České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra kybernetiky

DIPLOMOVÁ PRÁCE

Návrh SW platformy pro formulaci a řešení optimalizačních úloh v energetice

Autor: Bc. Josef Hák

Vedoucí práce: Ing. Michal Dvořák

Praha, 2012

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 9. května 2012 Josef Hák

Poděkování

Děkuji panu Ing. Michalu Dvořákovi za užitečné rady a vedení mé diplomové práce. Dále bych rád poděkoval svým rodičům Heleně a Josefovi za jejich všestrannou podporu po celou dobu studia.

Název práce: Návrh SW platformy pro formulaci a řešení optimalizačních úloh v energetice

Autor: Bc. Josef Hák

Katedra: Katedra kybernetiky

Vedoucí bakalářské práce: Ing. Michal Dvořák

e-mail vedoucího: michal.dvorak@fel.cvut.cz

Abstrakt Práce má za cíl určit SW vybavení pro formulaci a řešení MILP optimalizačních problémů, zejména pak pro optimalizaci provozu evropské přenosové soustavy. Důraz je kladen na rychlost a snadnou použitelnost nové platformy i pro komerční účely. Dosud užívaný program Matlab je pro danou úlohu nevhodný pro svou pomalost a vysokou cenu. Po analýze dostupných možností je zvolena varianta založená na použití .NET API solverů Gurobi a Cplex. Doporučeným vývojovým prostředím je Visual C# Express. Umožňuje programy snadno vyvíjet, kompilovat a ladit a je dostupný zdarma i pro komerční užití. Jádro práce tvoří návrh a implementace frameworku JD. Ten tvoří jednotné rozhraní solverů a umožňuje snadno formulovat optimalizační úlohy pomocí matic. V rámci testování nové platformy je implementován zjednodušený model přenosové soustavy. Test je zaměřen především na rychlost zpracování úloh velké dimenze (řádově 10^6 optimalizačních proměnných). Z výsledků testu plyne, že nová platforma je o více než 90 % rychlejší než původní. Všechny její komponenty jsou navíc dostupné zdarma.

Klíčová slova: optimalizace, vývojové prostředí, modelovací nástroj

---

Title: Environment for Modelling and Optimization in Power Industry

Author: Bc. Josef Hák

Department: Department of Cybernetics

Supervisor: Ing. Michal Dvořák

Supervisor's e-mail address: michal.dvorak@fel.cvut.cz

Abstract This work aims to determine the SW environment for formulating and optimization of MILP problems, especially for optimizing the operation of the transmission system. The emphasis is on speed, ease of use and usability for commercial purposes. Matlab program, which is used so far, is unsuitable for its slowness and high cost. After analyzing the options the one based on .NET API of Gurobi and Cplex use is selected. The recommended development environment is Visual C# Express. This makes it easy to develop, compile and debug programs and it is available for commercial use. The core thesis is the JD framework design and implementation. It provides the universal solver interface and makes it easy to formulate optimization problems using matrices. For testing purposes a simplified model of the transmission system is implemented. The test focuses primarily on the speed of large dimension task processing (about 10^6 optimization variables). The test results show that the new platform is more than 90 % faster than the original one. In addition, all its components are available for free.

Keywords: optimization, integrated development environment (IDE), modelling tool

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Bc. Josef H á k

Studijní program: Kybernetika a robotika (magisterský)

Obor: Robotika

Název tématu: Návrh SW platformy pro formulaci a řešení optimalizačních úloh v energetice

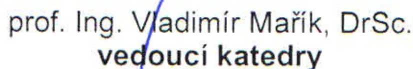
Pokyny pro vypracování:

1. Seznamte se se softwarem založeným na platformě Matlab, který je v současné době používán pro řešení optimalizačních problémů v energetice – modelu evropské přenosové soustavy a plánování provozu a obchodu tepláren. Shrňte nedostatky použité platformy.
2. Prostudujte dodanou literaturu.
3. Najděte a popište další platformy vhodné pro řešení uvedených optimalizačních problémů.
4. Platformy porovnejte na základě definovaných požadavků (rychlost, uživatelský komfort atd.) a vyberte užší okruh pro podrobnější vyhodnocení.
5. Detailně porovnejte vybrané platformy s důrazem na tu část, která bude použita pro formulaci optimalizačního problému.
6. Zvolte platformu, která bude použita. Výběr zdůvodněte.
7. Popište, jaká bude struktura softwaru založeného na zvolené platformě - slovně a schématem.
8. Ukažte možnosti platformy na zjednodušeném případu modelu evropské přenosové soustavy s velkou dimenzí (500 uzlů, 1500 linek, 300 elektráren, 2500 bloků, horizont 168 čas. vzorků).

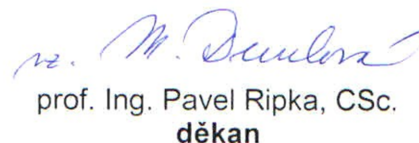
Seznam odborné literatury: Dodá vedoucí práce.

Vedoucí diplomové práce: Ing. Michal Dvořák

Platnost zadání: do konce letního semestru 2012/2013



prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry



prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 4. 1. 2012

Obsah

Abstrakt IV

Zadání práce VI

1 Úvod 1
1.1 Motivace a cíle práce . . . 1
1.1.1 Úloha typu MILP . . . 1
1.1.2 Model přenosové soustavy . . . 2
1.2 SW platforma . . . 3
1.2.1 Programovací jazyk . . . 3
1.2.2 Vývojové prostředí . . . 3
1.2.3 Modelovací nástroj . . . 4
1.3 Původní platforma . . . 4
1.3.1 Prostředí Matlab . . . 5
1.3.2 Yalmip . . . 5
1.3.3 Výhody . . . 5
1.3.4 Nevýhody . . . 5

2 Analýza dostupných řešení 7
2.1 Řešení posuzovaná v 1. fázi . . . 7
2.1.1 Metody posuzování . . . 7
2.1.2 Seznam variant . . . 7
2.1.3 Vyhodnocení . . . 10
2.2 Řešení posuzovaná ve 2. fázi . . . 11
2.2.1 Metody posuzování . . . 11
2.2.2 Seznam variant . . . 12
2.2.3 Vyhodnocení . . . 14

3 Zvolená varianta 17
3.1 Struktura a vlastnosti nové platformy . . . 17
3.1.1 Prostředí .NET . . . 17
3.1.2 Gurobi API pro .NET . . . 18
3.1.3 Cplex API pro .NET . . . 21
3.1.4 Visual C# Express . . . 23
3.2 Příklad vývoje aplikace . . . 25
3.2.1 Založení projektu . . . 25
3.2.2 Přidání knihovny do referencí . . . 25
3.2.3 Nastavení pro překlad . . . 26
3.2.4 Přeložení programu . . . 27
3.2.5 Debugování programu . . . 27

4 Framework JD 31
4.1 Návrh a implementace . . . 31
4.1.1 Účel frameworku . . . 31
4.1.2 Objektová struktura . . . 32

4.1.3 Dokumentace 35
4.2 Testování knihovny 36
4.2.1 Jednoduché příklady 37
4.2.2 Implementace modelu přenosové soustavy 38
4.2.3 Výsledky testů 40

5 Závěr **43**

Literatura **47**

Přiložené CD **49**

1 Úvod

Cílem této práce je nalézt SW platformu (1.2) pro řešení MILP (1.1.1) optimalizačních problémů. Důraz je kladen na využitelnost při optimalizaci provozu evropské přenosové soustavy. Při výběru hraje roli především rychlost zpracování úlohy, zohledňuje se ale také pracovní komfort ze strany programátora (složitost kódu, vývojové prostředí atd.) a v neposlední řadě finanční stránka věci. Platforma má být vhodná i pro vývoj komerčních aplikací. Dosud používaný program Matlab [2] je z hlediska uvedených požadavků nevyhovující.

První dvě kapitoly (včetně této) pojednávají o výběru samotné platformy, tři následující se pak zabývají jejím podrobným popisem, návodem k použití a testováním.

V této kapitole je nejprve popsána motivace práce. Druhá část kapitoly má za cíl objasnit, co vše zde zahrnuje pojem SW platforma. Ve třetí části se zaměřím na dosud používaný nástroj a především jeho nedostatky, které chceme výběrem nového nástroje eliminovat.

Předmětem druhé kapitoly je analýza všech možností, které lze při výběru nové platformy uvažovat. Jednotlivé varianty jsou posouzeny z hlediska stanovených kritérií a ve dvou fázích je proveden výběr nejlepší varianty.

Třetí kapitola má za cíl rozbor a podrobný popis vybrané platformy. Zahrnuje rovněž návod, jak na nové platformě vytvořit optimalizační aplikaci.

Ve čtvrté kapitole se zabývám návrhem a implementací doplňujícího modulu – frameworku pro matematický popis optimalizačních úloh a plnící funkci jednotného rozhraní solverů. Rovněž uvedu příklad použití.

V páté závěrečné kapitole shrnu výsledky práce.

1.1 Motivace a cíle práce

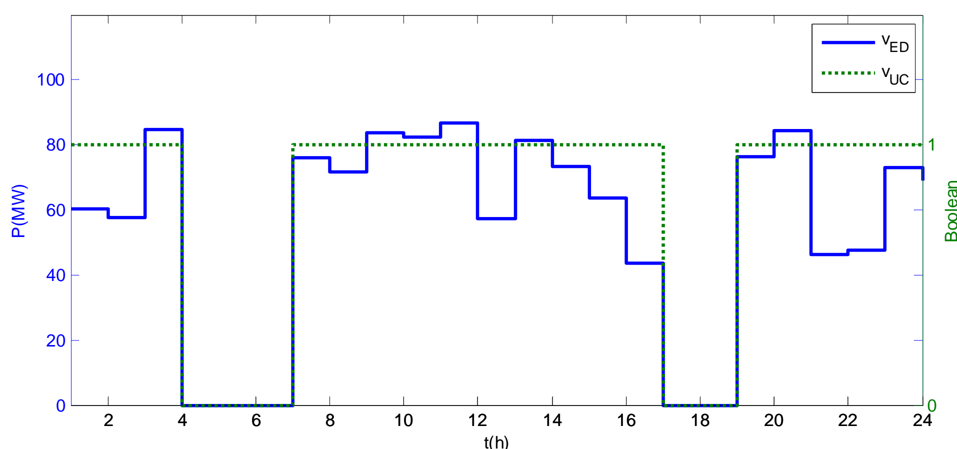
Zadání práce vychází z optimalizační úlohy – modelu evropské přenosové soustavy. Modelování je zde řešeno jako optimalizační úloha typu MILP. V této podkapitole se tedy nejprve zmíním o tomto druhu úloh obecně a následně popíšu konkrétní příklad – model přenosové soustavy.

1.1.1 Úloha typu MILP

Jedná se o optimalizační úlohu, kdy chceme minimalizovat cílovou funkci $f(\mathbf{x}) = \mathbf{c} \cdot \mathbf{x}$. Přitom platí, že $\mathbf{x} \in X$ je vektor přípustných řešení, který náleží do přípustné množiny X. Každý takový vektor zároveň splňuje podmínku $A\mathbf{x} \leq \mathbf{b}$. Prvky vektoru $\mathbf{x}$ jsou optimalizační proměnné. V případě úloh typu LP (Linear Programming) jsou všechny tyto proměnné reálné. Pro přesnější popis některých systémů se ale někdy zavádějí celočíselné optimalizační proměnné, a úlohu pak řadíme do kategorie tzv. smíšeně celočíselného lineárního programování – MILP (Mixed-Integer-Linear-Programming). Zavádění dalších nelinearit, jako např. kvadratické cílové funkce, vede k dalšímu růstu náročnosti úlohy. Proto se v této práci zaměřuji pouze na řešení LP a MILP problémů.

1.1.2 Model přenosové soustavy

Hlavní motivací pro tuto práci je modelování provozu evropské přenosové soustavy. Ta se modeluje jako síť uzlů spojených linkami. V uzlech jsou umístěny výrobní zdroje (elektrárny) a zátěže. Každá zátěž je reprezentována průběhem (hodinovou časovou řadou) požadovaného výkonu. Výrobní zdroj je pak omezen svým výkonovým rozsahem, náklady na výrobu a dalšími vlastnostmi. Výrobní zdroje mají svým koordinovaným provozem pokrýt požadovaný výkon zátěží. Informace o struktuře a vlastnostech soustavy včetně zátěží a výrobních zdrojů tvoří zadání optimalizační úlohy. Cílovou funkci $f(\mathbf{x})$ zde představuje funkce celkových nákladů, kterou chceme minimalizovat. Průběhy dodávaných výkonů od jednotlivých výrobních zdrojů pak tvoří hledané řešení úlohy $\mathbf{x}$. Řeší se při tom úloha nasazování UC (unit commitment) i řízení provozu ED (economic dispatch) jednotlivých zdrojů. To vede k úloze MILP, protože je nutné zavést binární, tedy celočíselné, optimalizační proměnné. Každý zdroj je modelován po jeho jednotlivých výrobních blocích, kterých může mít obecně n. Za předpokladu modelování soustavy v horizontu jednoho týdne při rozlišení na jednu hodinu, přináší každý blok do úlohy 168 reálných optimalizačních proměnných, které reprezentují vektor jeho hodinového provozu $\mathbf{v_{ED}}$. Protože ale neřešíme jen úlohu ED ale také UC, je třeba přidat ještě vektor $\mathbf{v_{UC}}$ binárních optimalizačních proměnných stejného rozměru, který indikuje v každé hodině, zda je blok v provozu či, nikoli. Pro dokreslení uvádím obrázek 1.1 ukazující příklad možného provoz bloku v horizontu dvaceti čtyř hodin. Je z něj patrný i vztah zmíněných vektorů $\mathbf{v_{ED}}$ a $\mathbf{v_{UC}}$.



Obrázek 1.1: Příklad provozu bloku v horizontu 24 hodin

Další složkou úlohy je distribuce vyrobené energie mezi uzly skrze linky. K uzlu může být připojeno n linek a uzel skrze každou linku může výkon odebírat nebo dodávat do uzlu sousedního. Přenos v síti je pak reprezentován toky energie jednotlivými linkami. Tento přenos je modelován metodou DC load flow, která je popsána například v [1]. To přináší další vektory optimalizačních proměnných. Při modelování evropské přenosové soustavy počítáme s přibližně 3000 bloky. To vede na MILP úlohu o řádově 10^6 optimalizačních proměnných. Fakt, že se nejedná jen o proměnné reálné, ale vektor řešení $\mathbf{x}$ tvoří z velké části i proměnné binární, má za následek, že prostor řešení X je mnohem větší, než by tomu bylo v případě problému typu LP. Při použití n binárních proměnných, totiž roste prostor řešení 2^nkrát. V našem případě tedy prostor

řešení roste 2^{10^5} krát a více. Úlohu také komplikuje strukturální složitost modelu. Vyhodnocované optimalizační proměnné reprezentují parametry prvků sítě. Vlastnosti těchto prvků v úloze reprezentuje sada omezení, kterou lze shrnout do maticové nerovnice $A\mathbf{x} \leq \mathbf{b}$.

Řešení úlohy takového rozsahu je dnes možné za použití výpočetní techniky. Nalezení vhodné SW platformy umožňující úlohy podobného typu formulovat a řešit je těžištěm této práce.

1.2 SW platforma

V této podkapitole bych rád objasnil, co tento text uvažuje pod pojmem SW platforma. Uvedu, co všechno zde tento pojem zahrnuje a popíši význam jednotlivých prvků, které sem patří.

SW platforma zahrnuje programové vybavení pro tvorbu aplikací schopných řešit optimalizační úlohy. Tvoří ji tři prvky – programovací jazyk, vývojové prostředí a modelovací nástroj.

