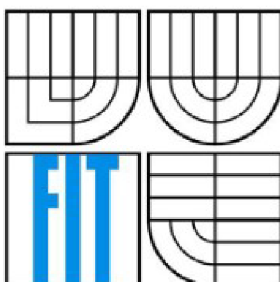
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEME

UNIVERZÁLNÍ GRAFICKÝ EDITOR JAKO KNIHOVNA A MODUL PRO PYTHON

UNIVERSAL GRAPH EDITOR AS A LIBRARY AND PYTHON MODULE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — BC. JAROSLAV KOŠULIČ
AUTHOR

VEDOUCÍ PRÁCE — ING. ALEŠ SMRČKA
SUPERVISOR

BRNO 2008

Univerzální grafický editor jako knihovna a modul pro Python

Odevzdáno na Fakultě informačních technologií Vysokého učení technického v Brně
dne 19. května 2008

© Jaroslav Košulič, 2008

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Prohlášení

Prohlašuji, že jsem tento semestrální projekt vypracoval samostatně pod vedením Aleše Smrčky. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . .
Jaroslav Košulič
19. května 2008

Abstrakt

Častým prostředkem pro přehledné zobrazení informací jsou různá schemata, diagramy, souhrně grafy. Používají se často například ve výuce pro prezentaci některých algoritmů a velké využití mají i v technické praxi v podobě UML diagramů, grafů topologie sítě a graf můžeme dokonce pozorovat v elektrotechnických schematech a návrhu HW architektury. Před dvěmi lety proto začal vznikat projekt Univerzálního grafického editoru. Jeho cílem je zaplnit mezeru na trhu poskytnutím dostupného univerzálního nástroje pro práci s grafy. Na tehdejší práci navázal semestrální projekt, jehož dalším pokračováním je tato diplomová práce. Cílem je navrhnout, implementovat, zdokumentovat a otestovat modul pro jazyk Python poskytující objektové rozhraní pro práci s grafy s využitím knihovny s rozhraním jazyka C vytvořené v rámci semestrálního projektu.

Klíčová slova

Univerzální grafický editor, UGE, knihovna, jazyk C, Python, graf, uzel, hrana, vektorová grafika

Košulič, Jaroslav: *Univerzální grafický editor jako knihovna a modul pro Python*. Diplomová práce, Brno, FIT VUT v Brně, 2008.

Abstract

The diagrams, schemes, and graphs in general are widely used in the field of easy-to-read information visualisation. We use them for example in the school lessons for an algorithm presentation, or in the technical jobs such as software and hardware development by modelling UML diagrams, database schemes, etc. The project Universal Graph Editor has been established two years ago to fill the gap with the software tool providing such a modelling engine. The previous work has been reasumed in semestral project by design of the dynamic graph drawing (or the drawing of a vector graphic in general) and the library for graph manipulation with C-language interface. This master thesis continues further by creating a Python module using the developed interface. The documentation and the testing phase is conluding the annual work.

Keywords

Universal Graphic Editor, UGE, library, graph-drawing library, graph, node, vertex, edge, vector graphic, C-language, Python

Obsah

Obsah 4

Seznam obrázků 6

1 Úvod 7

2 Univerzální grafický editor jako knihovna s jednoduchým rozhraním jazyka C 10
2.1 Analýza požadavků 10
2.2 Vysoce modulární architektura 11
2.2.1 Jádro knihovny UGE 11
2.2.2 Kreslení základních grafických primitiv 11
2.2.3 Práce s grafem 13
2.2.4 Vizualizace 13
2.2.5 Ukládání a načítání ze souboru 14
2.3 Použití knihovny 14

3 Současný stav projektu Univerzální grafický editor 16
3.1 Původní návrh architektury systému UGE 16
3.2 Stav implementace systému UGE 17
3.3 Možnosti využití stávajících zdrojových kódů 18

4 Univerzální grafický editor jako modul pro Python 19
4.1 Požadavky na modul 19
4.1.1 Využití knihovny UGE v modulu pro Python 19
4.1.2 Objektově orientovaný návrh 20
4.1.3 Jednoduchá instalace a použití 20
4.1.4 Snadná rozšiřitelnost 21
4.1.5 Kontrola konzistence grafu 21
4.2 Návrh architektury modulu 22
4.2.1 Práce s grafem 22
4.2.2 Grafické objekty 22
4.2.3 Vizualizace 25
4.2.4 Komunikace komponent 27

5 Implementace 30
5.1 UGE jako knihovna s rozhraním jazyka C 30
5.1.1 Aplikace principů OOP v neobjektovém jazyce 31
5.1.2 Použité algoritmy počítačové grafiky 31

5.2 UGE jako modul pro Python . . . 33

6 Testování **35**

7 Závěr **37**

Literatura **39**

A Dokumentace **41**

B Tutoriál **42**
B.1 Instalace . . . 42
B.2 Vytvoření rozšiřujícího modulu – práce s Petriho sítěmi . . . 43
B.3 Použití modulu UGE v jednoduché aplikaci s GUI . . . 45

C Obsah přiloženého CD **49**

Seznam obrázků

2.1 Architektura knihovny UGE 12
2.2 Jádro a základní moduly knihovny 13
2.3 Kontrola vytvářených hran 14
2.4 Ukázka použití knihovny UGE 15

3.1 Architektura systému UGE (převzato z [6]) 16
3.2 Uživatelské rozhraní stávající aplikace 17

4.1 Ukázka možného použití modulu UGE 20
4.2 Ukázka rozšíření modulu UGE 21
4.3 Diagram tříd objektů grafu 23
4.4 Diagram tříd grafických objektů 24
4.5 Vazba mezi semantickou a vizuální stránkou grafu 25
4.6 Diagram tříd sloužících pro vizualizaci 26
4.7 Vytváření nové hrany 27
4.8 Změna polohy grafického objektu reprezentujícího uzel 28
4.9 Odstranění uzlu z grafu 29

5.1 Struktura souborů implementace knihovny 31
5.2 Příklad vytváření hierarchie dědičnosti pomocí struktur v jazyce C 32
5.3 Uživatelské rozhraní aplikace vytvářené v tutoriálu (nahoře) a demonstrační aplikace semestrálního projektu (dole) 34

6.1 Chybný výsledek testování výpočtu polohy portu 35
6.2 Konfigurace použitá pro testování, jestli bod leží uvnitř objektu 36

Kapitola 1

Úvod

Během života se často setkáváme s nutnostní utřídit a následně přehledně zobrazit množství informací nebo například zachytit činnost nějakého algoritmu a podobně. Často je vhodné použít k tomu právě některý typ grafu. Spadají sem tedy všechny typy UML diagramů, grafická reprezentace konečných automatů a Petriho sítí nebo dokonce mapy (GPS) a schemata hardware architektury a mnohé další. Vidíme, že jenom vyjmenování všech oblastí, kde se s grafy setkáváme, by vydalo na samostatnou kapitolu. S trochou nadsázky můžeme tedy říci, že grafy jsou všude kolem nás.

Nabízí se ale otázka, jestli není tato práce zbytečná. Pro tak důležitou oblast, jako je práce s různými grafy, přece již určitě existuje řada knihoven a nástrojů. Na některé z nich se proto podíváme v následujících několika odstavcích spolu s důvody, proč má smysl vymýšlet a vyvíjet novou knihovnu pro práci s grafy (a vektorovou grafikou obecně).

První kategorií projektů jsou více či méně obecné knihovny pro práci s grafy. Zde dominuje The Boost Graph Library [12] – rozsáhlá knihovna grafových struktur a algoritmů v jazyce C++. Vše je zde implementováno formou šablon v duchu STL (Standard Template Library). BGL vůbec neřeší vizualizaci grafu, omezuje se zde na vstup a výstup do formátu programu GraphViz, o kterém se zmíním dále. Jako samostatný projekt existuje dokonce rozhraní BGL pro jazyk Python, v současné době ale neprobíhá další vývoj. Další zajímavou knihovnou je Goblin. Opět nabízí aplikační rozhraní pro manipulaci s grafy v jazyce C++, kromě toho ale také Tcl/Tk, grafické uživatelské prostředí pro manipulaci s grafy implementované v Tcl/Tk a některé algoritmy pro automatické rozvržení grafu.

Další důležitou oblast práce s grafy představují aplikace a nástroje pro automatické rozvržení grafů [7]. Vyspělým produktem s dlouholetou historií je bezesporu GraphViz. Tento opensource nástroj generuje vizuální reprezentaci grafu podle popisu v textovém souboru s poměrně jednoduchou syntaxí. Kromě toho existuje řada souvisejících projektů, které poskytují uživatelsky přijatelnější rozhraní nebo přinášejí možnosti GraphViz do jiných programovacích jazyků (např. Grappa, ZGRViewer).

Nemůžu vynechat rovněž komerční nástroje zabývající se problematikou rozvržení grafu: aiSee, yFiles a algoritmy automatického rozvržení obsahuje i prostředí Mathematica 6. První jmenovaný nástroj čte podobně jako GraphViz popis grafu z textového souboru a na jeho základě generuje rozvržený graf, který je poté možno interaktivně zkoumat a exportovat do různých obrazových formátů. Naproti tomu yFiles je knihovna v jazyce Java vyvíjená firmou yWorks. Kromě původní implementace je k dispozici i yFiles.NET, yFiles WPF (Windows Presentation Foundation) a yFiles pro tzv. Web 2.0 - yFiles FLEX a yFiles AJAX. Zdarma je dostupná aplikace yEd, která využívá yFiles Java, umožňuje vytvářet grafy, aplikovat automatické rozvržení, načítat a ukládat grafy ve formátu GraphML [3] a export do obrazových formátů. Knihovnu yFiles využívá také například DBVisualizer – komerční nástroj pro vizualizaci relačních databází. Největší nevýhodou těchto

velice kvalitních komerčních produktů je nedostupnost zdrojových kódů a jejich poměrně vysoká cena znemožňující jejich použití v nekomerčních aplikacích (v případě aiSee řádově stovky Euro, pro použití yFiles ve vlastních programech dokonce tisíce Euro).

Následuje skupina aplikací pro kreslení a manipulaci s více či méně vymezenými druhy grafů, hlavně pak UML diagramů. Opensource zástupci těchto programů jsou například Dia, ArgoUML nebo StarUML. Dia je jednoduchá aplikace s grafickým uživatelským rozhraním využívajícím GTK, sloužíví pro kreslení nejrůznějších schemat, vývojových a UML diagramů, s možností exportu do grafických formátů a s pomocí dalších nástrojů také například generování zdrojových kódů podle UML diagramu. ArgoUML je kvalitní program podporující všechny typy diagramů definované v UML 1.4 (verze uznaná jako ISO standard). Implmentačním jazykem je v tomto případě Java, což umožňuje běh na mnoha platformách a operačních systémech. StarUML si klade za cíl poskytnout alternativu svými možnostmi dosahující na komerční Rational Rose, na rozdíl od ostatních jmenovaných nástrojů je však dostupný pouze pro operační systémy Microsoft Windows. A to stále není všechno, v Pythonu a GTK implementovaný Gaphor, Umbrello UML pro desktopové prostředí KDE nebo Pyprus UML využívající platformu Eclipse RCP, to jsou jen některé další opensource nástroje pro práci s UML diagramy.

Mezi komerčními programy nalezneme rovněž mnoho nástrojů pro práci s obecnějšími schematy nebo přímo UML diagramy. Za všechny vyjmenuji například Visual Paradigm for UML nebo Poseidon for UML firmy GentleWare, což je nadstavbu nad ArgoUML. Vynechat nesmím ani snad už legendární Rational Rose a vytváření UML diagramů je často možné i přímo v integrovaných vývojových prostředích jako MS Visual Studio. Pro kreslení nejrůznějších obchodních diagramů a schemat pak slouží například SmartDraw.

E-R diagramy, ač nejsou součástí specifikace UML, jsou jistě rovněž velmi důležitou pomůckou při návrhu software a opět se jedná o grafy, se kterými je třeba manipulovat, udržovat je synchronizované s aktuálním schematem databáze, kterou modelují, a u větších databází je hodně užitečná možnost vygenerování E-R diagramu z metadat databáze. A přesně to umí například DBVisualizer – komerční nástroj s volně dostupnou verzí pro soukromé využití implementovaný v jazyce Java a využívající zmiňovanou knihovnu yFiles. Pro vizuální návrh databáze můžeme také použít MySQL Workbench nebo Toad Data Modeler a řadu dalších.

Zajímavým a užitečným typem grafů jsou tzv. myšlenkové mapy. Velmi populárním opensource nástrojem pro tvorby myšlenkových map je FreeMind. Z komerčních nástrojů například ConceptDraw, MindManager nebo MindMeister, který je spouštěn online ve webovém prohlížeči a umožňuje snadnou vzdálenou spolupráci více uživatelů nad jenou mapou.

V praxi při řízení projektů velmi často používaným grafem jsou Ganttovy diagramy. Pro práci s nimi může posloužit opensource program Planner nebo komerční MS Project či už jednou vzpomínaný SmartDraw.

Za další možný prostředek pro tvorbu schemat můžeme považovat obyčejné vektorové grafické editory jako je Inkscape nebo OpenOffice.org Draw. Tyto aplikace umožňují kreslit tzv. „konektory“ představující hrany grafu a následující při přesunu grafický objekt představující uzel grafu. V situacích, kdy nepotřebujeme různé vymoženosti týkající se práce se semantickou stránkou grafu, mohou být právě tyto programy zajímavou volbou pro rychlé vytvoření srozumitelných diagramů bez nutnosti učit se s novým specializovaným nástrojem.

Na grafy v různých podobách ale narazíme i při práci na projektech a úkolech, jejichž výstup samotný s grafem nemá vůbec nic společného. Příkladem jsou multimediální aplikace a frameworky. Grafy jsou tak využívány při vývoji s pomocí Microsoft DirectShow API – uzly představují obrazové (a jiné) filtry, hrany spojují výstupy a vstupy filtrů a představují tok dat mezi jednotlivými filtry. Podobně funguje opensource multimediální framework GStreamer a mladá knihovna GEGL (Generic Graphics Library). V aplikaci Terragen sloužící pro generování virtuální krajiny se rovněž

využívá tzv. graf scény, uzly představují různé operace jako například generování výškové mapy, shadery, osvětlení a hrany spojují vstupy a výstupy těchto operací.

Z přechozích odstavců by se mohlo zdát, že trh je kvalitními nástroji přímo přesycen a není třeba nic nového vyvíjet. To však není pravda, existuje sice řada kvalitních nástrojů pro konkrétní typy diagramů, ale vývoj žádného takového nástroje není přímým cílem tohoto projektu. Místo toho usiluje o poskytnutí jednoduchého a přitom univerzálního rozhraní pro práci s grafy, které by pak bylo využitelné v dalších projektech a usnadnilo a urychlilo jejich vývoj. Například tak bude snadné vyvinout nástroj pro vizuální programování složitých multimediálních filtrů frameworku GStreamer nebo GEGL – tedy podobný nástroj, jako pro MS DirectShow představuje komerční GraphEdit. Nebo nový nástroj pro vizualizaci, modelování a hlavně simulaci Petriho sítí. V současné době můžeme buď koupit některou nákladnou komerční knihovnu, nebo vyvíjet vše od začátku. Pokud bude volně dostupná knihovna pro práci s grafy, můžeme se soustředit jenom na to podstatné - vlastní simulaci systému popsaného Petriho sítí. A tak bych mohl pokračovat dalšími typy aplikací a proto věřím ve velkou užitečnost vývoje Univerzálního grafického editoru.

