BFS algoritmus dělení

Algoritmus dělení silniční sítě metodou BFS je v menu programu označen jako `BFS dividing algorithm`. Více o dělení mapy metodou BFS se lze dozvědět v kapitolách 3.1.3 a 7. Po kliknutí na název algoritmu se objeví formulář, který lze vidět na obr. 38.



Obrázek 38: Formulář BFS algoritmu dělení

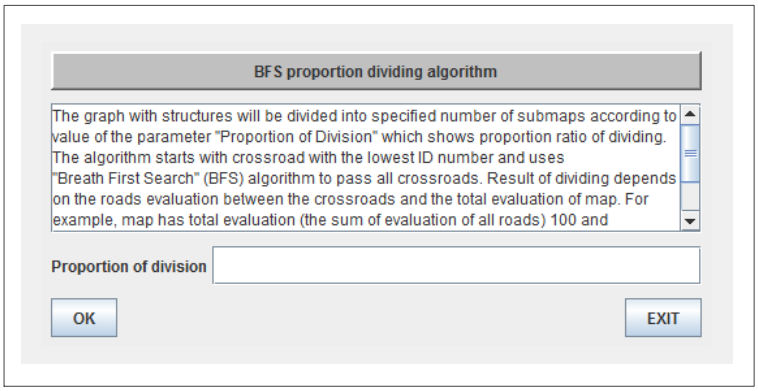
Ve formuláři je algoritmus popsán a obsahuje pole, kam se vyplňuje počet map, do kterých má být mapa rozdělena. Formulář provádí kontrolu, zda je zadaný vstup číslo.

Po vložení počtu map a stisknutí tlačítka OK je mapa rozdělena a do panelu se seznamem map jsou nové mapy vloženy. K vytvořeným mapám byly automaticky vygenerovány XML soubory popisující propojení nově vzniklých map. Nově vzniklé mapy sice byly vytvořeny, ale ještě nebyly uloženy ve formátu XML – pokud je uživatel s dělením spokojen, může mapy uložit.

A.7.3 Práce s pluginem obsahující proporcionální BFS algoritmus dělení

Algoritmus dělení silniční sítě proporcionální metodou BFS je v menu programu označen jako `BFS proportional dividing algorithm`. Více informací o tomto algoritmu dělení lze nalézt v kapitole 7.3. Po kliknutí na název algoritmu se objeví formulář, který lze vidět na obr. 39.

