VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Jednoduchý nástroj pro ověřování ekvivalence systémů: Rozhodování bisimilarity na BPP

Simple Equivalence Checking Tool: Deciding Bisimilarity of BPP

2013 Bc. Jakub Juřica

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jakub Juřica**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Jednoduchý nástroj pro ověřování ekvivalence systémů: Rozhodování bisimilarity na BPP
Simple Equivalence Checking Tool: Deciding Bisimilarity of BPP

Zásady pro vypracování:

Cílem práce je sestrojit jednoduchý nástroj pro ověřování bisimulační ekvivalence mezi dvěma systémy specifikovanými jako základní paralelní procesy. Příslušný algoritmus je popsán v anglicky psaném článku slovně nebo pseudokódem. Algoritmus bude vysvětlen i vedoucím práce na konzultacích a student má za úkol jej jen implementovat.

Postup práce:
1. Nastudovat pojmy Basic Parallel Processes (BPP, základní paralelní procesy) a bisimulační ekvivalence a popis požadovaného algoritmu.
2. Vytvořit nástroj rozhodující ekvivalenci mezi dvěma zadanými BPP systémy.
3. Nástroj bude co nejvíce modulární, aby se do něj daly v budoucnu doplňovat algoritmy rozhodující jiné ekvivalence nebo pracující s jinými modely apod.
4. Vstupní systémy bude moci vhodným způsobem zadat uživatel, ale taky bude k dispozici generátor náhodných vstupů podle parametrů (počet proměnných, abeceda atd.) zvolených uživatelem.
5. Součástí práce bude i porovnání časů výpočtu a velikosti použité paměti na vygenerovaných nebo zadaných vstupech různých velikostí.

Seznam doporučené odborné literatury:

Jančar P.: Strong Bisimilarity on Basic Parallel Processes is PSPACE-complete; In Proceedings 18th LiCS (Logic in Computer Science), IEEE Computer Society, 2003, pp. 218--227, http://www.cs.vsb.cz/jancar/lics2003.ps

Další podle pokynů vedoucího práce

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Martin Kot, Ph.D.**

Datum zadání: 16.11.2012

Datum odevzdání: 07.05.2013

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 26. dubna 2013 .....

Rád bych na tomto místě poděkoval svému vedoucímu práce Ing. Martinu Kotovi, Ph.D.
za odborné vedení, pomoc, věnovaný čas a užitečné rady při tvorbě této diplomové práce.

Abstrakt

Mezi standardní techniky verifikace systémů patří ověřování ekvivalencí. Ověřují se různé ekvivalence na různých modelech systému. Tato diplomová práce byla zaměřena na dva, v literatuře dostupné, algoritmy pro ověřování bisimulační ekvivalence mezi dvěma systémy specifikovanými jako základní paralelní procesy. Cílem bylo vytvoření verifikačního nástroje, který rozhodne bisimulační ekvivalenci na zadaných vstupech různých velikostí. V tomto textu je popsána tvorba tohoto nástroje včetně pojmů důležitých k pochopení použitých algoritmů. Korektnost použitých algoritmů a jejich asymptotická složitost vyplývá z citovaných zdrojů, práce se tímto nezabývá.

Klíčová slova: základní paralelní procesy, bisimulační ekvivalence, bisimilarita, verifikační nástroj, ověřování ekvivalencí

Abstract

The standard techniques of verification systems includes the verification of equivalences. On the various types of models of system are checking different equivalences. This thesis was focused on two algorithm verifying bisimulation equivalence between two systems specific as a basic parallel processes which are available in literature. The goal was to create a verification tool that will decide bisimulation equivalence on different inputs of various sizes. This text describe how was the tool created including concepts important to the understanding of the algorithms. Correctness of the algorithms and their asymptotic complexity results from the cited sources, thesis does not include this.

Keywords: basic parallel processes, bisimulation equivalence, bisimilarity, verification tool, equivalence checking

Seznam použitých zkratek a symbolů

BPP	–	Basic Parallel Processes
	–	základní paralelní procesy
LTS	–	Labelled Transition Systems
	–	ohodnocené přechodové systémy
MAX	–	maximum
SECT	–	Simple Equivalence Checking Tool
	–	jednoduchý nástroj pro ověřování ekvivalence systémů
kB	–	kilobyte, jednotka množství dat
s	–	sekunda, základní jednotka času

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

V této diplomové práci se budeme zabývat tvorbou nástroje, který je určený k ověřování bisimulační ekvivalence na vstupních systémech specifikovaných jako základní paralelní procesy. V kapitole 2 jsou popsány a vysvětleny důležité teoretické pojmy vyskytující se v této práci a vztahy mezi těmito pojmy.

Kapitola 3 se zaměřuje na modelování a analýzu systému. Popisuje vztahy mezi jednotlivými systémovými komponentami a možnostmi, jak nástroj využívat z pohledu uživatele.

Nejdůležitější částí je kapitola 4. Ta se zabývá implementací samotného nástroje. Nejprve je vysvětlena struktura a funkčnost jednotlivých komponent systému, následně se zaměřujeme na ověřování bisimulační ekvivalence pomocí dvou základních algoritmů. Těmi jsou exponenciální a polynomiálně prostorový algoritmus.

Kapitola 5 je vyhrazena pro práci se vstupními systémy. Popisuje části zabývající se uživatelsky definovaným vstupem a části generující náhodné vstupy podle zadaných parametrů.

Předposlední kapitola 6 se zaměřuje na výsledky experimentů s časovou a prostorovou složitostí při ověřování bisimulační ekvivalence pomocí exponenciálního i polynomiálního algoritmu nad vstupními daty.

Na závěr je v kapitole 7 uvedeno celkové zhodnocení výsledků práce a její přínos.

1.1 Cíle práce

Prvním cílem je vytvořit efektivní nástroj ověřující bisimulační ekvivalenci na zadaných vstupech. Ten by měl být co nejvíce modulární, aby do něj v budoucnu šly doplnit další algoritmy rozhodující jiné ekvivalence nebo pracující s jinými modely.

Druhým cílem je vytvořit možnosti vhodného zadávání vstupních systémů manuálně, ale také pomocí generátoru náhodných vstupů podle zadaných parametrů zvolených uživatelem.

Posledním cílem je porovnat zadané, popřípadě vygenerované, vstupy různých velikostí z hlediska časové a prostorové složitosti výpočtu bisimulační ekvivalence. Tedy z hlediska časů výpočtů a velikosti využité paměti při průběhu jednotlivých algoritmů.

2 Základní pojmy

Tato kapitola je zaměřená na teoretický popis a vysvětlení základních pojmů vyskytujících se v této práci. Čtenář se seznámí s principem ohodnocených přechodových systémů. Dále se dozví, co jsou to bisimulační ekvivalence, základní paralelní procesy a jak spolu tyto pojmy souvisí. Informace k této kapitole jsem čerpal z publikací [1, 2, 3].

2.1 Ohodnocené přechodové systémy

Základním konceptem pro popis systému ověřující bisimulační ekvivalenci jsou takzvané ohodnocené přechodové systémy (Labelled Transition Systems – LTS).

Definice 2.1 *LTS jsou formálně definovány jako trojice (S, A, $\longrightarrow$), kde:*

- *S je (potencionálně nekonečná) množina stavů,*
- *A je konečná množina akcí,*
- *$\longrightarrow \subseteq$ S x A x S je přechodová relace.*

Neboli můžeme říct, že S je (potencionálně nekonečná) množina všech možných stavů systému. A je množina navenek pozorovatelných akcí, které má daný systém. Přechodová relace představuje chování systému.

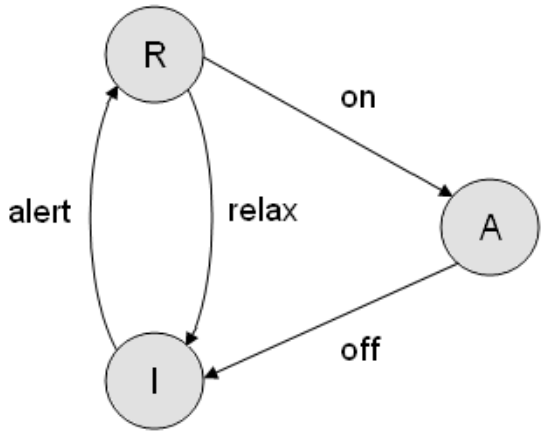
Místo $(s, a, s') \in \longrightarrow$ používáme $s \xrightarrow{a} s'$, což můžeme číst jako „systém ve stavu s může provést akci a a přejít do stavu s'“. Zápis $s \xrightarrow{a} s'$ může být rozšířen na posloupnost akcí $w = a_1, a_2, \ldots, a_n$, kde $w \in A^*$. Poté píšeme $s \xrightarrow{w} s'$ právě tehdy, když existuje posloupnost stavů $s_0, s_1, \ldots, s_n$ taková, že $s_0 = s, s_n = s'$ a $s_{i-1} \xrightarrow{a_i} s'_i$ pro každé i takové, že $1 \leq i \leq n$.

Stav s' je dosažitelný ze stavu s (psáno $s \longrightarrow^* s'$) právě tehdy, když existuje nějaké $w \in A^*$ takové, že $s \xrightarrow{w} s'$.

Příklad 2.1
Mějme následující LTS:
$S = I, R, A$
$A = alert, relax, on, off$
$\rightarrow = (I, alert, R), (R, relax, I), (R, on, A), (A, off, I)$

Jednoduchý LTS se třemi stavy, čtyřmi akcemi a čtyřmi přechodovými relacemi je zobrazen na obrázku 1.

■



Obrázek 1: Grafické zobrazení jednoduchého LTS

2.2 Bisimulační ekvivalence

Nechť (S, A, $\longrightarrow$) je LTS. Binární relace $R \subseteq S \times S$ je bisimulace právě tehdy, když pro každou dvojici stavů $(s,t) \in R$ a každou akci $a \in A$ jsou splněny následující podmínky:

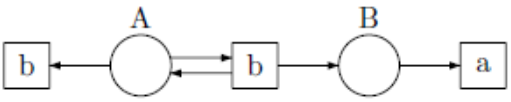
- Jestliže existuje nějaké $s' \in S$ takové, že $s \xrightarrow{a} s'$, pak musí existovat nějaké $t' \in S$ takové, že $t \xrightarrow{a} t'$ a $(s',t') \in R$.
- Jestliže existuje nějaké $t' \in S$ takové, že $t \xrightarrow{a} t'$, pak musí existovat nějaké $s' \in S$ takové, že $s \xrightarrow{a} s'$ a $(s',t') \in R$.

Poznámka 2.1 Toto nám říká, že $s \xrightarrow{a} s'$ odpovídá $t \xrightarrow{a} t'$, respektive $t \xrightarrow{a} t'$ odpovídá $s \xrightarrow{a} s'$.

Stavy s,t jsou bisimulačně ekvivalentní (bisimilární), psáno $s \sim t$, právě tehdy, když existuje nějaká bisimulace R taková, že $(s,t) \in R$. Relace $\sim$ se nazývá bisimulační ekvivalence neboli bisimilarita.

Bisimulační ekvivalence na LTS (S, A, $\longrightarrow$) může být alternativně popsána z hlediska podmínek bisimulační hry, kterou hrají dva hráči označení jako útočník a obránce. Pozice ve hře jsou dvojice $(s,t) \in S \times S$. Hra probíhá v kolech. V pozici (s,t) si útočník vybere jeden stav z dvojice s a t (řekněme s) a některý přechod vedoucí z tohoto vybraného stavu (řekněme $s \xrightarrow{a} s'$). Obránce poté vybere některý přechod z druhého stavu se stejným ohodnocením, které vybral útočník (řekněme $t \xrightarrow{a} t'$). Hra pokračuje v dalším kole v pozicích (s',t'). Pokud některý z hráčů uvázne v některé pozici ve hře, tedy nemá žádnou další možnost volby pohybu, vyhrává jeho soupeř. V případě, že je hra nekonečná, je považován za vítěze obránce.

Poznámka 2.2 $s \sim t$ právě tehdy, když v bisimulační hře startující v pozici (s,t) má obránce vítěznou strategii.



Obrázek 2: BPP z příkladu 2.2

Přestože je bisimulační ekvivalence definována na stavech jednoho LTS, může být vždy použita také na stavech dvou různých LTS, díky možnosti jejich disjunktního sjednocení.

2.3 Základní paralelní procesy

Základní paralelní procesy (Basic Parallel Processes – BPP) mohou být definovány následovně:

Definice 2.2 *BPP systém Δ je, vzhledem k množině atomických akcí $Act = \{a, b, \ldots\}$ a množině procesních proměnných $Var = \{X, Y, \ldots\}$, konečná množina pravidel ve tvaru*

- $X \xrightarrow{a} Y_1 Y_2 \ldots Y_n$.

Var_Δ je poté množina proměnných a Act_Δ množina akcí vyskytujících se v pravidlech Δ. Z BPP systému Δ může vzniknout LTS, které má Var^*_Δ jako stavy, Act_Δ jako akce, a kde

- $M \xrightarrow{a} M'$

právě tehdy, když existuje nějaké pravidlo $X \xrightarrow{a} \alpha$ v Δ takové, že $X \in M$
a $M' = (M - \{X\}) \cup mset(\alpha)$.

Sekvenci proměnných α neboli $mset(\alpha)$ značíme multimnožinu prvků z α. Je to tedy funkce přiřazující ke každé proměnné X počet výskytů v α .

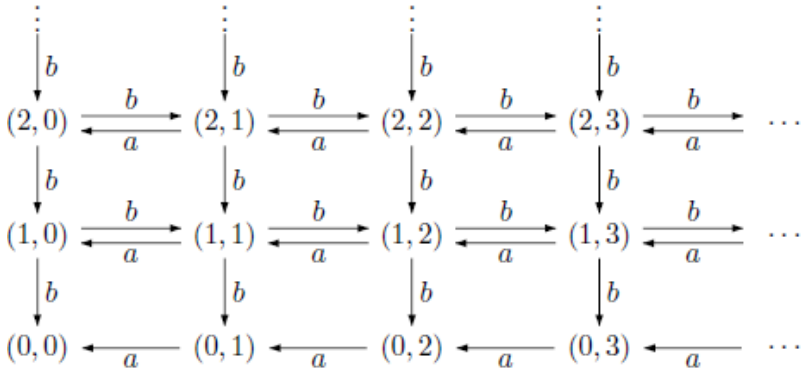
Příklad 2.2
Mějme následující gramatiku reprezentující BPP systém:

$Var = \{A, B\}$
$Act = \{a, b\}$

$A \xrightarrow{b} \varepsilon$
$A \xrightarrow{b} AB$
$B \xrightarrow{a} \varepsilon$

Grafické zobrazení tohoto systému je na obrázku 2.

LTS odpovídající zadanému BPP z příkladu pak zobrazuje obrázek 3. ■



Obrázek 3: LTS odpovídající zadanému BPP z příkladu 2.2

Pro účely této diplomové práce je vhodnější reprezentovat BPP systém jako BPP síť, tedy speciální typ Petriho sítě, kde má každý přechod právě jedno vstupní místo.