Projekt Univerzální grafický editor (zkráceně UGE) vzniknul na Fakultě informačních technologií před dvěma lety jako několik souvisejících bakalářských prací. Tehdy byl položen základ modulární aplikace (viz [6]), jejíž aktuální stav je popsán ve třetí kapitole. Aplikace UGE nebyla dokončena a to je jeden z impulsů pro vytvoření knihovny UGE a modulu pro Python. Jazyk C pro rozhraní knihovny a dříve rovněž pro komunikační rozhraní jádra aplikace UGE a rozšiřujících modulů byl zvolen pro svou stabilitu, vyspělost a dostupnost na mnoha platformách, de-facto jde o průmyslový standard. Relativní nevýhodou programování v jazyce C je ale daleko vyšší pracnost ve srovnání s objektově orientovanými a nebo dynamickými jazyky. A Python je právě takovým dynamickým jazykem, je vhodný pro rychlé prototypování a rapidní vývoj aplikací s uživatelským rozhraním. Navíc je použitelný i v prostředí internetu a může tak v budoucnu umožnit vývoj online verze univerzálního grafického editoru (podobně jako dříve zmiňovaná knihovna yFiles pro Web 2.0). Proto se právě Python nabízí jako vhodný jazyk pro vytvoření rozhraní knihovny UGE.

Tato diplomová práce přímo navazuje na stejnojmenný semestrální projekt, během kterého byla navržena a implementována knihovna s rozhraním v jazyce C. Tato knihovna je dále vylepšována a upravována v souvislosti s návrhem modulu pro jazyk Python. Tento modul bude plně využívat možnosti objektově orientovaného programování v jazyce Python a poskytne podporu pro rychlý vývoj aplikací pracujících s grafy nebo jednoduchou vektorovou grafikou.

Následující kapitola rozebírá požadavky na knihovnu jazyka C a návrh architektury knihovny, která má být podobně jako původní projekt velice modulární. Následuje popis stavu projektu UGE s rozborem možností využití existujících zdrojových kódů ve vyvíjené knihovně. Tyto dvě kapitoly jsou s minimálními úpravami převzaty ze semestrálního projektu. Následující čtvrtá kapitola se zabývá analýzou požadavků a objektovým návrhem modulu pro jazyk Python. V páté kapitole je stručně popsána implementace obou částí projektu (informace týkající se knihovny jazyka C jsou opět převzaty ze semestrálního projektu). Na konci práce je popsán způsob testování implementace modulu a v závěru shrnuty dosažené výsledky a naznačeny možnosti dalšího využití projektu. V přílohách jsou uvedeny tutoriály vysvětlující použití modulu.

Kapitola 2

Univerzální grafický editor jako knihovna s jednoduchým rozhraním jazyka C

V této kapitole se podíváme na návrh knihovny pro manipulaci s grafem v jazyce C. Kapitola začíná zkoumáním požadavků na knihovnu pro práci s grafem a jednoduchou vektorovou grafikou a pokračuje konkrétním návrhem modulární architektury knihovny.

2.1 Analýza požadavků

Cílem projektu není vytvořit obrovskou všehoschopnou knihovnu, která se bude snažit pokrýt každý možný aspekt práce s grafem. Cílem je naopak poskytnout jednoduše použitelný a hlavně otevřený systém. Musí být možné snadno vytvářet rozšiřující moduly poskytující různé typy grafových elementů (typy uzlů – např. HW komponenty) nebo grafických primitiv, stejně tak jako moduly pro import a export vytvořeného grafu.

Klíčový význam má také nezávislost na dalších knihovnách pro zachování maximální možné přenositelnosti. Pro výstup na obrazovku se nabízí zvolit jeden GUI toolkit a ten potom vyžadovat, což ale odporuje požadavku. Proto bude lepší tuto část knihovny řešit jako samostatnou komponentu s jednoduchým komunikačním rozhraním a jádro budovat důsledně nezávisle.

S předchozím požadavkem souvisí i možnost použít knihovnu úplně bez GUI, jenom pro práci s matematickým modelem grafu. Pro připomenutí, z matematického hlediska je graf dvojice $G(V,E)$, kde V je množina uzlů grafu (angl. *vertices*, *nodes*), E potom množina hran (angl. *edges*). Knihovna pro práci s grafem musí umět pracovat s neorientovanými i orientovanými grafy (v tomto případě jsou hrany uspořádanými dvojicemi - mají „směr“).

Na druhou stranu, jak vyplývá ze samotného názvu projektu – univerzální *grafický* editor – nebude návrh omezen jen pro práci s grafy, ale umožní i práci s obecnou vektorovou grafikou. Zde se však omezí na jednoduché grafické objekty (primitiva – čára, křivka, polygon, obdélník, elipsa, kružnice) s možností sdružovat tyto jednoduché objekty do skupin a nebo implementovat složitější objekty v rámci rozšiřujícího modulu.

Cílem tedy bude vyvinout projekt umožňující pracovat s grafy bez důrazu na jejich grafickou reprezentaci, vektorovou grafikou bez další semantiky, a nebo obojím dohromady – každý prvek grafu má i svůj vzhled, je reprezentován nějakým grafickým primitivem. Zároveň bude knihovna poskytovat prostředky pro kontrolu sémantiky grafu – například u Petriho sítí to znamená, že není možné spojit hranou dvě místa nebo dva přechody.

2.2 Vysoce modulární architektura

Celková architektura komponent, které budou tvořit knihovnu UGE, ja znázorněna na obrázcích 2.1 a 2.2. Je zde naznačena i role zásuvných modulů pro práci s různymi typy grafů (konečné automaty, Petriho sítě. . .) a dále pro vizualizaci grafu (buď na obrazovku nebo do souboru). Ve spodní části obrázku 2.1 je znázorněna role komponenty pro vstup a výstup, která se jako jediná musí starat o semantiku i vzhled zároveň. Podobný přístup se může objevit i v některcýh zásuvných modulech (například návrh HW architektury, kde každá komponenta má kromě funkce dánu i grafickou reprezentaci ve schematu).

2.2.1 Jádro knihovny UGE

Všechny uvedené požadavky vedou k tomu navrhnout samotné jádro systému co nejmenší a nejjednodušší. Z toho důvodu bude poskytovat pouze základní rozhraní pro práci s grafickými primitivy a definuje funkce a informace, které musí poskytovat každé primitivum:

- Informace o bodu, kde je možné k hranici objektu připojit čáru (tzv. port) vedoucí ze směru daného jiným bodem
- Ohraničující obdélníková oblast, do které je grafické primitivum vepsáno – ta je určena souřadnicemi levého horního rohu a dále svojí šířkou a výškou
- Odpověď na otázku, jestli bod o zadaných souřadnicích leží uvnitř grafického primitiva
- Přesun a změna velikosti
- Vykreslení pomocí volání základních funkcí poskytovaných vrstvou pro vizualizaci (např. křivka se kreslí jako mnoho krátkých úseček)

Jádro samotné se vůbec nezabývá konkrétními grafickými primitivy. Netuší, jestli se jedná o čáru, křivku, kružnici nebo obdélník. . . a ani semantikou kresby jako grafu. Poskytuje jenom rozhraní pro vytvoření tzv. listu a vytváření obecných grafických primitiv na tomto listu. List můžeme přirovnat například k dokumentu textového procesoru a grafická primitiva k jednotlivým znakům abecedy.

Kromě toho bude v jádře implementována i obecná komponenta sloužící pro vizualizaci grafických primitiv – renderer. Tím bude zajištěna úplnost jádra knihovny. Při vykreslování grafického primitiva bude důležitým parametrem právě reference na objekt implementující renderer.

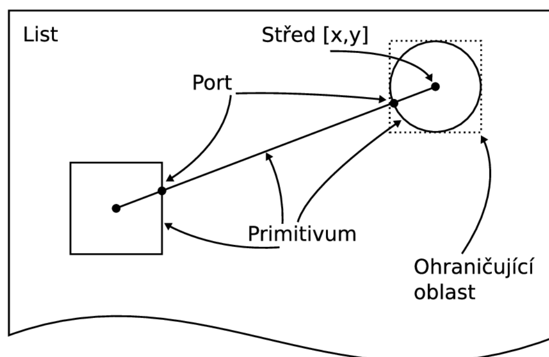
V souladu s terminologií objektově orientovaného návrhu tedy jádro knihovny obsahuje definici abstraktního grafického objektu a abstraktního rendereru.

2.2.2 Kreslení základních grafických primitiv

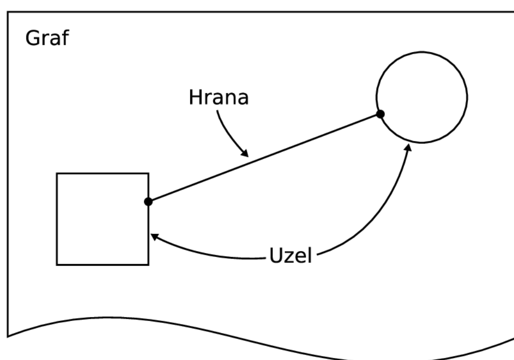
Tato komponenta už bude nabízet konkrétní grafická primitiva. Každé primitivum je popsáno svým typem („čára“, „kružnice“. . .) a funkcemi požadovanými jádrem. Tyto informace pak budou využívat další komponenty, zejména pro vizualizaci.

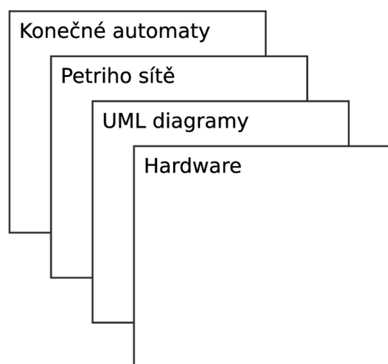
Pro dobrou využitelnost projektu stačí vytvořit následující objekty: čára (úsečka), lomená čára, Bézierova křivka, elipsa, kružnice, polygon, obdélník, čtverec a textový řetězec. Při bližsím prozkoumání jsou čára a křivka zvláštním případem lomené čáry (liší se v prvním případě maximálním počtem bodů, ve druhém způsobem vykreslení). Podobně kružnice je elipsa se stejně dlouhými poloosami a obdélník a čtverec konkretizované polygony.

Jádro

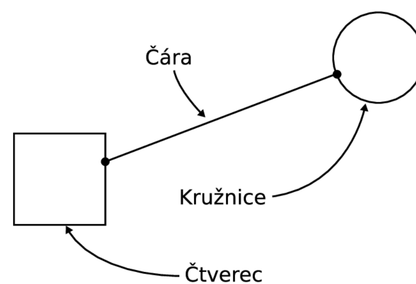


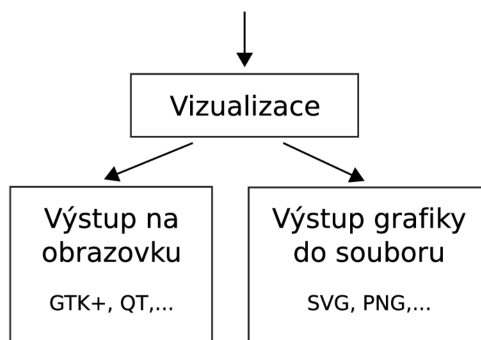
Semantika - graf

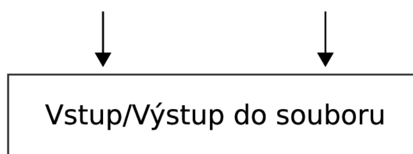




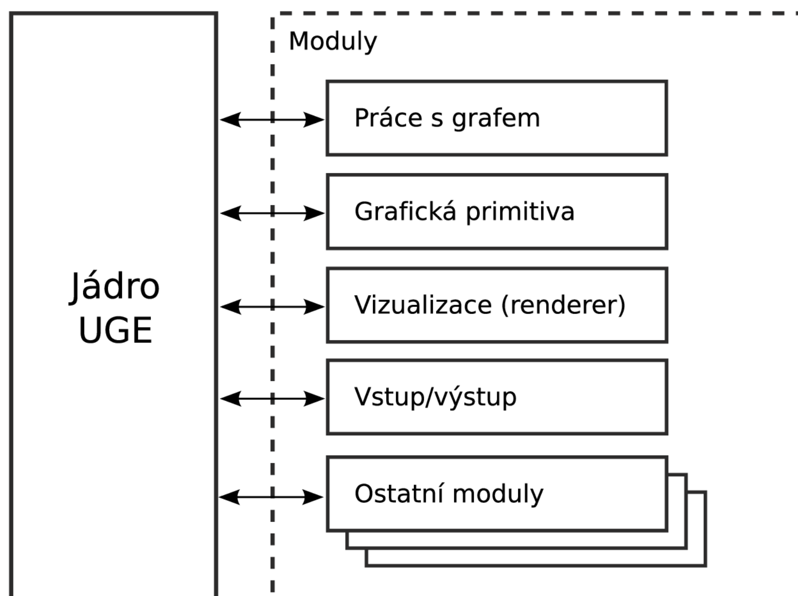
Vzhled - vektorová ilustrace







Obrázek 2.1: Architektura knihovny UGE



Obrázek 2.2: Jádro a základní moduly knihovny

2.2.3 Práce s grafem

Komponenta pro práci s grafem bude dodávat dosud nakresleným grafickým primitivům semantiku grafu. Bude poskytovat funkce pro vytvoření grafu, uzlu a hrany.

Graf kromě množiny uzlů a hran může obsahovat rovněž referenci na list s grafickými primitivy. Potom každý grafový prvek (uzel nebo hrana) odkazuje na grafické primitivum, kterým je reprezentován, a v omezené míře dále ovlivňuje jeho vzhled (například pokud je hrana reprezentována čárou, jsou její konce upraveny tak, aby se dotýkaly portů na hranici uzlů, které hrana spojuje).

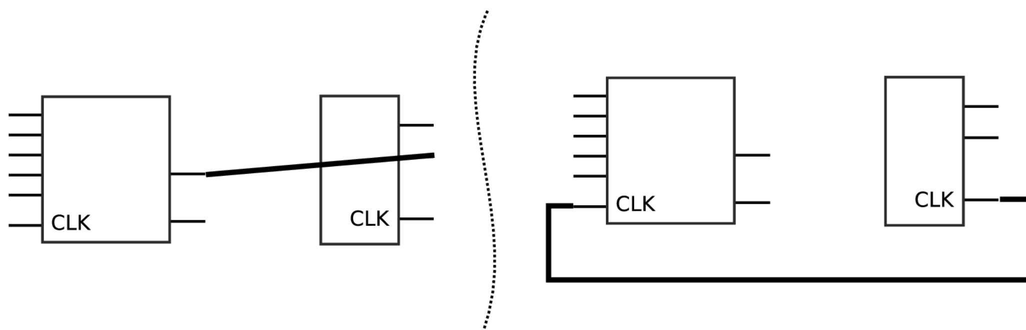
Dále bude knihovna poskytovat mechanismus, který umožní udržování konzistence grafu. Za určitých okolností může dojít například k situaci, kdy se uživatel pokusí spojit hranou dva uzly, u kterých to syntaxe daného typu grafu neumožňuje. V takovém případě nebude muset vše řešit aplikace postavená s využitím knihovny UGE, postačí když nadefinuje potřebná pravidla a UGE dohlédne na jejich dodržení.

Celou situaci předvedu na příkladu návrhu HW architektury. Na obrázku 2.3 je vlevo znázorněn pokus o spojení dvou komponent „obyčejnou“ hranou, která je vizuálně reprezentována čarou. Při správné implementaci uzlů grafu – HW komponent – bude pokus o vytvoření takovéto hrany zamítnut. Vpravo je potom znázorněno vytvoření správné hrany – signálu CLK – stačí vybrat správný typ hrany a uzly (HW komponenty) samy správně určí porty, kam je třeba hranu napojit. Hrana pak má v sobě vestavěnu inteligenci, která umožní vést čáru správným způsobem.

2.2.4 Vizualizace

Zvláštním typem komponenty (nebo zásuvného modulu) je takzvaný renderer. Tyto moduly (na obrázku 2.1 v pravé spodní části) mají na starost samotné zobrazení grafu.

Renderer poskytuje rozhraní pro kreslení těch nejzákladnějších objektů: text, čára, polygon, elipsa. Kromě toho uchovává další informace, potřebné pro kreslení, jako je velikost a řez použitého písma, barva popředí (čáry) a pozadí (výplně), styl a tloušťka čáry. Dále pak informace specifické pro různé renderery – například grafický kontext při kreslení na GTK+ widget.