1.2.1 Programovací jazyk

Volba jazyka má dopad jak na rychlost výsledného programu, tak na náročnost implementace. Kompilované jazyky (např. C++) jsou zpravidla náročnější na implementaci, protože mají složitější syntaxi. Umožňují ale vytvořit rychlejší programy. U interpretovaných jazyků (např. Java či Python) je tomu naopak.

Při implementaci výše popsaného modelu (1.1.2) se nabízí metodika objektově orientovaného programování OOP. Prvky s logickou závislostí i operace nad nimi se zde slučují do tzv. objektů. I velice rozsáhlý program si tak může zachovat přehlednou a logickou strukturu.

Aby nebylo nutné veškerou funkcionalitu aplikace implementovat zcela od nuly, lze při vývoji použít hotové kusy kódu či již zkompilované knihovny, které danou funkcionalitu zajišťují. Získáme tak potřebnou sadu funkcí i objektů např. pro práci s databází či pro vytvoření grafického uživatelského rozhraní. Takové specializované sadě funkcí a tříd říkáme framework. Pomocí frameworků lze realizovat velkou část funkcionalit naší aplikace.

Z předchozích odstavců je patrné, že programovací jazyk hraje při vývoji aplikace stěžejní roli.

1.2.2 Vývojové prostředí

Vývoj aplikace může značně usnadnit psaní kódu ve specializovaném editoru – tzv. vývojovém prostředí neboli IDE (Integrated Development Environment). Populární vývojová prostředí jsou např. Visual Studio [6], Eclipse IDE [7] nebo NetBeans IDE [8]. Dobré IDE poskytuje programátorovi řadu užitečných funkcí. Uvedu zde ty, které považuji za obzvláště užitečné.

- **Integrovaný překladač (Compiler)** – Umožňuje překládat program prostřednictvím IDE.
- **Ladění programu (Debugging)** – Velice důležitá funkcionalita. IDE umožňuje kód kompilovat a spouštět v ladícím režimu. Program je pak možné v libovolném místě zastavit a vidět aktuální hodnoty všech proměnných, následně lze v běhu programu pokračovat po krocích. Tímto způsobem lze odstranit systémové chyby programu. Vývoj rozsáhlého projektu je bez této možnosti nemyslitelný.
- **Automatické doplňování (Autocomplete)** – Tato vlastnost usnadňuje vlastní psaní kódu. Editor při psaní klíčových slov zobrazuje nabídku možných variant, což psaní značně

urychluje. Vlastnost je často spojena s automatickou kontrolou syntaxe kódu, která odhalí překlepy i další chyby ještě před kompilací.

- **Automatické formátování (Smart indentation)** – Zajišťuje přehledné formátování kódu. Odsazuje automaticky řádky dle úrovně zanoření, vkládá mezery okolo znamének atd.

- **Automatické generování kódu (Code generation)** – Ušetří psaní rutinních částí kódu. Např. generuje konstruktor ze seznamu atributů.

Při výběru IDE hraje roli také jeho cena. Něktará vývojová prostředí jsou zdarma (např. Eclipse IDE).

1.2.3 Modelovací nástroj

Předchozí dvě části (1.2.1 a 1.2.2) jsou nutné pro vývoj obecné softwarové aplikace. Modelovací nástroj doplňuje platformu tak, aby pomocí ní bylo možné vytvořit aplikaci pro řešení optimalizační úlohy. Pro vlastní optimalizační výpočet se využívá externí program – solver. Ten řeší úlohu ve tvaru $\{f(\mathbf{x}) = \mathbf{c}\cdot\mathbf{x}, A\mathbf{x} \leq \mathbf{b}\}$ pomocí specializovaných algoritmů. Modelovací nástroj tvoří komunikační vrstvu mezi aplikací a solverem. Nástroj interpretuje úlohu do tvaru, ve kterém je řešena solverem. Rovněž načítá výsledky vyhodnocené solverem zpět do aplikace. Modelovací nástroj může mít dvě následující podoby:

1. **Modelovací jazyk a překladač** – Pro formulování optimalizační úlohy je možné použít specializovaný jazyk. Ten umožňuje zadávat optimalizační proměnné a omezení nad nimi jednotlivě. Zadání zapsané v daném jazyku se kompiluje do tvaru pro solver. Výhodou této varianty je možnost formulovat úlohu v jazyce k tomuto účelu navrženém a při překladu volit cílový solver. Nevýhodou je, že se fakticky jedná o další programovací jazyk, který je třeba explicitně překládat. Rovněž je nutné řešit přelévání dat z modelovacího jazyka, ve kterém se řeší optimalizace, do programovacího jazyka, pomocí něhož se data prezentují, načítají a ukládají.

2. **API[1] solveru** – Za předpokladu, že solver poskytuje programovací rozhraní pro jazyk, ve kterém je napsána naše aplikace, lze jej k danému účelu využít. Narozdíl od předchozí varianty slibuje tato přímou konektivitu na solver. Implementace ale z principu zajistí použitelnost pouze jednoho solveru.

1.3 Původní platforma

V této podkapitole představím nahrazovanou platformu. Ukáži přednosti i nedostatky, které její použití přináší. Platforma je postavena na programu Matlab. Ten se využívá při vývoji aplikace i při jejím používání. Jako modelovací nástroj zde slouží Yalmip [3], což je balíček Matlabu. V následujících dvou bodech se nejprve podrobněji zaměřím na program Matlab a zmíněný balíček, v bodech následujících uvedu výhody a nevýhody jejich použití.

[1] Rozhraní pro programování aplikací (Application Programming Interface)

1.3.1 Prostředí Matlab

Stručný přehled vlastností Matlabu nalezneme třeba v [4]. Matlab tvoří programovací prostředí poskytující vlastní skriptovací jazyk. Jeho primární funkcí je vykonávání příkazů čtených z příkazové řádky nebo vykonávání celých sekvencí příkazů – skriptů. Druhý způsob umožňuje v Matlabu tvořit klasické počítačové aplikace. Matlab umožňuje snadnou práci s maticemi a vektory, vykreslování grafů, testování různých algoritmů, počítačovou simulaci atd.

1.3.2 Yalmip

Jedná se o balíček pro Matlab plnící funkci modelovacího jazyka. Umožňuje pracovat s optimalizačními proměnnými stejným způsobem jako s ostatními proměnnými Matlabu. Zároveň podporuje celou řadu solverů, jak je uvedeno v [5].

1.3.3 Výhody

V této sekci uvádím v bodech výhody použití původní varianty platformy.

1. Podpora práce s maticemi a vektory – Jelikož je koncept Matlabu na maticovém počtu přímo založen, je matematické modelování pomocí něho poměrně snadné. Tato vlastnost je zejména u řešení optimalizačních úloh velice užitečná.

2. Jednoduchá syntaxe jazyka – Skriptovací jazyk Matlab se syntaxí blíží pseudokódu. Nevyžaduje explicitní deklarování proměnných podle datových typů.

3. Interaktivní režim programování – Zadávání příkazů přes příkazový řádek je výhodné zejména při testování funkcí.

4. Snadná vizualizace dat – Tato vlastnost umožňuje rychle zobrazovat datové sady v různé formě. To je pro analýzu velkého objemu dat výhodné.

5. Vlastní IDE – Matlab zahrnuje postačující vývojové prostředí pro psaní programů. Obsahuje nástroj pro debugování i funkce usnadňující psaní kódu.

6. Integrovaný modelovací nástroj – Z hlediska rozdělení modelovacích nástrojů v sekci 1.2.3 lze Yalmip zařadit do první kategorie. Protože celý nástroj je ale implementován v Matlabu, odpadá potřeba dalšího programu – překladače modelovacího jazyka. Syntaxe modelovacího i programovacího jazyka se shodují.

1.3.4 Nevýhody

1. Drahá komerční licence – Použití této platformy vyžaduje program Matlab, jehož licence stojí stovky tisíc Kč.

2. Nízká rychlost při OOP – Matlab je silný nástroj pro práci s maticemi a vektory. V této oblasti je poměrně rychlý. Při zpracování objektů ale rychlostně výrazně zaostává za jinými jazyky vyšší úrovně (např. Java, C#). To se projevuje mimo jiné pomalým procesem formulování optimalizačních problémů nástrojem Yalmip, což potvrzuje práce [9].

3. Špatná podpora OOP – V Matlabu nelze explicitně zavést interface, což výrazně omezuje polymorfismus – jednu z klíčových vlastností objektově orientovaného programování.

Použitím nové platformy by nevýhody měly být eliminovány.

2 Analýza dostupných řešení

V této kapitole uvádím postup i výsledky výběru nové varianty platformy. Jelikož seznam možných variant byl poměrně obsáhlý, rozdělil jsme proces výběru do dvou fází. Cílem první fáze je vyčlenit užší skupinu možných variant, ze které je ve druhé fázi vybrána ta konečná.

2.1 Řešení posuzovaná v 1. fázi

Podkapitola shrnuje postup první fáze výběru. I když hledáme celkovou platformu, jak je popsána v části 1.2, při výběru se soustředím na stěžejní část platformy, kterou je modelovací nástroj (viz 1.2.3). V prvním bodě uvádím kritéria, na základě kterých varianty hodnotím, a tvořím skupinu nejvhodnějších. Následující bod je věnován krátkému popisu jednotlivých variant. Poslední bod přináší vyhodnocení této fáze výběru.

2.1.1 Metody posuzování

Kritéria pro výběr užší skupiny variant je nutné stanovit tak, aby hodnocení možností z daného hlediska nevyžadovalo instalaci a přímé testování daného nástroje, protože možností je poměrně velké množství. Je třeba stanovit klíčové vlastnosti, které vybíraný nástroj musí splňovat. Informaci o tom, zda daná varianta tyto vlastnosti splňuje, by mělo být možné zjistit z webu či dokumentace. Klíčové vlastnosti shrnuji ve dvou bodech:

1. **Programovací rozhraní (API)** – Varianta umožňuje vytvářet samostatné aplikace prostřednictvím API obecných programovacích jazyků. Vývoj celé aplikace tak lze provádět v jednom vývojovém prostředí, což značně zjednodušuje debugování i vlastní programování aplikace.

2. **Přímá konektivita na solvery Gurobi a Cplex** – Nástroj poskytuje rozhraní k solverům Gurobi a Cplex. Jedná se o osvědčené výkonné solvery schopné řešit poměrně rychle i rozsáhlé MILP úlohy. Podpora dalších solverů výslovně vyžadována není.

2.1.2 Seznam variant

V této části uvedu stručný popis všech uvažovaných variant. Každé variantě je věnován odstavec, ve kterém jsou shrnuty hlavní vlastnosti nástroje. Řadu informací přebírám z [9], další z webů samotných produktů.

AIMMS

Jedná se o pokročilé vývojové prostředí pro řešení optimalizačních úloh. Poskytuje konektivitu na celou řadu solverů, mezi nimi i na Gurobi a Cplex. Faktem je, že celá funkcionalita tohoto nástroje je vázána na AIMMS IDE – vlastní vývojové prostředí, a nástroj je tak nevyhovující

z hlediska požadované vlastnosti 1. Využití nástroje ze strany další aplikace je tak možné jen využitím standardního rozhraní Component Object Model (COM) pro komunikaci mezi různými programy. Debugování je možné prostřednictvím vlastního vývojového prostředí, ovšem jen formou chybových výpisů. Uvedené vlastnosti čerpám z [9]. Nevýhodou je také cena € 6500 uvedená v [10]).

AMPL

AMPL (A Mathematical Programming Language) je algebraický modelovací jazyk pro lineární i nelineární optimalizaci. Umožňuje rychlé zpracování modelů do formy pro solver. Podporuje celou řadu solverů, také Gurobi a Cplex. Další jeho vlastnosti nalezneme v [9], mimo jiné absenci dostačující dokumentace. AMPL rovněž neposkytuje API standardních programovacích jazyků, nesplňuje vlastnost 1.

ASCEND

Program pro matematické modelování. Přehled jeho vlastností čerpám z [11]. Umožňuje řešit soustavy lineárních i nelineárních rovnic, lineární a nelineární optimalizační úlohy a modelovat dynamické systémy. K uvedeným účelům nabízí vlastní modelovací jazyk. Zahrnuje rovněž vlastní solver a z toho důvodu neposkytuje konektivitu na žádný další.

Cplex API a Gurobi API

Programovací rozhraní obou solverů umožňují využívat jejich funkcionalitu v externích aplikacích. V obou případech jsou podporovány jazyky C++, Java, Python a jazyky prostředí .NET. Prostřednictvím API lze formulovat optimalizační úlohy objektově přímo v kódu aplikace. Podrobná struktura obou API je popsána v dokumentacích [12] a [17]. Nevýhodou této varianty je, že vyžaduje nadstavbu jednotící přístup k oběma solverům.

CVXOPT

Optimalizační balíček pro Python. Umožňuje vytvářet aplikace pro řešení optimalizačních úloh v tomto jazyce. Neumožňuje formulovat MILP úlohy. Neposkytuje rozhraní k solverům Gurobi a Cplex, čímž nesplňuje požadovanou vlastnost 2. Zdrojem informací je [13].

FICO Xpress-Mosel

Nástroj umožňuje formulovat a řešit rozsáhlé optimalizační úlohy prostřednictvím vlastního modelovacího jazyka. Následně lze úlohy řešit solverem Xpress-Optimizer. Jiný solver ale podporován není. Modelovací nástroj lze do aplikací integrovat pomocí API jazyků C, Java nebo C#. Uvedené vlastnosti jsem čerpal z [14].

FlopC++

Jedná se o open-source algebraický modelovací jazyk implementovaný jako knihovna jazyka C++. Název tvoří zkratka anglického výrazu **F**ormulation of **L**inear **O**ptimization **P**roblems in **C++**. Nástroj umožňuje formulovat optimalizační úlohy podobným stylem jako v jazyku

AMPL či GAMS, a to přímo v kódu programu jinak obecného jazyka C++. Zdrojem informací je webová stránka [15], odkud lze rovněž stáhnout zdrojový kód knihovny.

GAMS

Informace o produktu čerpám ze [16]. Jde o program pro modelování a řešení optimalizačních problémů. Zahrnuje překladač vlastního modelovacího jazyka a sadu integrovaných solverů. Rovněž poskytuje rozhraní k řadě externích solverů, ke kterým patří i Gurobi a Cplex. Z jiných aplikací lze program využívat voláním funkcí příkazového řádku.

JOptimizer

Open-source Java knihovna pro formulování a řešení optimalizačních úloh. Neumožňuje ale řešit MILP problémy a postrádá rozhraní k solverům Gurobi a Cplex. K řešení úloh využívá vlastní integrovaný solver. Informace jsou převzaty z [18].

JModelica

Rozšiřitelná open-source platforma pro optimalizaci, simulaci a analýzu dynamických systémů postavená na jazyku Modelica. Neposkytuje rozhraní k solverům Gurobi ani Cplex. Více informací o nástroji lze nalézt v [19].

LINGO

Kompletní balík pro řešení velkého spektra optimalizačních úloh. Obsahuje modelovací nástroj i sadu integrovaných výkonných solverů. Neposkytuje rozhraní k solverům Gurobi ani Cplex. Umožňuje přístup externím aplikacím přes LINDO API. Informace jsou převzaty z [20].

Microsoft Solver Foundation

Sada vývojových balíčků pro matematickou simulaci, optimalizaci a modelování. Nástroj je postaven na platformě .NET. Součástí licence je i solver Gurobi, jak je patrné z [21]. Solver Cplex naopak není přímo podporován. Nástroj je možné získat pod třemi možnými licencemi – Express (zdarma), Standard a Enterprise. První dvě verze umožňují pouze řádově 1000 proměnných definovaných MILP úloh.