Formulář obsahuje popis algoritmu a pole pro vložení parametrů. Jako para-

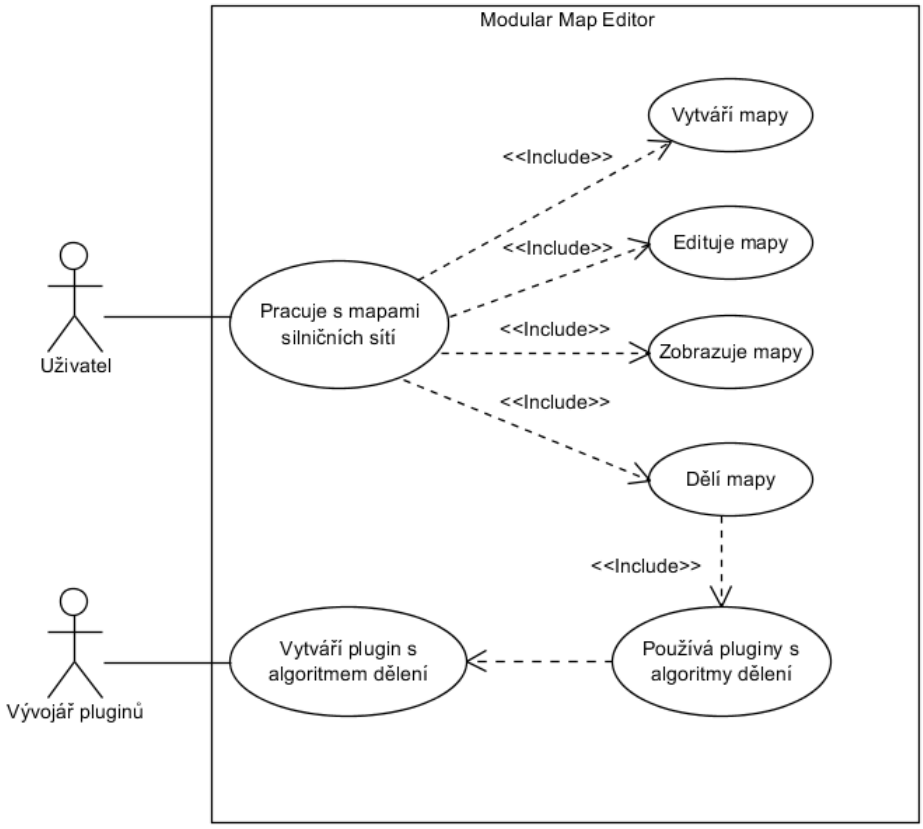


Obrázek 39: Formulář proporcionálního BFS algoritmu dělení

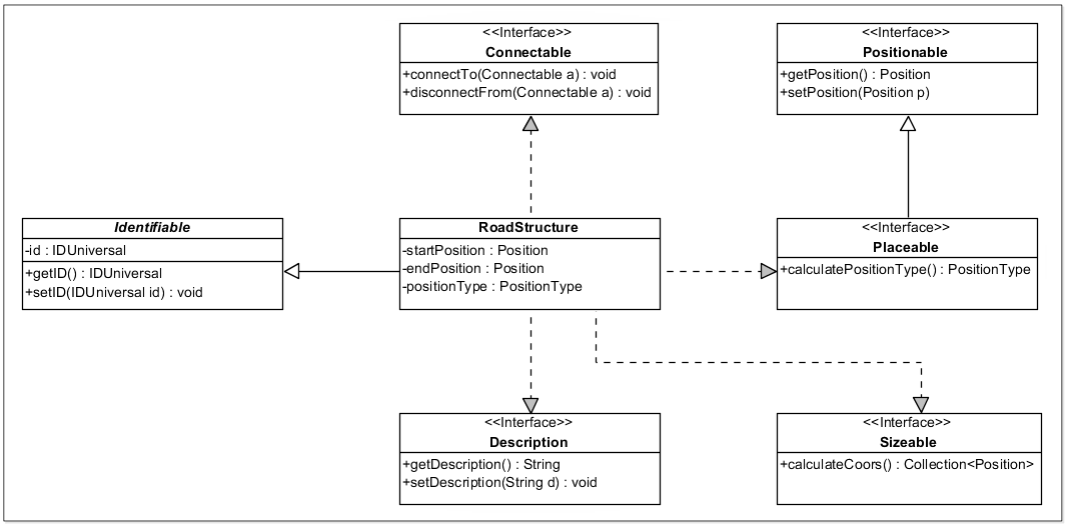
metr se do pole udává poměr, se kterým má být silniční síť rozdělena do více map. Poměr se zapisuje tak, že jednotlivé části jsou odděleny dvojtečkou. Například pro rozdělení mapy v poměru jedna ku dvěma ku pěti se poměr zapíše jako: „1:2:5“.

Po vložení poměru dělení a stisknutím tlačítka OK se mapa rozdělí v daném poměru. Program vytvoří nové mapy a zároveň s nimi vygeneruje XML soubory o propojení jako u předchozího algoritmu dělení. Pokud je uživatel s dělením spokojen, může mapy uložit.