Obrázek 2.3: Kontrola vytvářených hran

2.2.5 Ukládání a načítání ze souboru

Komponenty pro ukládání a načítání musí manipulovat jak se sémantickou, tak s vizuální stránkou grafu. Každý prvek grafu a grafické primitivum musí pro správné uložení samo znát způsob zakódování do souboru a opětovné rekonstrukce z uložených dat (tomuto se někdy říká serializace).

2.3 Použití knihovny

Na výpisu zdrojového kódu 2.4 je ilustrováno předpokládané použití knihovny UGE pro vytvoření jednoduchého grafu se dvěmi uzly spojenými hranou. Uzly jsou reprezentovány vizuálně kružnicí a čtvercem, hrana pak čarou.

List pro umístění grafických objektů není nutné vytvářet – je vytvořen a asociován s grafem automaticky v případě, že je parametr funkce `uge_graph_new` `NULL`.

Konce čáry jsou získány jako porty vždy ve směru protilehlého uzlu. Pro získání portu kromě souřadnice směru zadáváme i protilehlý uzel grafu, to umožňuje ovlivnit polohu portu v závislosti na typu uzlu, se kterým hrana spojuje (například signál CLK u HW komponent bude mít vyhrazený jediný port bez ohledu na směr, ve kterém se nachází)

Parametr funkce `uge_graph_node_get_port`, který je v ukázce nastaven na `NULL`, je vyhrazen pro souřadnice místa na uzlu, kself.guiněmuž port patří a ke kterému má ležet co nejblíže (použití může nalézt v uživatelském rozhraní, kde pro určité typy uzlu bude žádoucí ovlivnit souřadnice portu místem v uzlu, kam uživatel kliknul myší.

Příklad je ukončen vykreslením grafu do souboru SVG.

```
#include <uge_graph.h>
#include <uge_geom.h>
#include <uge_render_svg.h>

void test_graph(void)
{
  UgeGraph *graph;
  UgeGraphNode *nodeA, *nodeB;
  UgeGraphEdge *edge;

  UgeGeomLine *line;
  UgeGeomCircle *circ;
  UgeGeomSquare *square;
  UgeCoord *line_start, *line_end;

  UgeRenderer *r;

  graph = uge_graph_new(NULL);

  // parametry: list, stred x, stred y, polomer
  circ = uge_geom_circle_new(graph->sheet, uge_coord_new(10, 10), 5);

  nodeA = uge_graph_node_new(graph, circ);

  square = uge_geom_square_new(graph->sheet, uge_coord_new(30, 30), 10);

  nodeB = uge_graph_node_new(graph, square);

  start = uge_graph_node_get_port(nodeA, nodeB, NULL, circ->top_left);
  end   = uge_graph_node_get_port(nodeB, nodeA, NULL, square->top_left);

  line = uge_geom_line_new(graph->sheet, start, end);

  edge = uge_graph_edge(graph, nodeA, nodeB, 0, line);

  if (edge == NULL) printf(''Hranu neni mozne vytvorit\n'');

  r = (UgeRenderer *)uge_render_svg_new(''test.svg'');
  uge_sheet_draw_all(graph->sheet, r);
}
```

Obrázek 2.4: Ukázka použití knihovny UGE

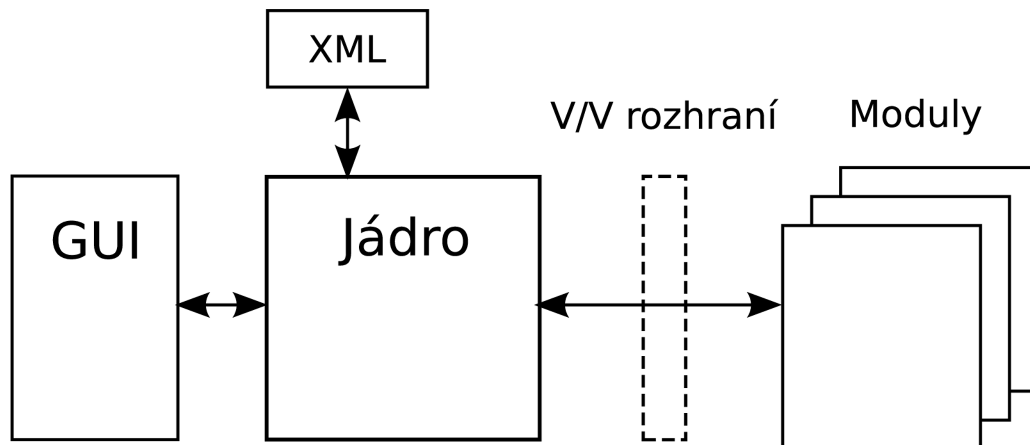
Kapitola 3

Současný stav projektu Univerzální grafický editor

Tato kapitola velmi stručně představuje návrh UGE jako celku tak, jak byl původně navržen v rámci bakalářských prací. Dále pak zkoumá skutečný stav implementace UGE se zaměřením na možnost využití existujících zdrojových kódů v projektu UGE jako knihovny.

3.1 Původní návrh architektury systému UGE

Architektura sytému UGE je schematicky znázorněna na obrázku 3.1. Univerzální grafický editor byl od počátku navrhován jako vysoce modulární systém s minimálními závislostmi na knihovnách grafického uživatelského rozhraní. Jádro systému mělo poskytovat základní operace s grafem a veškeré náročnější operace nad grafy měly být realizovány zásuvnými moduly. Mezi vyvíjené moduly patřily moduly pro práci s konečnými automaty a Petriho sítěmi ([5]), modul pro návrh HW architektury ([14]) a modul pro automatické přehledné rozvržení grafu ([7]).



Obrázek 3.1: Architektura systému UGE (převzato z [6])

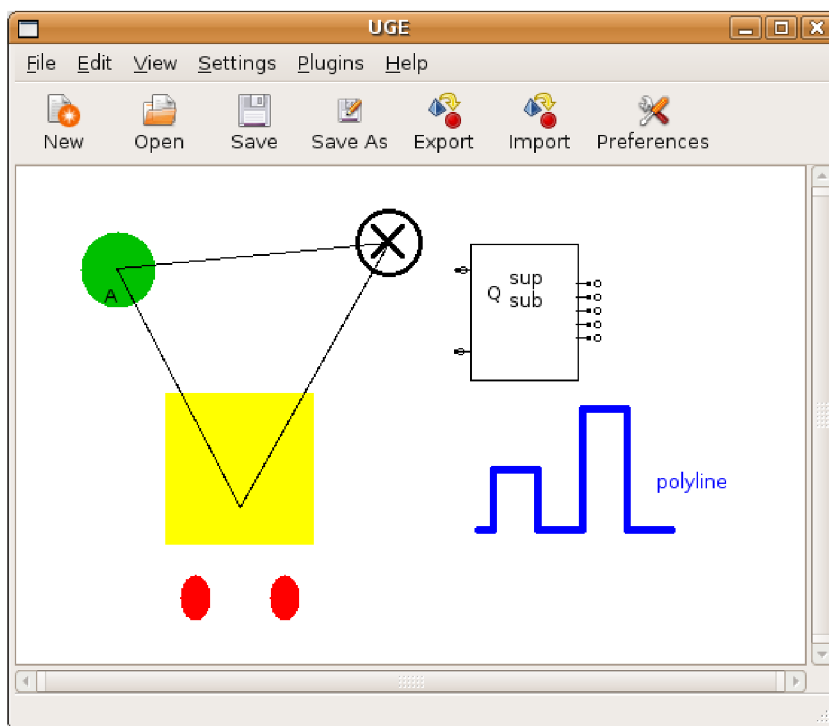
3.2 Stav implementace systému UGE

Implementace systému se bohužel v několika aspektech odchýlila od původního návrhu a není úplně dokončena. Zejména nebyla dodržena nezávislost na grafickém rozhraní a konkrétní knihovně grafického uživatelského rozhraní.

Zdrojové kódy systému včetně modulů tak, jak vznikly v roce 2006, jsou tvořeny několika desítkami tisíc řádků zdrojového kódu převážne v jazyce C++. Rozhraní pro komunikaci jádra se zásuvnými moduly je potom implementováno v jazyce C (V/V rozhraní na obrázku 3.1). Tato komunikační vrstva však zůstala nedokončena a hlavně špatně otestována.

Samotné jádro – odhadem 15–20 tisíc řádků kódu – zahrnuje třídy pro práci s grafem a grafickými primitivy a pro grafické uživatelské rozhraní aplikace. Jak jsem už naznačil dříve, chybí důsledné oddělení funkcí pro práci s grafem a obsluhy uživatelského rozhraní. Kód uživatelského rozhraní je sice umístěn odděleně od zbytku jádra, tvoří ale poměrně tenkou obálku nad knihovnou GTK+, která byla zvolena jako výchozí GUI knihovna projektu. Z tohoto důvodu by například původně plánovaná snadná možnost implementace alternativního GUI s využitím QT mohla stát značné úsilí.

Stav zásuvných modulů není z pohledu projektu UGE jako knihovny příliš důležitý. Úkoly řešenými těchmit modul se knihovna jako taková zabývat nebude. Stav implementace modulů je ještě rozpačitější než jádra UGE, existující zdrojové kódy nejsou dobře otestovány a odladěny a obsahují řadu i poměrně zásadních chyb (například neoprávněné přístupy do paměti způsobjující pády celé aplikace v modulu přehledného rozvržení).



Obrázek 3.2: Uživatelské rozhraní stávající aplikace

3.3 Možnosti využití stávajících zdrojových kódů

Z několika důvodů rozebíraných dále v této kapitole jsem se rozhodl v projektu UGE jako knihovny existující zdrojové kódy nevyužívat a začít s implementací od začátku.

Orientace na jedinou instanci grafu Jádro systému UGE pracuje s jedinou instancí grafu. Naopak od obecné knihovny pro práci s grafem očekáváme možnost manipulace s více grafy v jedné aplikaci (podobně jako od knihovny pro práci s grafikou očekáváme možnost manipulovat s libovolným množstvím obrázků v jedné aplikaci).

Funkčnost nevyužitelná v knihovně, signály Značná část stávajcích zdrojových kódů je pro knihovny nepoužitelná, protože řeší úkoly, kterými se knihovna zabývat nebude. Příkladem budiž inicializace a komunikace se zásuvným modulem.

Samostatnou skupinu těchto funkcí toví obsluha signálů neboli zpětných volání. UGE poskytovalo prostřednictvím vstupně/výstupního rozhraní svým modulům možnost registrovat vlastní funkce, které potom byly volány při daných událostech. Například při vytváření nového uzly grafu mohla být volána funkce, která mohla případně vytvoření uzlu zakázat. Část této funkcionality bude v knihovně zachována. Dalšími signály jsou ale například vytvoření nové položky v menu aplikace, kliknutí na oblast grafu a podobně - těmito událostmi těsně souvisejícími s GUI se knihovna nebude zabývat vůbec.

Potenciálně značné množství chyb Systém UGE vzniknul během poměrně krátké doby, což vedlo k zanedbání testování a ladění. Velké množství zdrojového kódu tak může skrývat řadu na první pohled neodhalených chyb. Toto je nepříjemné zvlášť s přihlédnutím k tomu, že velká část kódu nemá v knihovně přímé využití a přitom ji není možné snadno odbourat.

Provázanost s GUI a orientace na vizualizaci grafu Knihovna UGE může být použita i pro práci s matematickým modelem grafu, bez nutnosti jakékoli manipulace nebo samotné znalosti vzhledu grafu. Ve stávající implementaci jsou přitom uzly a hrany grafu pevně svázány se svým vzhledem. A naopak – jak vyplývá z názvu projektu „Univerzální *grafický* editor" – knihovna UGE může být využita v aplikaci pro práci s vektorovou grafikou bez jakéhokoli vědomí o tom, že vůbec něco jako graf existuje.

Kapitola 4

Univerzální grafický editor jako modul pro Python

Kapitola o modulu pro Python začíná podobně jako předchozí kapitola analýzou požadavků na tento modul a pokračuje návrhem architektury modulu.

4.1 Požadavky na modul

Nejprve v několika bodech uvedu požadavky kladené na modul UGE a potom v následujících podkapitolách budou tyto požadavky podrobně popsány a odůvodněny.

- Využití knihovny UGE popsané v předchozích kapitolách pro kreslení grafických objektů a pro práci s grafem
- Objektově orientovaný návrh modulu
- Jednoduchá instalace a použití
- Snadná rozšiřitelnost o nové typy grafů, uzlů a hran, případně složitějších grafických objektů
- Podpora pro kontrolu konzistence grafu (např. v Petriho síti není možné spojit hranou místo s místem nebo přechod s přechodem)

4.1.1 Využití knihovny UGE v modulu pro Python

Vytvářený modul by měl využívat dynamickou knihovnu vytvořenou v první části práce všude, kde to bude možné. Tím se zabrání duplikování zdrojových kódů a algoritmů a vzniku řady chyb, které s duplicitami ve zdrojových kódech bývají spojeny. S trochou nadsázky můžeme modul považovat za objektový wrapper nad knihovnou UGE. Už při implementaci knihovny byl kladen důraz na objektově orientovaný přístup k problému, přestože jazyk C jako takový není objektově orientovaným jazykem.

Tento požadavek rovněž předurčuje jazyk C jako jasnou volbu pro implementaci modulu. Tímto by zároveň výsledný modul měl získat na rychlosti ve srovnání s tím, kdyby byl celý implementován v jazyce Python. Daní ovšem bude vyšší programátorská náročnost implementace a možnost vzniku některých chyb, ke kterým by jinak nemohlo dojít, hlavně v souvislosti s alokací a uvolňováním paměti a počítáním referencí na objekty Pythonu.

4.1.2 Objektově orientovaný návrh

Objekty v oblasti práce s grafem představují v první řadě samotný graf, potom uzly a hrany grafu. V kontextu knihovny a modulu UGE pak dále grafické objekty (primitiva), jimiž jsou například čára, křivka, elipsa, polygon...

Modul UGE bude tedy navržen objektově, tento způsob návrhu a implementace umožňuje snadno a srozumitelně modelovat realitu práce s grafy. Zároveň bude tento styl přirozeně odrážet implementaci knihovny jazyka C, kde jsou objekty představovány strukturami obsahujícími kromě dat ukazatele na funkce pracující s daty objektu (analogie virtuálních metod v objektových jazycích).

4.1.3 Jednoduchá instalace a použití

Instalace modulu musí být v rámci možností jednoduchá, aby neodrazovala od použití modulu. Jako schůdný způsob se jeví nakopírování dynamické knihovny do adresáře, kde bude dostupný interpretu Pythonu.

Pro vytvoření a vizualizaci jednoduchého grafu musí stačit jednoduchý kód. Příklad takového kódu je naznačen na výpise 4.1 (výsledkem je totéž jako v případě kódu v jazyce C na ukázce 2.4). Rozhraní zároveň odráží rozhraní knihovny UGE, staví na stejných principech.

```
import uge

def test_graph():

  graph = uge.Graph(uge.Sheet(640, 480))

  circle = uge.Circle(graph.sheet, 10, 10, 5)

  nodeA = uge.Node(graph, circle)

  square = uge.Square(graph.sheet, 30, 30, 10)

  nodeB = uge.Node(graph, square)

  start = nodeA.get_port(nodeB, circle.x, circle.y, square.x, square.y)
  end   = nodeB.get_port(nodeA, square.x, square.y, circle.x, circle.y)

  line = uge.Line(graph.sheet, start[0], start[1], end[0], end[1])

  edge = graph.edge(nodeA, nodeB, line)

  if (edge == None):
    print 'Hranu neni mozne vytvorit'

  graph.draw(uge.SvgRenderer(''test.svg''))
```

Obrázek 4.1: Ukázka možného použití modulu UGE

4.1.4 Snadná rozšiřitelnost

Vytváření a manipulace s triviálními grafy jako je naznačeno na ukázce 4.1 však není hlavním posláním modulu. Tím je poskytnout základní rámec pro další rozšiřování a vytváření nových modulů pro práci se specifickými typy grafů – například konečnými automaty, Petriho sítěmi a dalšími.

