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

Bakalářská práce

Generování grafů předávání informace

Plzeň 2015 Jakub Morávka

Zadání

Prohlášení

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

V Plzni dne 6. května 2015

Jakub Morávka

Abstract

Sideways information passing generation

This bachelor thesis deals with sideways information passing, which is a crucial part of the Magic Sets method. This method's purpose is optimization of a logical program query's evaluation. As there is lack of open implementations of the Magic Sets method, this work's output is a Java application, which implements its part – sideways information passing algorithm. The second output is a text format, in which implemented application saves its results into text files.

Key words: sideways information passing, adornment string, logical programming, optimization

Abstrakt

Generování grafů předávání informace

Tato bakalářská práce se zabývá konstrukcí grafu předávání informace, stěžejní součásti metody magických množin, která slouží k optimalizaci vyhodnocení dotazu logickému programu. Práce se tímto tématem zabývá, neboť otevřená implementace metody magických množin je nedostupná. Výstupem je aplikace v jazyce Java, která implementuje algoritmus konstrukce grafu předávání informace, a návrh formátu, ve kterém jsou vypočtené grafy předávání informace ukládány do textových souborů.

Klíčová slova: graf předávání informace, řetězec ozdobení, logické programování, optimalizace

Obsah

Seznam obrázků a tabulek

1 Úvod

Vyhodnocování cílového dotazu ve složitějších logických programech, obsahujících rekurzi, může trvat nepřijatelně dlouho. Metoda magických množin je jeden ze způsobů, jak vyhodnocení dotazu optimalizovat. Skládá se z několika algoritmů, jimiž transformujeme logický program na ekvivalentní, tzv. magický program, pro který vyhodnocení stejného dotazu trvá výrazně kratší dobu.

Dostupných programů, implementujících metodu magických množin, je velice omezený počet, navíc nemají otevřený zdrojový kód. Tato práce si proto klade za cíl prozkoumat a implementovat v jazyce Java jednu její část – konstrukci grafu předávání informace. Dalším cílem je navržení formátu, v jakém budou vytvořené grafy předávání informace uloženy v textových souborech.

Aplikaci bude možno spustit ve dvou módech – dávkovém a s grafickým uživatelským rozhraním. V dávkovém módu, spouštěném s konzole bez další interakce s uživatelem, vytvoří aplikace pro každé pravidlo logického programu (se syntaxí podobnou jazyku Datalog) příslušné grafy předávání informace a uloží je do textových souborů v navrženém formátu. V módu s grafickým rozhraním bude tvorba konkrétního grafu řízena uživatelem.

2 Logický jazyk

V této kapitole bude čtenář nejprve seznámen se základními pojmy a definicemi, týkajícími se logického jazyka obecně a poté s úpravami syntaxe konkrétního logického jazyka pro účely této práce.

2.1 Terminologie

Proměnná

Proměnná je alfanumerická posloupnost znaků doplněná o znak „_“ (podtržítko), která začíná velkým písmenem. Příklady proměnných:

```
Jmeno, Delka_hrany, X, Y_1
```

Toto nejsou proměnné:

```
jmeno, _delka, x, 1Y
```

Anonymní proměnná je taková proměnná, jejíž hodnota není podstatná. Značí se znakem „_“ (podtržítko).

Konstanta

Konstanta je řetězec znaků, ohraničený apostrofy, nebo číslo.

```
20, 'vyska', '21', 'Hmotnost_auta'
```

Predikátový symbol

Predikátový symbol je řetězec alfanumerických znaků, začínající malým písmenem, doplněný o znak „_“.

```
hmotnost_auta, vyska
```

Predikát

Pokud p je predikátový symbol a $(a_1, a_2, \ldots, a_n)$ seznam **argumentů**, kde a_i je proměnná (může být i anonymní) nebo konstanta, pak $p(a_1, a_2, \ldots, a_n)$ je predikát, **atom** nebo **literál**.