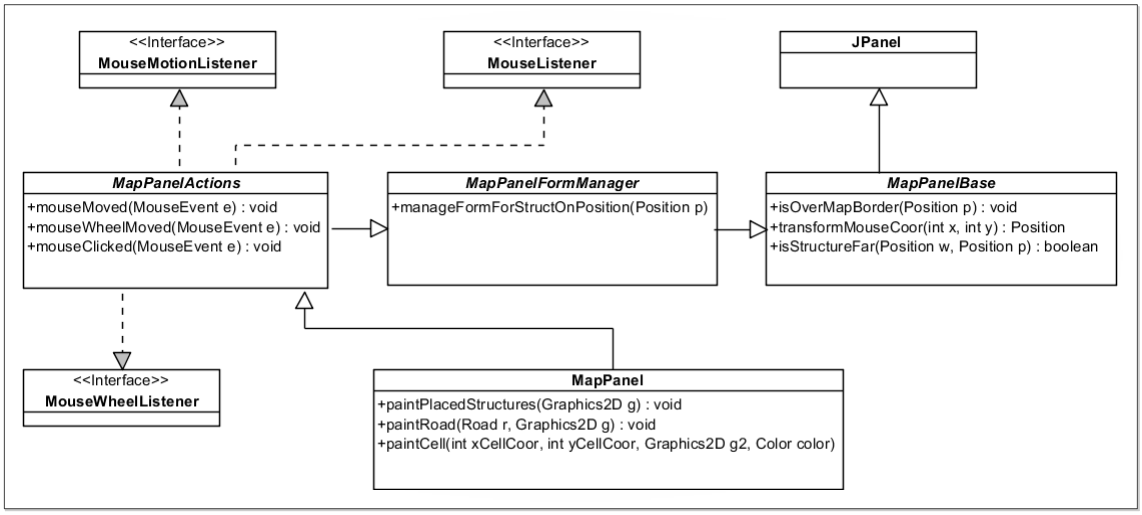
B UML diagramy



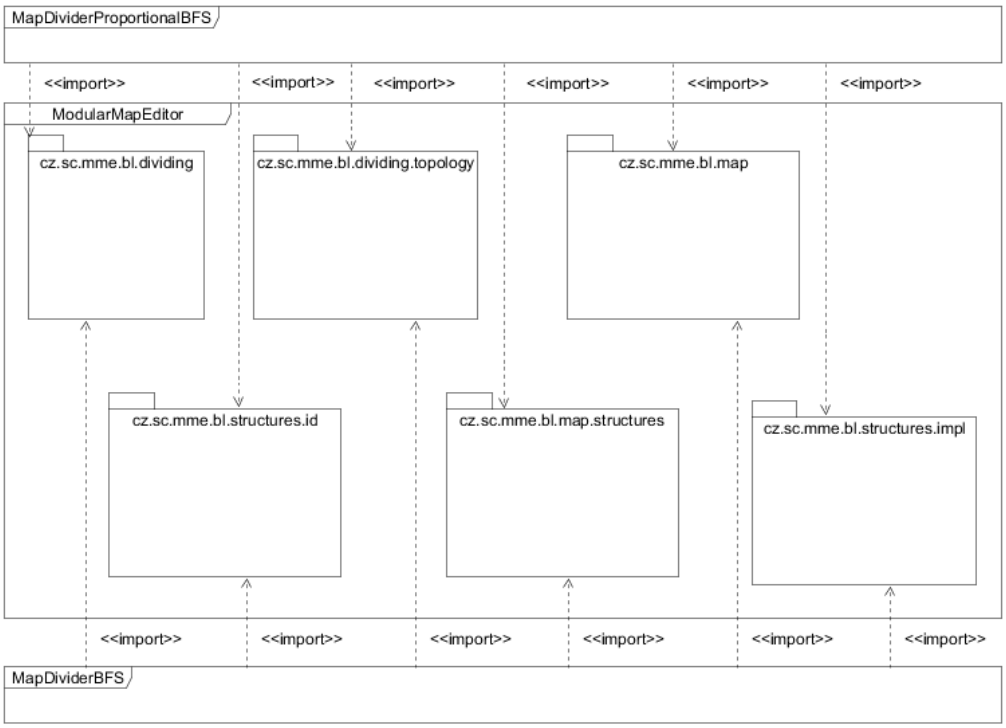
Obrázek 40: UML diagram případů užití.