MPL

Další modelovací nástroj realizovaný samostatným programem a disponující vlastním modelovacím jazykem podobně jak je tomu u nástrojů AMPL, AIMMS či GAMS. Program poskytuje konektivitu na řadu solverů (také Gurobi a Cplex). Nástroj lze rovněž využít ve vlastních aplikacích prostřednictvím knihovny OptiMax využitím jazyků Visual Basic, Visual C++, Visual J++, Java nebo Delphi. Tyto a další vlastnosti lze nalézt v [22].

OpenModelica

Open-source prostředí pro modelování a simulaci založené na jazyce Modelica. Obsahuje též optimalizační modul OMOptim. Neposkytuje přímou konektivitu na Gurobi ani Cplex. Další vlastnosti programu viz [23].

OpenOpt

Balíček jazyka Python pro řešení optimalizačních problémů. Má tvořit svobodnou alternativu komerčních modelovacích nástrojů jako např. AMPL, GAMS a AIMMS. Tyto i další informace jsou dostupné na [24]. Nástroj nabízí velice stručnou a nepřehlednou dokumentaci. Navíc v oblasti modelování MILP problémů nepodporuje solver Gurobi.

PuLP

Balíček jazyka Python. Slouží pro modelování LP a MILP úloh. Disponuje přímou konektivitou na solvery Gurobi i Cplex. Umožňuje snadno formulovat optimalizační úlohy zaváděním proměnných a omezení přímo v kódu programu. Zdrojem informací je [25]. Dokumentace dostupná v [26] pokrývá jen část funkcionality nástroje.

Pyomo

Nástroj podobný jako PuLP – modelovací nástroj implementovaný v jazyce Python. Přehled vlastností je uveden v [27]. Způsob formulování optimalizačních problémů vychází z konceptu oddělení modelu a dat. Nevýhodou je absence klasické programátorské dokumentace. Funkcionalitu nástroje je nutné zkoumat z dostupné prezentace [28] a dokumentu [29]. Text dokumentu [29] mimo jiné upozorňuje na pomalost nástroje Pyomo v porovnání s jinými modelovacími nástroji.

TOMNET Optimization

Jedná se o nástroj pro modelování optimalizačních problémů na platformě .NET. Přehled vlastností je dostupný v [30]. Použití solveru je závislé na druhu řešené optimalizační úlohy. Jak se ukazuje v [31], pro řešení úloh nelze využít Gurobi solver.

Zimpl

Je poslední uvažovaný modelovací nástroj. Jedná se o open-source program vytvořený pro akademické účely, jak se uvádí v [9]. Ve zmíněné publikaci je také uváděn jako nejrychlejší testovaný nástroj při formulování optimalizační úlohy. Bohužel neposkytuje přímou konektivitu na solver Gurobi ani Cplex. Další informace o produktu jsou dostupné v [32].

2.1.3 Vyhodnocení

Jak jednotlivé varianty splňují dané požadavky je patrné z tabulky 2.1. Symbol „X“ značí, že varianta danou vlastnost splňuje, „-“ znamená, že nikoli. Dále se tedy budu zabývat pěticí vyhovujících nástrojů.

	API	Gurobi & Cplex
AIMMS	-	X
AMPL	-	X
ASCEND	-	-
Cplex API a Gurobi API	**X**	**X**
CVXOPT	X	-
FICO Xpress-Mosel	X	-
FlopC++	**X**	**X**
GAMS	-	X
JOptimizer	X	-
JModelica	-	-
LINGO	X	-
MS Solver Foundation	X	-
MPL	**X**	**X**
OpenModelica	-	-
OpenOpt	X	-
PuLP	**X**	**X**
Pyomo	**X**	**X**
TOMNET Optimization	X	-
Zimpl	-	-

Tabulka 2.1: Hodnocení nástrojů z hlediska stanovených kritérií ve fázi 1

2.2 Řešení posuzovaná ve 2. fázi

Výstupem druhé fáze výběru má být již konečná varianta, která bude použita při vývoji aplikací. Podobně jako ve fázi první nejprve uvedu kritéria posuzování zbylých možností. Druhá část podkapitoly se zabývá podrobněji jednotlivými zbylými variantami. Jelikož se jejich počet značně zmenšil, lze nástroje i testovat. Třetí část přináší hodnocení a uvádí zvolenou variantu.

2.2.1 Metody posuzování

Nyní už možnosti nehodnotím na základě dvou vlastností, které by měly splňovat, ale zabývám se komplexní použitelností při vývoji nových aplikací pro optimalizaci. V některých případech je za účelem testu implementován zjednodušený příklad úlohy popsané v části 1.1.2.

Rychlost formulování úlohy

Modelovací nástroj by měl být schopen rychlé kompilace zadání úlohy do tvaru posílaného na vstup solveru.

Finanční náročnost

Není žádoucí, aby použitý modelovací nástroj zvyšoval náklady vyvíjené aplikace. Preferována jsou proto open-source řešení.

2.2.2 Seznam variant

Tato část je věnována podrobnějšímu zkoumání zbylých variant. Jednotlivé odstavce popisují možnosti z hlediska kritérií uvedených v předchozí části.

Cplex API a Gurobi API

Formulování optimalizačních úloh prostřednictvím programovacích rozhraní samotných solverů omezuje modelovací nástroj na sadu funkcí ve vybraném programovacím jazyku (C, C++, Java, jazyky .NET, Python). Eliminuje se tak potřeba modelovacího nástroje jako externího programu.

Protože varianta z principu postrádá jednotný přístup k oběma API, její použití by vyžadovalo implementaci jednotící mezivrstvy. Struktura obou API je ale velice podobná, jak je zřejmé z jejich dokumentace. Obě rozhraní jsou založena na objektové reprezentaci skalárních optimalizačních proměnných, jejich lineárních kombinací – lineárních výrazů a nerovnic, které představují omezení. Vytvořit jednotné rozhraní pro obě API je proveditelné využitím vlastností OOP (dědičnost, rozhraní, polymorfismus). Výhodou je možnost navrhnout další vrstvu pro matematické modelování na míru podle potřeb.

Rychlosti vytváření modelu a formulování úlohy pro solver ukazuje zátěžový test (2.2.3). Zvolené jazyky jsou C# a Python. Oba jazyky umožňují přetěžování operátorů, jejichž využitím lze zjednodušit zápis optimalizačních úloh. V testu (2.2.3) je použito API solveru Gurobi, přičemž předpokládám podobné vlastnosti v případě API solveru Cplex. Program implementovaný v jazyce C# se ukazuje více než desetkrát rychlejší oproti tomu napsaném v Pythonu. Přesné porovnání uvádím v závěrečném hodnocení 2.2.3. Zdrojové kódy testovaných programů jsou na přiloženém CD (5, třetí položka).

Jak již bylo zmíněno, použití jazyků C# i Python obecně díky jejich jednoduché syntaxi umožňuje poměrně rychlý vývoj aplikací. Pro psaní programů lze využít populárních IDE. V případě Pythonu je dostupný plug-in PyDev do Eclipse IDE. Vývoj programů v jazyce C# značně usnadní Visual C# ve verzi Express, která je zdarma.

Jelikož obě API jsou poskytována spolu se solvery, odpadá nutnost pořizovat modelovací nástroj zvlášť a s tím spojené náklady.

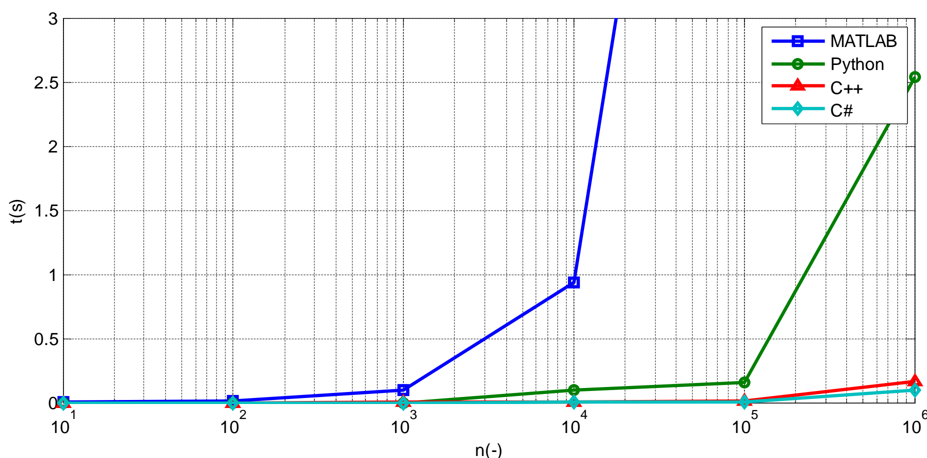
FlopC++

Varianta předpokládá použití jazyka C++ jako implementačního jazyka aplikace. To je nevýhodné, protože se jedná o jazyk nízké úrovně a vývoj aplikací v něm trvá obecně delší dobu. Rychlost vytváření modelu reprezentujícího optimalizační úlohu za použití OOP je přitom téměř stejná, jak ukazuje srovnávací test (2.1). Porovnávám zde rychlost, s jakou vytvářejí velké množství objektů programy implementované v různých jazycích. Vytváření velkého počtu malých objektů totiž úzce souvisí s tvorbou objektové reprezentace úlohy v aplikaci a tak s vlastním formulováním úlohy. Zdrojové kódy testovaných programů jsou součástí přiloženého CD (5, druhá položka).

Použití nástroje FlopC++ by z finančního hlediska problém nebyl. Jedná se o open-source projekt.

MPL

Vlastnosti tohoto nástroje jsou poměrně podrobně rozebrány v [9].



Obrázek 2.1: Porovnání programovacích jazyků z hlediska rychlosti vytváření velkého množství objektů

Lze předpokládat, že nástroj nabízí uspokojivý výkon při formulování optimalizačních úloh.

Knihovna OptiMax umožňuje využít nástroj v samostatných aplikacích postavených např. na platformě Java. Dostupná je i postačující dokumentace.

Zásadní nevýhodou této varianty je její komerční licence, jejíž cena znamená další náklady při pořizování vývojové platformy i při užívání vytvořených aplikací.

PuLP

Použití nástroje přináší výhody i nevýhody typické pro práci s jazykem Python. Rychlost a jednoduchost implementace programů je vyvážena jejich nízkým výkonem.

Rychlost s jakou nástroj vytváří objektový model optimalizačního problému (viz 2.2.3) a následně ho převádí do tvaru pro solver je jeho jedinou, ale bohužel velice významnou nevýhodou. Pro tuto variantu byl implementován zjednodušený příklad úlohy 1.1.2, výsledek je uveden ve vyhodnocení na konci této kapitoly (viz 2.2.3). Zdrojový kód testovaného programu je na přiloženém CD (5, třetí položka).

Díky jednoduché syntaxi jazyka Python je vytváření aplikací jednoduché a rychlé. Psaní programů rovněž usnadňuje možnost programování z příkazového řádku podobně jako je tomu u Matlabu. Zápis optimalizačních úloh je pomocí PuLPu snadný díky přetížení operátorů nad objekty reprezentujícími proměnné, nerovnice i cílovou funkci. Pro představu uvádím příklad zápisu jednoduché optimalizační úlohy:

```
# Definovani maximalizacni ulohy:
prob = LpProblem('prob1', LpMaximize)
# Pridani realne optimalizacni promenne z intervalu
   (-Inf, 10>:
x = LpVariable('x', upBound = 10, cat = 'Continuous
   ')
# Pridani omezeni x <= 4.3:
prob += x <= 4.3
```

```
# Stanoveni cilove funkce:
prob += x
# Spusteni optimalizace pomoci solveru Gurobi:
GUROBI().solve(prob)
# Vypis vyresene hodnoty promenne x:
print x.varValue

>> 4.3
```

Použití této varianty nepřináší žádné finanční nároky. Balíček PuLP je dostupný zdarma z [25].

Pyomo

Použití tohoto nástroje sebou nese podobné výhody i úskalí jako předchozí varianta (PuLP). Jedná se rovněž o balíček jazyka Python.

Časová náročnost je velkou slabinou nástroje Pyomo. Plyne to z použití jazyka Python (viz 2.1) a rovněž to potvrzuje měření v [29].

Ze snahy tvořit alternativu modelovacím jazykům jako je AMPL či MPL vychází i syntaxe nástroje Pyomo, což ale nepřináší výrazné výhody. Výrazným nedostatkem této varianty je špatná dokumentace, jak je uvedeno v předchozí části výběru.

2.2.3 Vyhodnocení

V této části uvádím vyhodnocení druhé fáze výběru, tedy i výsledek celého výběrového procesu.

Nástroj FlopC++ je vyřazen, neboť by vyžadoval vývoj aplikací v C++. Použití tohoto jazyka by jinak nepřineslo žádné výrazné výhody. Testovaný program napsaný v C++ je rychlostně srovnatelný s tím napsaným v jazyku C# (viz 2.1).

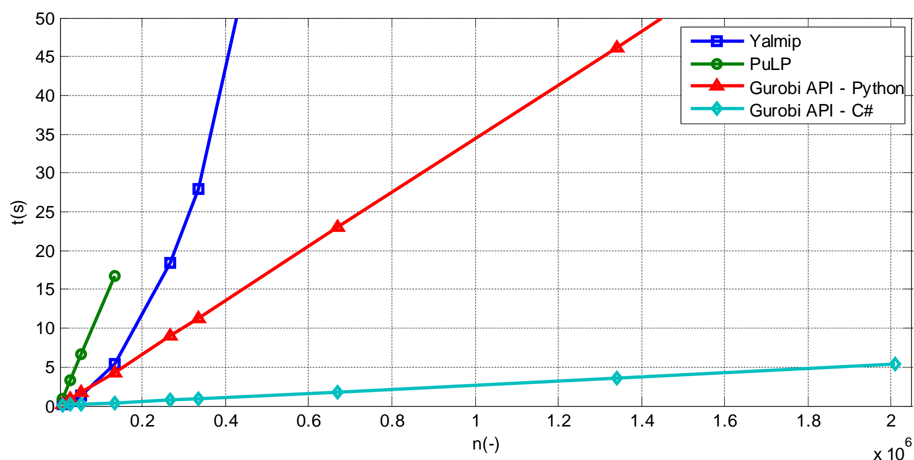
Varianta MPL je vyřazena, protože vyžaduje použití modelovacího nástroje jako samostatného programu s nutností zakoupení komerční licence.

Balíček Pyomo je nevhodný kvůli absenci programátorské dokumentace.

Testovací příklad

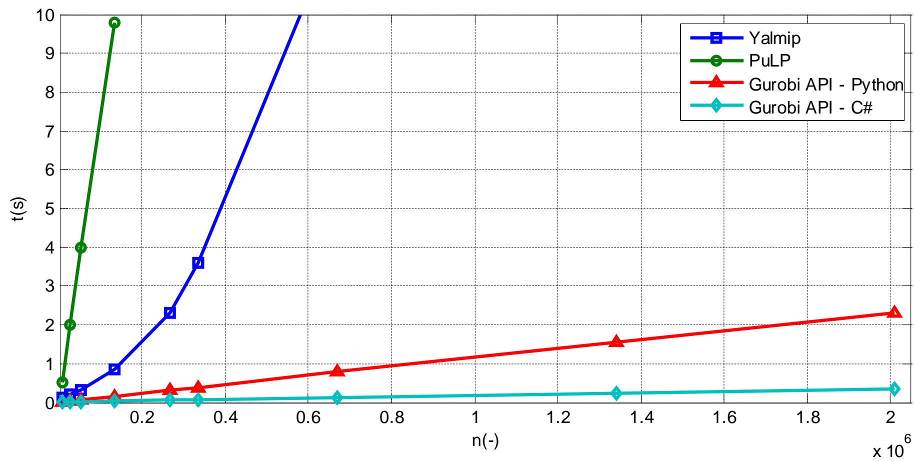
Hodnocení zbývajících variant je patrný z výsledků zátěžového testu (viz grafy 2.2, 2.3 a 2.4). Pomocí těchto nástrojů byl implementován zjednodušený příklad úlohy modelování energetické sítě. Jedná se tedy o typ úlohy, pro který je platforma určena. Úkolem bylo pokrýt definovanou zátěž řízeným provozem elektráren v horizontu 168 hodin. Každá elektrárna má dva bloky. Test je v každé variantě proveden několikrát pro počty 5 až 1500 elektráren, což odpovídá úlohám v rozsahu řádově 10^3 až 10^6 optimalizačních proměnných (reálných i binárních celkem). Testována je rychlost vytváření modelu, která souvisí s rychlostí vytváření objektů, a rychlost převedení úlohy do tvaru $A\mathbf{x} \leq \mathbf{b}$ pro solver. Rovněž je testována paměťová náročnost. Zdrojové kódy testovaných programů jsou na přiloženém CD (5, třetí položka).