Uvažujme predikátový symbol *ps*, proměnné *A* a *B*, jednu anonymní proměnnou a řetězcovou konstantu *'cde'*. Potom následující zápis je predikát a *A*, *B*, _ a *'cde'* jsou jeho argumenty.

```
ps(A, B, _, 'cde')
```

Porovnání

Pokud a_i a a_j jsou proměnné, potom

a_i `>` a_j

a_i `<` a_j

a_i `>=` a_j

a_i `<=` a_j

a_i `=` a_j

a_i `\=` a_j

jsou porovnání [4] nebo **binární porovnávací predikáty** [1]. Porovnání taktéž považujeme za literál.

Vyčíslení

Vyčíslení [4] je literál ve tvaru

```
a is e
```

kde *a* je proměnná a *e* výraz, jehož hodnota je přiřazena proměnné *a*. Může vypadat například takto:

```
a is b * 2 * c
```

kde *a*, *b* a *c* jsou proměnné a *2* je číselná konstanta.

Speciálním případem vyčíslení jsou **inkrementace** a **dekrementace**.

```
a is b + 1
```

```
a is b - 1
```

Pravidlo

Pravidlo pro predikát $p(a_1, a_2, \ldots, a_n)$ zapíšeme ve tvaru

$$\texttt{p(a}_1\texttt{, a}_2\texttt{, . . . , a}_n\texttt{) :- L}_1\texttt{, L}_2\texttt{, . . . , L}_m\texttt{.}$$

kde konstanty m a n jsou přirozená čísla, $a_1, a_2, \ldots, a_n$ jsou argumenty (tedy proměnné nebo konstanty) predikátu p a $L_1, L_2, \ldots, L_m$ jsou literály.

Zápis $p(a_1, a_2, \ldots, a_n)$ reprezentuje **hlavičku** pravidla[1] (predikát p někdy nazýván jako **predikát hlavy** nebo **hlavičkový predikát**) a $L_1, L_2, \ldots, L_m$ **tělo** pravidla.

Symbol „,“ (čárka) značí v logických jazycích konjunkci a „:-“ (dvojtečka a pomlčka) implikaci. Toto pravidlo tedy odpovídá následující implikaci.

$$\texttt{L}_1 \wedge \texttt{L}_2 \wedge \texttt{. . . } \wedge \texttt{L}_m \texttt{ => p(a}_1\texttt{, a}_2\texttt{, . . . , a}_n\texttt{)}$$

Správně zapsaná pravidla[2]:

```
p2(X, Y) :- q1(Y, X).

p3(X, Y) :- q2(_, Y), q3(X, _).

p4(X, Y ,Z) :- q4(X, Y), Y >= Z.

p5(X, Y) :- X is Y + 1.
```

Nesprávně zapsané pravidlo – v hlavičce je anonymní proměnná:

```
p(X, _) :- q(X, Y).
```

Pravidlo p také může být negováno pomocí klíčového slova *not*. V tom případě jej zapíšeme takto:

```
not (p)
```

[1] V hlavičce pravidla se **nesmí** vyskytovat anonymní proměnná

[2] Pro jediné pravidlo je každá anonymní proměnná unikátní, pouze jsou všechny stejně značeny. To znamená, že anonymní proměnné v predikátech q_2 a q_3 pravidla p_3 spolu žádným způsobem nesouvisí.

Bezpečné pravidlo

Pravidlo r nazveme bezpečným [1], pokud se každá proměnná, obsažená v hlavičce pravidla, vyskytuje jako argument literálu těla tohoto pravidla. To samé platí pro negované predikáty – každý jejich argument se musí v pravidle vyskytovat v predikátu, který negován není. O pravidle, které jednu z těchto podmínek nesplňuje, řekneme, že **není bezpečné**.