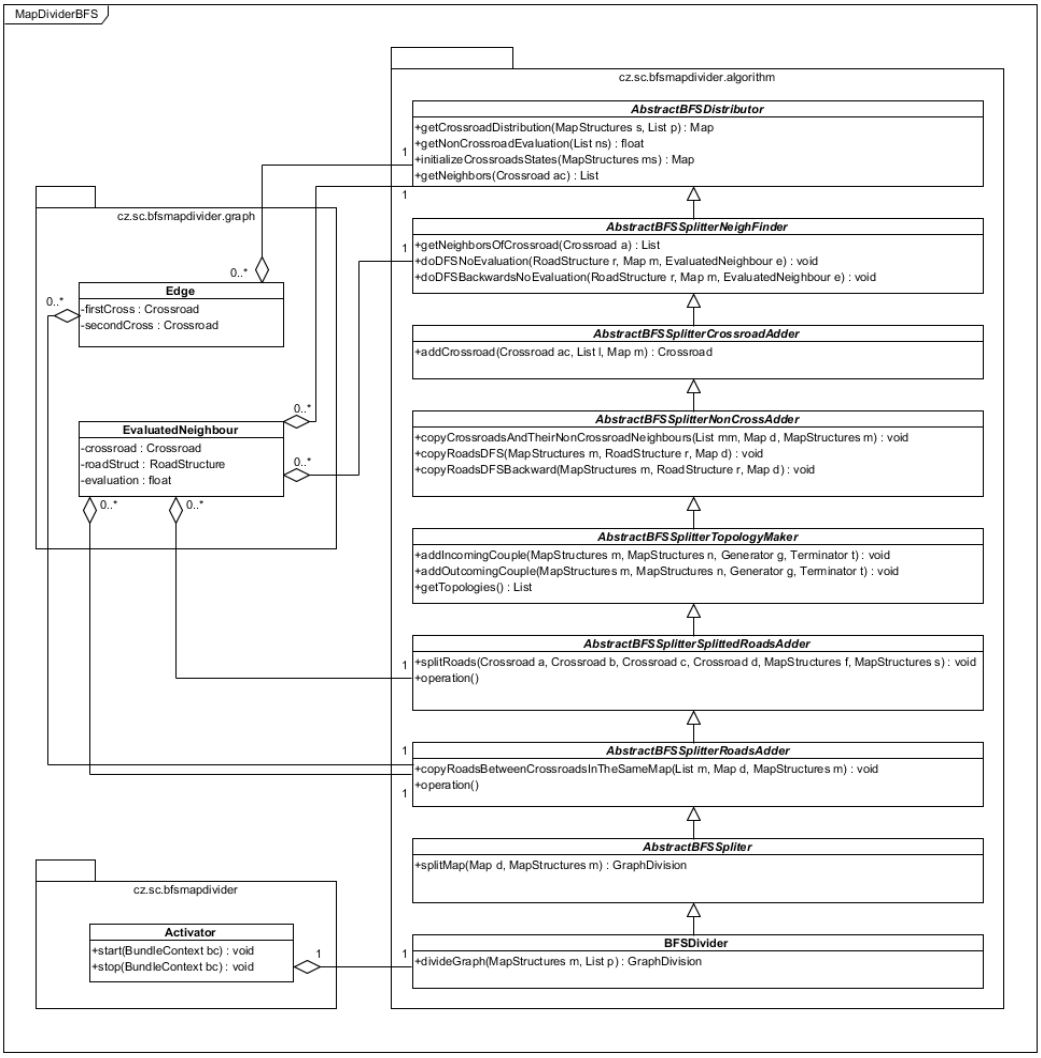
Obrázek 41: UML diagram implementace společného předka objektů mapy.