Definice 2.3 *Formálně je BPP síť n-tice* $\Sigma = (P, Tr, \text{PRE}, \text{POST}, \lambda)$*, kde:*

- P *je konečná množina míst,*
- Tr *je konečná množina přechodů,*
- PRE: $Tr \to P$ *je funkce, která přiřazuje vstupní místo* $\text{PRE}(t)$ *každému přechodu* t*,*
- POST: $Tr \to P^{\oplus}$ *($P^{\oplus}$ je množina multimnožin nad P) je funkce, která přiřazuje množinu výstupních míst* $\text{POST}(t)$ *každému přechodu* t*, kde počet výskytů místa* p *v* $\text{POST}(t)$ *reprezentuje počet tokenů vkládaných do místa* p *přechodem* t*;*
- $\lambda : Tr \to Act$ *je funkce přiřazující přechodům akce z* Act*.*

Používáme funkci $F : Tr \times P \to N$ takovou, že $F(t, p)$ je počet výskytů p v $\text{POST}(t)$, tj. $\text{POST}(t)(p)$. Hodnota $F(t, p)$ reprezentuje počet tokenů vkládaných přechodem t do místa p. $PostPlaces(t)$ značíme množinu výstupních míst přechodu, tj. $PostPlaces(t) = \{p \in P | F(t, p) > 0\}$. Definujeme $tr(p) = \{t \in Tr |\ \text{PRE}(t) = p\}$

Značení ($Marking$) M je množina míst, tj. prvek z $P^{\oplus}$. Množinu všech značení z Σ označujeme $M(\Sigma)$. Pro označení prázdného značení používáme M_0, tj. značení M takové, že $\forall p \in P : M(p) = 0$. Místo p je označeno v značení M pokud $M(p) > 0$, v opačném případě v něm označeno není.

Přechod t je uschopněn v M jestliže $M(\text{PRE}(t)) \geq 1$. Uschopněný přechod může být provedený s výsledkem $M' = (M - \{\text{PRE}(t)\}) \cup \text{POST}(t)$, poté píšeme $M \xrightarrow{t} M'$.

BPP síť $\Sigma = (P, Tr, \text{PRE}, \text{POST}, \lambda)$ definuje LTS s množinou stavů $M(\Sigma)$, abecedou Act a přechodovou relací, kde $M \xrightarrow{a} M'$ právě tehdy, když $M \xrightarrow{t} M'$ pro nějaké $t \in Tr$

takové, že $\lambda(t) = a$.

BPP proces je dvojice (M_0, Σ), kde je Σ BPP systém a $M_0 \in M(\Sigma)$ je počáteční značení.

BPP systém Σ je normovaný právě tehdy, když pro každé $M \in M(\Sigma)$ platí $M \longrightarrow^* M_0$.

2.4 Bisimilarita na BPP

Hlavní částí, kterou se v průběhu této práce budeme zabývat je rozhodování, zda pro BPP systém (M_1, Σ) a (M_2, Σ) platí $M_1 \sim M_2$. Pokud bude vstupem dvojice BPP systému (M_1, Σ_1) a (M_2, Σ_2), pak využijeme možnosti jejich disjunktního sjednocení do jednoho systému Σ.

- Problém: Bisimilarita na BPP
- Vstup: BPP systém Σ, značení $M, M' \in M(\Sigma)$.
- Otázka: Platí $M \sim M'$?

Problém rozhodování bisimilarity na normovaném BPP (nBPP) je speciální případ rozhodování bisimilarity na BPP, kde je zapotřebí, aby byl vstupní BPP systém Σ normovaný.

Aby byla BPP síť Σ se dvěma značeními M_1, M_2 bisimilární, je zřejmou (nutnou) podmínkou, aby pro každou akci $a \in Act(\Sigma)$ buď oba značení M_1, M_2 uschopňují a-přechod nebo oba značení M_1, M_2 zneschopňují všechny a-přechody. Obecněji tedy vzdálenost k zneschopnění a, tj. délka nejkratší sekvence přechodů vedoucí k značení, které zneschopňuje všechny a-přechody, musí být stejná pro oba značení M_1, M_2. Je potřeba vzít na vědomí, že tato vzdálenost může být i nekonečná. Tím dostaneme několik funkcí $d_1, d_2, \ldots, d_k$ typu $M(\Sigma) \to N_\omega$ (jednu pro každé $a \in Act(\Sigma)$), na kterých se musí značení M_1, M_2 schodovat ($d_i(M_1) = d_i(M_2)$ pro všechna $i = 1, 2, \ldots, k$).

U takové BPP sítě Σ má každé d_i přiřazenou množinu míst Q_i, kde $Q_i = \text{PRE}(T)$ pro množinu přechodů T takovou, že $d_i(M)$ je délka nejkratší cesty z M do nějakého značení, kde není označeno Q_i, jinými slovy $d_i = \text{NORM}_{Q_i}$. Pro funkci $\text{NORM}_Q : M(\Sigma) \to N_\omega$ platí:

- $\text{NORM}_Q = 0$,
- $\text{NORM}_Q(p) = 0$ pro $p \in (P - Q)$,
- $\text{NORM}_Q(p) = 1 + \min\{\text{NORM}_Q(\text{POST}(t)) | t \in tr(p)\}$ pro $p \in Q$,
- $\text{NORM}_Q(M) = \Sigma_{p \in P}\ \text{NORM}_Q(p) \cdot M(p) = \Sigma_{p \in Q}\ \text{NORM}_Q(p) \cdot M(p)$.

Říkáme, že $R \subseteq P$ je past, jestliže pro jakékoliv t, takové že $\text{PRE}(t) \in R$, platí $\text{POST}(t) \cap R \neq \emptyset$. Vzhledem k tomu, že $\emptyset$ je past a sjednocení pastí je rovněž past, sjednocení všech pastí v množině $Q \subseteq P$ je největší past v Q označována $\text{TRAP}(Q)$.

3 Modelování

3.1 Specifikace požadavků

Tato kapitola se zabývá specifikací funkčních a nefunkčních požadavků, kladených na jednoduchý nástroj pro ověřování ekvivalence BPP systémů (Simple equivalence checking tool – SECT) a na jeho podsystém generující vstupní systémy. Specifikuje také jednotlivé případy užití SECT uživatelem.

3.1.1 Funkční požadavky

Cílem je sestrojit jednoduchý nástroj pro ověřování bisimulační ekvivalence mezi dvěma systémy specifikovanými jako základní paralelní procesy. Tento nástroj musí být co nejvíce modulární tak, aby do něj v budoucnu šly přidávat algoritmy rozhodující i jiné ekvivalence nebo pracující s jinými modely.

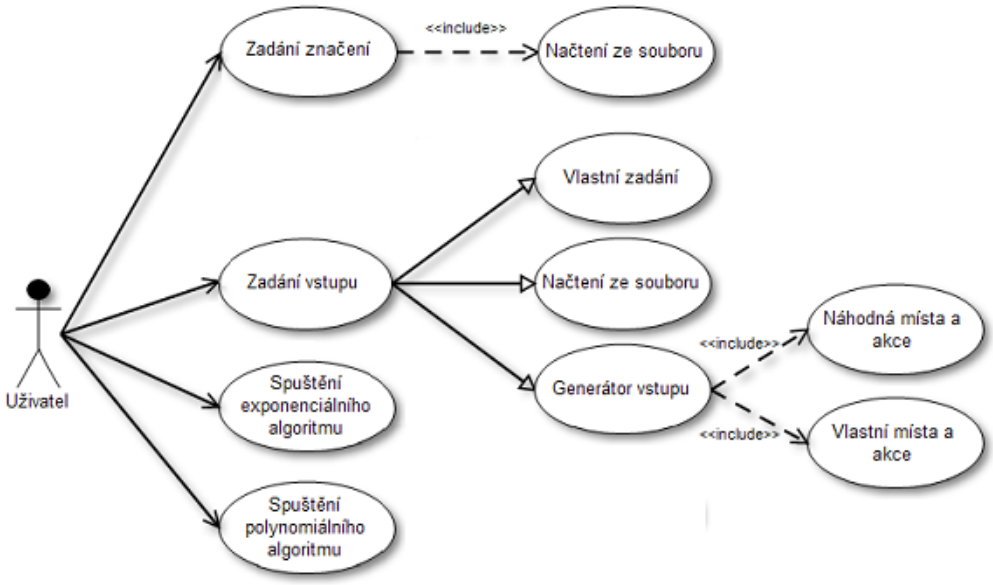
Vstupem tohoto nástroje budou dva BPP systémy (popřípadě jeden systém) a dva značení ($Marking$), pomocí kterých se bude zjišťovat bisimilarita. Zadávání vstupních BPP systémů bude probíhat třemi způsoby – vlastním zadáním, otevřením ze souboru nebo náhodným vygenerováním. Vstupům se budeme podrobněji věnovat především v kapitole 5. Značení bude možno zadat načtením ze souboru, ale také se náhodně vygeneruje při generování vstupu.

Výstupem nástroje pak bude ohodnocení, zda-li jsou zadané vstupy bisimulačně ekvivalentní či nikoliv. Zde použijeme dvou různých způsobů tohoto ověření. Tedy exponenciální nebo polynomiální algoritmus.

U obou algoritmů provedeme experimentování s porovnáváním časů výpočtu a velikosti použité paměti na zadaných vstupech různých velikostí (viz. kapitola 6) při ověřování bisimilarity.

3.1.2 Nefunkční požadavky

Nástroj pro ověřování ekvivalence BPP systémů je naimplementován pomocí jazyka C#, ve vývojovém prostředí Microsoft Visual Studio 2010 společnosti Microsoft a využívá platformu Microsoft .NET framework 4.0.



Obrázek 4: Use Case – Diagram případů užití

3.1.3 Případy užití

Diagram případů užití (obrázek 4) popisuje, jak lze SECT využívat uživatelem. K dispozici jsou tyto hlavní možnosti:

- Zadání vstupu
- Zadání značení
- Spuštění exponenciálního algoritmu
- Spuštění polynomiálního algoritmu

Scénáře jednotlivých případů užití, tak jak jsou uvedeny na obrázku 4, slouží k následujícím účelům.

- Zadání značení – využívá scénář případu užití načtení značení ze souboru. Uživatel vybere soubor, který chce načíst jako značení. Systém ověří korektnost načítaných dat. V případě, že jsou korektní, zobrazí načtené značení v SECT, v opačném případě upozorní na chybná data v souboru.
- Zadání vstupu – se dělí na tři speciální případy zadání vstupního BPP systému:
    1. Vlastní zadání – Uživatel zadává jednotlivé přechody vstupního systému. U každého z nich musí vytvořit množinu výstupních míst. Ta se vytváří tak, že do ní uživatel přidává jednotlivá výstupní místa společně s jejich násobností hrany, přičemž se kontroluje korektnost takto zadávaných údajů. Následně uživatel vyplní vstupní místo s akcí a přiřadí přechod do zvoleného BPP systému.

Systém následně ověřuje korektnost i těchto zadávaných údajů. V případě jakékoliv nekorektnosti zobrazí uživateli informaci o chybě v zadávaných datech, jinak přiřadí přechod do BPP systému a zobrazí jej uživateli v SECT. Uživatel může následně přidávat další přechody.

2. Načtení ze souboru – Uživatel vybere soubor, který chce načíst jako vstupní BPP systém. Systém ověří korektnost načítaných dat. V případě, že jsou korektní, zobrazí daný vstupní systém uživateli v SECT, v opačném případě upozorní na chybná data v souboru.
3. Generátor náhodných vstupů – Zde uživatel spustí generátor vstupů. Systém jej zobrazí a uživatel musí zadat veškerá vstupní data pro generování. Opět systém kontroluje, zda jsou takováto data zadána správně. Pokud ne, dojde k upozornění na chybu, pokud ano, systém vygeneruje vstupní systém společně se vstupním značením a zobrazí je v SECT. K zadávaní vstupních dat patří zadání míst a akcí, které budou použity ke generování. Proto připojujeme dva následující scénáře:
   - Náhodná místa a akce – Uživatel zadá počet míst a počet akcí, ze kterých se bude generovat vstup. Systém tyto položky automaticky vygeneruje podle zadaného počtu a zobrazí jejich seznam v generátoru.
   - Vlastní místa a akce – Uživatel načte seznam míst a seznam akcí ze souboru, popřípadě si manuálně zadává místa a akce, která chce při generování využívat. Systém znova kontroluje korektnost jako v předešlých scénářích. Obě tyto možnosti lze kombinovat.

   Dalšími zadávanými daty je počet přechodů generovaného BPP, maximální počet výstupních míst u jednotlivých přechodů, maximální násobnost hrany kteréhokoliv výstupního místa, maximální počet vyprazdňujících (tj. $epsilon$) přechodů, maximální počet tokenů v kterémkoliv místě v značení.

- Spuštění exponenciálního algoritmu – Uživatel zvolí možnost ověření bisimulační ekvivalence mezi zadanými vstupy pomocí exponenciálního algoritmu. Systém provede výpočet, ověření, změření času a množství využité paměti při průběhu algoritmu. Tyto údaje následně zobrazí uživateli v SECT.

- Spuštění polynomiálního algoritmu – Uživatel zvolí možnost ověření bisimulační ekvivalence mezi zadanými vstupy pomocí polynomiálního algoritmu. Systém provede výpočet, ověření, změření času a množství využité paměti při průběhu algoritmu. Tyto údaje následně zobrazí uživateli v SECT.

3.2 Analýza a návrh

Následující kapitola je zaměřena na architekturu SECT. Pomocí diagramu aktivit popisuje dynamické aspekty systému, vysvětluje jeho procesy a jak spolu komunikují. V části návrhu se zabýváme tím, jak byly jednotlivé části programu i celý nástroj navrženy.

3.2.1 Diagram aktivit

Zde jsou uvedeny některé základní diagramy aktivit. Nejprve je na obrázku 5 znázorněn průběh ověření ekvivalence BPP systémů z hlediska bisimulační ekvivalence. Jako první probíhá načtení samotného vstupu, které lze provést třemi způsoby. Rozhoduje se mezi uživatelsky definovaným, načteným nebo náhodně vygenerovaným vstupním BPP systémem. Je-li potřeba zadat druhý vstup dojde k opakování tohoto procesu. Poté proběhne načtení prvního a druhého značení. Následující krok zahrnuje rozhodnutí, zda chceme ověřit bisimilaritu pomocí exponenciálního nebo pomocí polynomiálního algoritmu. Poté proběhne zobrazení celkového výsledku ověření a aktivita končí.

Diagram na obrázku 6 znázorňuje vygenerování náhodného vstupního BPP systému. Nejdříve se musí zadat množina míst a množina akcí BPP systému. K tomuto účelu slouží tři možnosti. Tedy vlastní uživatelsky definovaná množina míst a množina akcí, náhodně vygenerovaná množina míst a množina akcí, popřípadě množina míst a množina akcí načtená ze souboru. Následujícím krokem proběhne vložení zbývajících důležitých vstupních parametrů (tj. počet přechodů, atd.) budoucího BPP systému. Nakonec se provede samotné vygenerování a aktivita končí.