Následující pravidla nejsou bezpečná:

```
p(X, Y, Z) :- q1(X), q2(X, Y).

r(A, B) :- q2(B), not (q3(A)).
```

Pravidlo p není bezpečné, neboť se proměnná Z nevyskytuje v jeho těle. Pravidlo r není bezpečné, protože proměnná A negovaného predikátu q_3 se jinde v těle pravidla r nevyskytuje.

Fakt

Pravidlo, které nemá tělo ($m = 0$), zapíšeme takto:

```
p(a1, a2, . . . , an).
```

Jestliže argumenty predikátu hlavy tohoto pravidla $(a_1, a_2, \ldots, a_n)$ jsou pouze konstanty, řekneme, že toto pravidlo je **fakt** [1, 4].

Pokud by některý argument hlavičky pravidla bez těla byl proměnnou, není toto pravidlo bezpečné, neboť nesplňuje podmínku, že každá proměnná hlavy pravidla se musí vyskytovat v těle tohoto pravidla.

Následující pravidla jsou fakty:

```
p(10).

p('12', 'Slovensko').

p(14, 'Rumunsko').
```

Tato pravidla nejsou fakty:

```
p1(X).

p2(2, _).
```

P_1 obsahuje proměnnou a p_2 není bezpečné pravidlo, protože v hlavičce má anonymní proměnnou.

Cílový dotaz

Jestliže $p(a_1, a_2, \ldots, a_n)$ je atom a $a_1, a_2, \ldots, a_n$ konstanty nebo proměnné, potom

```
? - p(a1, a2, . . . , an)
```

je cílový dotaz.

Logický program

Logický program je neprázdná množina bezpečných pravidel.

Extenzionální databáze

Extenzionální databází nazveme množinu faktů logického programu. Predikát definovaný extenzionální databází nazveme **extenzionální predikát**.

Intenzionální databáze

Intenzionální databáze vznikne vyhodnocením pravidel logického programu. Predikát, který definuje intenzionální databáze, je **intenzionální predikát**. Intenzionální predikáty jsou takové predikáty, které jsou definovány v hlavičkách pravidel logického programu.

Vestavěný predikát

Logické jazyky obvykle poskytují vestavěné predikáty, jejichž pozitivní vyhodnocení vyvolá nějakou systémovou funkci, například sečtení dvou čísel, výpis textu na obrazovku atd.

Ekvivalentní logický program

Řekneme, že dvě množiny bezpečných pravidel P_1 a P_2 jsou ekvivalentní, pokud pro jakoukoliv extenzionální databázi E produkují logické programy $P_1 \bigcup E$ a $P_2 \bigcup E$ stejné odpovědi na libovolný cílový dotaz.

2.2 Úpravy jazyka pro účely této práce

Vyčíslení, konkrétně inkrementace a dekrementace, a porovnání budou nahrazeny vestavěnými predikáty podle tabulky 2.1, kde a_1 a a_2 jsou proměnné. Ve třetím sloupci této tabulky je uvedeno, čeho je predikátový symbol zkratkou.

Původní literál	**Nahrazující vestavěný predikát**	**Predikátový symbol**
a_1 `is` a_2 + 1	`inc(`a_1`,` a_2`)`	increment
a_1 `is` a_2 − 1	`dec(`a_1`,` a_2`)`	decrement
a_1 < a_2	`lt(`a_1`,` a_2`)`	less than
a_1 > a_2	`gt(`a_1`,` a_2`)`	greater than
a_1 <= a_2	`lte(`a_1`,` a_2`)`	less than or equal
a_1 >= a_2	`gte(`a_1`,` a_2`)`	greater than or equal
a_1 = a_2	`eq(`a_1`,` a_2`)`	equal
a_1 \= a_2	`neq(`a_1`,` a_2`)`	not equal

Obrázek 2.1: Nahrazení vyčíslení a porovnání vestavěnými predikáty.