Rozšiřitelnost a modularita modulu je proto pro budoucí použití velmi důležitým požadavkem a musí jim být při návrhu věnována odpovídající péče. Předpoklady pro rozšiřitelnost podpoří rovněž objektový návrh modulu.

4.1.5 Kontrola konzistence grafu

Modul pro Python musí podobně jako knihovna, nad kterou je postaven, podporovat mechanismy pro zajištění konzistence a správnosti vytvářeného grafu. Cílem je umožnit relativně snadnou implementaci podmínek formou algoritmů pro určení, jestli dva uzly je možné spojit hranou či nikoli.

Na ukázce 4.2 je naznačen možný způsob vytvoření velmi jednoduchého rozšíření modulu UGE, které spojuje vizuální a semantickou stránku grafu do jednoho celku (grafické objekty reprezentující prvky grafu jsou tedy vytvářeny automaticky a mají přesně daný vzhled). Zároveň je naznačena možnost kontroly konzistence znemožněním spojení dvou stejných typů uzlu.

```
import uge

class PN_Place(uge.Node):
  def __init__(self, graph, x, y):
    p = uge.Circle(graph.sheet, x, y, 10)
    super(PN_Place, self).__init__(graph, p)

class PN_Transition(uge.Node):
  def __init__(self, graph, x, y):
    p = uge.Rectangle(graph.sheet, x-3, y-10, 6, 20)
    super(PN_Transition, self).__init__(graph, p)

class PN_Edge(uge.Edge):
  def __init__(self, graph, a, b):
    p_start = a.get_port(b, 0, 0, b.primitive.x, b.primitive.y)
    p_end = b.get_port(a, 0, 0, a.primitive.x, a.primitive.y)
    p = uge.Line(graph.sheet, p_start[0], p_start[1], p_end[0], p_end[1])
    p.set_arrows('end')
    super(PN_Edge, self).__init__(graph, a, b, p)

class PN_Net(uge.Graph):
  def edge(self, a, b):
    if (type(a) != type(b)):
      return PN_Edge(self, a, b)
    else:
      return None
```

Obrázek 4.2: Ukázka rozšíření modulu UGE

4.2 Návrh architektury modulu

Návrh modulu vychází z implementované knihovny jazyka C. Prezentován bude formou UML diagramů, převážně pomocí komentovaných diagramů tříd, které se pro objektově orientovaný systém přímo nabízí. Pro popis komunikace jednotlivých objektů systému ve složitějších situacích poslouží sequence diagramy.

Na diagramech je rozlišeno několik komponent. První z nich je jádro systému (na diagramech `UgeCore`), které obsahuje podobně jako jádro knihovny třídu pro tzv. list – prostor pro umísťování grafických objektu, dále definici abstraktního grafického primitiva a třídu umožňující seskupování primitiv a tím vytváření složitějších objektů a nakonec abstraktní renderer. Kromě toho je v diagramech v komponentě `UgeCore` znázorněna třída `Object`, která slouží jako předek všech ostatních tříd. Její ekvivalent existuje v implementaci knihovny s rozhraním jazyka C, v modulu pro Python bude její úlohu plnit `PyObject` (o tom ale až dále v kapitole týkající se implementace).

Další z komponent obsahuje třídy konkrétních grafických objektů – v diagramech je označena jako `UgeGeometry`. Veškeré objekty související se semantickou stránkou grafu jsou soustředěny do komponenty `UgeGraph`. Poslední komponentou je `UgeRenderer` obsahující konkrétní implementace tříd pro vizualizaci grafických objektů.

4.2.1 Práce s grafem

Třídy objektů pro objekty grafu jsou znázorněny na diagramu 4.3. Třída uzlu (`Node`) a hrany (`Edge`) mají společného předka v podobě abstraktní třídy pro obecný grafový objekt (`GraphObject`). Graf se pak skládá z těchto objektů – obsahuje seznam uzlů a seznam hran, což přesně odráží matematickou teorii grafů, která graf definuje jako dvojici $G(V,E)$, kde V je množina uzlů grafu a E množina hran.

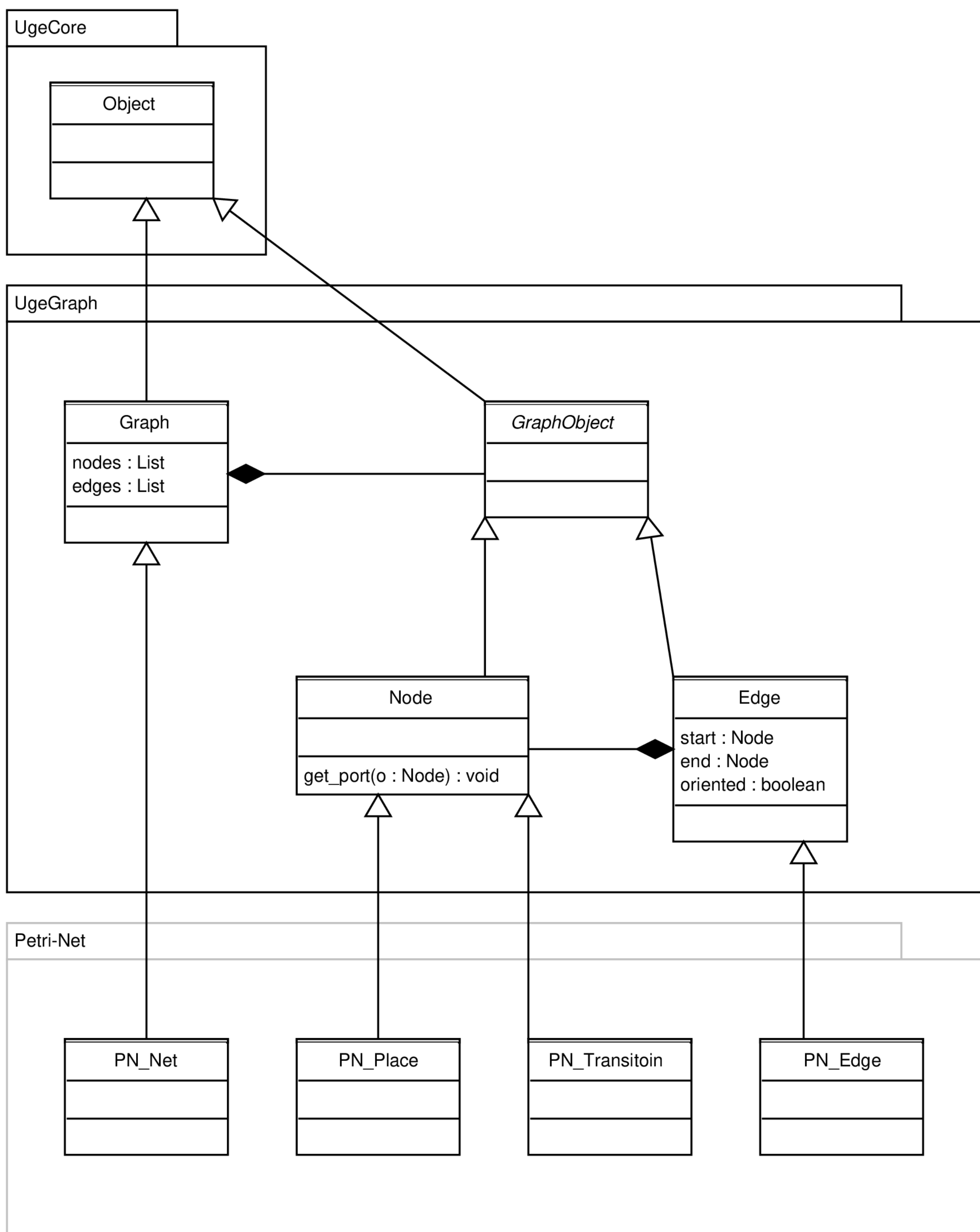
V diagramu je rovněž naznačen pravděpodobně nejčastější způsob rozšiřování knihovny – vytvoření nového typu grafu, uzlů a hran. Zde se jedná o Petriho síť se dvěma typy uzlů (místo a přechod).

4.2.2 Grafické objekty

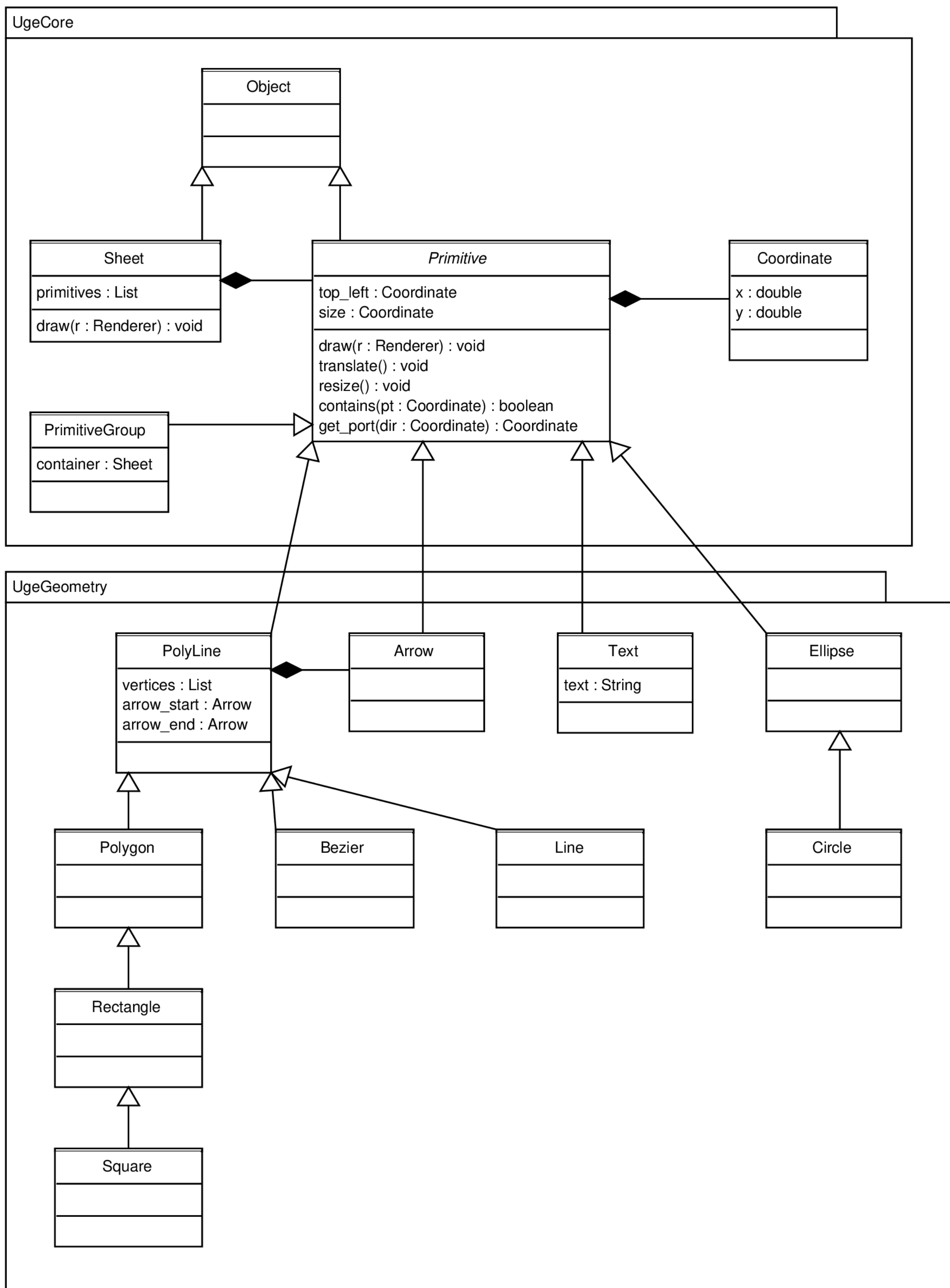
Diagram tříd grafických objektů je znázorněn na obrázku 4.4. Z grafu je patrná role abstraktní třídy `Primitive` definující rozhraní povinné pro všechny grafické objekty. Součástí jádra je kromě toho třída poskytující možnost konstrukce složitějších objektů seskupením více základních primitiv z balíku geometrie. Roli kontejneru pro seskupené grafické objekty zastupuje instance třídy list. To umožní snadné zobrazení a manipulaci se skupinou jako celkem bez nutnosti duplikování zdrojového kódu.

Nezvykle může působit odvození polygonu od lomené čáry. Na první pohled se jedná o úplně odlišné objekty, jsou však popsány stejným způsobem – uspořádanou množinou bodů (vrcholů, anglicky *vertices*). Liší se vykreslením – polygon je uzavřen, jeho poslední a první vrchol jsou spojeny čarou (a je možné jej vyplnit barvou). U všech typů čar můžeme doplnit šipku na začátek a konec čáry (tato vlastnost je u polygonů samozřejmě potlačena). Šipka je určena bodem, na který ukazuje (splývá s koncem čáry) a směrem („odkud“ šipka ukazuje – bude typicky splývat s bodem čáry sousedícím s koncovým bodem).

Obrázek 4.5 znázorňuje propojení mezi prvky grafu a grafickými objekty. Graf obsahuje referenci na list s grafickými objekty. Metoda pro vykreslení grafu s využitím rendereru pak nebude dělat nic jiného, než že zavolá metodu vykreslení listu. Každý prvek grafu (uzel nebo hrana) pak obsahuje referenci na příslušný grafický objekt, který je jeho vizuální reprezentací. A naopak rovněž každý grafický objekt obsahuje referenci na příslušný prvek grafu. To samozřejmě nebude platit



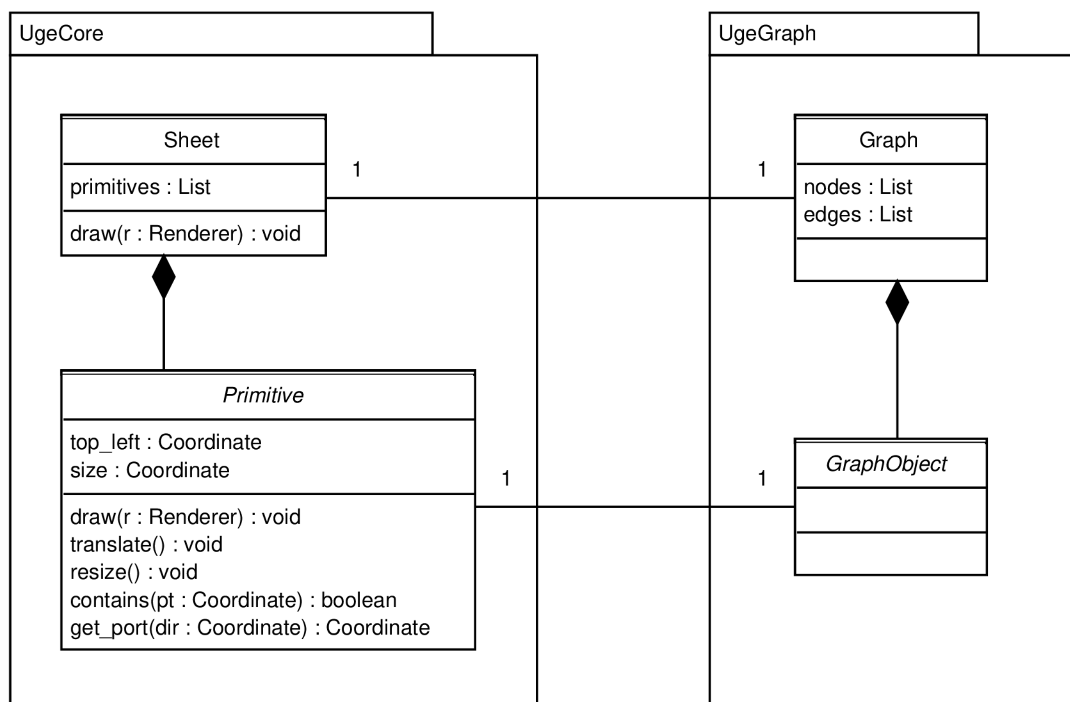
Obrázek 4.3: Diagram tříd objektů grafu



Obrázek 4.4: Diagram tříd grafických objektů

v případě, že aplikace bude využívat pouze semantické možnosti modulu a grafické objekty tak vůbec nebude využívat, a nebo bude naopak sloužit jenom jako vektorový editor.