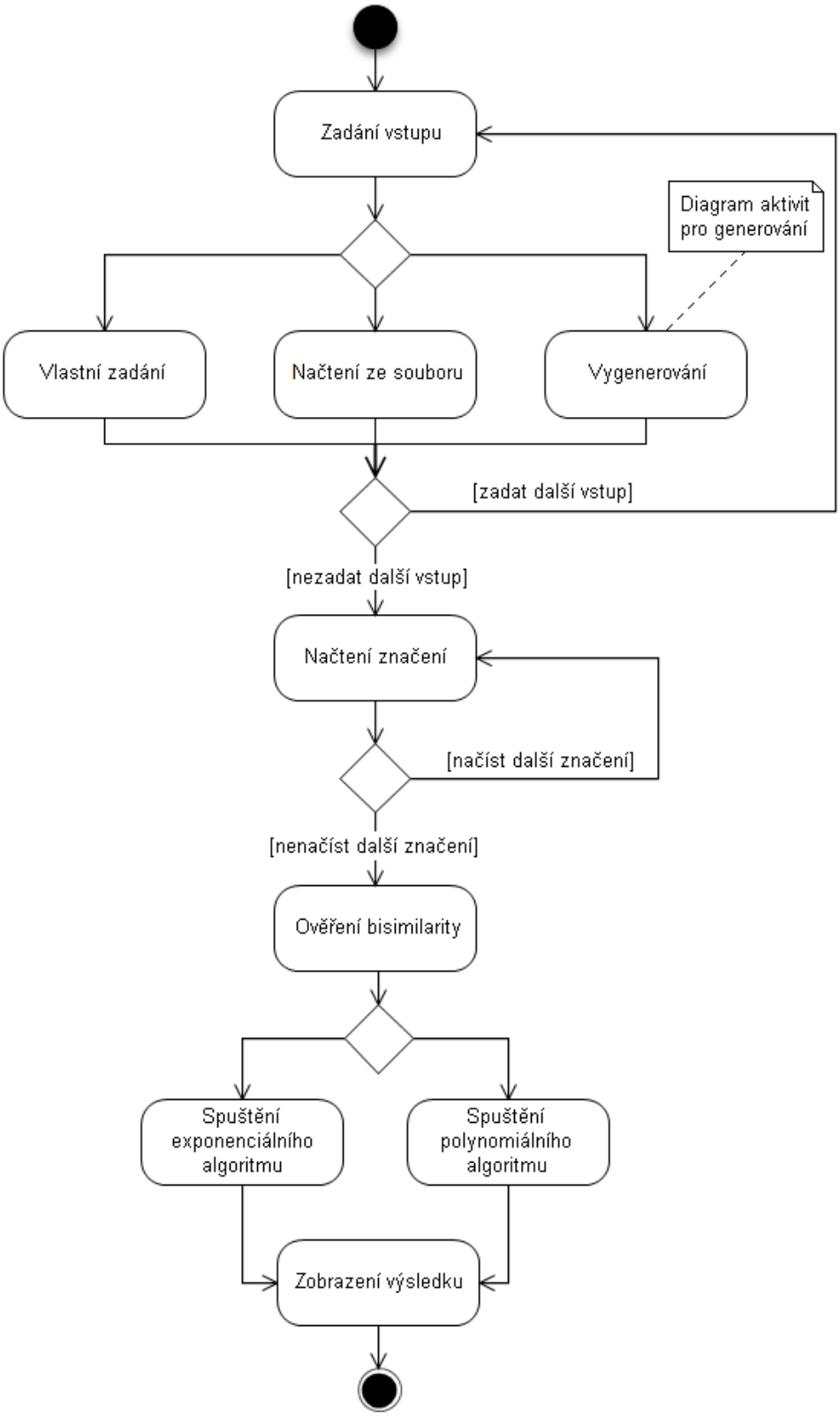
3.2.2 Diagram tříd

Diagram tříd popisuje statickou strukturu systému, znázorňuje datové struktury a operace u objektů a souvislosti mezi objekty. Znázorňuje datový model systému od konceptuální úrovně až po implementaci. Datové struktury zařazuje do tříd a zobrazuje vztahy těchto tříd[4]. Diagram uvedený na obrázku 7 znázorňuje rozvržení tříd v SECT.

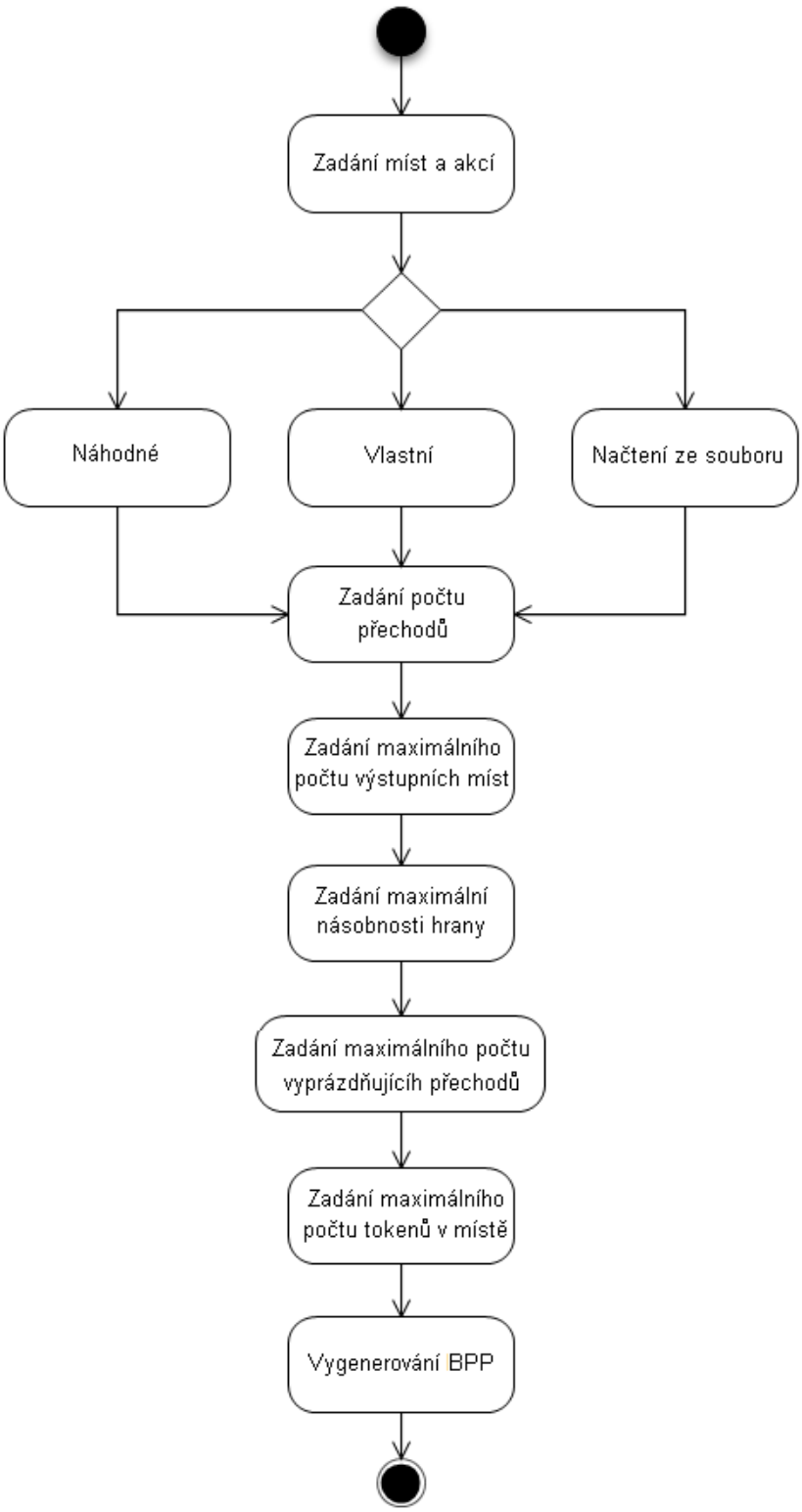
Hlavní částí je třída $Tool$ představující nástroj rozhodující bisimulační ekvivalenci na BPP systémech (třída BPP) a zadaných značeních (třída $Marking$). K ověřování bisimilarity využíváme exponenciální algoritmus (třída $Exponencial$) nebo polynomiální algoritmus (třída $Polynomial$). Nástroj navíc umožňuje vygenerovat vstupní systém pomocí generátoru (třída $Generator$).

Algoritmy dále počítají normy na BPP pomocí třídy $Norm$, využívají množinové operace třídy Set a počítají značení pomocí třídy $Marking$. Exponenciální algoritmus navíc vytváří rozklady reprezentovány třídou $Decomposition$. Samotný BPP systém se skládá z jednotlivých přechodů (třída $Transition$), přechody pak z jednotlivých míst (třída $Place$).

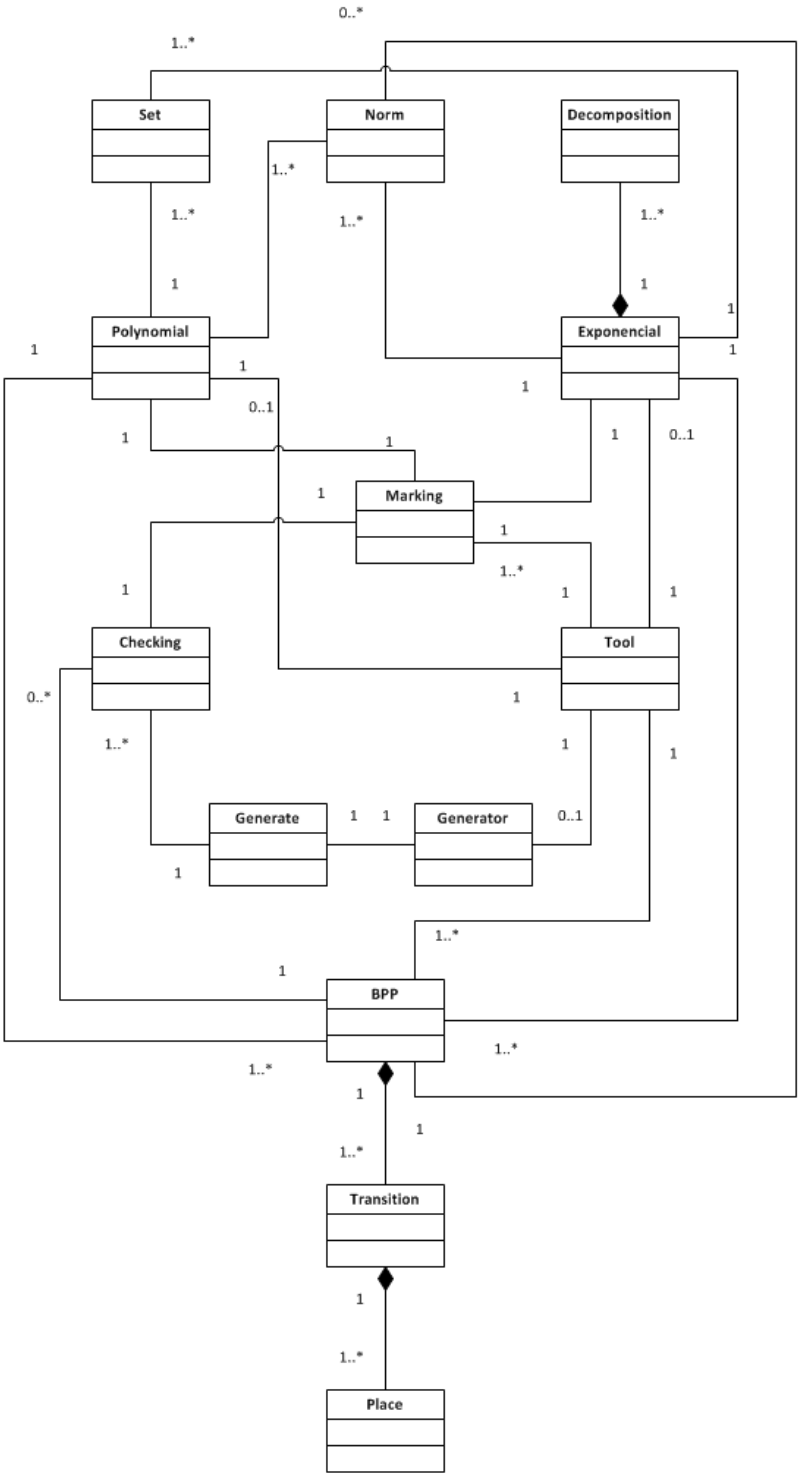
Generátor ke generování BPP systému využívá třídu $Generate$. Ta stejně jako třída BPP a třída $Marking$ ověřuje korektnost vstupních údajů pomocí třídy $Checking$.



Obrázek 5: Diagram aktivit – Ověřování bisimilarity na BPP



Obrázek 6: Diagram aktivit – Generování náhodného vstupního BPP systému



Obrázek 7: Třídní diagram – SECT

4 Implementace

V této kapitola se budeme zabývat popisem implementace nástroje ověřujícího bisimulační ekvivalenci na zadaných vstupech. Rozebereme si jednotlivé třídy diagramu tříd z obrázku 7, vysvětlíme jejich funkčnost a jaké metody implementují.

4.1 Vzhled nástroje

Na obrázku 8 je zobrazen vzhled naprogramovaného SECT. Ten se skládá z menu, kde jsou (mimo jiné) možnosti otevření BPP systému ze souboru a otevření značení ($Marking$) ze souboru, kterým se budeme blíže věnovat v kapitole 5.

Hlavní část obsahuje čtyři seznamy. Tedy BPP1 a BPP2, které zobrazují vstupní systémy a Marking M1 s Marking M2, které zobrazují vstupní značení. Uprostřed se nachází volba vytvoření vlastního přechodu. Díky tomu má uživatel možnost přidat nový přechod do již existujícího vstupního systému, ale také takto vytvořit vlastní uživatelsky definovaný BPP systém (viz. kapitola 5). Pod tím následuje spuštění generátoru náhodného vstupního BPP1, popřípadě BPP2. Nakonec jsou zde volby ověření bisimulační ekvivalence, spolu s vyhodnocením časové a prostorové složitosti výpočtu, pomocí exponenciálního nebo polynomiálního algoritmu.

4.2 Reprezentace nástroje

Jak je patrno z obrázku 9, samotný SECT reprezentuje třída $Tool$. Ta pro svou funkcionalitu dále asociuje třídy BPP, $Marking$ a $Generator$.