Obrázek 42: UML diagram implementace struktury map panelu.



Obrázek 43: UML diagram importu balíků nezbytných pro dělení mapy.



Obrázek 44: UML diagram dělícího modulu BFS algoritmu.

C Popis XML dokumentů

```
<?xml version="1.0" encoding="windows-1250"?>
<topology id="6x6S1"> <!-- mapa, jejíž spojení s ostatními mapami je popisováno -->
  <neighbours> <!-- seznam sousedů mapy, se kterými má 6x6S1 spojení -->
    <neighbour id="6x6S0"> <!-- spojení s mapou 6x6S0 -->
      <incomingPair terminatorId="T_6x6S0_24" generatorId="G_6x6S1_11"/> <!-- spojení z 6x6S0 do 6x6S1 -->
      <incomingPair terminatorId="T_6x6S0_28" generatorId="G_6x6S1_30"/> <!-- spojení z 6x6S0 do 6x6S1 -->
      <incomingPair terminatorId="T_6x6S0_29" generatorId="G_6x6S1_31"/> <!-- spojení z 6x6S0 do 6x6S1 -->
      <outcomingPair terminatorId="T_6x6S0_24" generatorId="G_6x6S1_11"/> <!-- spojení do 6x6S0 z 6x6S1 -->
      <outcomingPair terminatorId="T_6x6S0_25" generatorId="G_6x6S1_12"/> <!-- spojení do 6x6S0 z 6x6S1 -->
      <outcomingPair terminatorId="T_6x6S0_26" generatorId="G_6x6S1_13"/> <!-- spojení do 6x6S0 z 6x6S1 -->
    </neighbour>
    <neighbour id="6x6S2"> <!-- spojení s mapou 6x6S2 -->
      <incomingPair terminatorId="T_6x6S2_33" generatorId="G_6x6S1_40"/> <!-- spojení z 6x6S2 do 6x6S1 -->
      <incomingPair terminatorId="T_6x6S2_34" generatorId="G_6x6S1_41"/> <!-- spojení z 6x6S2 do 6x6S1 -->
      <outcomingPair terminatorId="T_6x6S2_25" generatorId="G_6x6S1_14"/> <!-- spojení do 6x6S2 z 6x6S1 -->
      <outcomingPair terminatorId="T_6x6S2_26" generatorId="G_6x6S1_15"/> <!-- spojení do 6x6S2 z 6x6S1 -->
    </neighbour>
  </neighbours>
</topology>
```