Toto propojení je velmi důležité z pohledu tvorby uživatelského rozhraní s využitím modulu. V uživatelském rozhraní uživatel manipuluje s grafickými objekty a je na modulu, aby provedené změny neporušily vizuální konzistenci grafu – například při změně polohy objektu představujícího uzel (například kružnice) je nutné odpovídajícím zpusobem upravit i čáry představující hrany spojené s uzlem (upravit polohu koncových bodů).



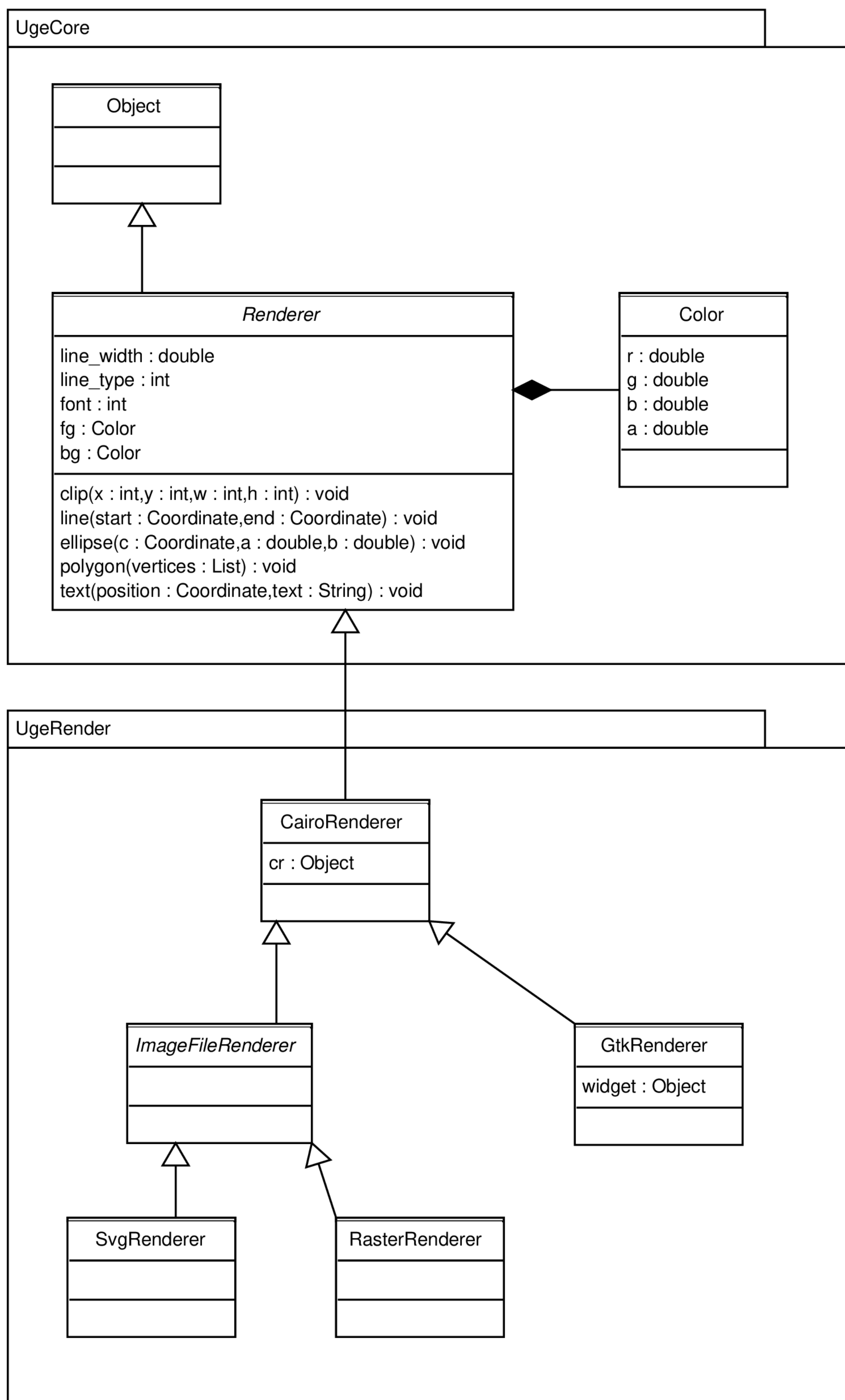
Obrázek 4.5: Vazba mezi semantickou a vizuální stránkou grafu

4.2.3 Vizualizace

Poslední v řadě diagramů tříd zachycuje část systému týkající se vizualizace, která bude využívat vektorové grafické knihovny Cairo. Objekty poskytující metody pro zobrazení grafických objektů implementují pouze jednoduché rozhraní. To umožní snadné využití jiných grafických toolkitů než je zde použité Cairo.

Abstraktní renderer je přímo součástí jádra a zajišťuje úplnost systému i v případě, že by byla úplně odstraněna komponenta označená v diagramu 4.6 jako `UgeRenderer`. To znamená, že ačkoli mluvím o abstraktní třídě, při důsledném dodržení terminologie to není pravda – bude možné vytvářet instance třídy `Renderer`, ale metody pro kreslení budou prázdné.

Každý renderer poskytuje metody pro vykreslení čáry (přesně úsečky), elipsy, textu a polygonu. Metoda `clip` slouží pro ořezání kresleného výstupu – účelem je zvýšení výkonu u velkých grafů, protože není třeba počítat a kreslit celý rozsáhlý graf, když viditelná je aktuálně na obrazovce pouze jeho část.



Obrázek 4.6: Diagram tříd sloužících pro vizualizaci

4.2.4 Komunikace komponent

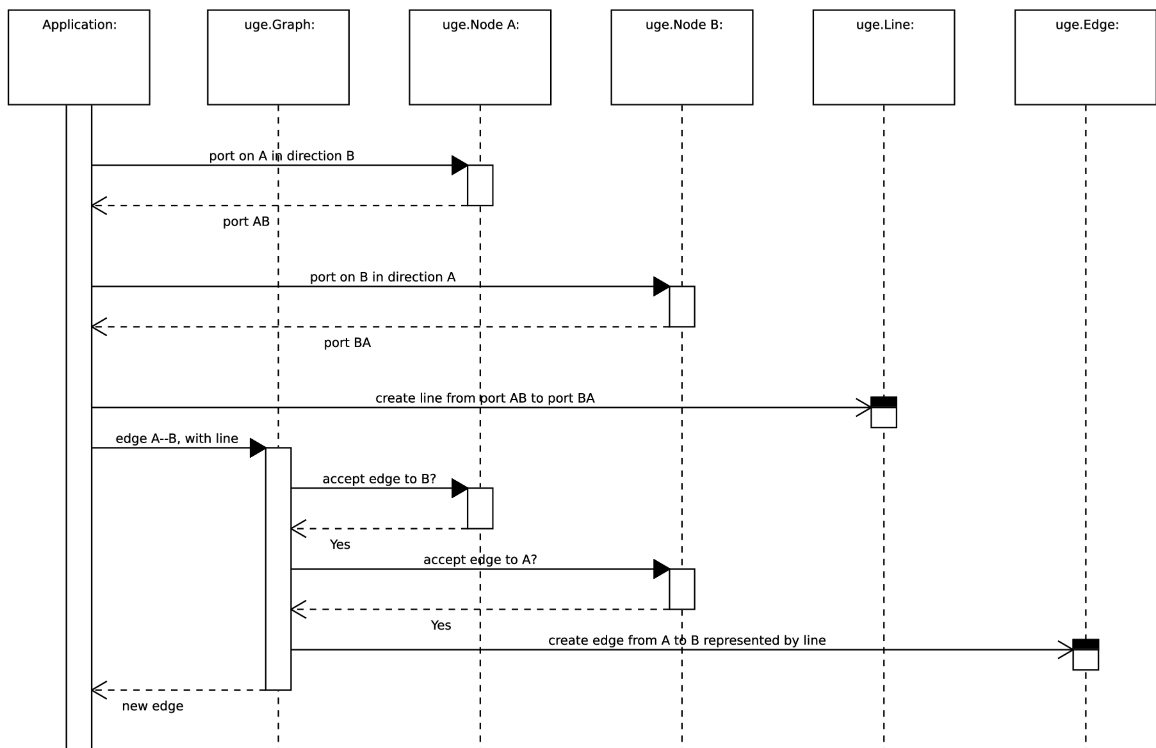
V tomto oddíle je s pomocí sequence diagramů navržen a popsán proces vytváření nové hrany spojující dva uzly v grafu spolu s vytvořením odpovídající čáry reprezentující hranu, změna polohy vizuální reprezentace uzlu a odstranění uzlu z grafu.

Vytváření hrany Vytváření hrany začíná získáváním portů a vytvořením čáry spojující tyto dva porty. Aplikace žádá o souřadnice portů spojované uzly grafu. V diagramu není pro zvýšení přehlednosti zaznamenán fakt, že uzel polohu portu získává od grafického objektu, kterým je vizuálně reprezentován. Tento postup je zvolen proto, aby jak uzel, tak grafický objekt, mohly polohu portu ovlivnit (grafický objekt vrací souřadnice bodu na obvodu v zadaném směru, uzel ale může polohu ovlivnit ve prospěch například pinu HW koponenty).

Požadavky na kontrolu konzistence grafu dělají z vytváření hrany komplexní proces znázorněný na diagramu na obrázku 4.7. Samotná kontrola může probíhat na dvou úrovních (mimo samotnou aplikaci využívající modul).

První kontrolu provádí objekt grafu. Zde může proběhnout například kontrola typu uzlů a v případě konfliktních typů zamítnutí vytvoření hrany (například už dříve vzpomínané Petriho sítě, kde není možné spojit dvě místa nebo dva přechody).

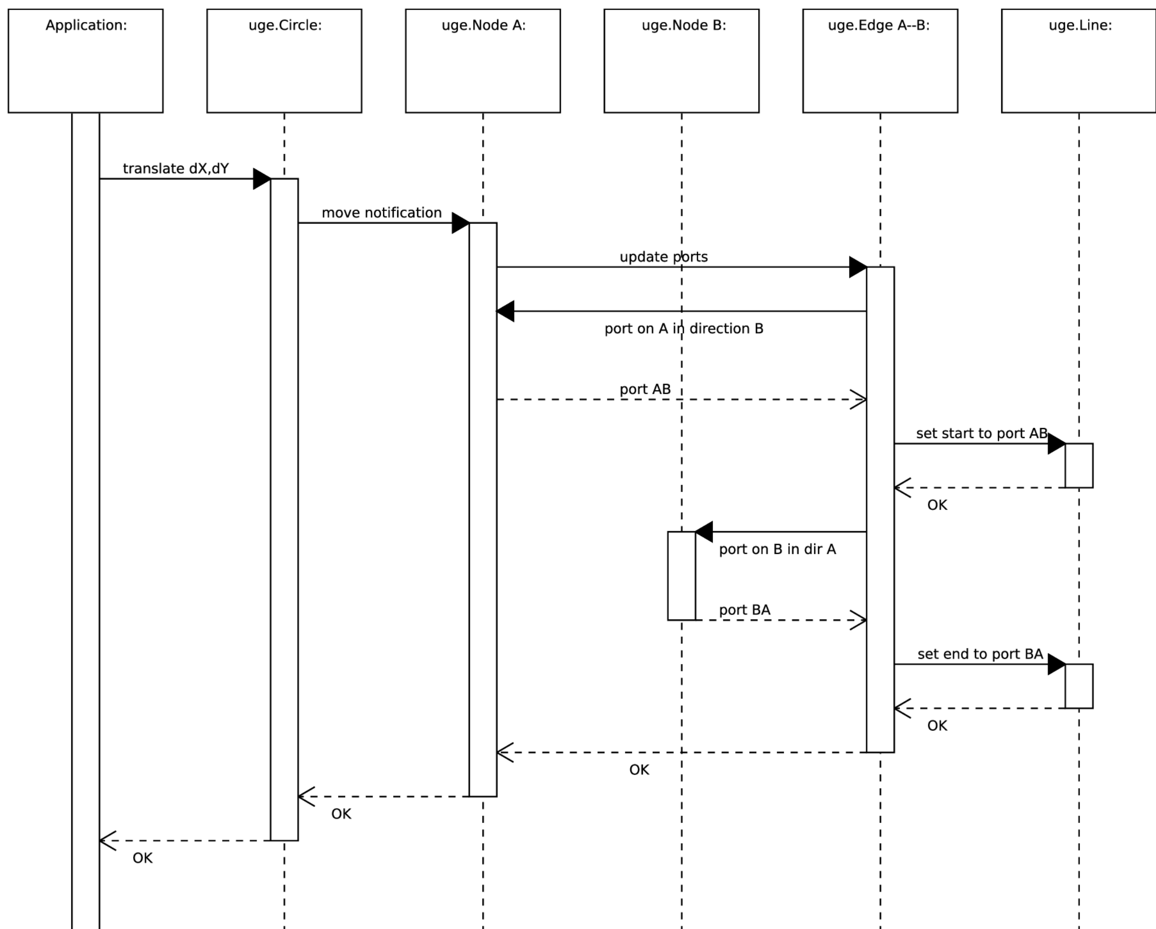
Pokud graf samotný neodhalí problém, ptá se každého z uzlů, které má hrana spojovat, jestli je možné propojení s druhým uzlem. To znamená pokud hrana spojuje uzly A a B, graf se nejprve ptá uzlu A, jestli může být spojen hranou s uzlem B, potom se ptá uzlu B, jestli může být spojen s A. Toto je prostor pro řešení situací, kdy uzel může být spojen s jiným uzlem pouze jednou



Obrázek 4.7: Vytváření nové hrany

hranou, nebo například jenom s jedním uzlem určitého typu. Například HW komponenta má jenom jeden port určený pro signál CLK a proto další hranu spojující s tímto signálem odmítne. Tyto složité kontroly by nebylo vhodné řešit přímo na úrovni grafu, protože podobných pravidel může být v aplikační doméně obrovské nebo dokonce předem neurčené množství.

Změna polohy objektu Při změně polohy grafického objektu reprezentujícího uzel grafu (v grafickém uživatelském rozhraní například jeho potažením myší) je kromě samotné změny polohy nutné správným způsobem upravit také konce čar představujících hrany spojující jej s jinými uzly grafu. To je umožněno propojením objektů grafu s grafickými primitivy popsaným na obrázku 4.5. Modul UGE je tedy zodpovědný za aktualizaci vizuální reprezentace všech hran souvisejících s přesouvaným uzlem a uživatelská aplikace je od tohoto procesu naprosto odstíněna, musí se postarat jenom o překreslení plátna. Celý proces je zachycen na diagramu 4.8.