4.2.1 Třída BPP

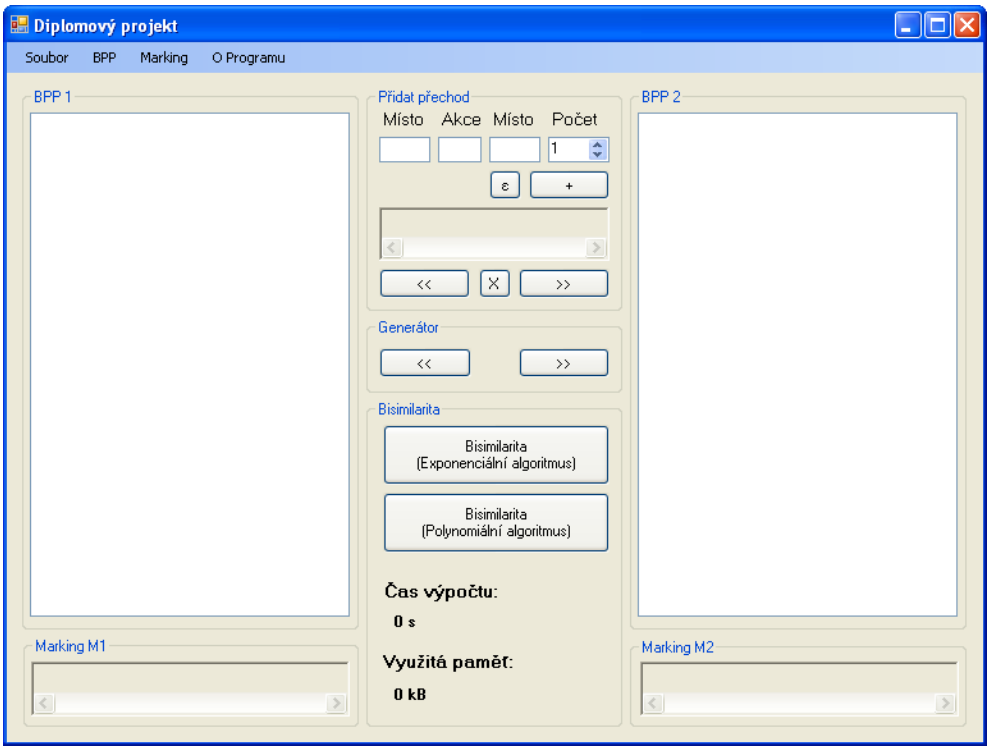
Představuje vstupní BPP systém a je jednou z nejdůležitějších tříd SECT. Její metody slouží k vytváření BPP systému, ale také k ověření bisimulační ekvivalence na zadaných vstupech pomocí exponenciálního a polynomiálního algoritmu. Funkcionalitě jednotlivých metod budeme věnovat pozornost v dalším průběhu práce.

4.2.2 Třída Marking

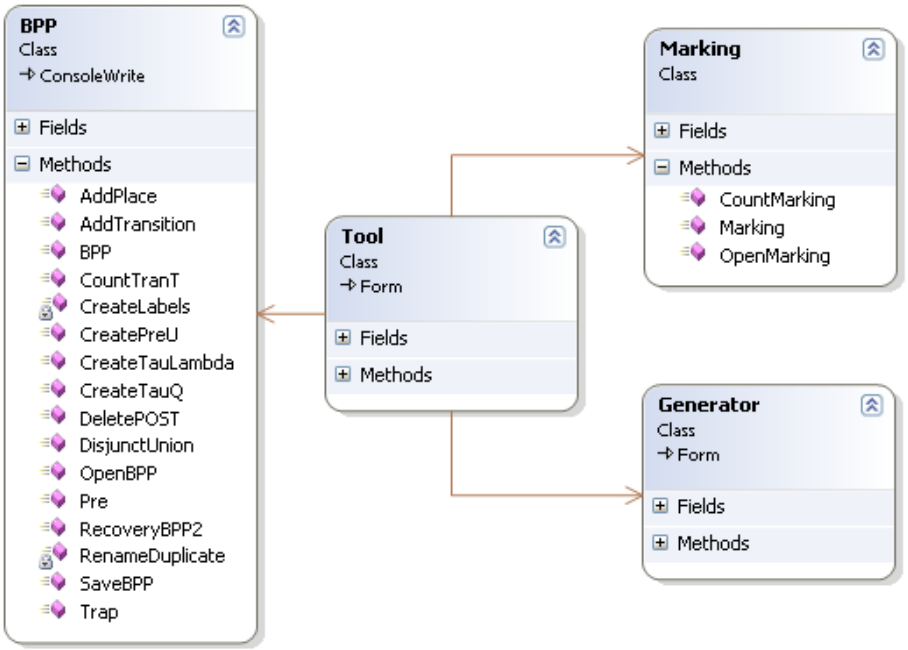
Zde nástroj využívá metodu $OpenMarking$, pomocí které načítáme vstupní značení ze souboru. Ty reprezentují počet tokenů v jednotlivých místech BPP. Metoda $SaveMarking$ poté slouží k uložení takovýchto dat k dalšímu použití.

4.2.3 Třída Generator

Tato třída reprezentuje Generátor, který umožňuje vygenerovat náhodný vstupní BPP systém. K tomuto účelu využívá třídu $Generate$ a dohromady tvoří subsystém, jehož funkcionalitě budeme věnovat pozornost v kapitole 5.



Obrázek 8: Jednoduchý nástroj ověřující ekvivalenci systémů



Obrázek 9: Třídy – Tool, BPP, Generator a Marking

4.3 Reprezentace BPP systému

Třídy reprezentující BPP systém jsou zobrazeny na obrázku 10 včetně jejich proměnných, vlastností a metod. Jak je z tohoto obrázku patrno, každý vstupní systém reprezentuje třída BPP. Ta vytváří seznam objektů (BPP systém) typu $Transition$ a ty se skládají z objektů typu $Place$.

4.3.1 Třída Place

Reprezentuje místo z konečné množiny $P \in \Sigma$, tedy místo z konečné množiny míst BPP systému. Každé místo má označení $\_Place$, dále normu $Norm$ a nakonec násobnost hrany $Number$.

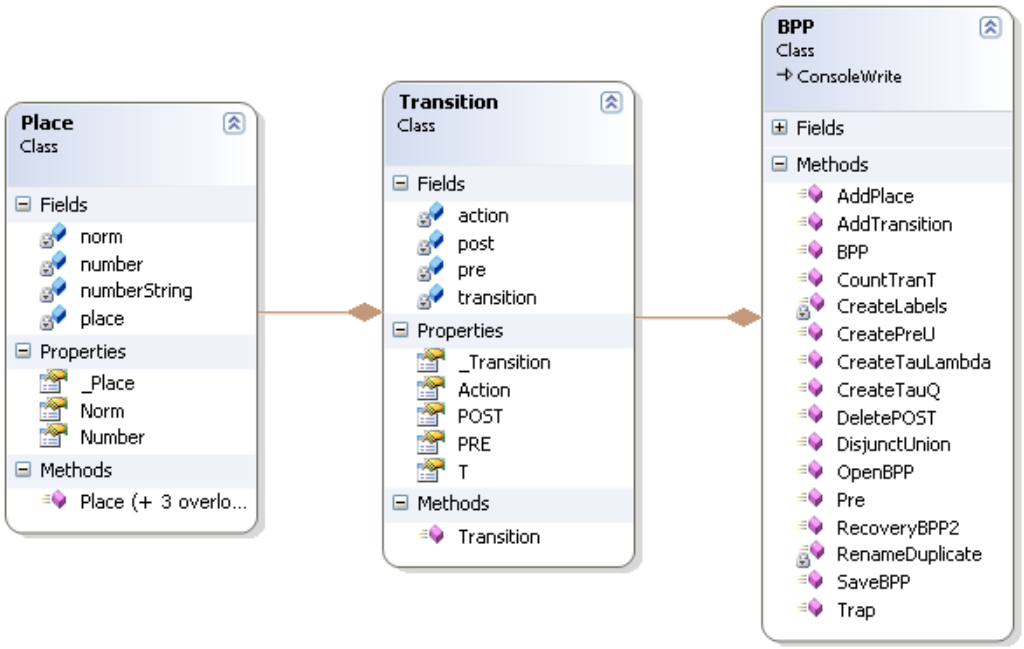
4.3.2 Třída Transition

Reprezentuje přechod z konečné množiny $Tr \in \Sigma$, tedy přechod z konečné množiny přechodů BPP systému. Skládá se z označení přechodu $\_Transition$, vstupního místa PRE (objekt typu $Place$), ohodnocení přechodu $action$ (akce), množiny výstupních míst $POST$ (objekty typu $Place$) a pomocné hodnoty přechodu T.

4.3.3 Třída BPP

Lze rozdělit na tři oddíly. První část metod $OpenBPP$, $SaveBPP$, $AddPlace$, $DeletePOST$ a $AddTransition$ slouží k práci se vstupními BPP systémy. Těmi se budeme blíže zabývat v kapitole 5. Druhá část se věnuje sjednocení BPP systémů. Zde patří metody $DisjunctUnion$, $RecoveryBPP$, $CreateLabels$ a $RenameDuplicate$. Poslední oddíl metod využíváme v kapitolách exponenciální a polynomiální algoritmus a jejich funkcionalitu probereme právě nyní.

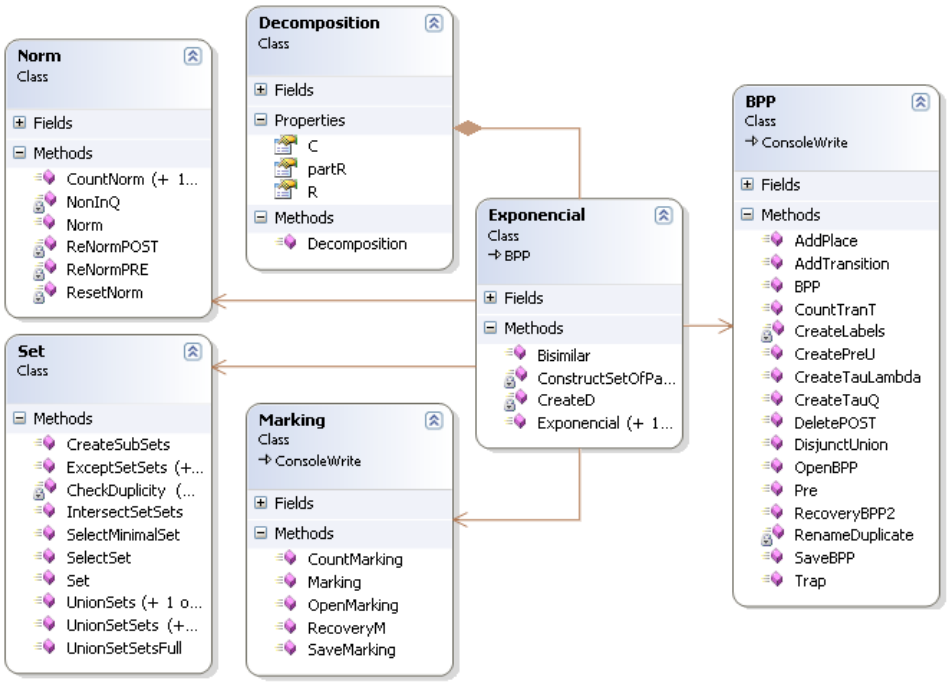
- CreatePreU – Jako vstupní parametr získá množinu míst Q, přičemž $Q \subseteq P$. Následně zavolá funkci $CountNorm$ s parametrem Q, pro spočítání norem u příslušných míst (viz. třída $Norm$), kde Q je právě tato vstupní množina. Poté prochází všechny přechody a vytváří množinu $preU$, tedy seznam dvojic vstupní místo a norma tohoto místa. Tu vrátí jako výsledek.
- Metoda CreateTauLambda – Očekává na vstupu množinu přechodů BPP systému. Každý takovýto přechod má některé ohodnocení (akci – $action$). Tato metoda postupně rozděluje všechny přechody do množin právě podle jejich shodného ohodnocení. Návratová hodnota je množina množin přechodů, rozdělených podle stejné akce.
- Metoda CreateTauQ – Má opět jako vstupní parametr množinu přechodů BPP. K nim nejprve pomocí metody $CountNorm$ spočítá normy míst (viz. třída $Norm$). Takovýto BPP systém, mající spočítané normy, předá parametrem funkci $CountTranT$



Obrázek 10: Třídy – BPP, Transition, Place

k vypočtení pomocných hodnot přechodů T. Podle těchto hodnot pak rozdělí přechody do množin podle stejného T (podobně jako metoda $CreateTauLambda$) a tu vrátí jako výsledek.

- Metoda CountTranT – Jak bylo uvedeno výše, ke každému přechodu s místy ohodnocenými normami vypočítá hodnotu T. Pokud je norma vstupního nebo kterékoliv z výstupních míst rovna nekonečnu, bude tato hodnota u daného přechodu taktéž rovna nekonečnu. V opačném případě do proměnné $numPOST$ sečte součin normy a násobnosti hrany každého příslušného výstupního místa. Od něj odečte normu vstupního místa a výsledek uloží jako hodnotu T.
- Metoda Pre – K vstupní množině přechodů označené T, hledá jejich vstupní místa. Tedy prochází všechny přechody BPP systému a tam, kde se přechod rovná přechodu z T, přiřadí do množiny vstupních míst pre jeho vstupní místo. Tuto množinu pak vrací jako výsledek.
- Metoda Trap – Vstupním parametrem je množina míst označená Q, kde $Q \subseteq P$ a množina všech přechodů systému. Metoda prochází obě množiny, dokud nenajde všechna místa z Q taková, která mají normu ve vstupních místech BPP rovnou nekonečnu. Nalezená místa uloží do množiny $trap$ a tu vrátí jako výsledek.



Obrázek 11: Třída Exponencial a třídy, které využívá

4.4 Exponenciální algoritmus

Třída $Exponencial$ slouží k ověření bisimilarity na zadaných vstupech exponenciálním algoritmem. K tomuto účelu využívá třídy BPP, $Norm$, $Marking$, Set a $Decomposition$. Celé schéma je uvedeno na obrázku 11.

4.4.1 Třída Norm

Souží výhradně k počítání norem míst BPP systému. Norma místa je číslo, které udává nejkratší cestu (posloupnost přechodů) k odstranění jednoho tokenu v tomto místě pryč z BPP. Na začátku algoritmu jsou normy všech míst nastaveny na nekonečno. Pokud nemá systém výstupní přechody (tj. přechody $epsilon$), pak nemůže být normovaný a token v kterémkoliv místě nikdy neodstraníme z BPP. Podobně i pasti nikdy neodstraňují token z daného místa, a proto norma tohoto místa je vždy nekonečná. V ostatních případech používáme k výpočtu následující metody:

- Metoda CountNorm – Je uvedena ve výpisu kódu 1. Na úvod zavolá privátní metodu $ResetNorm$. Dále následuje vnější cyklus, na jehož začátku se vždy nastaví logická proměnná $unprocessed$ na hodnotu $nepravda$. V něm postupně prochází všechny přechody a ke každému výstupnímu místu (z množiny výstupních míst $POST$) ve vnitřním cyklu kontroluje následující podmínky:

- Pokud je norma výstupního místa rovna nekonečnu nebo pokud má vstupní místo (PRE) normu různou od nekonečna, výpočet neproběhne a vnitřní cyklus končí.
- Pokud je norma vstupního místa rovna 0, výpočet také neproběhne a vnitřní cyklus končí.
- Pokud je výstupní místo $epsilon$ a zároveň je norma vstupního místa různá od 1, nastaví se norma vstupního místa na 1. Logické proměnné $unprocessed$ je přiřazena hodnota $pravda$, výpočet opět neproběhne a vnitřní cyklus končí.
- Pokud neplatí ani jeden z výše zmíněných bodů, do proměnné $normPOST$ přičteme součin násobnosti hrany daného výstupního místa s jeho normou a pokračujeme ve vnitřním cyklu dalším místem z $POST$.

Z předcházejících bodů vyplývá, že pouze v případě provedení bodu 4 u všech míst z $POST$, můžeme provést výpočet normy místa PRE. Ten se provádí tak, že k proměnné $normPOST$ přičteme 1 a výsledek uložíme jako novou normu tohoto vstupního místa daného přechodu. Poté nastavíme logickou proměnnou $unprocessed$ na hodnotu $pravda$.