3 Graf předávání informace

Tvorba grafu předávání informace je stěžejní součástí tzv. metody magických množin, což je postup, kterým transformujeme vstupní logický program na ekvivalentní logický program, jež nazýváme magický program. Metodu magických množin aplikujeme proto, že vyhodnocení cílového dotazu rozsáhlejšího logického programu obsahujícího rekurzi může trvat příliš dlouho. Vyhodnocení stejného dotazu v ekvivalentním magickém programu zabere výrazně méně času. Tato práce se ovšem metodou magických množin jako takovou a jejími dalšími součástmi zabývat nebude; v případě zájmu se lze více dozvědět zde: [1, 2, 3].

V této kapitole budou nejprve vysvětleny pojmy týkající se grafu předávání informace a poté zaveden textový formát, ve kterém bude graf předávání informace v této práci uváděn. V tomto formátu také bude vypočtené grafy předávání informace ukládat implementovaná aplikace do textových souborů.

3.1 Terminologie

Řetězec ozdobení

Řetězec ozdobení predikátu $p(a_1, a_2, \ldots, a_n)$ je řetězec o délky n nad abecedou $\{b, f\}$. Pořadí znaků v tomto řetězci odpovídá pořadí argumentů predikátu p a zápis p_o značí predikát p ozdobený řetězcem o. Písmena **b** a **f** reprezentují zkratky *bound* (vázaný), resp. *free* (volný). Vázaný argument je takový argument, který odpovídá znaku **b**, volný argument odpovídá znaku **f**.

Uvažujme predikát *p(W, X, Y, Z)* a řetězec ozdobení **fbbf**. Ozdobený predikát bude vypadat následovně:

```
p_fbbf(W, X, Y, Z)
```

Speciální hlavičkový predikát

Nechť hlavičkou pravidla r je ozdobený predikát $p\_o(a_1, a_2, \ldots, a_n)$. Potom $p\_h(b_1, \ldots, b_m)$, kde $b_1, \ldots, b_m$ jsou vázané argumenty a m je jejich počet, je speciální hlavičkový predikát pravidla r. Platí $m < n$. Jestliže $m = 0$, potom speciální hlavičkový predikát pro daný hlavičkový predikát nevytváříme.

Uvažujme následující pravidlo s ozdobeným predikátem hlavy:

```
p_bfb(X, Y, Z) :- q1(X), q2(X, Y), q3(Z, Y).
```

Potom speciálním hlavičkovým predikátem pro toto pravidlo je:

```
p_h(X, Z)
```

Graf

Grafem rozumíme reprezentaci množiny objektů, u kterých je důležité, a tedy tento fakt chceme nějakým způsobem znázornit, že některé jsou určitým způsobem propojeny. Graf G zapisujeme jako

$G = \langle V, E \rangle$

kde V je množina vrcholů (reprezentace objektů) a E množina hran (reprezentace propojení objektů).

Neorientovaný a orientovaný graf

U neorientovaného grafu hranu e chápeme jako dvouprvkovou množinu $e = \{u, v\}$, kde u a v jsou nějaké vrcholy z množiny V grafu G, a říkáme, že tato (neorientovaná) hrana spojuje vrcholy u a v.

U orientovaného grafu chápeme hranu jako uspořádanou dvojici $\langle u, v \rangle$, protože záleží, jak jsou vrcholy propojeny. Zápis $e = \langle u, v \rangle$ znamená, že (orientovaná) hrana e vede z vrcholu u do vrcholu v, kdežto $e = \langle v, u \rangle$ značí, že tato hrana vede z vrcholu $v$do vrcholu u (je orientována na opačnou stranu).

Představme si následující příklad:

```
G = <V, E>

V - množina měst

E - množina silnic; každá vede právě mezi dvěma městy

a, b, c, d - nějaká města z množiny V

e1 = {a, b}, e2 = ⟨c, d⟩
```

Potom po neorientované hraně e_1 můžeme libovolně cestovat mezi městy a a b, kdežto po orientované hrané e_2 lze cestovat pouze směrem z města c do města

d, nikoliv naopak.