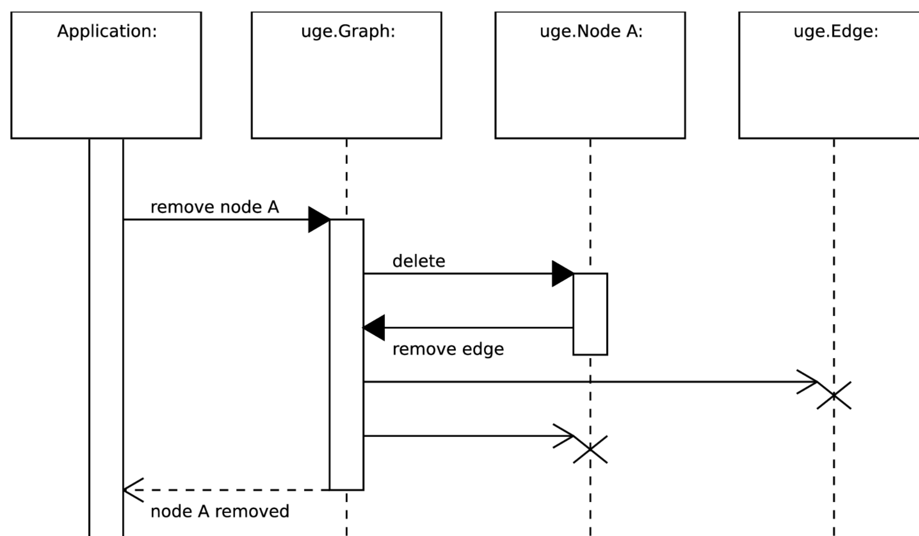
Obrázek 4.8: Změna polohy grafického objektu reprezentujícího uzel

Nabízí se však otázka, proč při každé změně polohy vyhledávat všechny hrany a přepočítávat porty. Elegantním řešením by bylo navrhnout port jako zvláštní typ bodu. Přesné souřadnice tohoto bodu by nebyly uloženy staticky, ale vypočítávány dynamicky až při vykreslování grafu. S tímto přístupem dokonce počítal semestrální projekt. Toto řešení má ovšem některé slabiny: Překreslování grafu v aplikaci s grafickým uživatelským rozhraním se zpravidla děje poměrně často, zároveň se ale často překresluje pouze část plátna. Vždy by se však musely počítat polohy všech portů minimálně

za účelem zjištění, jestli port vůbec leží v překreslované části plátna. A algoritmus pro výpočet polohy portu nemusí být vždy rychlý. Například pro výpočet bodu na obvodu elipsy se neobejdeme bez odmocňování a navíc ani není možné předvídat složitost grafických objektů potřebných v budoucnu ve specifických oblastech použití modulu.

Z uvedených výkonostních důvodů bude výhodnější přepočítat polohy portů jenom pokud dojde v grafu k nějaké změně. Přitom stačí přepočítat jenom ty porty, jichž se změna bezprostředně dotkne.

Odstranění uzlu z grafu Při odstranění uzlu z grafu je nutné zachovat konzistenci grafu. To znamená odstranit i všechny hrany související s uzlem. Dále pak jsou při odstraňování uzlu (a hran) odstraněna i všechny grafické objekty reprezentující uzel a odstraněné hrany. Proces je znázorněn na diagramu 4.9, pro zjednodušení je však vynecháno odstraňování grafických objektů.



Obrázek 4.9: Odstranění uzlu z grafu

Kapitola 5

Implementace

První část kapitoly se věnuje implementaci Univerzálního grafického editoru jako knihovny s rozhraním jazyka C. Tato část práce byla zahájena v rámci semestrálního projektu a nyní je dále rozšířena o popis těch částí implementace, které v semestrálním projektu nebyly dokončeny. Hlavním důvodem tohoto zdržení je rozhodnutí nevyužívat zdrojové kódy z původního projektu a místo toho implementovat celou knihovnu od začátku.

V další části kapitoly je potom popsán způsob implementace modulu pro Python v jazyce C, s využitím knihovny UGE a Python C API.

5.1 UGE jako knihovna s rozhraním jazyka C

Programovací jazyk rozhraní je dán zadáním. Jako implementační jazyk samotné knihovny se nabízí C nebo C++. Druhý jmenovaný jazyk je sice silnější, ale vzhledem k požadavku na jednoduchost a kompaktnost celé knihovny jsem se rozhodl pro implementaci celé knihovny v jazyce C.

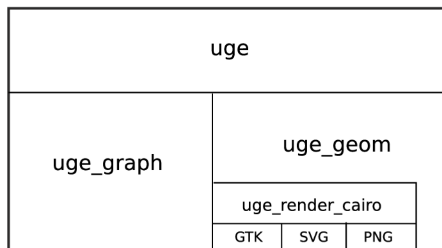
Pro demonstrační modul vizualizace grafu jsem zvolil GTK+ – stejně jako tomu bylo v původním projektu (viz [6]). Samotné kreslení je ale realizováno pomocí vektorové grafické knihovny Cairo. Ta je součástí GTK+ od verze 2.8 a umožňuje kromě kvalitnějšího výstupu s vyhlazováním hran výrazně rychlejší překreslování grafu než původně použité kreslení do pixmapy mimo obrazovku a její následné kopírování do zobrazovacího bufferu. Navíc Cairo přímo podporuje výstup do souboru ve vektorovém formátu SVG (Scalable Vector Graphics) a rastrovém PNG (Portable Network Graphics). Důležitým zdrojem informací při implementaci této části byl manuál [1] a tutoriál [13].

Rozdělení implementovaných komponent do souborů přehledně znázorňuje obrázek 5.1 (rozvržení odpovídá návrhu na obrázku 2.1).

Soubory `uge.h` a `uge.c` obsahují hlavičky a implementaci jádra knihovny. Kromě základních operací popsaných v kapitole 2 jsou v jádru implementovány rutiny pro práci s obousměrnými lineárními seznamy (předpona `uge_list`), které jsou používány k ukládání primitiv listu, uzlů a hran grafu, bodů čáry atd.

Hlavičky komponenty pro práci s grafem se nachází v souboru `uge_graph.h`. Byla zde dokončena v semestrálním projektu nehotová logika pro kontrolu konzistence grafu odpovídající potřebám popsaným v návrhu modulu pro Python, který bude tyto funkce využívat.

Dále v `uge_geom.h` jsou hlavičky komponenty poskytující základní grafické objekty – geometrická tělesa, primitiva. Tato část byla opět oproti semestrálnímu projektu dopracována, naprogramovány jsou všechny typy objektů popsané v návrhu včetně vykreslení, určení portů a polohy zadaného bodu vzhledem k objektu.



Obrázek 5.1: Struktura souborů implementace knihovny

Výsledkem sestavení knihovny je jeden dynamický objekt `libuge.so`. Neustále zdůrazňovaná oddělitelnost jednotlivých komponent je tak v současné verzi možná pouze na úrovni zdrojového kódu. V budoucnu v případě rozrůstání knihovny je pravděpodobné rozdělení do více dynamických objektů (např. `libugecore.so`, `libugegraph.so`…).

5.1.1 Aplikace principů OOP v neobjektovém jazyce

Protože cílem práce je využít vytvořenou knihovnu v objektově orientovaném modulu pro jazyk Python, byl na tento fakt kladen důraz už při implementaci knihovny.

Objekty jsou v jazyce C reprezentovány strukturami, které obsahují kromě dat také ukazatele na funkce manipulující s daty. Řešení inspirované knihovnou GTK a aplikačním rozhraním Pythonu je vhodné vysvětlit na jednoduchém příkladu – viz obrázek 5.2.

Nejprve vytvoříme strukturu pro obecný grafický objekt. Ta mimo dat obsahuje ukazatel na funkci starající se o vykreslení objektu (řádek označený číslem 1). Následuje vytvoření struktury konkrétního grafického objektu – lomené čáry. Její definice začíná vložením obecnější struktury (řádek 2) a pokračuje daty specifickými pro tento typ objektu. Potom implementujeme funkci starající se o vykreslení obecného objektu zavoláním jeho „virtuální metody“ (řádek 3). Přetypování na řádku 4 je bezproblémové a je umožněno konstrukcí z řádku 2.

Pro inicializaci objektu každého typu jsou vytvořeny tři funkce:

- `uge_object_type_new`,
- `uge_object_type_create`,
- `uge_object_type_init`.

První jmenovaná je určena pro použití v aplikacích. Plní úlohu konstruktoru tak jak ji známe z OOP. Jejím výsledkem je „objekt“ připravený k použití. Funkce s příponou create slouží k alokaci paměti potřebné pro příslušný objekt. Poslední jmenovaná plní úlohu implementace těla konstruktoru. Odvozené typy jsou povinny volat tuto funkci příslušnou odvozovaného typu. Typická práce této funkce spočívá v inicializaci proměnných ve struktuře a nastavení ukazatelů na funkce, jejichž implementace se liší od nadřazeného typu.

5.1.2 Použité algoritmy počítačové grafiky

V totmo krátkém oddíle bude popsán způsob zobrazení Bézierových křivek a postupy pro výpočet polohy portu a zjištění, jestli grafický objekt obsahuje zadaný bod či nikoli.

```
typedef struct _UgeGraphic
{
  double x;
  double y;
  void (*draw)(struct _UgeGraphic *); // 1
} UgeGraphic;

typedef struct _UgePolyLine
{
  UgeGraphic g; // 2
  UgeList *vertices;
} UgePolyLine;

UgePolyLine *uge_poly_line_new(void);

void uge_graphic_draw(UgeGraphic *self)
{
  self->draw(); // 3
}

void test()
{
  UgePolyLine *poly = uge_poly_line_new();
  uge_graphic_draw((UgeGraphic *)poly); // 4
}
```

Obrázek 5.2: Příklad vytváření hierarchie dědičnosti pomocí struktur v jazyce C

Bézierovy křivky Pro kreslení křivek se nabízí využití funkcí poskytovaných knihovnou Cairo. To však odporuje požadavku na tenké rozhraní rendereru, který umí kreslit pouze úsečky. Další komplikací by byla skutečnost, že Cairo umožňuje kreslit křivky určené čtyřmi řídicími body, což je pro knihovnu a modul UGE nedostačující, je nutné umět kreslit křivky zadané libovolným počtem bodů.

Křivka je tedy vykreslována jako řada navazujících krátkých úseček nebo jinými slovy lomená čára. Body této čáry jsou vypočítány s využitím Bernsteinových polygonů, použitý algoritmus je podrobně a přesně popsán například v [15].

Poloha portů Výpočet polohy portu se liší pro objekty odvozené od elipsy a od polygonu. Vstupem pro výpočet polohy portu jsou dva body, první je bod v objektu, na němž má port ležet, druhý pak směr, ve kterém se bude na port napojovat hrana. První bod většinou splývá se středem objektu, jeho smysl v rozšiřujících modulech je popsán v 4.2.4. Poloha portu se pak zjednodušeně řečeno určí jako průsečík přímky spojující oba body určující port s obvodem objektu.

Při výpočtu polohy portu na elipse se vychází z analytických rovnic elipsy a přímky:

$$\frac{(x-x_0)^2}{a^2} + \frac{(y-y_0)^2}{b^2} = 1 \quad (5.1)$$

$$y = kx + q. \quad (5.2)$$

V rovnici elipsy jsou x_0, y_0 souřadnice středu elipsy, a, b pak velikost hlavní a vedlejší poloosy (v tomto pořadí). Rovnice přímky je zapsána ve směrnicovém tvaru. Pro usnadnění výpočtu je nejdříve počátek soustavy souřadnic umístěn do středu elipsy, následně pomocí dosazení rovnice přímky do rovnice elipsy jsou vypočítány souřadnice průsečíku přímky a elipsy. Tento postup není aplikován ve speciálních případech pokud přímka leží na hlavní nebo vedlejší poloose. Pak je poloha portu určena posunutím středu elipsy ve směru portu o délku hlavní nebo vedlejší poloosy. V případě kružnice je postup stejný, za velikosti poloos je dosazen poloměr kružnice.

U obecného polygonu je port umístěn na stranu, která se nachází nejblíže zadanému směru portu. To znamená, že se nejdříve určí vzdálenost středů všech stran polygonu k zadanému směru, pak se vybere nejbližší hrana a port se (pro jednoduchost) umístí na střed nalezené hrany. Tento postup je jednoduchý na implementaci a poskytuje vizuálně pěkný výsledek. Algoritmus pracuje i pro určení polohy portu na obdélníku a čtverci.

Přítomnost bodu v objektu Pro elipsu a kružnici je výpočet jednoduchý. Opět vychází z analytické rovnice elipsy 5.1. Přítomnost bodu v elipse získáme dosazením souřadnic zadaného bodu do rovnice za x a y a vyhodnocením nerovnosti.

U polygonu je situace o něco komplikovanější, jednoduchý analytický výpočet zde neexistuje. Autorem použitého algoritmu je Wm. Randolph Franklin a je podrobně popsán v [8].

5.2 UGE jako modul pro Python

Modul pro Python je implementován rovněž v jazyce C s využitím Python C API, které je dobře zdokumentováno v [11]. Zdrojové kódy modulu vycházejí z demonstračních příkladů popsaných v [9]. Rozdělení do souborů je jemnější než u implementace knihovny jazyka C. Je to zejména kvůli relativně velkému počtu řádků kódu potřebných k implementaci každého typu objektu a více typů v jednom souboru tak vedlo k nepřehlednému kódu. Pro každý typ objektu tedy existuje samostatný hlavičkový soubor a zdrojový soubor jazyka C.

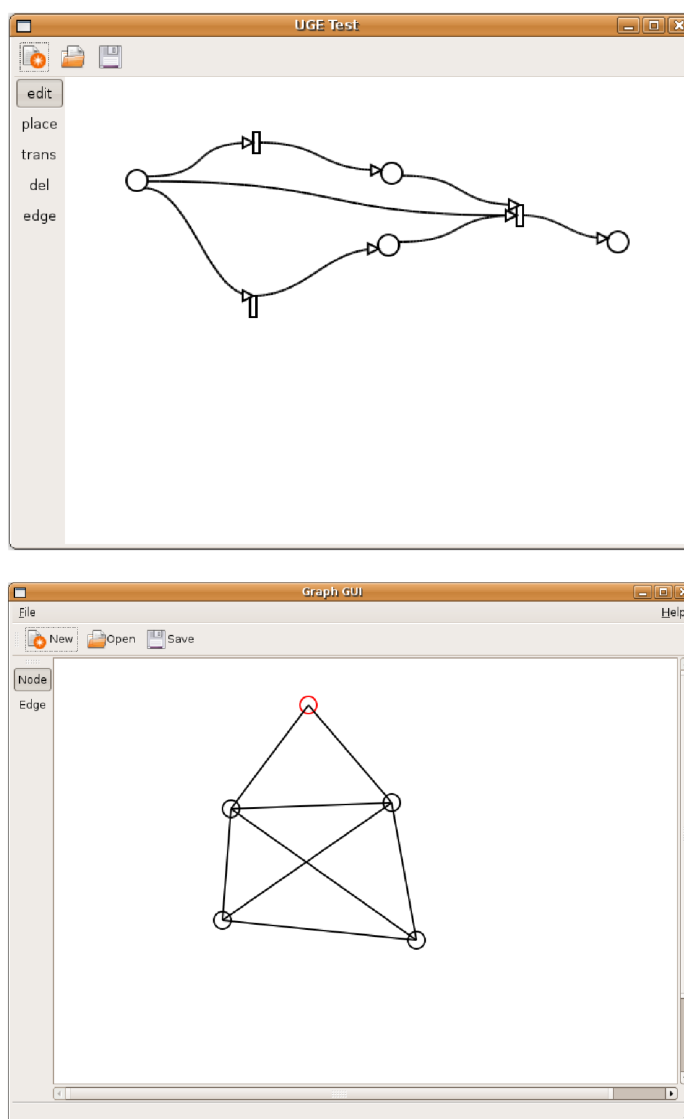
Implementace přesně odráží strukturu tříd popsanou na diagramech v kapitole 4.2. Úlohu společného předka všech tříd tvoří přímo `PyObject`, což je třída poskytovaná v Python C API.

`PyObject` je vůbec při vytváření modulu pro Python velice důležitý. Vše je odvozeno od tohoto typu – další objekty, čísla, seznamy. . . Funkce jazyka C implementující metody objektů vytvářeného modulu mají přesně stanovené rozhraní popsané v [11]. Návratové hodnoty funkcí využitelné potom v Pythonu jsou všechny referencí na `PyObject`. Stejně tak parametry jsou funkci předávány jako reference na `PyObject` a pro jejich zpracování jsou v API poskytnuty funkce převádějící je na běžné typy jazyka C.