Zpracování přechodu končí kontrolou různosti vstupního místa od nekonečna. V případě nerovnosti voláme metodu $ReNormPOST$. Nakonec vyhodnotíme proměnnou $unprocessed$. Pokud je rovna hodnotě $pravda$, znovu procházíme všechny přechody v BPP. V případě hodnoty $nepravda$ zavoláme metodu $ReNormPRE$, ukončíme vnější cyklus a tím výpočet norem míst končí.

```
public List<Transition> CountNorm() {
    BigInteger norm = 0;
    BigInteger normTransition = 1;
    bool unprocessed = false;

    ResetNorm();
    do {
        unprocessed = false;
        foreach (var tran in setTransition) {
            bool change = true;
            BigInteger normPOST = 0;
            foreach (var POST in tran.POST) {
                if ((POST.Norm == infinite) || (tran.PRE.Norm != infinite)) {
                    change = false;
                    break;
                } else if (tran.PRE.Norm == "0") {
                    change = false;
                    break;
                } else if ((POST._Place == epsilon) && (tran.PRE.Norm != "1")) {
                    tran.PRE.Norm = "1";
                    change = false;
                    unprocessed = true;
                    break;
                } else {
                    BigInteger POST_Norm = BigInteger.Parse(POST.Norm);
```

```
                    normPOST += (POST.Number * POST_Norm);
                }
            }
            if (change) {
                norm = normPOST + normTransition;
                tran.PRE.Norm = norm.ToString();
                unprocessed = true;
            }
            if (tran.PRE.Norm != infinite) {
                ReNormPOST(tran.PRE);
            }
        }
    }
    while (unprocessed);
    ReNormPRE();
    return setTransition;
}
```

Výpis 1: Ukázka počítání norem

- Metoda CountNorm s parametrem Q – Se velmi podobá metodě $CountNorm$ bez parametru. Jediným rozdílem je, že po zavolání funkce $ResetNorm$ zavolá navíc funkci $NonInQ$ a předá ji parametrem množinu míst $nonQ$.
- Metoda ReNormPOST – Jako vstup dostane vstupní místo PRE. Následně porovnává jeho normu se všemi výstupními místy ze všech přechodů, které jsou s ním shodné. Pokud je daná norma místa PRE menší než norma porovnávaného místa, nastaví ji tuto hodnotu.
- Metoda ReNormPRE – Projde v cyklech postupně všechny dvojice vstupních míst, které jsou shodné. Porovná hodnoty jejich norem a všem nastaví nejmenší nalezenou hodnotu.
- Metoda ResetNorm – Provádí resetování norem na základní hodnotu. To znamená, že nastaví normy všech míst ve všech přechodech BPP systému na nekonečno.
- Metoda NonInQ – Má jako vstupní parametr množinu míst Q. Poté prochází všechny přechody BPP systému a místům nevyskytujícím se v této množině nastaví normu na 0.

4.4.2 Třída Set

Zahrnuje operace s množinami. K reprezentaci všech množin, uvedených v této práci, jsem použil C# kolekci Hashset[5], jelikož nabízí efektivní způsob provádění množinových operací potřebných v dále popsaných algoritmech. Metody $UnionSets$, $UnionSetSets$, $ExceptSetSets$, $CheckDuplicity$ jsou přetížené[6] a mohou jako vstup zpracovat jak množiny přechodů, tak množiny míst.

4.4.3 Třída Exponencial

Obsahuje konstruktor, jednu veřejnou a dvě privátní metody. Jak lze vidět ve výpisu kódu 2, existují dva typy konstruktoru. První má jako vstup pouze jeden BPP systém, na kterém snadno ověříme bisimilaritu pomocí značení M1 a M2. Druhý případ je komplikovanější tím, že je zadaný také druhý vstup. Jak jsme si ovšem uvedli v kapitole 1, lze jednoduše provést disjunktní sjednocení BPP1 s BPP2 a ověření bisimulační ekvivalence pomocí značení M1 a M2 aplikovat na sjednocený systém.

```
public Exponencial(List<Transition> _BPP, Dictionary<string, int> M1, Dictionary<string, int>
    M2)
{
   this.setTran = _BPP;
   this.M1 = M1;
   this.M2 = M2;
}
public Exponencial(List<Transition> BPP1, List<Transition> BPP2, Dictionary<string, int> M1,
    Dictionary<string, int> M2)
{
   this.setTran = DisjunctUnion(BPP1, BPP2, M2);
   this.M1 = M1;
   this.M2 = M2;
}
```

Výpis 2: Přetížený konstruktor třídy Exponencial

K sjednocení BPP systémů se využívá tří metod třídy BPP:

- DisjunctUnion,
- CreateLabels,
- RenameDuplicate.

$DisjunctUnion$ nejprve zavolá metodu $CreateLabels$. Ta vrátí množinu všech míst, které v daných vstupech nejsou duplicitní a množinu míst, která duplicitní jsou. Poté zavolá metodu $RenameDuplicate$, která pomocí identifikovaných duplicit přepíše místa v BPP2 a značení M2 tak, aby byla unikátní. Nakonec je provedeno sjednoceni $BPP1$ s $BPP2$ do jednoho unikátního vstupního systému.

4.4.3.1 Metoda Bisimilar Jak je uvedeno ve výpisu 3, funkčnost spočívá ve vytvoření množiny D pomocí metody $CreateD$ a následné ověření, zda-li pro každé $Q \in D$ platí $NORM_Q(M_1) = NORM_Q(M_2)$. Funkce $CreatePreU$ patřící do třídy BPP, metodou $CountNorm$ z třídy $Norm$ spočítá normy míst určených množinou Q a poté vytvoří seznam těchto míst společně s jejich normami, označený $preU$. $NORM_Q$ představuje funkci $CountMarking$ z třídy $Marking$, která má za úkol spočítat hodnoty značení $M1, M2$ vzhledem k $preU$ a vyhodnotit jejich rovnost. V případě, že se rovnají, jsou zadaná značení BPP systému bisimulačně ekvivalentní, v opačném případě ekvivalentní nejsou.

```
public bool Bisimilar()
{
    Dictionary<string, string> preU;
    Marking M = new Marking();
    int actualQ = 1;
    string m1 = "";
    string m2 = "";

    List<HashSet<string>> D = CreateD();
    Console.WriteLine("Pocet Q je: " + D.Count);
    foreach (var Q in D) {
        Console.WriteLine("Zpracovavam Q: " + actualQ++);
        Console.SetCursorPosition(0, Console.CursorTop - 1);
        preU = BPP.CreatePreU(setTran, Q);
        m1 = M.CountMarking(preU, M1);
        m2 = M.CountMarking(preU, M2);
        if (!m1.Equals(m2)) {
            return false;
        }
    }
    return true;
}
```

Výpis 3: Ukázka ověřování bisimilarity pomocí exponenciálního algoritmu

4.4.3.2 Metoda CreateD

Vytváří množinu D, která vypadá následovně:

- $D = \{R \cup \text{PRE}(T) \mid R \in C, T \in \text{PART}(R)\}$,

kde C a $\text{PART}(R)$ představují prvky Q, v našem případě tedy objekty typu $Decomposition$. Tyto jednotlivé objekty se vytvářejí metodou $ConstructSetOfPairs$ popsanou v následující kapitole.

4.4.3.3 Metoda ConstructSetOfPairs

Představuje metodu naprogramovanou podle pseudoalgoritmu z obrázku 12. Jeho princip spočívá v odhalení pastí, které se v BPP systému objevují. Past je definována jako takové vstupní místo, kde přechod odebírá a zároveň vrací token zpět do tohoto místa. Tím pádem nikdy nemůže dojít k jeho vyprázdnění.

V prvním kroku se vytvoří množina C, tedy množina obsahující množiny jednotlivých pastí, kde první pastí BPP systému je prázdná množina. Dále potřebujeme množinu $ProcessedTraps$, již zpracovaných množin z množiny C, která bude v kroku 2 prozatím prázdná.

Krok číslo 3 představuje začátek cyklu, který končí krokem 22. Ten se bude provádět do té doby, dokud algoritmus nezpracuje všechny množiny z C. Proto v následujícím kroku 4 dojde k zavolání metody $ExceptSetSets$ z třídy Set, která vrátí množinu množin všech doposud nezpracovaných pastí. Z té se pak metodou $SelectMinimalSets$ vybere minimální

```
1   C := {∅}
2   ProcessedTraps := ∅
3   while C − ProcessedTraps ≠ ∅ do
4       select R from C − ProcessedTraps minimal wrt. ⊆
5       T := T_λ
6       for each R′ ∈ ProcessedTraps s.t. R′ ⊂ R do
7           for each T′ ∈ PART(R′) do
8               Q := R′ ∪ PRE(T′)
9               if TRAP(Q) ⊆ R then
10                  T := T ⊓ T_Q
11      ProcessedClasses := ∅
12      while T −ProcessedClasses ≠ ∅ do
12          select T from T −ProcessedClasses
14          Q := R ∪ PRE(T)
15          R′ := TRAP(Q)
16          if R′ ⊆ R then
17              T := T ⊓ T_Q
18          else
19              C := C ∪ {R′ ∪ R″ | R″ ∈ C}
20          ProcessedClasses := ProcessedClasses ∪ {T}
21      PART(R) := T
22      ProcessedTraps := ProcessedTraps ∪ {R}
```

Obrázek 12: Pseudoalgoritmus metody ConstructSetOfPairs převzatý z [3]

nezpracovaná past R. Krok 5 vytvoří množinu $\mathcal{T}$ zavoláním metody $CreateTauLambda$ z třídy BPP. Ta rozdělí přechody do množin podle toho, jakou akcí jsou označeny a vrátí množinu obsahující tyto množiny, čili rozklad množiny přechodů podle akcí.

Následují cyklus začínající krokem 6 zpracovává všechny ty množiny R' ze zpracovaných množin pastí $ProcessedTraps$, které jsou vlastními podmnožinami aktuálně zpracovávané pasti R. Cyklus v kroku 7 postupně vybírá všechny množiny přechodů T' z rozkladu PART. Funkcí Pre z třídy BPP získá vstupní místa těchto přechodů a ty následně metodou $UnionSets$ třídy Set sjednotí s množinou z kroku 6 a uloží do množiny Q. Funkce $Trap$ v kroku 9 poté rozhodne, zda některé vstupní místo, které bylo přidáno, je past. Předtím proto musí dojít k spočítání norem na všech místech BPP systému zavoláním metody $CountNorm$ z třídy $Norm$, kde ve výsledku místa s normou rovnající se nekonečnu jsou pasti. Pokud je tedy nalezena past ve Q, a zároveň je toto místo podmnožinou zpracovávané pasti R, proběhne krok 10. Zde využijeme funkce $CreateTauQ$ pro vytvoření množin přechodů podle Q a poté provedeme průnik této množiny s množinou z kroku 5. Tím získáme nový rozklad množin přechodů $\mathcal{T}$. Při prvním průchodu cyklu

z kroku 3 se kroky 6 – 10 provádět nebudou, proběhnou až v případě, že bude nalezena a zpracována některá past v dalším průběhu algoritmu.

Krok 11 nastaví množinu zpracovaných tříd $\mathcal{T}$ tedy množinu $ProcessedClasses$ na prázdnou. Cyklus v krocích 12 – 20 pak zpracovává každou nezpracovanou množinu z množiny $\mathcal{T}$. Krokem 13 se opět zavolá metoda $ExceptSetSets$, která vrátí všechny nezpracované třídy a jedna z nich se uloží do množiny T. Krokem 14 dojde k vytvoření množiny Q sjednocením metodou $UnionSets$ aktuálně zpracovávané pasti z kroku 4 a vstupních míst množiny T získaných stejně jako v kroku 8 metodou Pre. Dále opět spočítáme normy na všech místech BPP systému zavoláním metody $CountNorm$ z třídy $Norm$ a krokem 15 uložíme do množiny R' ty místa, která jsou ve Q pastmi. Jestliže je nová množina R' podmnožinou aktuálně zpracovávané pasti R, je proveden krok 17 ekvivalentně jako krok 10. V opačném případě zavoláme metodu $UnionSetSetsFull$ z třídy Set, která nám sjednotí všechny množiny z C s množinou R a poté přiřadí množinu R do C. Průchod cyklem končí krokem 20, kdy je množina T z kroku 13 sjednocena do zpracovaných množin $ProcessedClasses$ pomocí funkce $UnionSetSets$.

Krokem 21 zaznamenáme nový rozklad PART rovnající se zpracovanému rozkladu $\mathcal{T}$. Do seznamu objektů typu $Decomposition$ zaznamenáme past R, k ní daný rozklad PART a vytvořenou množinu C z kroku 19. Nakonec krokem 22 sjednotíme metodou $UnionSetSets$ past R se zpracovanými množinami $ProcessedTraps$. Pokud jsou zpracovány všechny množiny z C, algoritmus končí a vrací seznam objektů $Decomposition$.

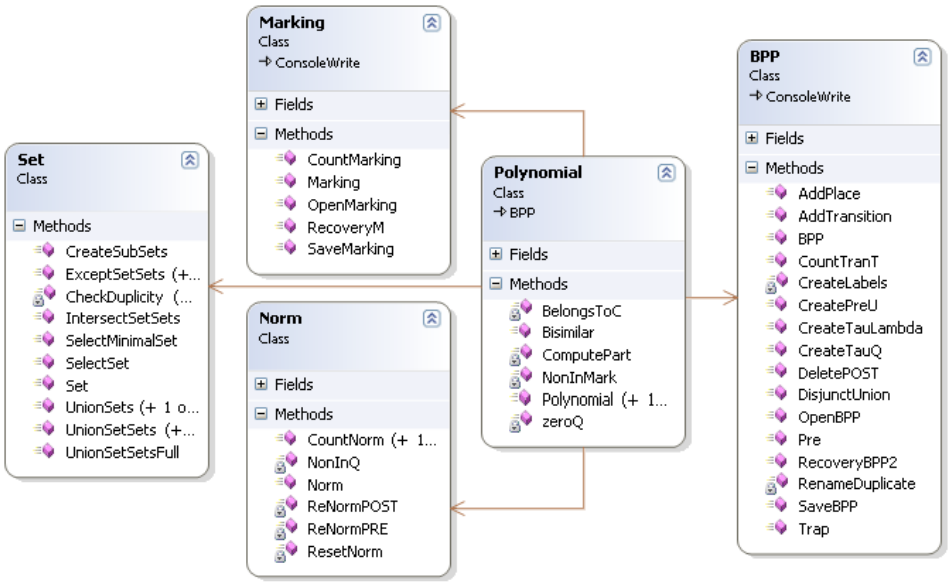
4.5 Polynomiální algoritmus

Třída $Polynomial$ slouží k ověření bisimilarity na zadaných vstupech polynomiálním algoritmem. K tomuto účelu stejně jako třída $Exponenciual$ využívá třídy BPP, $Norm$, $Marking$ a Set. Celé schéma je uvedeno na obrázku 13.