Ohodnocený graf

Ohodnocený orientovaný graf [5] *(G, w)* je orientovaný graf G spolu s reálnou funkcí $w : E(G) \rightarrow (0,\infty)$. Je-li e hrana grafu G, číslo $w(e)$ se nazývá její ohodnocení nebo váha.

Řekneme-li například, že funkce w určuje kvalitu silnice a do příkladu z předchozího bodu přidáme informaci

```
w(e2) = 2
```

znamená to, že z města c do města d vede silnice druhé třídy.

Graf předávání informace

Graf předávání informace (někdy nazýváme také **úplný graf předávání informace**, dále také SIP) pro pravidlo r s ozdobenou hlavičkou je ohodnocený orientovaný graf, ve kterém vrcholy jsou reprezentací speciálního hlavičkového predikátu a všech predikátů těla pravidla r. Orientované hrany tohoto grafu reprezentují směr předávání informace a jsou ohodnoceny množinou proměnných, přes které se daná informace předává.

Jestliže v hlavičce pravidla není žádný vázaný argument, potom pro toto pravidlo graf předávání informace neexistuje.

Pokud pravidlo nemá tělo (je faktem), graf předávání informace pro něj taktéž neexistuje.

Finální graf předávání informace

Finální graf předávání informace (dále také FSIP) je úpravou (viz kapitolu 3.2.2) úplného grafu předávání informace . Tuto úpravu bychom mohli brát jako zjednodušení původního (úplného) grafu a provádíme ji proto, že do dalšího kroku metody magických množin není potřeba úplný graf předávání informace, resp. údaje, které z něj touto úpravou odstraníme, jsou pro metodu magických množin nadbytečné a jejich absence nijak neovlivní její výstup.

3.2 Textový formát

3.2.1 Úplný graf předávání informace

Formát jedné hrany

Uvažujme upravený predikát hlavy p (speciální hlavičkový predikát), ze kterého hrana vychází, cílový predikát těla q_i a množinu proměnných $X_i = \{x_1, x_2, \ldots, x_n\}$, kterými je hrana ohodnocena. Poté hranu obecně zapíšeme takto:

`{p} -- x1, x2, . . . , xn --> qi`

Příklad 1:

`p = intersection_h`

`X1 = Elem1, Level2`

`q1 = xml`

Výsledná hrana:

```
{intersection_h} -- Elem1, Level2 --> xml
```

Pokud p není predikát hlavy, zapíšeme hranu obecně takto[1]:

`{p-n, p-n+1, ..., p-2, p-1; p} -- x1, x2, ..., xn --> qi`

kde p$_{-n}$ je predikát hlavy a $\langle p_{-n}$, p$_{-n+1}\rangle$, $\langle p_{-n+1}$, p$_{-n+2}\rangle$, ..., $\langle p_{-2}$, p$_{-1}\rangle$, $\langle p_{-1}$, p$\rangle$ představují cestu přes orientované hrany z předchozích iterací[2] z predikátu p$_{-n}$ do predikátu p.

Příklad 2:

`p-n = intersection_h`

`p-1 = xml`

[1] Všechny predikáty ve složené závorce jsou odděleny čárkami, kromě posledního, který je oddělen středníkem.

[2] Iterací rozumíme kroky algoritmu **(4)** až **(7)** z kapitoly Algoritmus na str. 17

`p = dec`

$\texttt{X}_i$ `= {Level1}`

$\texttt{q}_i$ `= self`

Dále uvažujme, že v předchozích dvou krocích vznikly postupně tyto hrany:

`{intersection_h}` -- $\texttt{X}_a$ --> `xml`

`{intersection_h; xml}` -- $\texttt{X}_b$ --> `dec`

kde X_a a X_b jsou množiny proměnných, kterými jsou tyto dvě hrany ohodnoceny a *intersection_h* je speciální predikát hlavy. Výsledná hrana v tomto kroku bude vypadat takto:

`{intersection_h, xml; dec}` -- `Level1` --> `self`

Formát celého grafu pro jedno pravidlo

Nejprve kvůli přehlednosti a rozlišení stejně se jmenujících predikátů těla pravidla zaveďme konvenci, že za název predikátu (predikátový symbol) těla vždy přidáme tečku a pořadové číslo (číslujeme zleva od 1 stejně se jmenující predikáty[3]). Nečíslujeme hlavičkový predikát, neboť ten je vždy převeden na speciální hlavičkový predikát.

Uvažujme následující pravidlo logického programu a řetězec ozdobení **fbfb**:

```
intersection(Line1, Line2, Element, Level1) :-
  xml(Line1, Element, Order, Level1),
  xml(Line2, Element, Order, Level1),
  dec(Level2, Level1),
  intersection(Line1, Line2, _, Level2).
```

Upravíme predikát hlavy na speciální predikát hlavy a dle zavedené konvence přepíšeme pravidlo do této podoby:

```
intersection_h(Line2, Level1) :-
  xml.1(Line1, Element, Order, Level1),
```

[3]Číslujeme i predikáty s neopakujícími se názvy.

```
xml.2(Line2, Element, Order, Level1),
dec.1(Level2, Level1),
intersection.1(Line1, Line2, _, Level2).
```

Hrany, vytvořené podle algoritmu konstrukce grafu předávání informace (kapitola 4.1) a upravené do správného formátu[4] můžeme vidět na obrázku 3.1. Důležité je zarovnání do tří sloupců. Dále také na každé řádce mezi každými dvěma sousedními sloupci jsou alespoň dvě pomlčky.

```
{intersection_h} ---------- Level1 ----------> xml.1
{intersection_h} ---------- Line2, Level1 ---> xml.2
{intersection_h} ---------- Level1 ----------> dec.1
{intersection_h} ---------- Line2 ----------> intersection.1
{intersection_h; xml.1} -- Element, Order --> xml.2
{intersection_h; xml.1} -- Line1 ----------> intersection.1
{intersection_h; dec.1} -- Level2 ----------> intersection.1
```

Obrázek 3.1: Hrany grafu předávání informace.

[4]Vždy jedna hrana na jedné řádce.

3.2.2 Finální graf předávání informace

Finální graf předávání informace vznikne jednoduchou úpravou z úplného grafu předávání informace. Formát, co se týče zarovnání sloupců, zápisu proměnných atd., je stejný.

Uvažujme, že

$$\texttt{\{p}_{-n}\texttt{, ..., p}_{-2}\texttt{, }_{-1}\texttt{; p\} -- x}_1\texttt{, x}_2\texttt{, ... , x}_n\texttt{ --> q}_i$$

je hrana grafu předávání informace ve správném formátu, zavedeném v předchozí kapitole (3.2.1).

Potom, pokud p je intenzionální predikát, bude hrana ve finálním grafu předávání informace vypadat takto:

$$\texttt{\{p\} -- x}_1\texttt{, x}_2\texttt{, ... , x}_n\texttt{ --> q}_i$$

Pokud je ovšem p extenzionální nebo vestavěný[5] predikát, zapíšeme hranu finálního grafu předávání informace obecně takto:

$$\texttt{\{p}_{-m}\texttt{, ..., p}_{-2}\texttt{, p}_{-1}\texttt{; p\} -- x}_1\texttt{, x}_2\texttt{, ... , x}_n\texttt{ --> q}_i$$

kde $m <= n$, p_{-m} je intenzionální predikát a p_{-m+1}, ..., p_{-1} *a* p jsou predikáty extenzionální, přičemž $\langle p_{-m}, p_{-m+1}\rangle, \langle p_{-m+1}, p_{-m+2}\rangle, ..., \langle p_{-1}, p\rangle$ představují cestu přes orientované hrany z předchozích iterací z predikátu p_{-m} do predikátu p.