Z grafů 2.2 a 2.3 lze vyvodit, že v obou fázích modelovacího procesu je nejefektivnější variantou Gurobi API pro C#. Tato varianta je také nejméně paměťově náročná, jak ukazují průběhy na obr. 2.4. Protože rychlost vyvíjených aplikací hraje při výběru stěžejní roli, volím tuto variantu jako vhodný modelovací nástroj do nové vývojové platformy.

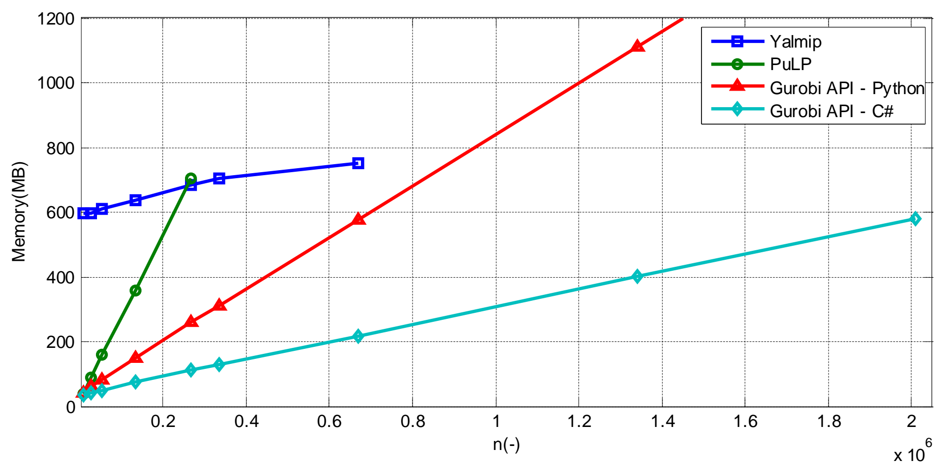


Obrázek 2.2: Porovnání rychlostí modelovacích nástrojů ve fázi vytváření objektové reprezentace úlohy

Konečným řešením je tedy .NET API solverů Gurobi a Cplex. Jako programovací jazyk pak volím C#, neboť je široce používaný a programy v něm napsané jsou dostatečně rychlé (viz 2.1 a 2.2). Dále doporučuji vývojové prostředí Visual C# Express [36], které umožňuje programy psát, kompilovat i ladit a je dostupné zdarma. Nová platforma je detailně popsána v následující kapitole.



Obrázek 2.3: Porovnání rychlostí modelovacích nástrojů ve fázi transformace úlohy do tvaru $A\mathbf{x} \leq \mathbf{b}$ pro solver



Obrázek 2.4: Porovnání paměťových náročností modelovacích nástrojů

3 Zvolená varianta

Cílem této kapitoly je představit zevrubně novou platformu postavenou na frameworku .NET.

V první části popíši strukturu a vlastnosti jednotlivých prvků platformy. Ve druhé uvádím příklad vývoje aplikace, který může sloužit jako návod k použití platformy.

3.1 Struktura a vlastnosti nové platformy

V této podkapitole uvedu jednotlivé složky vývojové platformy a popíši jejich vlastnosti a vzájemné vztahy.

V prvním bodě se zabývám prostředím .NET, jelikož jazyk C# patří do rodiny jazyků tohoto prostředí. Poté podrobněji popíšu .NET programovací rozhraní solverů Gurobi a Cplex. Ve třetí části uvedu důležité vlastnosti IDE Visual C# Express.

3.1.1 Prostředí .NET

Informace zde uvedené čerpám z [34] a [33], odtud přebírám i strukturální schéma 3.1.

Rozhraní .NET Framework je nedílnou součástí systému Windows. Poskytuje konzistentní objektově orientované programovací prostředí pro různé druhy aplikací. Rozhraní .NET Framework má dvě hlavní složky:

1. Společný jazykový modul CLR (Common Language Runtime)
2. Knihovnu tříd .NET (Class Library)

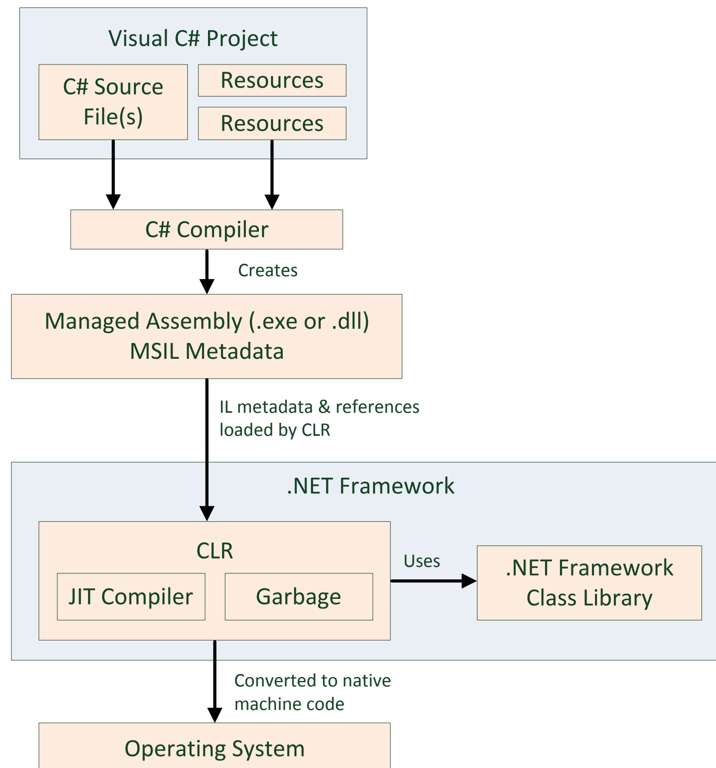
Modul CLR je komerční implementace mezinárodního standardu CLI (Common Language Infrastructure). Jde v podstatě o virtuální počítač, který dokáže předkompilovaný program (byte kód) překládat v reálném čase na strojový kód. Přitom využívá funkce z knihovny tříd .NET.

Zdrojový kód napsaný v jazyku C# je kompilován do byte kódu IL (intermediate language), který odpovídá standardním CLI specifikacím. Kód IL je uložen na disku spolu s dalšími zdroji (bitmapami, textovými soubory atd.) ve spustitelném souboru nazývaném assembly. Ten má typicky příponu .exe nebo .dll. Obsahuje manifest s informacemi o daném assembly.

Při spuštění programu je assembly načten do CLR a v souladu s informacemi v manifestu provádí v reálném čase JIT (just in time) překlad tak, že konvertuje kód IL do nativního strojového kódu. CLR se také stará o automatickou správu paměti pomocí GC (Garbage Collection). Kód IL vykonávaný přes CLR se někdy nazývá spravovaný kód MC (managed code). Naproti tomu nespravovaný kód UC (unmanaged code) je kompilován přímo do nativního strojového kódu.

Diagram na obrázku 3.1 ilustruje vztahy mezi zdrojovým kódem napsaným v jazyce C#, knihovnou tříd .NET, předkompilovaným programem do IL a modulem CLR.

Interoperabilita různých programovacích jazyků je klíčovou vlastností .NET frameworku. Jelikož IL kód produkovaný C# překladačem odpovídá standardu CTS (Common Type Specification), může spolupracovat se kterýmkoli jazykem prostředí .NET. Sem patří Visual Basic,



Obrázek 3.1: Schéma prostředí .NET

Visual C++ a dalších více než 20 jazyků překládaných dle CTS specifikace. Přitom může jeden spustitelný soubor obsahovat moduly napsané v rozdílných jazycích .NET, jako kdyby byly napsány v jazyce stejném.

Další silnou vlastností je, že modul CLR může využívat knihovnu tříd .NET obsahující přes 4000 tříd. Ty poskytují široké spektrum funkcí (vstupně výstupní operace, zpracování XML, vytváření grafického uživatelského rozhraní atd.).

Operační systém Windows 7 neobsahuje nejnovější verzi rozhraní (Microsoft .NET Framework 4), ale lze ji zdarma stáhnout z [38].

3.1.2 Gurobi API pro .NET

V této části podrobněji popíši .NET programovací rozhraní solveru Gurobi. Představím funkcionalitu některých hojně využívaných tříd a uvedu několik příkladů použití.

Ke všem funkcím solveru se přistupuje přes knihovnu GurobiXX.NET.dll, kde „XX“ značí konkrétní verzi solveru. Uváděný popis se vztahuje konkrétně ke Gurobi46.NET.dll. Knihovna je dostupná po instalaci solveru ve složce bin.

Model

Konkrétní optimalizační úloha je v programu uložena v instanci třídy GRBModel. Pro vytvoření instance modelu je třeba inicializovat prostředí solveru. K tomu slouží třída GRBEnv. Pomocí té se nastavují parametry řešení úlohy, konkrétně metodou GRBEnv.Set(). Vytvoření objektu model pak v kódu vypadá následovně:

```
// Inicializace prostredi solveru:
GRBEnv env = new GRBEnv();

// Nastaveni maximalni doby reseni ulohy (10s):
env.Set(GRB.DoubleParam.TimeLimit, 10);

// Vytvoreni modelu - objektove reprezentace ulohy:
GRBModel model = new GRBModel(env);
```

Model nyní neobsahuje žádné optimalizační proměnné, cílovou funkci ani omezení. To vše je nutné do modelu přidat prostřednictvím dalších objektů.

Proměnné

Optimalizační proměnná je reprezentována třídou GRBVar. Objekt nelze vytvořit standardním voláním konstruktoru GRBVar(), ale je nutné volání funkce GRBModel.AddVar(). Tak se zajistí příslušnost proměnné ke konkrétnímu modelu. Při vytváření se stanovuje charakter proměnné (reálný, celočíselný či binární) a její rozsah. Volitelně lze zadat název a koeficient proměnné v cílové funkci. Běžně jsou proměnné do modelu přidávány následujícím způsobem:

```
// Pridani realne opt. promenne z intervalu <0,5>:
GRBVar x = model.AddVar(0, 5, 0, GRB.CONTINUOUS,
    null);

// Pridani celociselne opt. promenne z intervalu
    <-3,10>:
GRBVar y = model.AddVar(-3, 10, 0, GRB.INTEGER, null
    );

// Pridani binarni opt. promenne:
GRBVar z = model.AddVar(0, 1, 0, GRB.BINARY, null);
```

Proměnné je také možné přidávat hromadně pomocí funkce GRBModel.AddVars(). Dolní a horní meze jsou pak zadávány pomocí polí.

Po inicializování všech potřebných proměnných se volá funkce GRBModel.Update() zajišťující jejich integraci do modelu.

Lineární výrazy

Formulování optimalizační úlohy se neobejde bez lineárních výrazů. Pomocí nich jsou definována omezení modelu a rovněž cílová funkce. Lineární výraz je součet lineární kombinace

optimalizačních proměnných a konstanty. V Gurobi API je lineární výraz reprezentován třídou GRBLinExpr. Díky přetížení operátorů nad třídou GRBVar je možné lineární výraz vytvořit následovně:

```
// a) Vytvoreni linearniho vyrazu "2x + 3y + 4z +
   5":
GRBLinExpr linExpr = 2 * x + 3 * y + 4 * z + 5;
```

Alternativní způsob je následující:

```
// b) Vytvoreni linearniho vyrazu "2x + 3y + 4z +
   5":
GRBLinExpr linExpr = new GRBLinExpr();
linExpr.AddTerm(2, x);
linExpr.AddTerm(3, y);
linExpr.AddTerm(4, z);
linExpr.AddConstant(5);
```

Omezení

Omezení jsou reprezentována třídou GRBTempConstr, jejíž instance jsou vytvářeny přetížením operátorů ==, <= a >=. Pravou a levou stranu tvoří instance tříd GRBLinExpr, GRBVar nebo reálné číslo (double). Dva způsoby zadávání omezení ukazuje následující příklad:

```
// a) Pridani omezeni "x + y <= 100"
model.AddConstr(x + y <= 100, "constraint name");
...
// b) Pridani omezeni "x + y <= 100"
GRBLinExpr linExpr = x + y;
model.AddConstr(linExpr, GRB.LESS_EQUAL, 100, "
   constraint name");
```

Cílová funkce

Cílová funkce se do modelu zadává pomocí GRBModel.SetObjective(). Zde se také udává, zda má úloha maximalizační či minimalizační charakter. Příklad:

```
// a) Zadat cilovou funkci "f(x,y) = x + y", pro
   maximalizaci:
model.SetObjective(x + y, GRB.MAXIMIZE);
...
// b) Zadat cilovou funkci "f(x,y) = x + y", pro
   minimalizaci:
model.SetObjective(x + y, GRB.MINIMIZE);
```

Optimalizace a přístup k výsledkům

Je-li úloha zadána, může být spuštěn optimalizační proces. To se provede zavoláním funkce GRBModel.Optimize(). Pokud optimalizační proces proběhne úspěšně, lze načíst výsledné hodnoty optimalizačních proměnných následujícím způsobem:

```
// Spusteni optimalizace solverem Gurobi:
model.Optimize();

// Nacteni vyresenych hodnot:
double xValue = x.Get(GRB.DoubleAttr.X);
double yValue = y.Get(GRB.DoubleAttr.X);
double zValue = z.Get(GRB.DoubleAttr.X);
```

Kompletní funkcionalita knihovny je popsána v dokumentaci 3.1.2.

3.1.3 Cplex API pro .NET

Obdobným způsobem jako v předchozí podkapitole přestavím zde základní funkcionalitu .NET programovacího rozhraní solveru Cplex.

API je realizované dvěma knihovnami:

- ILOG.CPLEX.dll,
- ILOG.Concert.dll.

První slouží pro vlastní přístup k solveru, druhá poskytuje funkce potřebné pro formulování úloh. Obě jsou dodávány jako součást balíku Cplex Optimization Studio.

Model

Zde má třída reprezentující model název Cplex. Není třeba předchozí inicializace prostředí, jak je tomu v případě Gurobi. K založení optimalizační úlohy tedy stačí jediný příkaz:

```
// Zalozeni optimalizacni ulohy:
Cplex model = new Cplex();
```

Proměnné

Rozhraní Cplex API obecně používá zapouzdření objektů pomocí interface. Tak je tomu při práci s optimalizačními proměnnými i lineárními výrazy. Přístup je ale jinak velice podobný jako u Gurobi. Proměnné se přidávají přes instanci modelu, zde instanci třídy Cplex. Analogické jsou i vstupní argumenty funkce Cplex.NumVar(), jak je patrné z příkladu:

```
// Pridani realne opt. promenne z intervalu <0,5>:
INumVar x = model.NumVar(0, 5, NumVarType.Float);

// Pridani celociselne opt. promenne z intervalu
   <-3,10>:
```

```
INumVar y = model.NumVar(-3, 10, NumVarType.Int);

// Pridani binarni opt. promenne:
INumVar z = model.NumVar(0, 1, NumVarType.Bool);
```

Hromadné přidávání proměnných do modelu je možné voláním funkce Cplex.NumVarArray(). Volání funkce Update() se zde neprovádí.