4.5.1 Třída Polynomial

Obsahuje konstruktor, jednu veřejnou a čtyři privátní metody. Jak lze vidět ve výpisu kódu 4, opět existují dva typy konstruktoru. Tedy podobně jako u Exponenciálního algoritmu pokud máme dva vstupy, musíme provést disjunktní sjednocení BPP1 s BPP2 a ověření bisimulační ekvivalence pomocí značení M1 a M2 aplikovat na sjednocený systém. V případě jednoho vstupu ověřujeme bisimilaritu rovnou pomocí vstupních značení M1, M2.

```
public Polynomial(List<Transition> _BPP, Dictionary<string, int> M1, Dictionary<string, int> M2
    )
{
    this.setTran = _BPP;
    this.M1 = M1;
    this.M2 = M2;
}
```



Obrázek 13: Třída Polynomial a třídy, které využívá

```
public Polynomial(List<Transition> BPP1, List<Transition> BPP2, Dictionary<string, int> M1,
    Dictionary<string, int> M2)
{
    this.setTran = DisjunctUnion(BPP1, BPP2, M2);
    this.M1 = M1;
    this.M2 = M2;
}
```

Výpis 4: Přetížený konstruktor třídy Polynomial

4.5.1.1 Metoda Bisimilar Jak je uvedeno ve výpisu 5, nejprve vytvoříme množinu $setPlaces$ těch míst BPP systému, do kterých vstupní značení nepřiřazují žádný token. K určení, o která místa jde, slouží privátní metoda $NonInMark$. Následně vytvoříme seznam množin $setQ$. Do něj společně s prázdnou množinou přiřadíme za pomocí metody $CreateSubSets$ z třídy Set všechny podmnožiny množiny $setPlaces$.

Dalším krokem je cyklus zpracovávající všechny množiny Q ze seznamu množin $setQ$. Nejprve kontrolujeme, zda opravdu žádné místo aktuálně zpracovávané množiny Q neobsahuje token, a to metodou $zeroQ$ na obou značeních M1 i M2. Pokud ne, zkontrolujeme, zda by měla aktuálně zapracovávaná množina patřit do množiny pastí C. Pro tento účel voláme metodu $BelongsToC$ vycházející z pseudoalgoritmu 14. Při vrácení pozitivní hodnoty vytvoříme množinu rozkladů přechodů $\mathcal{T}$, s využitím metody $ComputePart$ (vycházející z pseudoalgoritmu 15), nad danou množinou Q.

Poté pro každý takovýto rozklad T z množiny rozkladů $\mathcal{T}$ určíme metodou Pre vstupní

místa všech přechodů z daného rozkladu. Ty sjednotíme společně se zpracovávanou množinou míst Q do nové množiny $Q1$.

Nakonec musíme ověřit, zda-li pro $Q1$ platí $NORM_{Q1}(M_1) \neq NORM_{Q1}(M_2)$. Využijeme tedy funkci $CreatePreU$ patřící do třídy BPP. Ta metodou $CountNorm$ z třídy $Norm$ spočítá normy míst určených množinou $Q1$ a poté vytvoří seznam těchto míst společně s jejich normami označený $preU$. $NORM_Q$, představující funkci $CountMarking$ z třídy $Marking$, dále spočítá hodnoty značení $M1, M2$ vzhledem k $preU$ a vyhodnotit jejich rovnost. V případě nerovnosti nejsou zadaná značení M1, M2 BPP systému bisimulačně ekvivalentní,v opačném případě jsou.

```
public bool Bisimilar()
{
    Set set = new Set();
    Dictionary<string, string> preU;
    Marking M = new Marking();
    int actualQ = 1;
    string m1 = "";
    string m2 = "";

    HashSet<string> setPlaces = NonInMark();
    List<HashSet<string>> setsQ = new List<HashSet<string>>(){new HashSet<string>()};
    setsQ.AddRange(set.CreateSubSets(setPlaces));
    foreach (var Q in setsQ) {
        if ((zeroQ(Q, M1)) && (zeroQ(Q, M2))) {
            if (BelongsToC(Q)) {
                HashSet<HashSet<int>> Tau = new HashSet<HashSet<int>>();
                Tau = ComputePart(Q);
                foreach (var T in Tau) {
                    HashSet<string> Q1 = new HashSet<string>();
                    Q1 = set.UnionSets(Q, _BPP.Pre(T, setTran));
                    preU = _BPP.CreatePreU(setTran, Q1);
                    m1 = M.CountMarking(preU, M1);
                    m2 = M.CountMarking(preU, M2);
                    if (!m1.Equals(m2)) {
                        return false;
                    }
                }
            }
        }
    }
    return true;
}
```

Výpis 5: Ukázka ověřování bisimilarity pomocí polynomiálního algoritmu

4.5.1.2 Metoda BelongsToC Funkčnost této metody vychází z pseudoalgoritmu na obrázku 14. Má za úkol rozhodnout, zda by vstupní množina míst označená R měla nebo neměla patřit do množiny množin pastí C. Proto v kroku 1 vstupní množina rovna prázdné množině vrací hodnotu pravda, jelikož ta je pastí, a tudíž náleží do množiny C.

```
1   if R = ∅ then return True;
2   for all R1, R2 ⊂ R s.t. R1 ∪ R2 = R do
3       if BelongsToC(R1) and BelongsToC(R2) then
4           return True;
5   for each R' ⊂ R s.t. BelongsToC(R') do
6       (T := ComputePart(R'));
7       for each class T of T do
8           if R = TRAP(R' ∪ PRE(T)) than
9               return True;
10  return False
```

Obrázek 14: Pseudoalgoritmus metody BelongsToC převzatý z [3]

Před provedením dalšího kroku využijeme metody $CreateSubSets$ z třídy Set, abychom získali všechny podmnožiny vstupní množiny R. Ty následně uložíme do seznamu podmnožin $subsetsR$.

Nyní jsme připraveni provést krok 2. Postupně vybíráme všechny možné dvojice podmnožin R_1,R_2 ze seznamu $subsetsR$, které jsou vlastními podmnožinami R a zároveň metodou $UnionSets$ ověříme, zda je jejich sjednocení rovno vstupnímu R. Pokud ano, krokem 3 rekurzivně zavoláme metodu $BelongsToC$ jak na R_1, tak na R_2. Vrátí-li se u obou metod hodnota pravda, tedy R_1 i R_2 náleží do množiny pastí C, vrací i aktuální metoda $BelongsToC$ pro danou množinu R hodnotu pravda (krok 4).

Následující cyklus v kroku 5 zpracovává všechny ty podmnožiny R' z $subsetsR$, které jsou jednak vlastními podmnožinami vstupní množiny R a zároveň patří mezi množiny pastí. Tedy metodou $BelongsToC$ rekurzivně ověříme, zda vybraná množina R' náleží do C. Při splnění obou podmínek vytvoříme krokem 6 novou množinu rozkladů přechodů $\mathcal{T}$ pomocí metody $ComputePart$ z pseudoalgoritmu 15. Pro každý takový rozklad T z množiny rozkladů $\mathcal{T}$ v kroku 7 se krokem 8 ověří, zda je vstupní R rovno pasti. Tedy že se množina R rovná množině vzniklé sjednocením podmnožiny R' a množiny vstupních míst aktuálně vybraného T. Pro zjištění vstupních míst k rozkladu přechodů T využíváme metodu Pre z třídy BPP. K sjednocení množin pak metodu $UnionSets$ z třídy Set. Pakliže je R pastí, celá funkce vrací krokem 10 hodnotu pravda, tedy zpracovávaná množina patří do C. Nedošlo-li v celém průběhu algoritmu k vrácení hodnoty pravda, vrací se krokem 11 hodnota nepravda, tudíž množina R mezi množiny pastí C nepatří.

4.5.1.3 Metoda ComputePart Tato metoda provádí rozklad přechodů do množin. Její funkčnost vychází z pseudoalgoritmu uvedeném na obrázku 15. Jako vstup má množinu míst označenou R. V kroku 1 pomocí metody $BelongsToC$ zkontrolujeme, jestli tato množina náleží mezi pasti. V případě, že nenáleží, vracíme selhání algoritmu, jinak pro-

1 **if** $BelongsToC(R1) = False$ **then** fail **else** $\mathcal{T} := \mathcal{T}_\lambda$
2 **for** each $R' \subset R$ s.t. $BelongsToC(R')$ **do** $\mathcal{T} := \mathcal{T} \sqcap ComputePart(R')$;
3 $ProcessedClasses := \emptyset$
4 **while** $\mathcal{T} - ProcessedClasses \neq \emptyset$ **do**
5 select T from $\mathcal{T} - ProcessedClasses$
6 $Q := R \cup \text{PRE}(T)$
7 $R' := \text{TRAP}(Q)$
8 **if** $R' \subseteq R$ **then**
9 $\mathcal{T} := \mathcal{T} \sqcap \mathcal{T}_Q$
10 $ProcessedClasses := ProcessedClasses \cup \{T\}$
11 **return** $\mathcal{T}$

Obrázek 15: Pseudoalgoritmus metody ComputePart převzatý z [3]

vedeme nový rozklad přechodů podle jejich ohodnocení $\mathcal{T}_\lambda$ a ten uložíme do množiny $\mathcal{T}$.

Před krokem 2 je zapotřebí vytvořit, opět pomocí funkce $CreateSubSets$ ze třídy Set, množinu všech podmnožin $subsetsR$ daného R. Krokem 2 následně pro každou množinu R' náležící do $subsetsR$ ověříme, zda je vlastní podmnožinou R a metodou $BelongsToC$, zda patří do množiny pastí C. Pokud ano, vytvoříme nový rozklad přechodů $\mathcal{T}$ tak, že pronikneme funkcí $IntersectSetSets$ aktuální množinu rozkladů $\mathcal{T}$ s množinou rozkladů vrácenou z rekurzivního zavolání metody $ComputePart$ nad danou podmnožinou R'. V kroku 3 nastavíme zpracované třídy $ProcessedClasses$ na prázdnou množinu.

Kroky 4 – 10 představují cyklus, zpracovávající všechny rozklady z $\mathcal{T}$ a postupně je přiřazující do zpracovaných tříd $ProcessedClasses$. Jestliže v kroku 4 nevrátí funkce $ExceptSetSets$ z třídy Set, provádějící rozdíl množin $\mathcal{T}$ a $ProcessedClasses$, prázdnou množinu, vybereme krokem 6 libovolnou, doposud nezpracovanou množinu rozkladu přechodů z $\mathcal{T}$. V kroku 7 získáme pomocí metody Pre třídy BPP vstupní místa přechodů v této množině, ty sjednotíme metodou $UnionSets$ s místy množiny R a uložíme do množiny Q. Dalším mezikrokem zavoláme metodu $CountNorm$, pro spočítání norem jednotlivých míst BPP systému. Použitím metody $Trap$ v kroku 7 pak určíme, která místa jsou v množině Q pastmi a ty přidáme do množiny R'.

Na závěr krokem 8 zjistíme, zda je takto vzniklá množina R' podmnožinou R. Pokud ano, vytvoříme krokem 9 nový rozklad přechodů $\mathcal{T}$ pomocí průniku množin $\mathcal{T}$ a $\mathcal{T}_Q$. Množinu $\mathcal{T}_Q$ vytváří funkce $CreateTauQ$ z třídy BPP. Průnik pak má na starosti metoda $IntersectSetSets$ v třídě Set. Zpracovanou množinu T přidáme krokem 10 do zpracovaných tříd $ProcessedClasses$. Pokud neexistuje žádné další T k zpracování, vrátí se krokem 11 aktuální rozklad přechodů $\mathcal{T}$. Tím metoda končí.

5 Vstupní systémy

V této kapitole se budeme zabývat zadáváním vstupních BPP systémů. Pro tento účel existují celkem tři možnosti, tedy:

- vlastní zadání,
- načtení ze souboru,
- vygenerování náhodného vstupu.

5.1 Vlastní zadávání

Zahrnuje vytvoření systému pomocí samotného přidávání jednotlivých přechodů, ale také uživateli umožňuje rozšířit již existující vstup (tj. systém vygenerovaný nebo otevřený ze souboru) o další přechody. Na obrázku 16 je zobrazeno, jak vypadá rozhraní pro zadávání přechodů. Zde zadáváme postupně vstupní místo, ohodnocení přechodu (tj. akce vedoucí ze vstupního místa) a množinu výstupních míst, kterou tvoří jednotlivá výstupní místa s přiřazenou násobností hrany. K reprezentaci takovýchto dat jsem zvolil následující omezující podmínky:

- Vstupní místo musí být velké písmeno abecedy, popřípadě k němu lze přiřadit libovolně dlouhý řetězec čísel.
- Každá akce (ohodnocení přechodu) je omezena na jedno malé písmeno abecedy v rozmezí od a po z.
- Výstupní místo omezují stejné podmínky jako místo vstupní. Pouze pro případ, kdy bude přechod vyprazdňující, volíme jeho výstupním místem jediné písmeno Řecké abecedy a to písmeno $epsilon(\varepsilon)$.
- Násobnost hrany u výstupních míst pak reprezentuje číslo od 1 do námi zvolené libovolné hodnoty.

5.1.1 Třídy BPP a Checking

Vlastní zadávání pracuje se třídou BPP, kde využívá metody $AddPlace$, $DeletePOST$ a $AddTransition$. K ověřování korektnosti jednotlivých vstupů asociuje třída BPP třídu $Checking$ a její metody $PlaceCorrect$, $ActionCorrect$, $PlaceExist$ a $TransitionExist$. Toto schéma je uvedeno na obrázku 17.

Pří vytváření přechodu musíme přidat alespoň jedno výstupní místo do množiny výstupních míst. K tomuto účelu voláme metodu $AddPlace$. Ta pracuje s metodami $PlaceCorrect$ a $PlaceExist$, které ověří, zda-li je přidávané místo korektní a zatím nebylo v množině použito. Při splnění těchto podmínek se dané místo přidá do množiny výstupních míst.

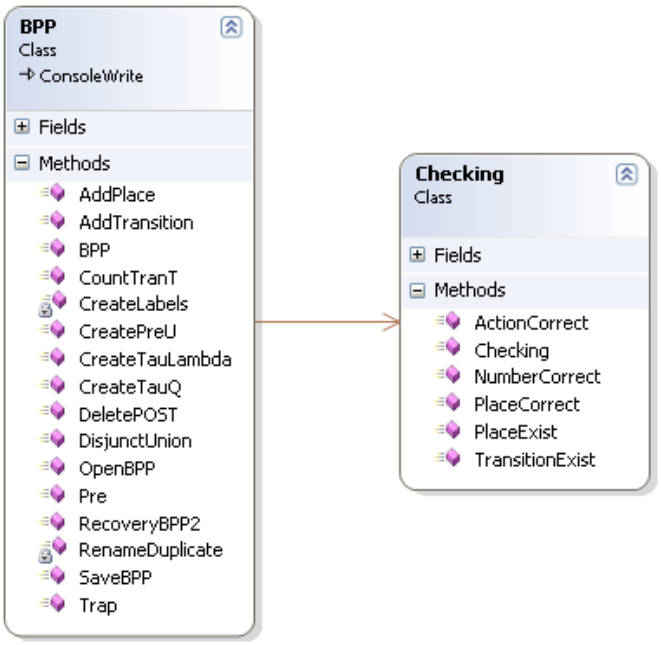
Obrázek 16: Přidání přechodu do BPP systému

V případě omylu a nutnosti zadání jiných výstupních míst existuje metoda $DeletePOST$, která tuto množinu vymaže.