Obrázek 45: Příklad XML souboru s topologií.

```xml
<?xml version="1.0" encoding="windows-1250"?> <!-- objekt mapy -->
<map xmlns:xs="http://www.w3.org/2001/XMLSchema-instance" id="ukazkova" description="Description" width="100" height="50">
  <roads> <!-- seznam jízdních pruhů -->
    <road id="R_ukazkova_0" description="Description" evaluation="1.0" positionX="46" positionY="14" position_endX="46" position_endY="21">
      <accessors> <!-- seznam objektů, které jsou na objekt připojené -->
        <G_ukazkova_0/>
      </accessors>
      <attachers> <!-- seznam objektů, kam je objekt připojený -->
        <E_ukazkova_0_0_0/>
      </attachers>
    </road>
    <road id="R_ukazkova_1" description="Description" evaluation="1.0" positionX="47" positionY="21" position_endX="47" position_endY="14">
      <accessors> <!-- seznam objektů, které jsou na objekt připojené -->
        <X_ukazkova_0_0_1/>
      </accessors>
      <attachers> <!-- seznam objektů, kam je objekt připojený -->
        <T_ukazkova_0/>
      </attachers>
    </road>
  </roads>
  <generators> <!-- seznam generátorů -->
    <generator id="G_ukazkova_0" description="Description" positionX="46" positionY="8" position_endX="46" position_endY="13" lambda="1.0">
      <accessors> <!-- seznam objektů, které jsou na objekt připojené -->
      </accessors>
      <attachers> <!-- seznam objektů, kam je objekt připojený -->
        <R_ukazkova_0/>
      </attachers>
    </generator>
  </generators>
  <terminators> <!-- seznam terminátorů -->
    <terminator id="T_ukazkova_0" description="Description" positionX="47" positionY="8" position_endX="47" position_endY="13">
      <accessors> <!-- seznam objektů, které jsou na objekt připojené -->
        <R_ukazkova_1/>
      </accessors>
      <attachers> <!-- seznam objektů, kam je objekt připojený -->
      </attachers>
    </terminator>
  </terminators>
  <crossroads> <!-- seznam křižovatek -->
    <crossroad id="C_ukazkova_0" description="Description" positionX="45" positionY="22" width="4" height="4">
      <arms> <!-- seznam ramen křižovatky -->
        <arm id="A_ukazkova_0_0" description="Description" position="NORTH" main_road="true">
          <entries> <!-- seznam vjezdů do ramene křižovatky -->
            <entry id="E_ukazkova_0_0_0" description="Description" positionX="46" positionY="22">
              <accessors> <!-- seznam objektů, které jsou na objekt připojené -->
                <R_ukazkova_0/>
              </accessors>
              <attachers> <!-- seznam objektů, kam je objekt připojený -->
              </attachers>
            </entry>
          </entries>
          <exits> <!-- seznam výjezdů z ramene křižovatky -->
            <exit id="X_ukazkova_0_0_1" description="Description" positionX="47" positionY="22">
              <accessors> <!-- seznam objektů, které jsou na objekt připojené -->
              </accessors>
              <attachers> <!-- seznam objektů, kam je objekt připojený -->
                <R_ukazkova_1/>
              </attachers>
            </exit>
          </exits>
        </arm>
        <arm id="A_ukazkova_0_1" description="Description" position="EAST" main_road="false">
          <entries> <!-- seznam vjezdů do ramene křižovatky -->
            <entry id="E_ukazkova_0_1_0" description="Description" positionX="48" positionY="23">
              <accessors> <!-- seznam objektů, které jsou na objekt připojené -->
              </accessors>
              <attachers> <!-- seznam objektů, kam je objekt připojený -->
              </attachers>
            </entry>
          </entries>
          <exits> <!-- seznam výjezdů z ramene křižovatky -->
            <exit id="X_ukazkova_0_1_1" description="Description" positionX="48" positionY="24">
              <accessors> <!-- seznam objektů, které jsou na objekt připojené -->
              </accessors>
```