Lineární výrazy

Lineární výraz reprezentuje v Cplex API rozhraní ILinearNumExpr. Práce s lineárními výrazy zde není usnadněna přetížením operátorů nad třídou proměnných, proto je jejich vytváření poměrně zdlouhavý proces:

```
// Vytvoreni linearniho vyrazu "2x + 3y + 4z + 5":
ILinearNumExpr linExpr = model.LinearNumExpr();
linExpr.AddTerm(2, x);
linExpr.AddTerm(3, y);
linExpr.AddTerm(4, z);
linExpr.Constant = 5;
```

Omezení

Jelikož vytváření omezení není implementováno pomocí přetížení operátorů ==, <= a >=, je nutné zadávat levou a pravou stranu rovnice jako argumenty funkce přidávající omezení. Takové funkce jsou tři a odpovídají třem možným druhům porovnání:

1. Cplex.AddLe(): <=,
2. Cplex.AddGe(): >=,
3. Cplex.AddEq(): ==.

Opět uvádím příklad použití:

```
// Vytvoreni lin. vyr.
// - leve strany omezeni:
ILinearNumExpr linExprConstr
        = model.LinearNumExpr();
linExprConstr.AddTerm(1, x);
linExprConstr.AddTerm(1, y);

// Pridani omezeni "x + y <= 100"
// do modelu:
model.AddLe(linExprConstr, 100,
        "constraint name");
```

Cílová funkce

Zadání cílové funkce se provádí téměř stejně jako u Gurobi. Je třeba pouze zavolat funkci Cplex.AddObjective(), předat jí lineární výraz odpovídající cílové funkci a parametr, který rozlišuje, zda se jedná o maximalizační či minimalizační úlohu:

```
// a) Zadat cilovou funkci
// "f(x,y) = x + y", pro maximalizaci:
model.AddObjective(ObjectiveSense.Maximize, linExpr)
   ;
...
// b) Zadat cilovou funkci
// "f(x,y) = x + y", pro minimalizaci:
model.AddObjective(ObjectiveSense.Minimize, linExpr)
   ;
```

Optimalizace a přístup k výsledkům

Optimalizační proces se spouští voláním funkce Cplex.Solve(). Vyřešené hodnoty optimalizačních proměnných pak lze načíst prostřednictvím funkce Cplex.GetValue():

```
// Spusteni optimalizace solverem Gurobi:
model.Solve();

// Nacteni vyresenych hodnot:
double xd = model.GetValue(x);
double yd = model.GetValue(y);
double zd = model.GetValue(z);
```

Dokumentace obou knihoven je dostupná na [12].

3.1.4 Visual C# Express

V této podkapitole uvedu některé vlastnosti zvoleného jazyka C# a vývojového prostředí Visual C# Express.

Jazyk C#

Jazyk patří do rodiny jazyků .NET. Syntakticky je o něco složitější než jazyky Python či Matlab. Vyžaduje například explicitní deklarování proměnných pomocí datových typů. Jeho syntaxe je podobná jazyku C++ a velice podobná jazyku Java. Pro představu uvádím implementace funkce, která spojí dvě n-tice čísel do n-tice jedné, ve třech verzích – Matlab, Python a C#:

Matlab:

```
function arr12 = joinArr(arr1, arr2)
        arr12 = [arr1 arr2];
end
```

Python:

```
def joinArr(arr1, arr2):
    return arr1 + arr2
```

C#:

```
static double[] joinArr(double[] arr1, double[] arr2
   ){
        double[] arr3 = new double[arr1.Length +
           arr2.Length];
        for (int i = 0; i < arr1.Length; i++){
        arr3[i] = arr1[i];
        }
        for (int i = 0; i < arr2.Length; i++){
                arr3[arr1.Length + i] = arr2[i];
        }
        return arr3;
}
```

Z příkladu je patrné, že vytváření aplikací v jazyku C# zřejmě zabere více času, než by tomu bylo v případě použití jazyků Python nebo Matlab. Výsledky rychlostních testů (2.1 a 2.2) ale potvrzují správnost volby jazyka C#.

Vývojové prostředí

Visual C# Express je verze populárního IDE Visual Studio firmy Microsoft. IDE lze zdarma stáhnout z [36]. Hlavní výhodou tohoto editoru je, že je přímo určen pro vývoj aplikací v jazyce C#. Umožňuje programy přímo kompilovat i debugovat. Nástroj je bezplatně využitelný i pro komerční účely, jak je uvedeno v [39].

Grafické rozhraní programu je organizováno do různých oken, která je možno libovolně rozmísťovat stylem „drag-and-drop“. Zmíním alespoň nejdůležitější okna v následujícím přehledu pomocí obrázku 3.2:

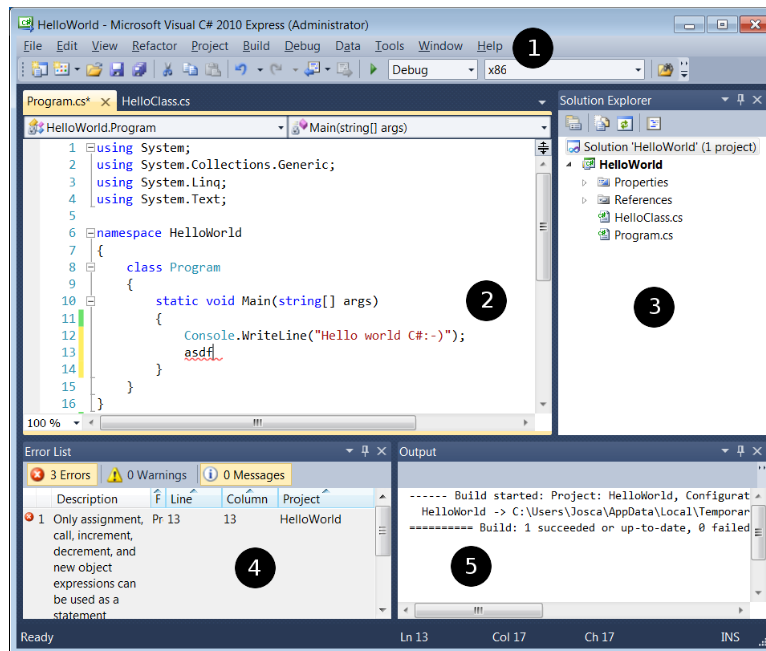
1. Hlavní menu – Standardní nabídka položek, přes které lze upravovat vlastnosti editoru, přidávat další panely nástrojů, založit nový projekt atd.

2. Editor kódu – Hlavní okno sloužící pro psaní programu. Programování zjednodušuje zvýrazněná syntaxe, automatická kontrola kódu, možnost automatického formátování a další programátorské pomůcky.

3. Solution Explorer – Náhled struktury zdrojových souborů. Pomocí tohoto průzkumníku lze soubory projektu organizovat do složek, zakládat nové soubory na správném místě a jednoduše odkazovat na použité knihovny.

4. Varovná a chybová hlášení – Během psaní je kód programu automaticky kontrolován. Na syntaktické chyby IDE okamžitě upozorňuje prostřednictvím tohoto okna. Ostatní chyby jsou odhaleny až při pokusu o překlad. Obdobné je to s varovnými výpisy.

5. Výstupní konzole – Okno zobrazující výstup překladače.



Obrázek 3.2: Okna vývojového prostředí Visual C# Express

Vývojové prostředí Visual C# Express je chudší ale bezplatná varianta IDE Visual Studio a pro vývoj standardních aplikací v jazyce C# postačuje, jak ukáže návod v následující podkapitole.

3.2 Příklad vývoje aplikace

V této podkapitole ukáži postup při vytváření optimalizační aplikace pomocí IDE Visual C# Express. Samotná struktura programu je zde vedlejší, proto použiji velice jednoduchý příklad práce s Gurobi API z části 3.1.2. Zdrojový kód aplikace je také součástí přiloženého CD (5, čtvrtá položka).

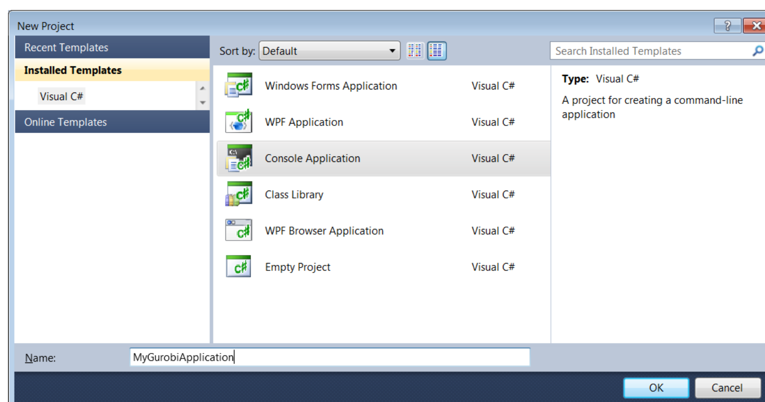
3.2.1 Založení projektu

Začneme spuštěním programu Visual C# Express. Úvodní stránka, která se objeví, umožňuje rychlé otevření posledních projektů, což teď nepotřebujeme. Nový projekt založíme výběrem z hlavního menu (File → New Project) nebo pomocí klávesové zkratky Ctrl+Shift+N. Z naběhnutého dialogového okna vybereme šablonu „Console Application“ pro vytvoření konzolové aplikace. Projekt pojmenujeme např. „MyGurobiApplication“ (viz obr. 3.3).

Po potvrzení je založen prázdný projekt a IDE se přepne do standardního režimu popsaného v části 3.1.4.

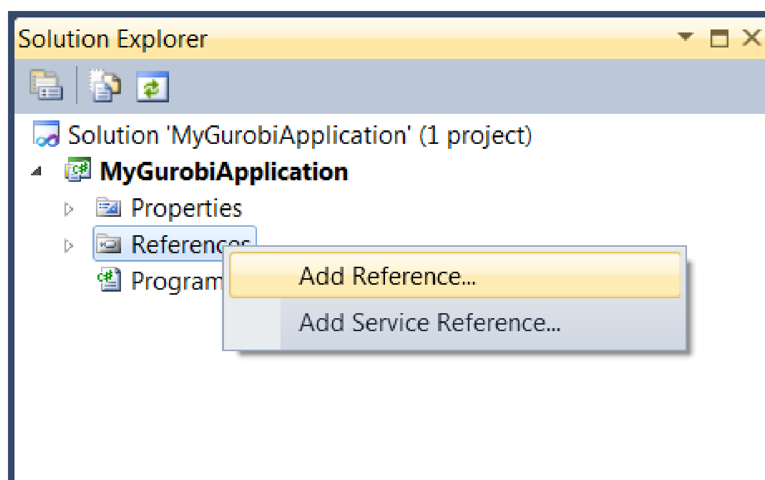
3.2.2 Přidání knihovny do referencí

Protože chceme využívat funkce a třídy knihovny Gurobi46.NET.dll, je nutné ji přidat do referencí v okně „Solution Explorer“. Postup je následující. Pravým tlačítkem klikneme na složku



Obrázek 3.3: Založení projektu

References a z kontextové nabídky vybereme možnost Add Reference (viz obr. 3.4).



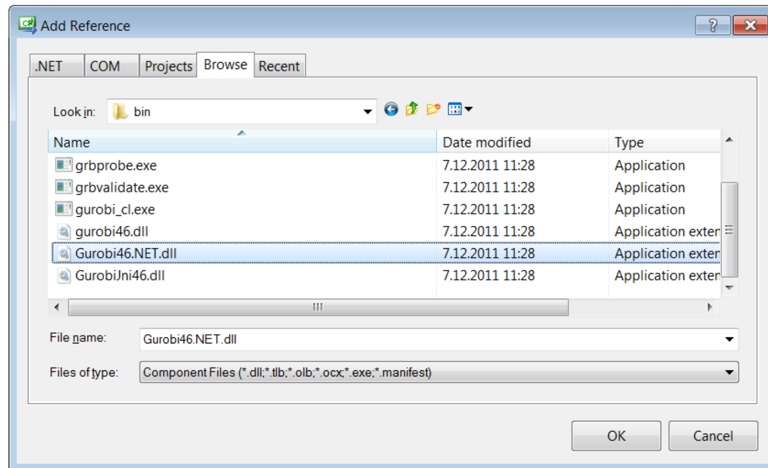
Obrázek 3.4: Přidání reference

Objeví se stejnojmenné dialogové okno. Přepneme se na kartu Browse a přidáme knihovnu ze složky bin programu Gurobi (viz obr. 3.5).

3.2.3 Nastavení pro překlad

Nyní je čas pro napsání kódu aplikace. Ten najdeme v příloze **??**. Pokud chceme vytvořit aplikaci pro 64-bitový systém, je nutné nastavit, aby se program překládal daným způsobem. Ve výchozím nastavení IDE není možné volit cílovou platformu překladu. Je třeba nejprve aktivovat nabídky Solution Configuration a Solution Platforms. To se provede následujícím způsobem. Z hlavního menu volíme: Tools →Options. V dialogovém okně zaškrtneme nejprve možnost Show all settings, následně v sekci Projects and Solutions ještě položku Show advanced build configurations (viz obr. 3.6). Zmíněné nabídky pro nastavení překladu jsou tak aktivovány.

Nyní je třeba změnit sílovou platformu na x64. To se provede přes Configuration Manager. Jeho spuštění je patrné z obrázku 3.7.



Obrázek 3.5: Přidání knihovny do referencí

V naběhnutém dialogovém okně vybereme z nabídky Active solution platform položku New. Zde vytvoříme novou cílovou platformu x64 dle obrázku 3.8. V nabídce Copy settings from ponecháme možnost x86. Volbu potvrdíme tlačítkem OK a zavřeme Configuration Manager.

3.2.4 Přeložení programu

Nyní lze program přeložit, a to dvěma způsoby. Klasický překlad pro účely používání aplikace provedeme zvolením položky Release z nabídky Solution Configuration (viz obr. 3.9). Následně lze překlad provést stiskem klávesy F6 nebo pomocí hlavní hlavní nabídky (Build → Build Solution). Spustitelný soubor – výsledek překladu nalezneme ve složce bin\x64\Release. Analogickým způsobem se provede i překlad pro ladění programu, z nabídky Solution Configuration se však v takovém případě vybere možnost Debug.

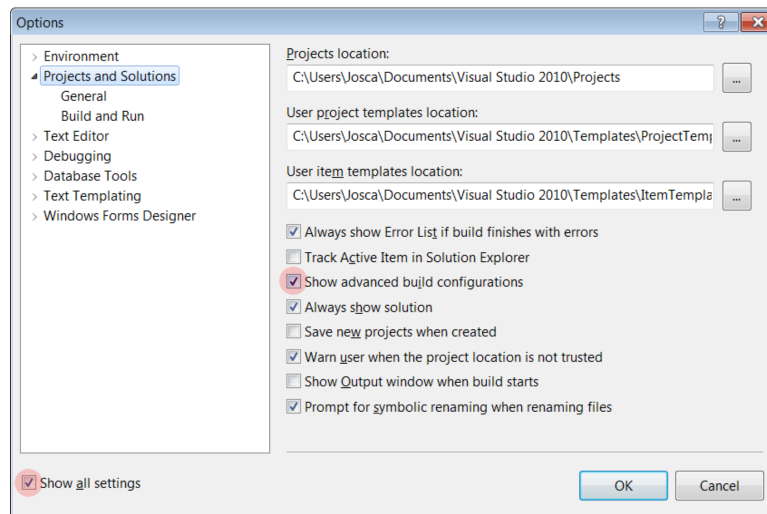
3.2.5 Debugování programu

Při vývoji aplikací je často užitečné program v určitém místě zastavit a prozkoumat jeho stav, zejména pak hodnoty vnitřních proměnných. K tomuto účelu slouží debugovací režim. Uvažujme, že chceme běh programu zastavit na řádku 26. Toho dosáhneme jednoduše tak, že na levý kraj tohoto řádku vložíme značku (breakpoint) kliknutím myši (viz obr. 3.10). Ladění lze nyní zahájit spuštěním programu v debugovacím režimu pomocí tlačítka Start Debugging (viz obr. 3.10) nebo klávesou F5.