Dalším krokem přiřadíme, pomocí metody $AddTransition$ uvedené ve výpisu kódu 6, nový přechod do BPP systému. Tato metoda zkontroluje korektnost vstupního místa metodou $PlaceCorrect$ a korektnost akce vedoucí ze vstupního místa metodou $ActionCorrect$. Poté ověří, zda takovýto přidávaný přechod již neexistuje v BPP systému. Nakonec vytvoří nový přechod a zobrazí jej uživateli v příslušné části SECT. Pokud vytvořená množina z předcházejícího kroku bude vyprazdňující, výstupní místo bude pouze $epsilon$.

```
public List<Transition> AddTransition(string PRE, string action, string POST, ListBox listbox,
      List<Transition> _BPP)
{
   if (check.PlaceCorrect(PRE))
   {
      if (check.ActionCorrect(action))
      {
         if (check.TransitionExist(PRE, action, _BPP))
         {
            if (POST == epsilon)
            {
               setPOST.Add(new Place(epsilon, "", "0"));
            }
            BPP.Add(new Transition(_BPP.Count + 1, new Place(PRE, infinite), action, setPOST));
            listbox.Items.Add(PRE + " -- > " + action + " -- > " + POST);
            setPOST = new List<Place>();
            return _BPP;
         }
      }
   }
}
```

Výpis 6: Ukázka přidávání přechodu do BPP systému



Obrázek 17: Třídy – BPP a Checking

5.2 Načtení ze souboru

Načtení BPP systému se provádí pomocí menu programu (např. BPP→BPP1→Otevřít). Na příkladu 5.1 je ukázáno, jaký má být formát vstupního souboru a jak se poté zobrazí výstup tohoto souboru uživateli v SECT.

Příklad 5.1
Formát vstupního souboru:
$A\ a\ B\ 2/C11$
$A\ b\ epsilon$
$B\ a\ 4/A\ B\ 3/C11$
$C11\ a\ 2/B$

Z uvedeného výpisu je patrné, že musí platit následující:

- Každý přechod BPP systému musí být v souboru uložen na novém řádku.
- Vstupní místo je velké písmeno abecedy (možno s přidaným řetězcem čísel), které uvádíme jako první znak ukončený mezerou.
- Následuje akce (písmeno a – z), také ukončena mezerou.
- Jestliže dále řádek obsahuje řetězec písmen $epsilon$, je přechod vyprazdňující.
- Pokud chceme místo $epsilon$ zadat množinu výstupních míst, píšeme jednotlivá výstupní místa rozdělená mezerou v jednom z těchto formátu:

- Pouze velké písmeno abecedy (možno s přidaným řetězcem čísel), toto výstupní místo bude mít násobnost hrany rovnu 1.
- Násobnost hrany výstupního místa (2 – libovolně zvolená hodnota) následována lomítkem / a velkým písmenem abecedy (možno s přidaným řetězcem čísel).

Při načítání BPP systému ze souboru dochází ke kontrole předešle uvedených dat. Pokud systém nalezne nekorektně zadanou hodnotu, zobrazí uživateli informaci o chybě společně s přibližným umístěním dané chyby v procházeném souboru. Není tedy problém daný omyl následně opravit.

Zobrazený vstupní systém v SECT pak vypadá takto:
$A \rightarrow a \rightarrow B\ 2C11$
$A \rightarrow b \rightarrow \varepsilon$
$B \rightarrow a \rightarrow 4A\ B\ 3C11$
$C11 \rightarrow a \rightarrow 2B$ ■

5.2.1 Třída BPP a Checking

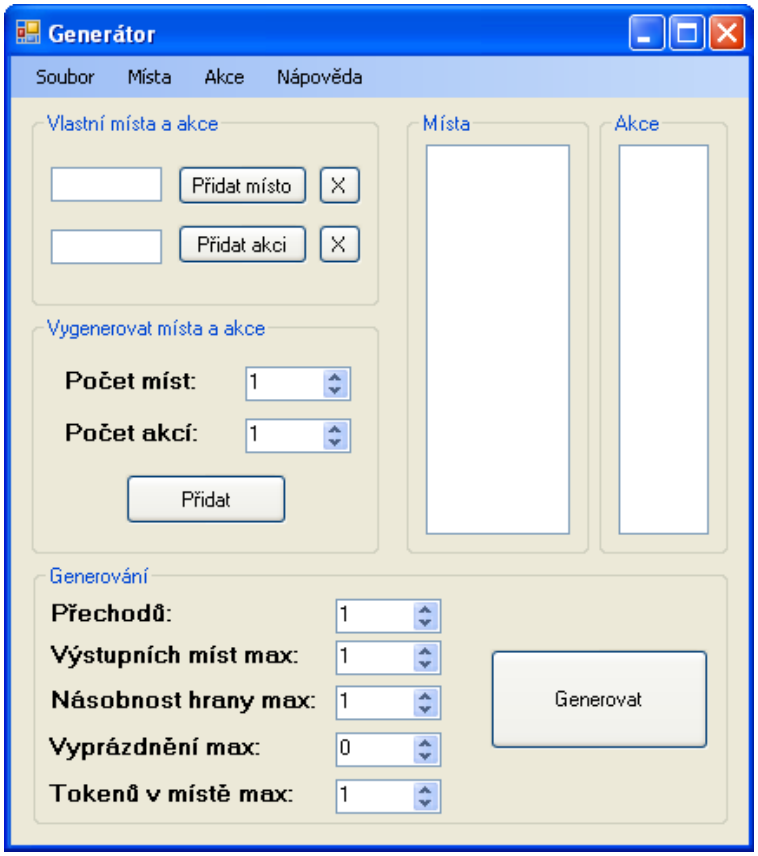
Načtení vstupu ze souboru využívá stejné třídy jako předchozí kapitola Vlastní zadání. Třída BPP volá metodu $OpenBPP$. K ověřování korektnosti jednotlivých vstupů asociuje třída BPP třídu $Checking$ a její metody $PlaceCorrect$, $ActionCorrect$, $NumberCorrect$, $PlaceExist$ a $TransitionExist$. Toto schéma je uvedeno na obrázku 17.

Samotné načítání tedy provádí metoda $OpenBPP$, kde procházíme postupně celý zdrojový soubor. Zpracováním každého řádku získáme jeden přechod. Korektnost vstupního místa ověříme metodou $PlaceCorrect$, korektnost akce metodou $ActionCorrect$. Pokud vede výstupní místo na $epsilon$, přechod je kompletní. Přiřadíme jej do BPP systému a zobrazíme uživateli v SECT.

V opačném případě se provádí část zpracování výstupních míst a kontrola, o jaké místa jde. Jestliže místo neobsahuje násobné hrany, ověříme metodou $PlaceCorrect$ jeho korektnost a přidáme jej do množiny výstupních míst. U výstupního místa s násobnou hranou musíme, metodou $NumberCorrect$, ověřit korektnost počtu násobných hran a správný formát zápisu. Teprve poté jej přidáme do množiny výstupních míst. V případě, že jsou všechna místa zpracována, množinu přiřadíme do přechodu. Ten opět do BPP systému a ten zobrazíme uživateli v SECT.

5.3 Generátor náhodných vstupů

Generátor se skládá z několika částí. Jak lze vidět na obrázku 18, zahrnuje seznam Místa a seznam Akce. Zde si uživatel navolí ty místa a akce, ze kterých se bude náhodný BPP systém generovat. Přitom může využít přidání vlastních míst a akcí, nechat si je



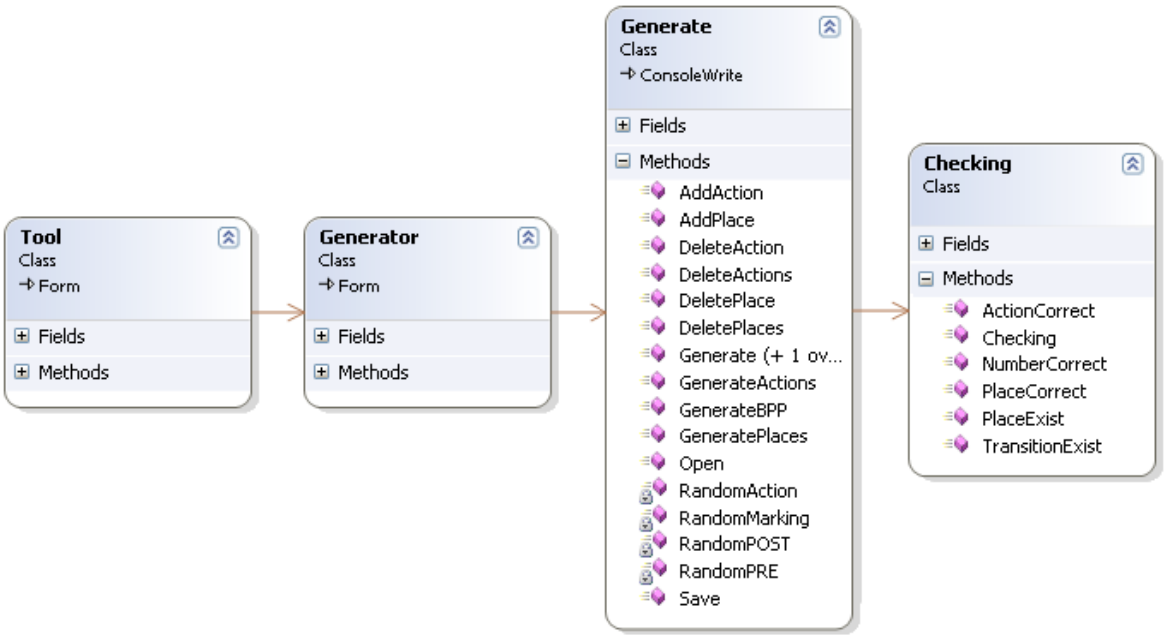
Obrázek 18: Generátor náhodných vstupů

vygenerovat, nebo pomocí menu generátoru načíst vstupy ze souboru. Opět zde musí být splněny následující podmínky:

- Místo musí být velké písmeno abecedy, s možností doplnění o libovolně dlouhý řetězec čísel.
- Akce musí být jedno malé písmeno abecedy v rozmezí od a po z.

Poslední část obsahuje samotné generování BPP, kde je potřeba zadat:

- Počet přechodů v BPP systému – omezený od 1 do všech možných kombinací míst s akcemi ze seznamů pro generování.
- MAX počet výstupních míst v rámci přechodu – omezený od 1 do počtu míst určených ke generování.
- MAX násobnost hrany v rámci výstupního místa – omezená od 1 do libovolného uživatelsky definovaného čísla.



Obrázek 19: Třídy – Tool, Generator a Generate

- MAX počet vyprazdňujících (*epsilon*) přechodů – omezený od 0 do zvoleného počtu přechodů.
- MAX počet tokenů vkládaných značením (*Marking*) do místa BPP systému – omezený od 1 do libovolného uživatelsky definovaného čísla.

5.3.1 Třídy Tool, Generator, Generate a Checking

Jak je uvedeno na obrázku 19, pokud chceme BPP systém vygenerovat, třída *Tool* využije třídu *Generator* a spustí podsystém pro generování náhodných vstupů. K tomuto účelu využívá třída *Generator* třídu *Generate* společně se všemi jejími metodami. Tato třída dále asociuje třídu *Checking*, díky které ověřuje korektnost zadávaných vstupních údajů.

5.3.2 Vlastní místa

Pokud chceme přidat uživatelsky definovaná vstupní místa, generátor využije metodu *AddPlace*. Ta vloží vlastní místo do seznamu míst, ze kterých bude následný BPP systém vygenerován. Předtím ovšem musíme zkontrolovat, zda seznam míst již takovéto místo neobsahuje a pomocí metody *PlaceCorrect* ověřit jeho korektnost. Lze také, metodou *DeletePlace*, kterékoliv místo odebrat, popřípadě metodou *DeletePlaces* odebrat všechna doposud vložená místa.

5.3.3 Vlastní akce

Podobně v případě uživatelsky definovaných vstupních akcí, generátor využije metodu $AddAction$. Ta, po ověření duplicity vzhledem k množině akcí a ověření korektnosti akce metodou $ActionCorrect$, vloží vlastní akci do seznamu akcí, ze kterých bude následný BPP systém vygenerován. Stejně můžeme metodou $DeleteAction$ kteroukoliv akci odebrat nebo metodou $DeleteActions$ odebrat všechny doposud vložené akce.

5.3.4 Náhodná místa a akce

Zde může uživatel zadat počet míst a počet akcí, které budou vygenerovány do seznamů míst a akcí. K tomuto účelu pak slouží metoda $GeneratePlaces$, pro generování zvoleného počtu míst a metoda $GenerateActions$, pro generování zvoleného počtu akcí. Pokud je počet míst menší nebo roven počtu písmen abecedy, přidají se pouze písmena. V případě většího počtu se k písmenu abecedy automaticky připojí navíc řetězec čísel. U tohoto postupu není potřeba nikterak kontrolovat korektnost, ta vyplývá z automatického generování vstupních dat podle zadaného počtu.

5.3.5 Načítání ze souboru

Provádí metodou $Open$, která má jako parametr typ otevíraného seznamu (tj. místa nebo akce). Při načítání míst zkontroluje metodou $PlaceCorrect$ jeho korektnost a přiřadí jej do množiny míst. U akcí zkontroluje metodou $ActionCorrect$ korektnost akce a tu uloží do seznamu akcí. Tento postup končí po přečtení celého načítaného souboru.

5.3.6 Uložení do souboru

Pokud využijeme kterýkoliv z předešlých postupů, přidáme nebo naopak ubereme současná místa (akce), můžeme uložit seznam míst i seznam akcí do souboru k dalšímu použití. Pro tento účel slouží metoda $Save$, u které opět parametrem předáváme, jaký typ seznamu chceme uložit (tj. místa nebo akce).

5.3.7 Generování

Provádíme metodou $GenerateBPP$, uvedená ve výpisu 7. Metodou $RandomPRE$ generujeme ke každému přechodu náhodné vstupní místo, metodou $RandomAction$ pak náhodnou akci. V případě, že metodou $TransitionExist$ zjistíme existenci přechodu se stejným místem a stejnou akcí, vybíráme tyto veličiny znovu. Následně metodou $RandomPOST$ vytvoříme množinu náhodných výstupních míst a tím máme přechod hotov. Nesmíme zapomenout přidat každé vstupní místo do množiny míst $placesToMark$. Jakmile totiž máme vytvořeny všechny přechody a tím i celý BPP systém, využijeme tuto množinu k vygenerování vstupního značení pomocí metody $RandomMarking$. Takovýto vygenerovaný vstupní systém a vstupní značení zobrazíme uživateli v SECT.

Obě metody $RandomPRE$ a $RandomPOST$ vybírají náhodné místa z množiny míst

pro generování (seznam Místa). Stejně tak metoda $RandomAction$ vybírá náhodné akce z množiny akcí pro generování (seznam Akce).

```
public List<Transition> GenerateBPP(ListBox listbox, RichTextBox richtextbox, int numberTran, int
    numberPostMax, int numberNumMax, int numberEpsMax, int numberTokenMax) {
  List<Transition> transitions = new List<Transition>();
  Place PRE = new Place();
  Checking check = new Checking();
  string action = "";
  string rule = "";

  for (int i = 1; i <= numberTran; i++) {
    if (transitions.Count != 0) {
      do {
        PRE = RandomPRE();
        action = RandomAction();
      } while (!check.TransitionExist(PRE._Place, action, transitions));
    }
    else {
      PRE = RandomPRE();
      action = RandomAction();
    }
    List<Place> setPOST = RandomPOST(numberTran, numberPostMax,numberNumMax,
      numberEpsMax);
    rule = PRE._Place + "␣␣-->␣" + action + "␣-->␣";
    foreach (var POST in setPOST) {
      if ((POST.Number == 1) || (POST._Place == epsilon)) {
        rule += "␣" + POST._Place;
      }
      else {
        rule += "␣" + POST.Number + POST._Place;
      }
    }
    listbox.Items.Add(rule);
    transitions.Add(new Transition(i, PRE, action, setPOST));
    if (!placesToMark.Contains(PRE._Place)) {
      placesToMark.Add(PRE._Place);
    }
  }
  richtextbox.Text = RandomMarking(numberTokenMax);
  return transitions;
}
```

Výpis 7: Ukázka generování náhodného vstupu

5.4 Ukládání do souboru

Existence této volby v menu SECT nám umožňuje vygenerovaný, vytvořený nebo poupravený BPP systém uložit do souboru. Podobně také existuje volba pro uložení vygenerovaného značení. Díky tomu lze se vstupními daty efektivně pracovat a využívat je k různým kombinacím ověřování bisimulační ekvivalence.