Modul tedy vytváří rozraní pro volání knihovny UGE z prostředí jazyka Python a umožňuje rozšiřování knihovny. S rozšiřováním je však drobný problém, který popíšu na příkladě vykreslení grafického objektu. Při požadavku na vykreslení celého listu jsou postupně vykreslována jednotlivé grafické objekty na tomto listu. Každý grafický objekt přitom implementuje metodu pro své vykreslení. Mělo by tedy stačit přetížit tuto metodu a tím dojít ke správnému vykreslení. To se ale nestane, protože metoda pro vykreslení listu volá přímo funkci C API knihovny UGE, která volá implementaci vykreslení objektu v jazyce C, kterou není ve stávající verzi modulu možné nahradit kódem implementovaným v Pythonu. Aby vykreslení fungovalo správně, bylo by třeba přetížit i metodu vykreslení celého listu a volání C API funkce nahradit postupným voláním metody pro vykreslení jednotlivých objektů – což není příliš náročné a zdá se to jako rozumná cena za vyšší rychlost výchozího přímého volání C API. Z tohoto důvodu také není možné vytvářet přímo v Pythonu nové renderery – nefungovalo by vykreslení grafických objektů implementovaných v jazyce C.

Pro demonstraci a vysvětlení principů práce s modulem UGE je dostupný krátký tutoriál, který je otištěn v příloze B. Tutoriál pokrývá problematiku instalace knihovny a modulu, vytvoření demonstračního modulu rozšiřujícího knihovnu o možnosti práce s Petriho sítěmi a následně použití modulu v jednoduché PyGTK aplikaci. Při implementaci aplikace popisované v tutoriálu jsem čerpal informace z tutoriálu jazyka Python [10] a referenční příručky [2] a tutoriálu [4] modulu PyGTK.

Na obrázku 5.3 je zachyceno uživatelské rozhraní aplikace vytvářené v tutoriálu a pro srovnání také snímek rozhraní demonstrační aplikace semestrálního projektu implementované v jazyce C s využitím knihovny. Na obrázcích jsou zřetelně vidět nedokonalosti původní implementace, které byly později odstraněny.



Obrázek 5.3: Uživatelské rozhraní aplikace vytvářené v tutoriálu (nahoře) a demonstrační aplikace semestrálního projektu (dole)

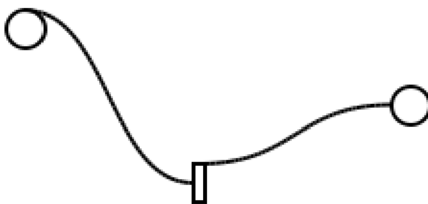
Kapitola 6

Testování

V kapitole věnované testování budou specifikovány testovací případy použité k ověření správné funkčnosti vytvořeného modulu. Všechny testy budu implementovány v jazyce Python, tím bude testován jak modul, tak knihovna využívaná modulem. Při testování byl kladen důraz zejména na výpočty polohy portů, přítomnost bodu uvnitř grafických objektů a složitější interakce popsané sekvenčními diagramy v kapitole 4.2. Zdrojové kódy testovacích případů jsou umístěny na přiloženém CD.

Výpočet polohy portů Tento test je zaměřen na ověření funkcí pro poskytnutí bodu na obvodu grafického objektu – tzv. portu. Při testu jsou použity celkem tři uzly reprezentované různými grafickými primitivy: kružnice a obdélník (což je specifický případ polygonu).

Na obrázku 6.1 je vidět výsledek prvního spuštění testu. Je zřejmé, že port na kružnici vlevo je určen špatně. Chyba v kódu způsobující nepřesnost byla nalezena a opravena, aktuální verze knihovny už touto chybou netrpí. Problém spočíval v umístění portu do špatného kvadrantu. Z analytické rovnice můžeme určit polohu portu ve dvou kvadrantech (pouze v kladném směru osy x). Pokud směr portu leží ve zbylých kvadrantech, je třeba jeho polohu upravit (násobením souřadnic vzhledem ke středu kružnice konstantou -1).

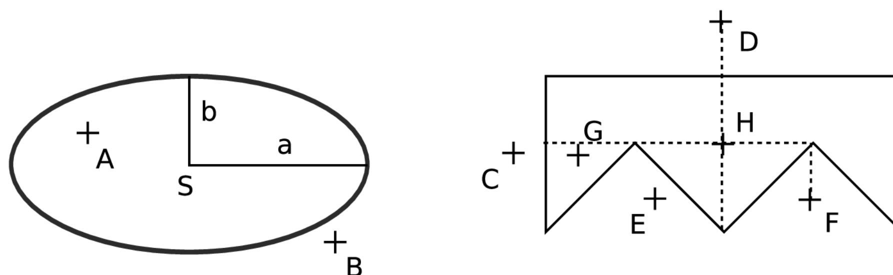


Obrázek 6.1: Chybný výsledek testování výpočtu polohy portu

Port na obdélníku spojující jej s kružnicí vpravo rovněž není umístěn úplně dokonale. Umístění však odpovídá algoritmu pro umístění portu na polygonu – krátká horní strana obdélníka je blíže ke kružnici. Toto chování by v budoucích verzích knihovny mělo být odstraněno (například zohledněním úhlu svíraného směrnicí portu a nejbližších hran polygonu – hrany svírající se směrnicí příliš malé úhly by nebyly pro umístění portu zvoleny).

Přítomnost bodu uvnitř grafického objektu Pro testování funkce ověřující přítomnost zadaného bodu byly zvoleny dva grafické objekty: elipsa a obecný nekonvexní polygon. Tím jsou pokryty

všechny ostatní případy (kružnice elipsou, čtverec a obdélník plygonem). Rozložení objektů a testovaných bodů je znázorněno na obrázku 6.2. Očekávaný výsledek testu: v elipse leží pouze bod A, uvnitř polygonu pak body G a H. Všechny ostatní body leží mimo. Konkrétní souřadnice bodů jsou uvedeny ve zdrojových kódech, důležitá je poloha bodů vzhledem k polygonu. Body D, F, H mají stejnou horizontální vzdálenost od levého horního rohu (souřadnice y) jako vrcholy polygonu. Bod H má i vertikální souřadnici x shodnou se dvěma vrcholy polygonu. Naopak body C, G, E leží mimo polohy vrcholů polygonu. Tím jsou testovány všechny varianty vzájemné polohy bodu a polygonu. Popsaný test byl modulem vyhodnocen úspěšně.



Obrázek 6.2: Konfigurace použitá pro testování, jestli bod leží uvnitř objektu

Kontrola konzistence grafu Podpora kontroly konzistence grafu pomocí definovaných pravidel je testována aplikací vyvinuté v rámci tutoriálu. Při pokusu spojit v GUI hranou místo s místem nebo přechod s přechodem nedojde k vytvoření uživatelem požadované hrany. Test byl tedy splněn úspěšně.

Změna polohy a smazání uzlu Testování změny polohy uzlu je opět prováděno pomocí GUI pro tovrbu jednoduchých Petriho sítí. Při uživatelském přesouvání grafických objektů představujících uzly grafu dochází podle očekávání k odpovídajícím přesunům napojených hran.

Při pokusu o smazání uzlu volbou příslušného nástroje a kliknutím na uzel se při prvních testech zdánlivě nic nedělo. Při další práci se pak aplikace chovala divně. Zkoumání příčiny problému vedlo ke zjištění, že sice dojde ke smazání uzlu a souvisejících hran z grafu, ale z listu nejsou odstraněny grafické objekty, které uzly a hrany reprezentují. Proto vizuálně nebylo odstranění objektů zaznamenáno. Popsaná chyba byla v modulu odstraněna, v implementaci však zůstal ještě jeden zatím neodstraněný problém, který způsobuje odstranění uzlu až na „druhý pokus“. Chyba je pravděpodobně ve špatném počítání referencí na objekt a bude odstraněna v následujících verzích modulu.

Kapitola 7

Závěr

Cílem tohoto diplomového projektu bylo zejména poskytnout programátorské vybavení pro snadnou práci s nejrůznějšími typy grafů, vytvořit knihovnu použitelnou při vývoji jakékoli aplikace pracující s libovolným typem grafu. Smyslem takovéto knihovny je zbavit programátora povinnosti soustředit se na vývoj komponenty pro práci s grafem a umožnit mu místo toho soustředit se na skutečný smysl programované aplikace. Pro co nejširší možnosti splnění tohoto poslání je hlavním implementačním jazykem knihovny jazyk C. Velmi důležitou součástí práce je ale vyvinutý modul pro jazyk Python, který využívá zmíněnou knihovnu a přináší možnosti Univerzálního grafického editoru do dynamického jazyka. Python je jazyk umožňující rapidní vývoj aplikací nebo rychlé vytvoření funkčního prototypu a to ve spojení s modulem pro práci sself.guigrafy může výrazně zjednodušit vývoj řady aplikací.

Vself.guinásledujících několika odstavcích na konkrétních příkladech představím možnosti praktického využití vytvořeného modulu nebo knihovny v různých typech aplikací zself.guirůzných oborů lidské činnosti.

Významnou oblastí využití různých typů grafů, schemat a diagramů je obor matematiky a informačních technologií. Zde je stále velký prostor pro vznik nástrojů jako jsou editory a hlavně simulátory konečných a zásobníkových automatů nebo Petriho sítí. Dalším využitím knihovny pro práci s grafy je vizualizace a simulace různých algoritmů teorie grafů – například hledání minimální kostry grafu, problém obchodního cestujícího. . . Zde by bylo zajímavé propojení knihovny UGE poskytující možnosti vizualizace sself.guiknihovnou Boost Graph Library [12] a jejími algoritmy implementovanými pomocí šablon.

Velkou skupinou grafů jsou různé typy UML diagramů. V této oblasti je poměrně silná konkurence, zvláště však v oblasti komerčních nástrojů. Nabídka opensource nástrojů už tak pestrá není, jedním z mála opravdu kvalitních je ArgoUML (který byl použit mimo jiné pro kreslení diagramů vself.guitéto práci). ArgoUML je však implementován v jazyce Java, což pro někoho může být výhoda, ale jiný s tím může mít velký problém. A když se podíváme například na programy pro prostředí GNOME, tak se nabízí poněkud ustrnulý Dia nebo mladý Gaphor. Je tedy vidět, že i tady je jistý prostor pro vývoj nových nástrojů.

Do skupiny UML diagramů nepatří E-R diagramy. Osobně mi ale zvláště tady více než kde jinde chybí kvalitní otevřený nástroj. Myslím, že svobodný program blížící se vlastnostmi komerčnímu DBVisualizeru, by své uživatele rozhodně nemusel hledat příliš dlouho. A přitom stačí naimplementovat některé algoritmy pro rozvržení grafu například podle [7], vytvořit typy uzlů a hran potřebné pro prezentaci E-R diagramu a vyvinout algoritmus pro převod relační databáze na graf. Není to sice práce na pár dní, ale ani na velmi dlouhou dobu.

Zajímavým využitím grafů je návrh hardware architektury. Toto téma již bylo zpracováváno v rámci dřívějších pracích na projektu UGE. Revize tehdejšího návrhu a jeho modifikace s využitím

vyvinutého modulu pro Python nebo knihovny s rozhraním jazyka C by mohlo vést k zajímavým výsledkům v rozumném čase. Pravděpodobně by však bylo třeba dopracovat a důkladněji otestovat ty části knihovny věnující se problematice portů a kontroly konzistence grafu, a dále vyvinout algoritmy pro vedení ortogonálních hran (odvozených od lomené čáry).

Podle mého názoru však z hlediska využití modulu UGE jednou z nejdůležitějších oblastí (ne-li tou úplně nejdůležitější) jsou aplikace, u nichž graf jako takový není hlavním cílem. Myslím tím například složitější grafické a multimediální filtry. Knihovny GEGL a GStreamer využívají principu skládání jednodušších filtrů za účelem komplikovaných transformací obrazu nebo videa. A takový složený filtr je reprezentován právě grafem, jednotlivé filtry představují uzly a hrany propojují vstupy a výstupy filtrů. Vývoj vizuálního editoru takového grafu by mohl GStreamer přiblížit programátorským komfortem k DirectShow, což je podobný framework od firmy Microsoft.

A v podobném duchu je možné pokračovat dál a dál: nejrůznější obchodní a organizační diagramy, Ganttovy grafy, myšlenkové mapy... Zajímavým a v dnešní době moderním využitím by mohlo být vytvoření verze použitelné pro tvorbu uživatelského rozhraní ve webovém prohlížeči s využitím JavaScriptu a technologie AJAX (Asynchronous Javascript And XML). Tím by vznikla konkurence komerční knihovny yFiles.NET zmiňované v úvodu práce.

Nicméně pro plné praktické nasazení potřebuje knihovna a hlavně modul pro Python urazit ještě kus cesty. Výsledkem implementační části této práce je funkční prototyp knihovny a modulu schopný dalšího rozšiřování a vylepšování. Pro zvýšení přidané hodnoty plynoucí zself.guivyužití modulu by bylo dobré doimplementovat alespoň komponenty pro vstup a výstup do formátu XML a export grafiky do obrazových formátů, zvláště protože ten by díky využité knihovně Cairo nebyl příliš pracný. Obě tyto funkce byly původně v návrhu naplánovány.

Vself.guiprvní fázi tohoto projektu (resp. v semestrálním projektu, který předcházel diplomovému) byly analyzovány zdrojové kódy původní implementace systému UGE a prvotní cíl byl v jejich využití. To bylo z důvodů uvedených v kapitole 3 zamítnuto, což s sebou ale přineslo větší časovou náročnost implementace knihovny, která tvoří základ modulu pro Python. To pak vyústilo ve vynechání implementace komponent pro vstup a výstup a fakt, že fázím dokumentace a testování bylo věnováno méně pozornosti, než by si zasloužily.

Projekt Univerzálního grafického editoru je myslím natolik zajímavý, že si zaslouží další pokračování. Krátkodobý plán je proto zveřejnit projekt na internetu pod svobodnou licencí a pokusit se získat programátory, kteří by měli zájem o využití projektu ve svých aplikacích, a nebo dokonce byli ochotni přispívat do vývoje projektu UGE samotného.

Literatura

[1] Cairo: A Vector Graphics Library. [online], revision for Cairo 1.6, checked 2008-05-16.
URL <`http://cairographics.org/manual/`>

[2] PyGTK 2.0 Reference Manual. [online], 2008, for PyGTK version 2.12.2, checked 2008-05-16.
URL <`http://www.pygtk.org/docs/pygtk/`>

[3] Brandes, U.; Eiglsperger, M.; Lerner, J.: GraphML Primer. [online], checked 2008-05-16.
URL <`http://graphml.graphdrawing.org/primer/graphml-primer.html`>

[4] Finlay, J.: PyGTK 2.0 Tutorial. [online], 2006, version 2.5, checked 2008-05-17.
URL <`http://pygtk.org/pygtk2tutorial/`>

[5] Golich, P.: *Univerzální grafický editor – konečné automaty a Petriho sítě*. Diplomová práce, FIT VUT, Brno, květen 2006.

[6] Jadrný, M.: *Univerzální grafický editor – V/V rozhraní*. Diplomová práce, FIT VUT, Brno, květen 2006.

[7] Košulič, J.: *Univerzální grafický editor – přehledné rozvržení grafu*. Diplomová práce, FIT VUT, Brno, květen 2006.

[8] O'Rourke, J.: Computer.graphics.algorithms Frequently Asked Questions, Section 2.03. [online], 2003, checked 2008-05-17.
URL <`http://www.faqs.org/faqs/graphics/algorithms-faq/`>

[9] van Rossum, G.: Extending and Embedding the Python Interpreter. [online], 2008, release 2.5.2, checked 2008-05-17.
URL <`http://docs.python.org/ext/`>

[10] van Rossum, G.: Python Tutorial. [online], 2008, release 2.5.2, checked 2008-05-17.
URL <`http://docs.python.org/tut/`>

[11] van Rossum, G.: Python/C API Reference Manual. [online], 2008, release 2.5.2, checked 2008-05-17.
URL <`http://www.python.org/doc/api/`>

[12] Siek, J. G.; Lee, L.-Q.; Lumsdaine, A.: *The Boost Graph Library: User Guide and Reference Manual*. Addison Wesley Professional, 2001.

[13] Urman, M.: Cairo Tutorial for Python Programmers. [online], checked 2008-05-16.
URL <`http://www.tortall.net/mu/wiki/CairoTutorial`>

[14] Varga, L.: *Univerzální grafický editor – návrh HW architektury*. Diplomová práce, FIT VUT, Brno, květen 2006.