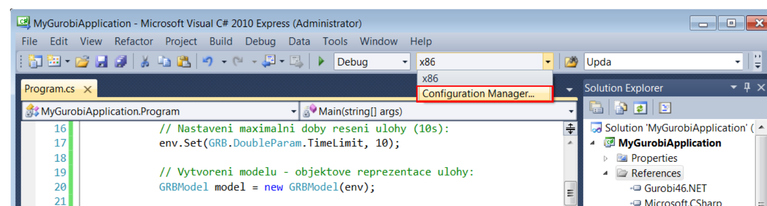
Běh programu se zastaví ve zvoleném místě a je možné zkoumat stav libovolné proměnné tím, že na ni najedeme myší (viz obr. 3.11).

Dále lze v běhu programu pokračovat stisknutím klávesy F5 nebo pokračovat krokováním pomocí klávesy F10. Po skončení ladění lze značku odstranit kliknutím na ni.

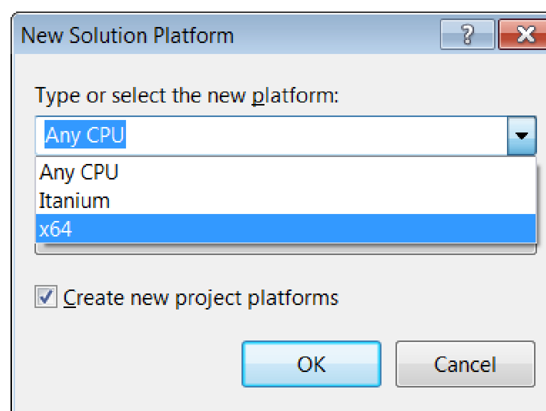
Hotový program je vždy tvořen spustitelným souborem (*.exe). Využívá-li program funkce knihoven, překladač je rovněž přidá do adresáře bin jako soubory s koncovkou .dll.



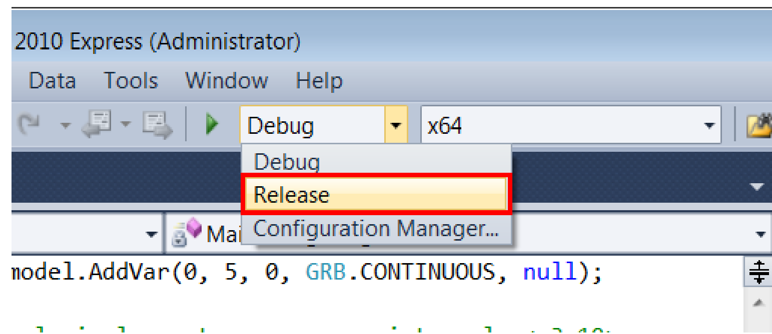
Obrázek 3.6: Aktivování nastavitelnosti překladu



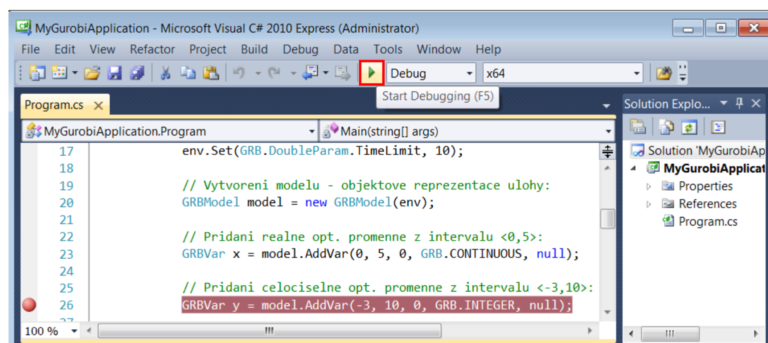
Obrázek 3.7: Spuštění dialogu Configuration Manager



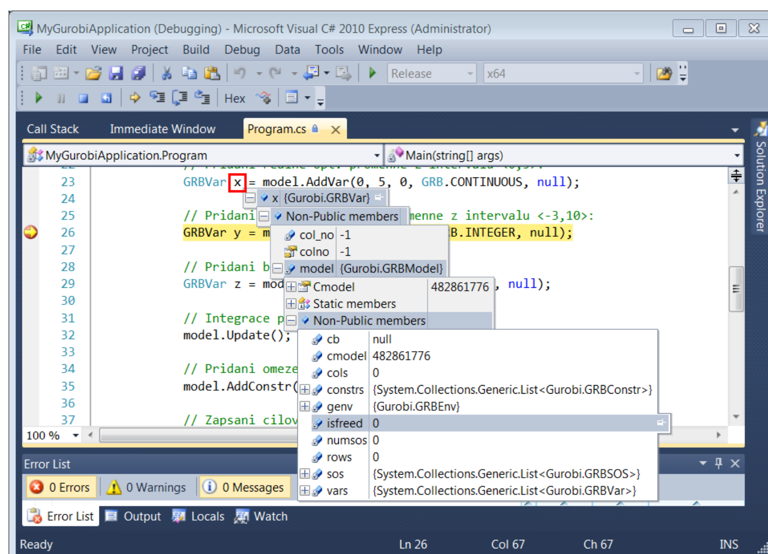
Obrázek 3.8: Vytvoření nové cílové platformy



Obrázek 3.9: Nastavení překladu programu pro účely užívání



Obrázek 3.10: Spuštění programu za účelem ladění



Obrázek 3.11: Ukázka ladění programu

4 Framework JD

Tato kapitola pojednává o návrhu, implementaci a testování .NET knihoven pro formulování optimalizačních úloh. Framework jsem pro zjednodušení pojmenoval, jeho název tvoří iniciály mé dívky.

Cílem první části kapitoly je popsat návrh a realizaci projektu. Je zde podrobně popsána funkcionalita, kterou knihovny zajišťují, objektová struktura, konektivita na solvery a struktura dokumentace.

Druhá část se zabývá testováním frameworku. K tomuto účelu je pomocí nového nástroje implementován model přenosové soustavy ve dvou verzích. První, jednodušší, je alternativou programů z části 2.2 a je realizován pro účely porovnání s nimi. Druhá verze má otestovat využitelnost nástroje při realizaci strukturálně složitých problémů s velkým počtem optimalizačních proměnných.

4.1 Návrh a implementace

Podkapitola pojednává o návrhu a realizaci projektu JD. V prvním bodě se zabývám účely, pro které je nástroj určen. Následující bod popisuje SW návrh, zejména pak objektovou strukturu. V bodě třetím se zmiňuji o materiálech usnadňujících práci s frameworkem.

4.1.1 Účel frameworku

Vývoj frameworku má dva cíle:

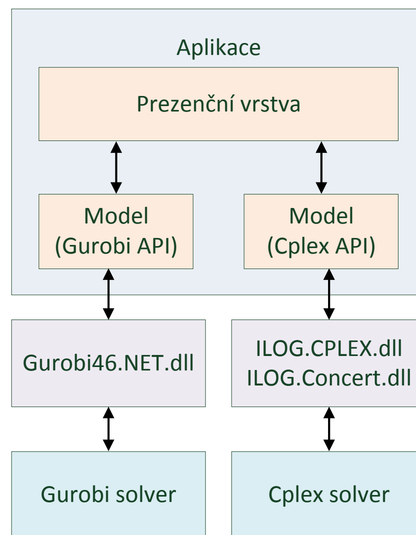
1. Sjednotit přístup k oběma solverům a zajistit rozšiřitelnost o podporu dalších solverů,

2. Zjednodušit způsob formulování optimalizačních úloh implementací maticového počtu, jak je tomu např. u nástroje Yalmip.

Jednotné rozhraní solverů

Přestože s .NET API obou solverů se pracuje velice podobným způsobem, jak bylo ukázáno v částech 3.1.2 a 3.1.3, každá z obou variant vyžaduje odlišnou implementaci konkrétní optimalizační úlohy. Proto je při stávajícím stavu poměrně časově náročné naprogramovat aplikaci schopnou využít oba solvery. Je to navíc konceptuálně špatné řešení, neboť v podstatě vyžaduje formulovat úlohu pro každý solver zvlášť. To také značně zvyšuje náročnost rozšiřitelnosti aplikace o podporu solverů dalších. Struktura takové aplikace by musela odpovídat schématu na obrázku 4.1.

Framework JD tvoří mezivrstvu, která poskytuje aplikaci ke všem solverům jednotné programovací rozhraní. Ta je pak kompatibilní se všemi solvery podporovanými frameworkem JD, jak je znázorněno na schématu 4.2.



Obrázek 4.1: Aplikace spolupracující se oběma solvery

Formulování úloh pomocí matic

Formulování úloh pomocí dostupných API je poměrně zdlouhavé a nepřehledné oproti tomu, jak to umožňuje Yalmipu. To potvrzuje následující příklad zápisu omezení $\mathbf{x} \leq \mathbf{a}$ pomocí Yalmipu a Gurobi API. Zde je příslušná část kódu pro Yalmip:

```
F = [F, x <= a];
```

Zde je omezení zapsáno pomocí Gurobi API:

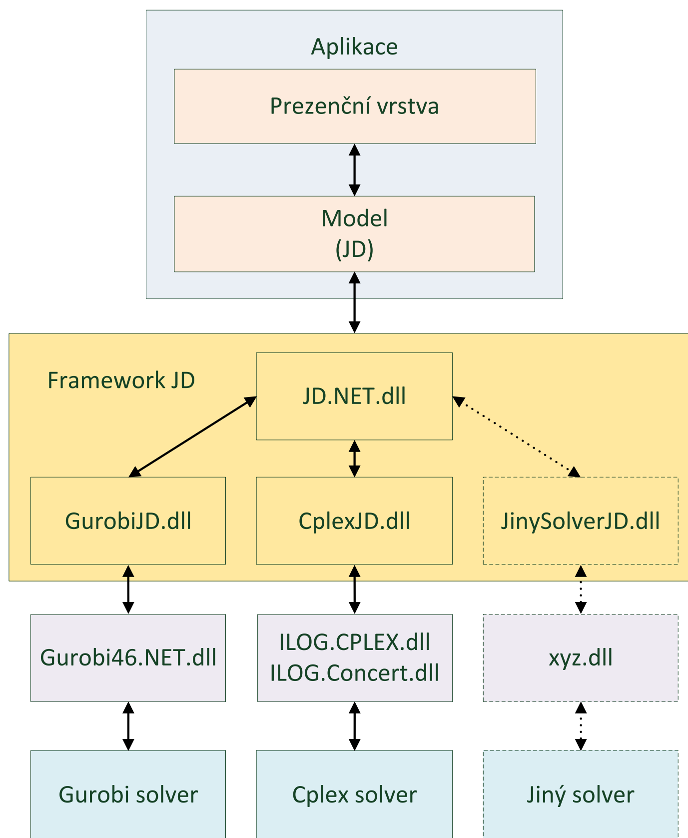
```
for (int i = 0; i < x.Lenght; i++){
        model.AddConstr(x[i], GRB.LESS_EQUAL, a[i],
           "constraint name");
}
```

Yalmip umožňuje operace nad celými vektory a dokonce nad celými maticemi. To rozhraní solverů neumožňují, a proto je formulování úloh pomocí nich mnohem složitější.

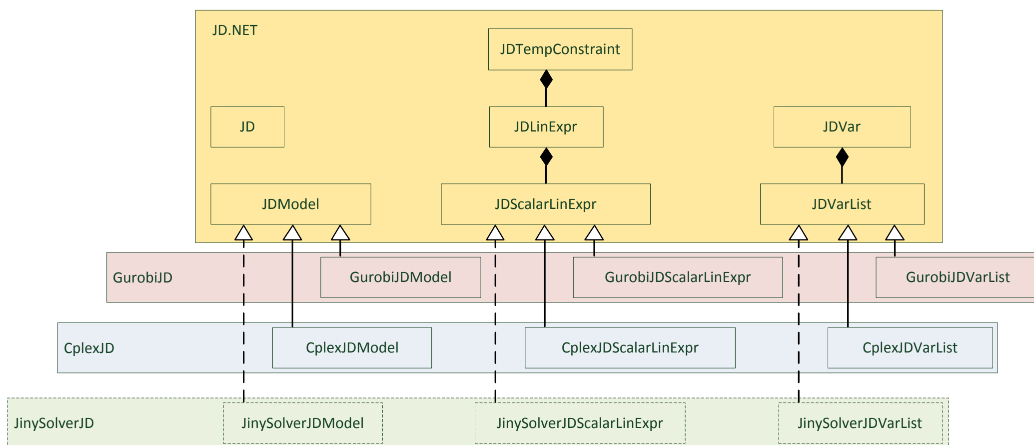
Framework JD tento problém odstraňuje zavedením objektů reprezentujících celé matice optimalizačních proměnných a operací nad nimi. Zápis optimalizačních úloh se pak značně usnadní.

4.1.2 Objektová struktura

V této části popíšu objektovou strukturu a význam jednotlivých tříd. Pro názornost uvádím schéma 4.3, pomocí kterého vztahy jednotlivých tříd vysvětlím.



Obrázek 4.2: Schéma funkcionality frameworku JD



Obrázek 4.3: Schéma objektové struktury frameworku JD

Třídy projektu se dělí na dvě skupiny podle dvou základních cílů zmíněných výše. Každou z nich tedy popíšu v samostatném bodě. Třetí bod je věnován popisu rozdělení tříd do knihoven.

Jednotné rozhraní solverů

První skupina tříd zajišťuje jednotný přístup k solverům. Jedná se o trojici abstraktních tříd JDModel, JDScalarLinExpr, JDVarList a třídy od nich odvozené. První tři třídy tvoří obecné rozhraní pro solver a poskytují základní funkcionalitu pro formulování úloh. Odvozené třídy tvoří ovladače konkrétních solverů. Formulování úloh použitím pouze této sady tříd je podobně náročné, jako v případě samotných Gurobi API či Cplex API, je však už pro všechny solvery jednotné.

- JDModel – Tato abstraktní třída reprezentuje optimalizační úlohu. Prostřednictvím ní jsou přidávány proměnné, omezení i cílová funkce. Tento koncept je převzat z Gurobi API.
- JDScalarLinExpr – Jedná se o abstraktní třídu, která reprezentuje lineární výraz ve smyslu uvedeném v části 3.1.2.
- JDVarList – Abstraktní třída zapouzdřující pole optimalizačních proměnných. Gurobi i Cplex API umožňuje vytvářet proměnné hromadně, proto je zde zvolen přístup jejich inicializace po celých skupinách. Pokud je nutné přidat proměnnou jen jednu, je jednoduše vytvořen list délky 1.

Děděním od této trojice dostáváme implementaci ovladače pro konkrétní solver. Pomocí Gurobi API je tak realizován ovladač pro Gurobi, pomocí Cplex API zase ovladač pro Cplex. Takto lze rozšířit kompatibilitu frameworku o další solvery, jak je znázorněno na obr. 4.3.