6 Experimenty časové a prostorové složitosti

V této kapitole nás bude především zajímat zjišťování množství využité paměti a celkový čas výpočtu při ověřování bisimulační ekvivalence na zadaných vstupech pomocí Exponenciálního a pomocí Polynomiálního algoritmu. Tyto vstupy budou mít různě velké vstupní parametry (počet přechodů, počet akcí, atd.) a budou prováděny na dvou rozdílných zařízeních. Získané výsledky následně porovnáme a vyhodnotíme.

6.1 Testovací zařízení

1.
   - Stolní počítač s operačním systémem Microsoft Windows XP Professional x86, Service Pack 3.
   - Procesor Intel Pentium 4E CPU 2800 MHz (3.5 x 800), 1 MB L2 Cache.
   - Paměť 2048 MB RAM (1 x 1024 MB DDR SDRAM, 2 x 512 MB DDR SDRAM).
2.
   - Notebook s operačním systémem Microsoft Windows XP Professional x86, Service Pack 3.
   - Procesor Intel Celeron M CPU 1400 MHz (3.5 x 400), 512 KB L2 Cache.
   - Paměť 752 MB RAM (1 x 512 MB DDR SDRAM, 1 x 256 MB DDR SDRAM).

6.2 Testovací případy

6.2.1 První testovací případ

První vstupní systém BPP1:

- Celkový počet přechodů 1.
- Celkový počet akcí 1.
- Celkový počet vyprazdňujících přechodů 0.

Druhý vstupní systém BPP2:

- Celkový počet přechodů 10/10 (ekvivalentní/neekvivalentní).
- Celkový počet akcí 1.
- Celkový počet vyprazdňujících přechodů 0/1.

V tabulce 1 jsou uvedeny výsledky experimentů prvního testovacího případu, kde platí M1 ~ M2. V tabulce 2 pak výsledky, kde bisimilarita na značeních M1, M2 neplatí.

Bisimulačně ekvivalentní vstupy						
První testovací zařízení	Exponenciální algoritmus					Průměr
Čas výpočtu (s)	0,000697	0,000699	0,00069	0,000695	0,000697	0,000696
Využité paměti (kB)	369	369	368	370	369	369
První testovací zařízení	Polynomiální algoritmus					Průměr
Čas výpočtu (s)	0,00048	0,00047	0,00046	0,00048	0,00045	0,00047
Využité paměti (kB)	360	360	359	360	361	360
Druhé testovací zařízení	Exponenciální algoritmus					Průměr
Čas výpočtu (s)	0,00086	0,00095	0,00089	0,00099	0,00087	0,00091
Využité paměti (kB)	369	381	371	371	369	372
Druhé testovací zařízení	Polynomiální algoritmus					Průměr
Čas výpočtu (s)	0,00063	0,00062	0,00264	0,00403	0,00399	0,00238
Využité paměti (kB)	360	361	360	368	360	362

Tabulka 1: Testovací případ 1 – bisimulačně ekvivalentní vstupy

Bisimulačně neekvivalentní vstupy						
První testovací zařízení	Exponenciální algoritmus					Průměr
Čas výpočtu (s)	0,00513	0,00514	0,00510	0,00518	0,00514	0,00514
Využité paměti (kB)	514	513	513	513	513	513
První testovací zařízení	Polynomiální algoritmus					Průměr
Čas výpočtu (s)	0,00172	0,00163	0,00162	0,00164	0,00344	0,00201
Využité paměti (kB)	401	400	400	400	400	400
Druhé testovací zařízení	Exponenciální algoritmus					Průměr
Čas výpočtu (s)	0,00644	0,00642	0,00645	0,00647	0,00641	0,00644
Využité paměti (kB)	514	514	513	514	514	514
Druhé testovací zařízení	Polynomiální algoritmus					Průměr
Čas výpočtu (s)	0,00208	0,00199	0,00482	0,00545	0,00534	0,00393
Využité paměti (kB)	415	401	401	401	402	404

Tabulka 2: Testovací případ 1 – bisimulačně neekvivalentní vstupy

Bisimulačně ekvivalentní vstupy						
První testovací zařízení	Exponenciální algoritmus					Průměr
Čas výpočtu (s)	31,24	35,20	35,99	38,28	40,61	36,26
Využité paměti (kB)	1643	1652	1540	1941	1949	1745
První testovací zařízení	Polynomiální algoritmus					Průměr
Čas výpočtu (s)	12,88	13,64	13,48	13,78	13,93	13,54
Využité paměti (kB)	2143	2144	2136	2136	2136	2139
Druhé testovací zařízení	Exponenciální algoritmus					Průměr
Čas výpočtu (s)	31,65	31,76	31,84	31,90	31,83	31,80
Využité paměti (kB)	1549	1549	1557	1549	1549	1550
Druhé testovací zařízení	Polynomiální algoritmus					Průměr
Čas výpočtu (s)	10,27	9,97	10,27	10,26	10,25	10,21
Využité paměti (kB)	1934	1677	1934	1560	1588	1739

Tabulka 3: Testovací případ 2 – bisimulačně ekvivalentní vstupy

6.2.2 Druhý testovací případ

První vstupní systém BPP1:

- Celkový počet přechodů 500.
- Celkový počet akcí 2.
- Celkový počet vyprazdňujících přechodů 16.

Druhý vstupní systém BPP2:

- Celkový počet přechodů 490/450 (ekvivalentní/neekvivalentní).
- Celkový počet akcí 2.
- Celkový počet vyprazdňujících přechodů 12/11.

V tabulce 3 jsou uvedeny výsledky experimentů prvního testovacího případu, kde platí M1 ~ M2. V tabulce 4 pak výsledky, kde bisimilarita na značeních M1, M2 neplatí.

6.2.3 Třetí testovací případ

První vstupní systém BPP1:

- Celkový počet přechodů 200.
- Celkový počet akcí 5.
- Celkový počet vyprazdňujících přechodů 14.

Bisimulačně neekvivalentní vstupy						
První testovací zařízení	Exponenciální algoritmus					Průměr
Čas výpočtu (s)	10,92	11,68	12,73	13,54	13,99	12,57
Využité paměti (kB)	1747	1785	1676	1668	1668	1709
První testovací zařízení	Polynomiální algoritmus					Průměr
Čas výpočtu (s)	5,43	6,05	6,10	6,32	5,17	29,07
Využité paměti (kB)	1147	1147	1147	1147	1147	1147
Druhé testovací zařízení	Exponenciální algoritmus					Průměr
Čas výpočtu (s)	13,43	13,40	13,34	13,39	13,39	13,39
Využité paměti (kB)	1409	1409	1317	1378	1317	1366
Druhé testovací zařízení	Polynomiální algoritmus					Průměr
Čas výpočtu (s)	7,19	7,18	7,20	7,19	7,18	7,19
Využité paměti (kB)	1261	1261	1261	1261	1261	1261

Tabulka 4: Testovací případ 2 – bisimulačně neekvivalentní vstupy

Bisimulačně ekvivalentní vstupy						
První testovací zařízení	Exponenciální algoritmus					Průměr
Čas výpočtu (s)	208,22	221,71	215,91	215,12	215,78	215,35
Využité paměti (kB)	2075	1855	2205	2125	2308	2114
První testovací zařízení	Polynomiální algoritmus					Průměr
Čas výpočtu (s)	2,432	2,549	2,555	2,791	2,791	2,627
Využité paměti (kB)	1149	637	1147	1142	1149	1045
Druhé testovací zařízení	Exponenciální algoritmus					Průměr
Čas výpočtu (s)	125,38	125,52	125,43	125,38	125,45	125,43
Využité paměti (kB)	2399	2356	2386	2422	2322	2377
Druhé testovací zařízení	Polynomiální algoritmus					Průměr
Čas výpočtu (s)	2,19	2,18	2,20	2,19	2,19	2,19
Využité paměti (kB)	712	862	638	831	892	787

Tabulka 5: Testovací případ 3 – bisimulačně ekvivalentní vstupy

Bisimulačně neekvivalentní vstupy						
První testovací zařízení	Exponenciální algoritmus					Průměr
Čas výpočtu (s)	108,13	115,90	116,99	114,55	116,46	114,41
Využité paměti (kB)	3176	2925	3040	3008	3092	3048
První testovací zařízení	Polynomiální algoritmus					Průměr
Čas výpočtu (s)	0,389	0,390	0,391	0,390	0,389	0,390
Využité paměti (kB)	920	920	920	924	920	921
Druhé testovací zařízení	Exponenciální algoritmus					Průměr
Čas výpočtu (s)	70,47	70,40	70,46	70,47	70,42	70,44
Využité paměti (kB)	2840	3245	3177	2933	3248	3089
Druhé testovací zařízení	Polynomiální algoritmus					Průměr
Čas výpočtu (s)	0,535	0,536	0,538	0,538	0,541	0,538
Využité paměti (kB)	657	761	657	769	657	700

Tabulka 6: Testovací případ 3 – bisimulačně neekvivalentní vstupy

Druhý vstupní systém BPP2:

- Celkový počet přechodů 200.
- Celkový počet akcí 5.
- Celkový počet vyprazdňujících přechodů 14.

V tabulce 5 jsou uvedeny výsledky experimentů prvního testovacího případu, kde platí M1 ~ M2. V tabulce 6 pak výsledky, kde bisimilarita na značeních M1, M2 neplatí.

6.2.4 Vyhodnocení

Z výsledků testů je očividné, že pro malé vstupy je složitost výpočtu pro oba algoritmy přibližně totožná (testovací případ 1). Naopak u větších vstupů čas i paměť logicky narůstá. U polynomiálního algoritmu je patrno, že pro větší počet přechodů s menším počtem akcí (testovací případ 3) využití paměti narůstá, ale doba výpočtu je nižší než u algoritmu exponenciálního. U něj naopak narůstá čas výpočtu spolu s využitou pamětí v případě většího počtu akcí a to i při nižším počtu přechodů (testovací případ 5). Z toho vyplývá, že s přibývajícím počtem akcí, nezávisle na počtu přechodů, roste složitost exponenciálního algoritmu. U polynomiálního algoritmu pak roste úměrně k zvětšujícímu se počtu přechodů i akcí, ale zároveň je nutno poznamenat, že čas průběhu výpočtu výrazně záleží na počtu míst, která mají přiřazený token.

7 Závěr

Všechny úkoly a cíle této diplomové práce se podařilo úspěšně zpracovat a splnit dle požadovaného zadání.

V první části bylo úkolem nastudovat veškeré teoretické pojmy společně s popisem a průběhem požadovaných algoritmů. Zde jsem pochopil, co znamená pojem ověřování bisimulační ekvivalence na zadaných vstupních BPP systémech, jak jsou tyto systémy reprezentovány a které veličiny využívají.

Posléze se bylo nutno zamyslet nad možností reprezentování daných pojmů v prostředí vybraného programovacího jazyka. Zde jsem volil takový model, který by byl modulární v případě potřeby dalšího rozšiřování programu. Jakmile jsem měl o těchto věcech představu, následoval hlavní cíl celého projektu, a to implementace základních algoritmů ověřujících bisimilaritu na zadaných vstupech.

Dalším neméně důležitým cílem bylo vytvořit způsoby zadávání těchto vstupních systému, podle potřeb a parametrů zvolených uživatelem. Jednou z možností je generátor náhodných vstupů podle předem určených parametrů, druhou pak načtení vstupu z vhodně definovaného souboru. Navíc jsem zvolil univerzální možnost přidávaní jednotlivých přechodů. Takto lze jednoduše nadefinovat vstupní BPP systém (převážně menšího rozsahu) nebo rozšířit stávající vstupy o další uživatelsky definované přechody.

Posledním splněným cílem je otestování různě velkých vstupů z hlediska porovnání využité paměti a času výpočtu při průběhu algoritmů ověřujících bisimulační ekvivalenci na těchto vstupech.

Tím, jak byl nástroj navržen a naimplementován nebude problém, z hlediska jeho dalšího vývoje, doplnit algoritmy rozhodující jiné ekvivalence, popřípadě pracující s jinými modely. V podstatě je možno navrhovat nové třídy algoritmů a objektů nebo využívat a rozšiřovat ty stávající. Problém by nemělo představovat ani vytvoření možností pro generování jiných náhodných vstupů, jelikož je generátor vytvořen nezávisle jako podsystém, kde takovýchto podsystému můžeme mít libovolný počet. Jediný problém tedy představuje rozmístění jednotlivých komponent a funkce samotného nástroje, které by v případě rozšíření o jiné vstupy vyžadovaly dodatečnou modifikaci a přestavbu.

Bc. Jakub Juřica

8 Reference

[1] JANČAR, Petr. *Strong Bisimilarity on Basic Parallel Processes is PSPACE-complete*. In Proceedings 18th LiCS. IEEE Computer Society, 2003, s. 218–227.

[2] KOT, Martin. *Selected Problems from the Area of Formal Verification*. Dizertační práce. Ostrava: Vysoká škola báňská–Technická univerzita Ostrava. Fakulta elektrotechniky a informatiky, 2009. Vedoucí dizertační práce prof. RNDr. Petr Jančar, CSc.

[3] JANČAR, Petr; KOT, Martin; SAWA, Zdeněk. *Bisimilarity on Basic Parallel Processes*. Připravovaný článek. Ostrava: Vysoká škola báňská–Technická univerzita Ostrava.

[4] Wikipedia. *Diagram tříd*. [online]. San Francisco (CA): 2001 [cit. 2013-03-04]. Dostupné z: <http://cs.wikipedia.org/wiki/Diagram_tříd>.

[5] Microsoft Developer Network. *HashSet<T> Class*. [online]. [cit. 2013-03-18]. Dostupné z: <http://http://msdn.microsoft.com/en-us/library/bb359438.aspx>.

[6] Projekty SIPVZ; KOVÁŘ, Dušan. *Přetěžování metod a operací*. [online]. Olomouc: 2006, [cit. 2013-03-15]. Dostupné z: <http://projektysipvz.gytool.cz/ProjektySIPVZ/Default.aspx?uid=241>.

A Příloha na CD

Adresářová struktura přiloženého CD (Obsah CD)

- Program
  - Diplomova_prace_jur506 – Složka obsahující program (zdrojové kódy) diplomové práce.
  - Documentation.chm – Soubor obsahující dokumentaci k programu.
  - Manual.pdf – Soubor obsahující uživatelskou příručku k programu.
- Text
  - Diplomova_prace_jur506.pdf – Soubor obsahující text diplomové práce.