Obrázek 46: Příklad XML mapy, 1. část.

```
            <attachers> <!-- seznam objektů, kam je objekt připojený -->
            </attachers>
          </exit>
        </exits>
      </arm>
      <arm id="A_ukazkova_0_2" description="Description" position="SOUTH" main_road="false">
        <entries> <!-- seznam vjezdů do ramene křižovatky -->
          <entry id="E_ukazkova_0_2_0" description="Description" positionX="47" positionY="25">
            <accessors> <!-- seznam objektů, které jsou na objekt připojené -->
            </accessors>
            <attachers> <!-- seznam objektů, kam je objekt připojený -->
            </attachers>
          </entry>
        </entries>
        <exits> <!-- seznam výjezdů z ramene křižovatky -->
          <exit id="X_ukazkova_0_2_1" description="Description" positionX="46" positionY="25">
            <accessors> <!-- seznam objektů, které jsou na objekt připojené -->
            </accessors>
            <attachers> <!-- seznam objektů, kam je objekt připojený -->
            </attachers>
          </exit>
        </exits>
      </arm>
      <arm id="A_ukazkova_0_3" description="Description" position="WEST" main_road="false">
        <entries> <!-- seznam vjezdů do ramene křižovatky -->
          <entry id="E_ukazkova_0_3_0" description="Description" positionX="45" positionY="24">
            <accessors> <!-- seznam objektů, které jsou na objekt připojené -->
            </accessors>
            <attachers> <!-- seznam objektů, kam je objekt připojený -->
            </attachers>
          </entry>
        </entries>
        <exits> <!-- seznam výjezdů z ramene křižovatky -->
          <exit id="X_ukazkova_0_3_1" description="Description" positionX="45" positionY="23">
            <accessors> <!-- seznam objektů, které jsou na objekt připojené -->
            </accessors>
            <attachers> <!-- seznam objektů, kam je objekt připojený -->
            </attachers>
          </exit>
        </exits>
      </arm>
    </arms>
    <paths_from>  <!-- vnitřní provoz v křižovatce -->
      <path_from arm="A_ukazkova_0_0" entry="E_ukazkova_0_0_0">  <!-- z jakého ramene a vjezdu -->
        <paths_to> <!-- do jakého ramene, vjezdu, po jaké straně a s jakou pravděpodobností -->
          <path_to arm="A_ukazkova_0_1" exit="X_ukazkova_0_1_1" travel_direction="LEFT" probability="1.0"/>
          <path_to arm="A_ukazkova_0_2" exit="X_ukazkova_0_2_1" travel_direction="CENTER" probability="1.0"/>
          <path_to arm="A_ukazkova_0_3" exit="X_ukazkova_0_3_1" travel_direction="RIGHT" probability="1.0"/>
        </paths_to>
      </path_from>
      <path_from arm="A_ukazkova_0_1" entry="E_ukazkova_0_1_0"> <!-- z jakého ramene a vjezdu -->
        <paths_to> <!-- do jakého ramene, vjezdu, po jaké straně a s jakou pravděpodobností -->
          <path_to arm="A_ukazkova_0_0" exit="X_ukazkova_0_0_1" travel_direction="RIGHT" probability="1.0"/>
          <path_to arm="A_ukazkova_0_2" exit="X_ukazkova_0_2_1" travel_direction="LEFT" probability="1.0"/>
          <path_to arm="A_ukazkova_0_3" exit="X_ukazkova_0_3_1" travel_direction="CENTER" probability="1.0"/>
        </paths_to>
      </path_from>
      <path_from arm="A_ukazkova_0_2" entry="E_ukazkova_0_2_0"> <!-- z jakého ramene a vjezdu -->
        <paths_to> <!-- do jakého ramene, vjezdu, po jaké straně a s jakou pravděpodobností -->
          <path_to arm="A_ukazkova_0_0" exit="X_ukazkova_0_0_1" travel_direction="CENTER" probability="1.0"/>
          <path_to arm="A_ukazkova_0_1" exit="X_ukazkova_0_1_1" travel_direction="RIGHT" probability="1.0"/>
          <path_to arm="A_ukazkova_0_3" exit="X_ukazkova_0_3_1" travel_direction="LEFT" probability="1.0"/>
        </paths_to>
      </path_from>
      <path_from arm="A_ukazkova_0_3" entry="E_ukazkova_0_3_0"> <!-- z jakého ramene a vjezdu -->
        <paths_to> <!-- do jakého ramene, vjezdu, po jaké straně a s jakou pravděpodobností -->
          <path_to arm="A_ukazkova_0_0" exit="X_ukazkova_0_0_1" travel_direction="LEFT" probability="1.0"/>
          <path_to arm="A_ukazkova_0_1" exit="X_ukazkova_0_1_1" travel_direction="CENTER" probability="1.0"/>
          <path_to arm="A_ukazkova_0_2" exit="X_ukazkova_0_2_1" travel_direction="RIGHT" probability="1.0"/>
        </paths_to>
      </path_from>
    </paths_from>
  </crossroad>
 </crossroads>
</map>
```

Obrázek 47: Příklad XML mapy, 2. část.