Formulování úloh pomocí matic

Implementace maticového způsobu formulování úlohy je nadstavbou jednotícího rozhraní. Je realizováno druhou skupinou tříd. Jejím jádrem jsou třídy JDVar a JDLinExpr. Obě třídy mají maticový charakter a umožňují operace jinak běžné pro jednotlivé proměnné provádět nad celými sadami. Pomocí dvoudimenzionálního indexování lze provádět operace nad podmnožinami podobně jako v Matlabu.

- JDVar – Třída představuje matici optimalizačních proměnných.
- JDLinExpr – Reprezentuje matici lineárních výrazů.

Přetížením operátorů +, - a * nad těmito třídami je umožněn snadný zápis úloh, který se blíží syntaxi Yalmipu.

- JDTempConstraint – Tato třída reprezentuje omezení nad maticovými objekty. Instance třídy se automaticky vytváří použitím operátorů porovnání <=, >= a == na objekty typu JDVar a JDLinExpr.
- JD – Jedná se o třídu zapouzdřující konstanty používané ve frameworku (např. JD.MAXIMIZE, JD.INFINITY, JD.INTEGER atd.), dále jsou zde doplňující funkce maticového počtu a několik funkcí pro testování funkcionality ovladačů.

Knihovny frameworku JD

Třídy frameworku jsou kompilovány do knihoven tříd prostředí .NET. Rozdělení tříd do knihoven je patrné ze schématu 4.3.

- JD.NET.dll – Knihovna zahrnuje rozhraní solverů a modelovací nadstavbu.
- GurobiJD.dll – Knihovna realizující ovladač solveru Gurobi, je ji třeba kompilovat spolu s konkrétní verzí knihovny .NET Gurobi API (Gurobi46.NET.dll).
- CplexJD.dll – Knihovna realizující ovladač solveru Cplex, je ji třeba kompilovat spolu s konkrétní verzí knihoven .NET Cplex API (ILOG.CPLEX.dll, ILOG.Concert.dll).

Knihovna JD.NET.dll je na přiloženém CD (5, pátá položka). Jelikož knihovny ovladačů (GurobiJD.dll a CplexJD.dll) je třeba kompilovat spolu s API knihovnami solverů, jsou v příloze (5, pátá položka) pouze jejich zdrojové soubory.

4.1.3 Dokumentace

Ke knihovně je dostupná kompletní dokumentace všech tříd. Vývojové prostředí Visual C# Express usnadňuje psaní přehledných komentářů generováním XML kostry podle hlaviček funkcí, jak ukazuje příklad:

```
/// <summary>
/// Add multiplication of another scalar linear
   expression.
/// </summary>
/// <param name="multiplier">Added linear expression
    multiplier.</param>
/// <param name="linExpr">Linear expression to be
   add.</param>
public abstract void Add(double multiplier,
   JDScalarLinExpr linExpr);
```

Pokud je tímto způsobem srozumitelně okomentován celý kód, přináší to hned dvě výhody:

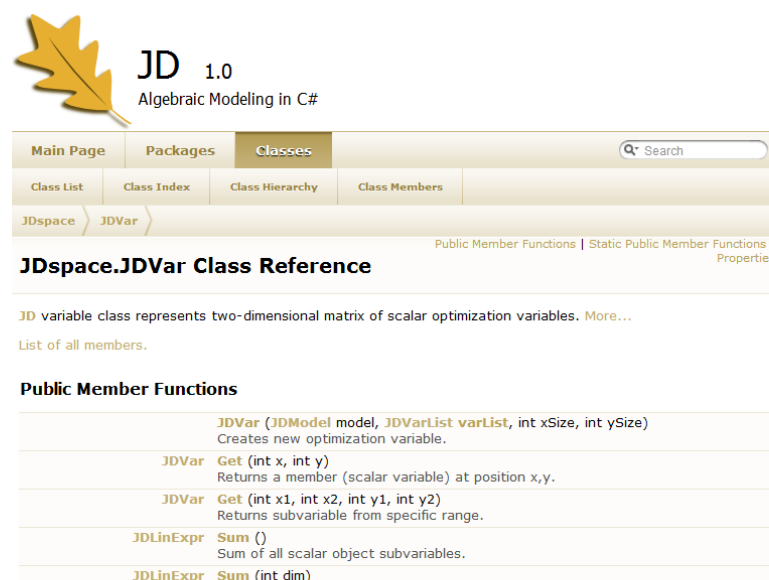
1. Visual C# Express zobrazuje komentáře funkcí jako nápovědu při jejich užívání jinde v kódu (viz obr. 4.4).
2. Pomocí programu Doxygen, který je volně stažitelný z [40], lze přímo z komentářů vygenerovat dokumentaci. Tímto způsobem je také vytvořena dokumentace frameworku JD. Ukázka dokumentace je na obr. 4.5. Dokumentace frameworku JD je v html a pdf na přiloženém CD (5, pátá položka).

```
/// <summary>
/// Soucet dvou realnych cisel.
/// </summary>
/// <param name="A">Prvni scitanec.</param>
/// <param name="B">Druhy scitanec.</param>
/// <returns>Soucet vstupnich parametru.</returns>
static double SumAandB(double A, double B)
{
    return A + B;
}

static void Main(string[] args)
{
    double C = SumAandB(3.22, 5.
}
```

double Program.SumAandB(double A, **double B**)
Soucet dvou realnych cisel.
B: *Druhy scitanec.*

Obrázek 4.4: Využití komentáře funkce jako nápovědy IDE



Obrázek 4.5: Ukázka dokumentace frameworku JD

4.2 Testování knihovny

V této kapitole uvedu testovací příklady použití frameworku JD. Nejprve se zaměřím na dva jednoduché příklady. Ve druhém kroku popíši implementaci dvou verzí modelu přenosové soustavy. Třetí bod obsahuje shrnutí výsledků testů.

4.2.1 Jednoduché příklady

V této části ukáži na dvou příkladech základy práce s frameworkem JD.

Různé solvery – stejný přístup

Tento příklad demonstruje, že formulování optimalizačních úloh pomocí funkcí frameworku je zcela nezávislé na použitém solveru. Úlohu lze jednoduše psát jako funkci, jejíž vstupním parametrem bude instance třídy JDModel.

```
static void OptimTask1(JDModel model){
        // Vytvoreni jednoprvkove matice
           optimalizacnich promennych:
        JDVar x = model.AddVar(1, 1);
        model.Update();

        // Pridani omezeni "x <= 100" do modelu:
        model += x <= 100;

        // Maximalizovat cilovou funkci "f(x) = x":
        model.SetObjective(x, JD.MAXIMIZE);
        model.Optimize();

        // Vypsat vysledek:
        Console.WriteLine("Reseno solverem: " +
           model.SolverName);
        x.Print();
}
```

Testovací funkce představuje optimalizační úlohu s jednou proměnnou x a jedním omezením $x \leq 100$, kdy chceme hodnotu proměnné maximalizovat. Následující program provede vyřešení úlohy nejprve solverem Gurobi a poté solverem Cplex:

```
// Reseni ulohy pomoci Gurobi:
JDModel grbModel = new GurobiJDModel();
OptimTask1(grbModel);

// Reseni ulohy pomoci Cplex:
JDModel cpModel = new CplexJDModel();
OptimTask1(cpModel);
```

Výstup programu bude následující:

```
Reseno solverem: Gurobi
100
Reseno solverem: Cplex
100
```

Maticové formulování úloh

Nyní ukážu formulování úlohy pomocí matic, které framework JD umožňuje:

```
// Vytvoreni matice opt. promennych rozmeru [2 x 3]:
JDVar X = model.AddVar(2, 3);
model.Update();

// Pridavani omezeni:
double[,] C = {{2, 4}, {6, 8}};
model += X <= 100;
model += 2 * X[0, 1, 1, 2] <= C;

// Nastaveni cilove funkce jako sumy vsech prvku
   matice X:
model.SetObjective(X.Sum(), JD.MAXIMIZE);

// Spusteni optimalizace:
model.Optimize();

// Vypsani vysledku:
Console.WriteLine("X = ");
X.Print();
```

Jedná se o úlohu, kdy maximalizujeme součet prvků matice X o rozměrech 2×3. Zároveň platí dvě pravidla:

- Žádný prvek matice není větší než 100.
- Dvojnásobek podmatice dané 2. a 3. sloupcem matice X není (po prvcích) větší než matice $C = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$.

Po provedení optimalizace program vypíše výslednou matici:

```
X =
100     1       2
100     3       4
```

Příklad ukazuje, že framework umožňuje indexování matic podobně jako Matlab a Python. Indexuje se od nuly na rozdíl od Matlabu, kde se indexuje od jedničky. Podmatici získáme z matice pomocí čtyř indexů. První dvojice udává rozsah řádků, druhá rozsah sloupců. V příkladu je toho využito při zápisu druhého omezení.

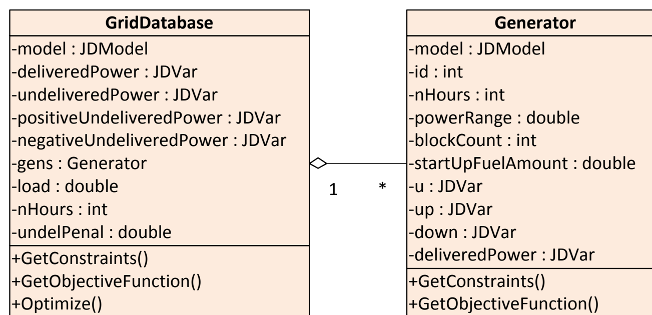
4.2.2 Implementace modelu přenosové soustavy

V této části uvedu dva příklady realizace modelu přenosové soustavy pomocí frameworku JD. První implementace odpovídá testovací úloze z 2. fáze výběru modelovacího nástroje (2.2.3). Cílem je porovnání rychlosti programu vyvinutého pomocí JD s ostatními testovanými variantami.

Druhý příklad představuje složitější model s uzly a vedeními. Cílem je odhadnout náročnost vytváření objektové struktury, která bude reprezentovat model evropské sítě. Zdrojové kódy obou programů jsou součástí přiloženého CD (5, třetí položka).

Jednoduchý model

Model je implementován pomocí dvou tříd podle UML diagramu 4.6:



Obrázek 4.6: Diagram tříd jednoduchého modelu přenosové soustavy

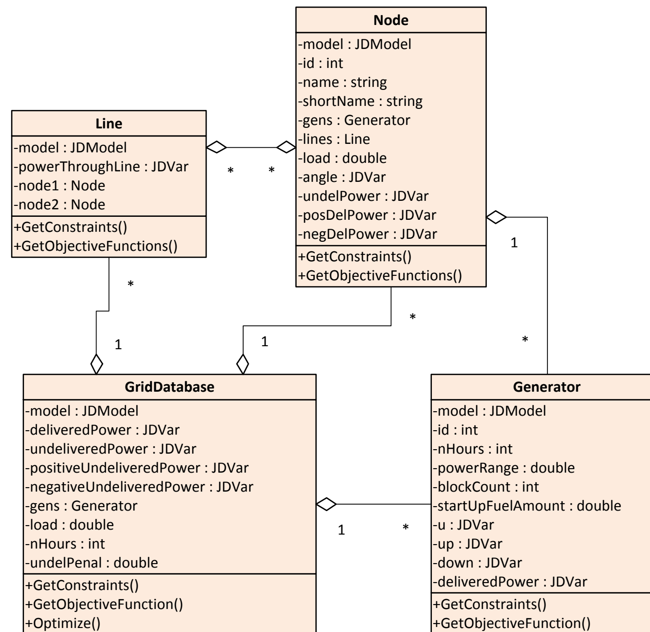
- Generator – Tato třída reprezentuje elektrárnu. Obsahuje její parametry jako je např. výkonový rozsah (powerRange) či počet bloků (blockCount). Rovněž nese průběhy dodávaných výkonou danou elektrárnou, ty jsou realizovány pomocí maticových proměnných typu JDVar. Funkce GetConstraits přidá do modelu omezení, která daný objekt popisují. Podobně přidá funkce GetObjectiveFunction příspěvek do cílové funkce.
- GridDatabase – Třída představuje celou přenosovou soustavu. Zapouzdřuje tedy sadu instancí třídy Generator, které představují výrobní zdroje v síti. Je zde také uložen průběh požadovaného výkonu (load) a další parametry soustavy. Funkce GetConstraints opět přidává do modelu omezení související s tímto objektem, mimo jiné také volá stejnojmennou funkci nad všemi vlastněnými instancemi třídy Generator.

Běh programu vypadá následovně. Nejprve jsou vytvořeny objekty reprezentující jednotlivé prvky sítě. Tím se inicializují i všechny optimalizační proměnné, které do nich patří. Poté je zavolána funkce GridDatabase.GetConstraints, která vytvoří všechna omezení. Následuje volání funkce GridDatabase.GetObjectiveFunctions vracející cílovou funkci. Nakonec je spuštěna optimalizace funkcí GridDatabase.Optimize.

Pokročilý model

Větší složitost druhého modelu spočívá v tom, že zahrnuje také vedení a uzly. To vyžaduje zavedení dalších tříd, takže diagram tříd pak vypadá dle obrázku 4.7.

- Node – Třída slučuje elektrárny umístěné v jedné lokalitě a přidává do sítě zátěž. Uzel může dodávat či odebírat elektrickou energii prostřednictvím připojených vedení.



Obrázek 4.7: Diagram tříd pokročilého modelu přenosové soustavy

- Line – Jedná se o třídu, která reprezentuje elektrické vedení. Spojuje dva uzly a přenáší mezi nimi výkon. Průběh této veličiny představuje proměnná typu JDVar s názvem powerThroughLine.

Instance těchto tříd přidávají do modelu omezení a příspěvky cílové funkce stejným způsobem jako třídy GridDatabase a Generator, tedy pomocí funkcí GetConstraints a GetObjectiveFunction.

Jelikož cílem této implementace je odhad rychlosti při vytváření modelu velké dimenze, není třeba podrobně implementovat logické závislosti, ale stačí obsadit objekty odpovídajícími vnitřními proměnnými.

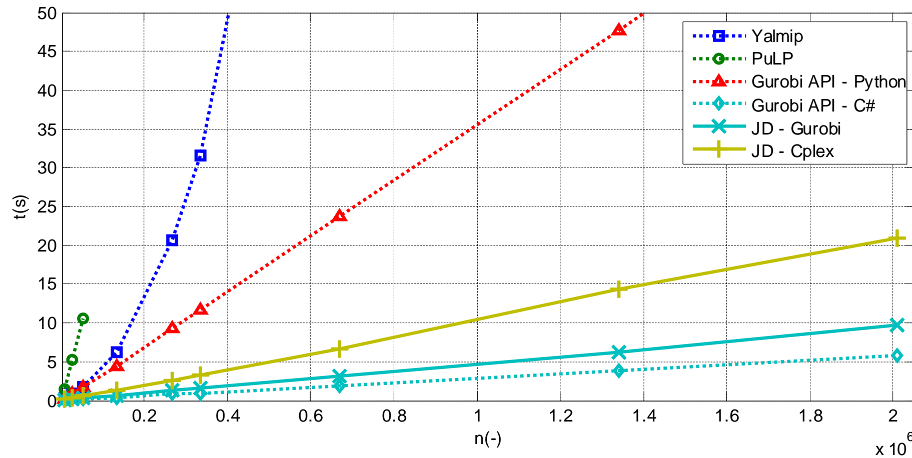
4.2.3 Výsledky testů

Tato část přináší vysledky testů dvou implementovaných modelů přenosové soustavy. V případě prvního modelu uvádím porovnání s modely implementovanými pomocí jiných modelovacích nástrojů (viz část 2.2.3). Výsledek testování složitějšího modelu pro několik dimenzí je pak v následujícím bodě.