[15] Žára, J.; Beneš, B.; Sochor, J.; aj.: *Moderní počítačová grafika*. Brno: Computer Press, a.s., 2004, ISBN 80-251-0454-0, 185-191 s.

Dodatek A

Dokumentace

Kompletní dostupná dokumentace knihovny a modulu je umístěna na přiloženém CD. K dispozici je dokumentace knihovny s rozhraním jazyka C ve formátu HTML vygenerovaná ze zdrojových kódů pomocí nástroje Doxygen. Zdrojové kódy jsou komentovány velmi stručně a v angličtině. Dokumentace je umístěna v adresáři `libuge/doc/html`.

Dále je na CD umístěna stručná dokumentace rozhraní modulu UGE pro jazyk Python, rovněž ve formátu HTML. Tato dokumentace je umístěna vself.guiadresáři `pyuge/doc`.

Velice důležitou dokumentací je také tutoriál vysvětlující použití modulu v jazyce Python na konkrétním příkladě. Tutoriál se nachází na CD v příslušném adresáři a je otištěn v následující příloze.

Dodatek B

Tutoriál

Tutoriál vysvětluje na praktické ukázce možnosti použití modulu UGE. Je rozdělen na tři části. V první části jsou uvedeny knihovny potřebné pro správnou funkci modulu a je uveden postup instalace modulu. Druhá část ukazuje vytvoření modulu specializujícího se na konkrétní typ grafu. Ve třetí části je pak tento modul využit pro vytvoření jednoduché aplikace s grafickým uživatelským rozhraním. Na přiloženém CD je umístěn kompletní tutoriál ve formátu HTML spolu s kompletním zdrojovým kódem výsledné aplikace.

B.1 Instalace

Modul je poskytován ve formě zdrojových kódů, které je třeba před instalací a použitím zkompilovat. V operačním systému musí být proto nainstalovány nástroje potřebné pro překlad programu v jazyce C – `gcc`, `automake` a `pkg-config`. Dále bude popisována instalace v prostředí operačního systému GNU/Linux. Všechny používané knihovny jsou však dostupné i na jiných platformách, takže překlad například na OS Microsoft Windows by neměl vyžadovat úpravy zdrojových kódů.

Pro úspěšnou kompilaci knihovny a modulu musí být v systému nainstalovány následující balíky (včetně hlavičkových souborů):

- GTK+ verze 2.8 a vyšší (testováno na 2.12)
- Python 2.x – verze 2.4 a vyšší (testováno na 2.5.1)
- PyGTK verze 2.8 a vyšší (testováno na 2.12)

Minimální verze balíků jsou určeny podle dostupné dokumentace odkazované v seznamu literatury. Tato konfigurace však není otestována.

Sestavení a instalace knihovny Knihovna UGE se nachází v adresáři `libuge`. Sestavení se provede zadáním příkazu `make` v tomto adresáři. Po úspěšném překladu se v adresáři objeví dynamická knihovna – soubor `libuge.so`. Pro použití této knihovny v programech je nutné při překladu použít přepínač `-luge`. Knihovna `libuge.so` přitom musí být vself.guiv adresáři, kde ji linker dokáže najít, stejně tak musí být překladači dostupné hlavičkové soubory knihovny.

Toho můžeme dosáhnout zkopírováním souboru `libuge.so` do adresáře `/usr/local/lib` nebo `/usr/lib`. Alternativní postup je předání cesty k adresáři s knihovnou linkeru při překladu a potom před spuštěním programu nastavením proměnné prostředí `LD_LIBRARY_PATH`.

Sestavení a instalace modulu pro Python Modul pro Python je umístěn v adresáři pyuge. Sestavení proběhne opět po zadání příkazu `make`. Přitom se počítá s tím, že v adresáři s relativní cestou `./../libuge` je dostupný soubor `libuge.so` a hlavičkové soubory knihovny UGE.

Překladem vznikne soubor `ugemodule.so`. Ten je třeba umístit do pracovního adresáře programu v Pythonu, který bude modul využívat.

V Pythonu jsou potom objekty implementované v modulu UGE dostupné po zadání příkazu `import uge`.

B.2 Vytvoření rozšiřujícího modulu – práce s Petriho sítěmi

V této části tutoriálu vytvoříme jednoduchý rozšiřující modul pro práci s jednoduchými Petriho sítěmi. Tento modul bude tvořen několika třídami, které budou rozšiřovat modul UGE způsobem naznačeným na obrázku 4.3. Celý modul bude naprogramován v Pythonu.

- `PN_Net` bude představovat samotnou Petriho síť a bude obsahovat implementaci kontroly konzistence grafu
- `PN_Place` a `PN_Transition` představují místa a přechody sítě, zároveň zapouzdřují svou grafickou reprezentaci
- `PN_Edge` představuje hranu grafu a rovněž se autonomně stará o vlastní vizuální reprezentaci

Zdrojový kód umístíme do souboru nazvaného `pn.py`. Začneme implementací místa a přechodu. Nejdříve ale musíme naimportovat modul UGE.

```
#!/usr/bin/env python

import uge

class PN_Place(uge.Node):
  'Petri-net place'

  def __init__(self, graph, x, y):
    p = uge.Circle(graph.sheet, x, y, 10)
    super(PN_Place, self).__init__(graph, p)

class PN_Transition(uge.Node):
  'Petri-net transition'

  def __init__(self, graph, x, y):
    p = uge.Rectangle(graph.sheet, x-3, y-10, 6, 20)
    super(PN_Transition, self).__init__(graph, p)
```

Obě právě vytvořené třídy jsou odvozeny od řídy `uge.Node` a předefinovávají metodu `__init__`, aby mohly automaticky vytvořit svou grafickou reprezentaci – kružnici nebo obdélník.

Pokračujeme vytvořením třídy pro hranu grafu:

```
class PN_Edge(uge.Edge):
  'Petri-net edge'

  def __init__(self, graph, a, b):
    port_start = a.get_port(b, 0, 0, b.primitive.x, b.primitive.y)
    port_end = b.get_port(a, 0, 0, a.primitive.x, b.primitive.y)
    p = uge.Bezier(graph.sheet, [
        port_start, [(port_start[0]+port_end[0])/2, port_start[1]],
        [(port_start[0]+port_end[0])/2, port_end[1]], port_end])
    p.set_arrows('end')
    p.keep_angle = True
    super(PN_Edge, self).__init__(graph, a, b, p)
```

Zde je inicializace nepatrně složitější. Vytváříme Bézierovu křivku s konci umístěnými do portů příslušících spojovaným uzlům grafu. Při dotazování na polohu portu je první dvojice souřadnic (druhý a třetí argument metody `get_port`) nastavena na $[0,0]$. Tyto souřadnice určují místo v objektu, na které uživatel kliknul. To však v našem případě polohu portu neovlivní a můžeme si proto dovolit nesmyslné zadání (jinak by tomu bylo například u uzlu představujícího mikroprocesor s řadou vývodů – tam by naopak bylo důležité, poblíž kterého vývodu uživatel kliknul myší).

Samotná Bézierova křivka je pak určena čtyřmi body a zde slouží pro demonstraci nepatrně hezčího přístupu než použít „obyčejné" úsečky. Křivka je zakončena šipkou, která je zapnuta voláním metody `set_arrows('end')` a vlastnost `keep_angle` říká křivce, že směr šipky se při pohybu koncového bodu nemění (to znamená s koncovým bodem se přesouvá i nejbližší další řídicí bod křivky).

Posledním úkolem této části je vytvoření třídy pro samotnou Petriho síť:

```
class PN_Net(uge.Graph):
  'Petri-net class'

  def edge(self, a, b):
    if (type(a) != type(b)):
      return PN_Edge(self, a, b);
    else:
      return None
```

Třída `PN_Net` přetěžuje metodu `edge`, kde je zakázáno spojení dvou uzlů grafu stejného typu (pro připomenutí v Petriho síti není možné spojit hranou dvě místa nebo dva přechody).

B.3 Použití modulu UGE v jednoduché aplikaci s GUI

V poslední části tutoriálu vytvoříme jednoduchou aplikaci sself.guis grafickým uživatelským rozhraním v PyGTK využívající modul implementovaný v předchozí kapitole. Aplikace bude umožňovat vkládání míst a přechodů a jejich spojování hranami.

Rozhraní aplikace bude opravdu jednoduché a vysvětlení jeho tvorby sahá mimo rámec tohoto tutoriálu (pro vysvětlení této problematiky prostudujte PyGTK tutoriál a dokumentaci dostupnou na internetu). V levé části okna bude panel s nástroji, v pravé pak plátno pro kreslení Petriho sítě.

Uživatel bude mít k dispozici tyto nástroje:

- **edit** pro změnu polohy uzlů
- **place** pro vložení nového místa (kliknutím na plátno)
- **trans** pro vložení nového přechodu
- **edge** pro vytvoření hrany (stisknutím tlačítka myši nad jedním uzlem, tažením a uvolněním nad jiným uzlem)
- **del** pro odstranění místa nebo přechodu

Zdrojový kód umístíme do souboru `pn_gui.py`. Začneme opět importem potřebných modulů:

```
#!/usr/bin/env python

import pygtk
pygtk.require('2.0')
import gtk, gobject, cairo

import uge
from pn import *
```

Pro přepínací tlačítka pro volbu nástrojů vytvoříme vlastní třídu. Každý nástroj bude identifikován svým názvem a podle tohoto názvu pak bude při uživatelských akcích nad plátnem volena správná akce (určitě existuje technologicky dokonalejší řešení, ale o to v tomto tutoriálu nejde).

Tlačítko při změně stavu na aktivní deaktivuje všechny ostatní nástroje, při pokusu o deaktivaci aktivního nástroje jiným způsobem než aktivací jiného nástroje je nástroj opět aktivován (zjednodušeně vždy je aktivní právě jeden nástroj).

```
class PN_GuiTool(gtk.ToggleToolButton):
  def __init__(self, gui, label):
    super(PN_GuiTool, self).__init__()
    self.set_label(label)
    self.gui = gui
    self.connect('toggled', self.on_toggled)
```

```
  def on_toggled(self, tb):
    if (tb.get_active()):
      self.gui.tool = tb.get_label()
      for k,b in self.gui.toggles.iteritems():
        if b != tb:
          b.set_active(False)
    elif (tb.get_label() == self.gui.tool):
      tb.set_active(True)
```

Pro kreslení nám poslouží widget `gtk.DrawingArea`. Na jeho základě vytvoříme vlastní widget pro kreslení Petriho sítě. Při inicializaci nesmíme zapomenout na nastavení masky událostí, následuje inicializace proměnných pro obsluhu uživatelských akcí (`selected` a `drag_start`). Vlastnost uge obsahuje objekt třídy `PN_Gui` popsané dále. Nakonec na posledních dvou řádcích vytvoříme list pro grafické objekty a samotný graf Petriho sítě.

```
class PN_Canvas(gtk.DrawingArea):
  __gsignals__ = {
    'expose-event': 'override',
    'button-press-event': 'override',
    'button-release-event': 'override',
    'motion-notify-event': 'override'
    }

  def __init__(self, gui):
    super(PN_Canvas, self).__init__()
    self.add_events(gtk.gdk.BUTTON_PRESS_MASK
      | gtk.gdk.BUTTON_RELEASE_MASK | gtk.gdk.POINTER_MOTION_MASK)
    self.gui = gui
    self.selected = None
    self.drag_start = [0,0]
    self.sheet = uge.Sheet(640,480)
    self.graph = PN_Net(self.sheet)
```

Dále musíme implementovat samotnou obsluhu signálů. Začneme bez váhání vykreslením grafu – signál `expose-event`.

```
  def do_expose_event(self, event):
    self.draw(*self.window.get_size())
```

```
def redraw_canvas(self):
  if self.window:
    alloc = self.get_allocation()
    rect = gtk.gdk.Rectangle(0, 0, alloc.width, alloc.height)
    self.window.invalidate_rect(rect, True)
    self.window.process_updates(True)

def draw(self, width, height):
  cr = self.window.cairo_create()
  cr.set_source_rgb(1.0, 1.0, 1.0)
  cr.rectangle(0, 0, width, height)
  cr.fill()
  renderer = uge.GtkRenderer(self)
  self.sheet.draw(renderer, 0, 0, width, height)
```

Při vykreslování nejdříve vyplníme celou plochu bílým pozadím a potom pro samotné vykreslení grafu používáme `uge.GtkRenderer`. Metoda `redraw_canvas` je volána při obsluze ostatních událostí, když dojde k nějaké změně grafu a je třeba překreslit plátno.

Pokračujeme v implementaci zbylých signálů týkajících se už beze zbytku práce s myší: stisknutí tlačítka, pohyb kurzoru a uvolnění tlačítka. Při stisknutí tlačítka pouze vyhledáme, jestli souřadnice myši leží v některém grafickém objektu a pokud ano, zapamatujeme si jej:

```
def do_button_press_event(self, event):
  self.selected = None
  for i~in self.sheet.primitives:
    if i.contains(event.x, event.y):
      self.selected = i
      self.drag_start = [event.x, event.y]
```

Pohyb myší je zajímavý jenom u nástroje **edit**, kdy měníme polohu vybraného objektu.

```
def do_motion_notify_event(self, event):
  if (self.gui.tool == 'edit' and self.selected != None):
    self.selected.translate(
      event.x - self.drag_start[0], event.y - self.drag_start[1])
    self.drag_start = [event.x, event.y]
    self.redraw_canvas()
```

Nejvíce práce se děje až při uvolnění tlačítka myši. To je místo pro vytváření všech nových objektů grafu. Ve zdrojovém kódu jsou v tomto pořadí: nová hrana, nové místo, nový přechod a nakonec smazání objektu. Při mazání se navíc kontroluje, jestli bylo tlačítko uvolněno nad stejným objektem, nad kterým bylo stistknuto.

```
def do_button_release_event(self, event):
  if (self.gui.tool == 'edge'):
    if (self.selected != None):
      for i~in self.sheet.primitives:
        if i.contains(event.x, event.y):
          self.graph.edge(self.selected.graph_object, i.graph_object)
          break
  elif (self.gui.tool == 'place'):
    PN_Place(self.graph, event.x, event.y)
  elif (self.gui.tool == 'trans'):
    PN_Transition(self.graph, event.x, event.y)
  elif (self.gui.tool == 'del'):
    if (self.selected != None):
      for i~in self.sheet.primitives:
        if i.contains(event.x, event.y) and i~== self.selected:
          grobj = i.graph_object
          if isinstance(grobj, PN_Edge):
            self.graph.remove_edge(grobj)
          else:
            self.graph.remove_node(grobj)
  self.selected = None
  self.redraw_canvas()
```

Vytvořená aplikace je značně jednoduchá a obsahuje jistě řadu nedostatků. Například by bylo vhodné pro reprezentaci místa seskupit kružnici s informací o počtu tokenů v místě. Rovněž nedokonalé je, že při vytváření hrany uživatel až do uvolnění tlačítka myši nedostává žádnou vizuální odezvu. Cílem tutoriálu ale bylo pouze na krátké ukázce seznámit se základy použití knihovny a implementace uvedených vlastností, ač nejsou náročné na programování, by zdrojový kód prodloužila a tutoriál by ztratil na stručnosti a přehlednosti.

Dodatek C

Obsah přiloženého CD

Soubory na přiloženém CD jsou umístěny v několika adresářích:

- libuge – zdrojové kódy knihovny UGE s rozhraním jazyka C
- libuge/doc/html – dokumentace knihovny vygenerovaná doxygenem
- pyuge – zdrojové kódy modulu pro Python
- pyuge/doc – HTML dokumentace rozhraní modulu
- tutorial – tutoriál modulu ve formátu HTML
- tutorial/src – zdrojové kódy aplikace z tutoriálu
- test – zdrojové kódy testovacích případů
- text – zdrojové kódy technické zprávy včetně obrázků
- model – UML diagramy použité v technické zprávě ve formátu ArgoUML
- uge.pdf – text této vytištěné zprávy
- README – tento popis adesářů