Jednoduchý model

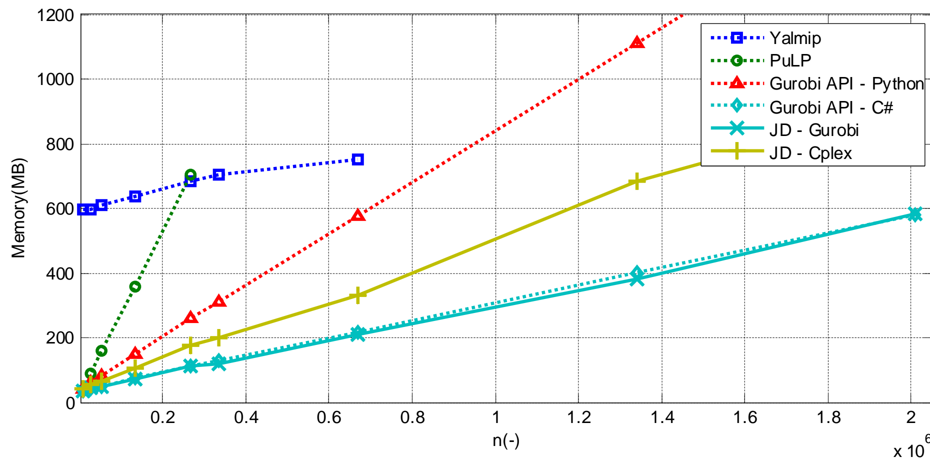
Rychlost programu implementovaného pomocí JD frameworku v porovnání s jinými nástroji je patrná z obrázku 4.8. Ukazuje se, že nadstavba jednotící přístup k solverům a zjednodušující formulování optimalizačních úloh prodlužuje modelovací proces přibližně o 40 % oproti variantě využívající holé .NET Gurobi API. Je to patrné z toho, jak se liší průběhy „Gurobi API – C#“ a

„JD – Gurobi“. Tento rozdíl je přijatelný, bereme-li v úvahu výhody, které použití frameworku JD přináší. Obě tyto varianty jsou pak o více než 90 % rychlejší než program implementovaný v Yalmipu, a to už při 200 000 proměnných. S rostoucím počtem proměnných se rozdíl stále zvětšuje, neboť průběh Yalmipu roste nelineárně.



Obrázek 4.8: Porovnání rychlostí modelovacích nástrojů

Paměťová náročnost se při použití frameworku JD téměř nezmění, jak ukazuje obr. 4.9.

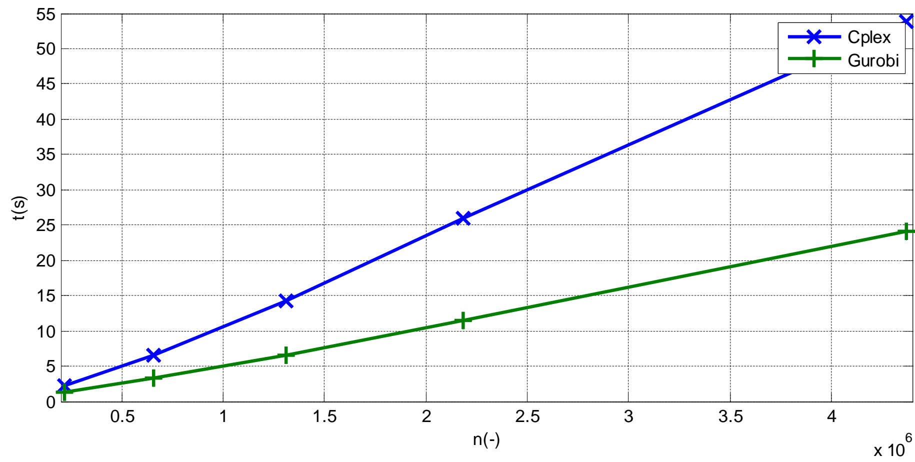


Obrázek 4.9: Porovnání paměťových náročností modelovacích nástrojů

Pokročilý model

Rychlostní test náročnějšího případu ukazuje, že nástroj JD je vhodný i pro modelování rozsáhlých problémů. Průběhy na obr. 4.10 ukazují, že pro Gurobi i pro Cplex trval proces formulování úlohy s více než 4 000 000 proměnnými méně než 1 minutu. Největší testovaný případ modeluje

soustavu s 2000 uzly, 2000 linkami, 2000 elektrárnami a 4000 bloky v horizontu 168 časových vzorků. Testy byly prováděny na osobním počítači s taktem 2,53 GHz.



Obrázek 4.10: Rychlost formulování úlohy modelující provoz evropské přenosové soustavy

5 Závěr

V rámci diplomové práce jsem navrhl vhodnou SW platformu pro vývoj optimalizačních aplikací. Tato platforma nahrazuje původní (viz 1.3), u které byla nalezena řada nevýhod.

Na základě testů (2.1 a 2.2.3) jsem vybral platformu založenou na použití .NET API solverů Gurobi a Cplex. Doporučené vývojové prostředí je Visual C# Express, protože umožňuje programy snadno překládat a ladit a je dostupné zdarma i pro komerční účely. V návodu (viz 3.2) jsem ukázal, jak v jazyce C# pomocí vybraných nástrojů vytvořit optimalizační aplikaci.

Dále jsem vytvořil framework JD, který poskytuje jednotné rozhraní solverů (viz 4.2.1) a umožňuje jednoduše formulovat optimalizační úlohy pomocí matic (viz 4.2.1).

Navržená platforma je vhodná pro implementaci modelu evropské přenosové soustavy, jak ukazují výsledky zátěžového testu (viz 4.10). Srovnávací test (viz 4.8) zase dokazuje, že program vytvořený na nové platformě je o více než 90 % rychlejší než ten vytvořený v Matlabu – původní platformě.

Literatura

[1] Antonio Goméz-Expósito, Antonio J. Conejo, Claudio Canizares. Electric Energy Systems Analysis and Operation. CRC Press, 2009.

[2] MathWorks – Accelerating the pace of engineering and science [online]. 2012 [cit. 2010-04-20]. Matlab – The Language of Technical Computing. Dostupné z WWW: <http://www.mathworks.com/products/matlab>

[3] YALMIP Wiki [online]. 2012 [cit. 2010-04-20]. What Is YALMIP. Dostupné z WWW: <http://users.isy.liu.se/johanl/yalmip/pmwiki.php?n=Main.WhatIsYALMIP>

[4] Wikipedie – Otevřená encyklopedie [online]. 2012 [cit. 2010-04-20]. MATLAB. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Matlab>

[5] YALMIP Wiki [online]. 2012 [cit. 2010-04-20]. Solvers. Dostupné z WWW: <http://users.isy.liu.se/johanl/yalmip/pmwiki.php?n=Solvers.Solvers>

[6] Microsoft Visual Studio [online]. 2012 [cit. 2010-04-20]. Dostupné z WWW: <http://www.microsoft.com/cze/msdn/vstudio/2010>

[7] Eclipse [online]. 2012 [cit. 2010-04-20]. Eclipse Downloads. Dostupné z WWW: <http://www.eclipse.org/downloads>

[8] NetBeans [online]. 2012 [cit. 2010-04-20]. NetBeans – The Smarter Way to Code. Dostupné z WWW: <http://netbeans.org/features/index.html>

[9] PODHRADSKÝ, Michal. Modelling languages for optimization [online]. Bachelor Thesis. Dostupné z WWW: <https://support.dce.felk.cvut.cz/mediawiki/images/0/0b/Bp_2010_podhradsky_michal.pdf>

[10] AIMMS – The Modeling System [online]. 2012 [cit. 2010-04-21]. AIMMS Purchase Order. Dostupné z WWW: <http://www.aimms.com/aimms/cgi/order.cgi?o=purchase>

[11] ASCEND overview [online]. 2012 [cit. 2010-04-21]. Dostupné z WWW: <http://ascend4.org/ASCEND_overview>

[12] IBM [online]. 2012 [cit. 2010-04-21]. Guide to API reference manuals of CPLEX. Dostupné z WWW: <http://publib.boulder.ibm.com/infocenter/cosinfoc/v12r2/index.jsp>

[13] CVXOPT – Python Software for Convex Optimization [online]. 2012 [cit. 2010-04-21]. Dostupné z WWW: <http://abel.ee.ucla.edu/cvxopt>

[14] FICO Xpress-Mosel [online]. 2012 [cit. 2010-04-21]. Mosel language. Dostupné z WWW: <http://www.fico.com/en/Products/DMTools/xpress-overview/Pages/Xpress-Mosel.aspx>

[15] Homepage of FLOPC++ [online]. 2012 [cit. 2010-04-21]. Dostupné z WWW: <https://projects.coin-or.org/FlopC++>

[16] GAMS [online]. 2012 [cit. 2010-04-21]. Welcome to the GAMS Home Page! Dostupné z WWW: <http://www.gams.com>

[17] Gurobi Optimization [online]. 2012 [cit. 2010-04-21]. Gurobi Optimizer Reference Manual. Dostupné z WWW: <http://www.gurobi.com/documentation/4.5/reference-manual>

[18] JOptimizer [online]. 2012 [cit. 2010-04-21]. About. Dostupné z WWW: <http://www.joptimizer.com/index.html>

[19] JModelica.org [online]. 2012 [cit. 2010-04-21]. About JModelica.org. Dostupné z WWW: <http://www.jmodelica.org>

[20] LINDO Systems Inc [online]. 2012 [cit. 2010-04-21]. An Overview of LINGO. Dostupné z WWW: <http://www.lindo.com/index.php?option=com_content&view=article&id=2&Itemid=10>

[21] MSDN Library [online]. 2012 [cit. 2010-04-22]. Microsoft Solver Foundation. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/ff524509(v=vs.93).aspx>

[22] Maximal Software [online]. 2012 [cit. 2010-04-22]. The Key Features of MPL. Dostupné z WWW: <http://www.maximal-usa.com/mpl/features.html>

[23] OpenModelica [online]. 2012 [cit. 2010-04-22]. Introduction. Dostupné z WWW: <http://www.openmodelica.org/index.php>

[24] OpenOpt Framework [online]. 2012 [cit. 2010-04-22]. Dostupné z WWW: <http://openopt.org/OOFramework>

[25] Python [online]. 2012 [cit. 2010-04-22]. PuLP 1.4.7. Dostupné z WWW: <http://pypi.python.org/pypi/PuLP/1.4.7>

[26] PuLP 1.4.9 documentation [online]. 2009 [cit. 2010-04-22]. Pulp classes. Dostupné z WWW: <http://packages.python.org/PuLP/pulp.html>

[27] Pyomo [online]. 2010 [cit. 2010-04-22]. Overview. Dostupné z WWW: <https://software.sandia.gov/trac/coopr/wiki/Pyomo>

[28] HART E. William. Pyomo: Python Optimization Modeling Objects [online]. Sandia National Laoratories. 2009. 23 s. Dostupné z WWW: <https://software.sandia.gov/trac/coopr/export/5691/coopr.misc/trunk/doc/pub/09-01-Hart%20Pyomo.pdf>

[29] HART E. William. Pyomo: Modeling and Solving Mathematical Programs in Python [online]. Sandia National Laoratories. 2011. 42 s. Dostupné z WWW: <https://software.sandia.gov/trac/coopr/raw-attachment/wiki/Pyomo/pyomo-jnl.pdf>

[30] TOMLAB Optimization [online]. 2011 [cit. 2010-04-22]. About Tomnet. Dostupné z WWW: <http://tomopt.com/tomnet/about/>

[31] TOMLAB Optimization [online]. 2011 [cit. 2010-04-22]. TOMNET Optimization. Dostupné z WWW: <http://tomopt.com/tomnet/optimization>

[32] Zimpl [online]. 2011 [cit. 2010-04-22] What is Zimpl. Dostupné z WWW: <http://zimpl.zib.de>

[33] MSDN Library [online]. 2012 [cit. 2010-04-22]. Introduction to the C# Language and the .NET Framework. Dostupné z WWW: <http://msdn.microsoft.com/cs-cz/library/z1zx9t92>

[34] MSDN Library [online]. 2012 [cit. 2010-04-22]. Konceptuální přehled rozhraní .NET framework. Dostupné z WWW: <http://msdn.microsoft.com/library/zw4w595w.aspx>

[35] Gurobi Optimization [online]. 2012 [cit. 2010-04-22]. .NET Reference Manual. Dostupné z WWW: <http://www.gurobi.com/documentation/4.6/reference-manual/node349>

[36] Microsoft Visual Studio [online]. 2012 [cit. 2010-04-22]. Visual C# Express – Free tools to create .NET applications on Windows using Visual C#. Dostupné z WWW: <http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-csharp-express>

[37] Microsoft Visual Studio [online]. 2012 [cit. 2010-04-22]. Porovnání a ceny. Dostupné z WWW: <http://www.microsoft.com/cze/msdn/vstudio/porovnani.aspx>

[38] Microsoft Download Center [online]. 2012 [cit. 2010-04-22]. Microsoft .NET Framework 4 (webová instalační služba). Dostupné z WWW: <http://www.microsoft.com/downloads/cs-cz/details.aspx?FamilyID=9cfb2d51-5ff4-4491-b0e5-b386f32c0992>

[39] Microsoft Visual Studio [online]. 2012 [cit. 2010-04-22]. Nejčastější dotazy. K čemu mohu použít nástroje Visual Studio Express? Dostupné z WWW: <http://www.microsoft.com/cze/msdn/vstudio/nejcastejsi-dotazy.aspx>

[40] Doxygen [online]. 2012 [cit. 2010-04-22]. Doxygen Download Page. Dostupné z WWW: <http://www.stack.nl/~dimitri/doxygen/download.html#latestsrc>

Přiložené CD

Přiložené CD obsahuje elektronickou verzi tohoto dokumentu, vytvořený framework JD (včetně zdrojového kódu a dokumentace) a další zdrojové kódy související s prací.

- **\dp_hakjosef.pdf** – Elektronická verze tohoto dokumentu.
- **\prog_languages_test** – Zdrojové kódy (případně i spustitelné soubory) programů pro testování rychlosti vytváření objektů různými programovacími jazyky. Složka také obsahuje soubor testResults.xlsx s výsledky měření a grafy.
  - **\C#** – Implementace v jazyce C#.
  - **\C++** – Implementace v jazyce C++.
  - **\matlab** – Implementace v jazyce Matlab.
  - **\python** – Implementace v jazyce Python.
- **\energy_grid_model** – Zdrojové kódy implementace modelu přenosové soustavy pomocí různých modelovacích nástrojů. Složka také obsahuje soubor testResults.xlsx s výsledky měření a grafy.
  - **\yalmi**p – Implementace pomocí kombinace Matlab & Yalmip.
  - **\csAPI** – Implementace pomocí .NET (C#) API solveru Gurobi.
  - **\pyAPI** – Implementace pomocí Python API solveru Gurobi.
  - **\pulp** – Implementace pomocí balčku PuLP jazyka Python.
  - **\jd** – Implementace pomocí kombinace .NET (C#) API solverů Gurobi a Cplex & framework JD.
    - **\simple** – Jednoduchá varianta pro učely porovnání s ostatními nástroji.
    - **\advanced** – Složitější varianta (uzly a linky).
- **\MyGurobiApplication** – Zdrojový kód optimalizační aplikace vytvořené ve Visual C# Express.
- **\JD** – Framework JD. Složka obsahuje zdrojové kódy projektu, knihovnu JD.NET.dll a zdrojové kódy JD ovladačů pro solvery Gurobi a Cplex.
  - **\JD.NET** – Zdrojové kódy (rozhraní solverů a modelovací nástroj).
  - **\GurobiJD** – Zdrojové kódy JD ovladače pro solver Gurobi.
  - **\CplexJD** – Zdrojové kódy JD ovladače pro solver Cplex.
  - **\doc** – Dokumentace frameworku JD (html a pdf).