V případě, že je extenzionální (nebo vestavěný) predikát q_i, bude příslušná hrana vynechána z finálního grafu předávání informace úplně.

Příklad 1:

$\texttt{p}_{-n}$ `= intersection_h`

$\texttt{p}_{-1}$ `= xml.1`

`p = dec.1`

$\texttt{X}_i$ `= Level1`

[5]Extenzionální a vestavěné predikáty mají na finální graf předávání informace stejný vliv. Budeme-li tedy kdykoliv dále v této práci mluvit o extenzionálním predikátu či predikátech, je možné, že některý z nich, nebo všechny, je predikát vestavěný.

`q`$_i$ `= self.1`

Dále uvažujme že v předchozích dvou iteracích vznikly postupně hrany:

`{intersection_h} -- X`$_a$ `--> xml.1`

`{intersection_h; xml.1} -- X`$_b$ `--> dec.1`

kde X_a a X_b jsou množiny proměnných, kterými jsou tyto dvě hrany ohodnoceny, a *xml* je extenzionální predikát.

Kdyby *dec* byl intenzionální predikát, výsledná hrana v této iteraci bude vypadat takto :

```
{dec.1} -- Level1 --> self.1
```

v opačném případě (predikát *dec* je extenzionální (nebo vestavěný)) zapíšeme tuto hranu následovně:

```
{intersection_h, xml.1; dec.1} -- Level1 --> self.1
```

Příklad 2:

Uvažujme pravidlo

```
intersection(Line1, Line2, Element, Level1) :-
  xml(Line1, Element, Order, Level1),
  xml(Line2, Element, Order, Level1),
  dec(Level2, Level1),
  intersection(Line1, Line2, _, Level2).
```

kde *xml* a *dec* jsou extenzionální predikáty. Dále uvažujme řetězec ozdobení **bfbb** a příslušný graf předávání informace, který můžeme vidět na obrázku 3.2a na str. 16.

Na stejné stránce na obrázku 3.2b vidíme finální graf předávání informace ve správném formátů po odstranění hran vedoucích do extenzionálních predikátů.

```
{intersection_h} ---------------- Line1, Element, Level1 --> xml.1
{intersection_h} ---------------- Element, Level1 ----------> xml.2
{intersection_h} ---------------- Level1 ------------------> dec
{intersection_h} ---------------- Line1 -------------------> intersection.1
{intersection_h; xml.1} --------- Order -------------------> xml.2
{intersection_h, xml.1; xml.2} -- Line2 -------------------> intersection.1
{intersection_h; dec.1} --------- Level2 -------------------> intersection.1
```

(a) Hrany úplného grafu.

```
{intersection_h} ---------------- Line1 ---> intersection.1
{intersection_h, xml.1; xml.2} -- Line2 ---> intersection.1
{intersection_h; dec.1} --------- Level2 ---> intersection.1
```

(b) Hrany finálního grafu.

Obrázek 3.2: Hrany grafu předávání informace.

4 Konstrukce grafu předávání informace

4.1 Algoritmus

Vstup

Bezpečné pravidlo a řetězec ozdobení.

Výstup

Graf předávání informace[1] pro toto pravidlo a daný řetězec ozdobení.

Kroky algoritmu

(1) Ozdobíme hlavičku pravidla zadaným řetězcem ozdobení.

(2) Transformujeme vstupní pravidlo nahrazením predikátu hlavičky speciálním hlavičkovým predikátem. Graf předávání informace tedy nyní obsahuje uzly, které reprezentují tento predikát a všechny predikáty těla pravidla. Neobsahuje zatím ale žádné hrany.

(3) Řekneme, že množina **B** je množina vázaných proměnných a obsahuje všechny proměnné speciálního hlavičkového predikátu. Množina **U** použitých proměnných je prozatím prázdná.

(4) Do grafu přidáme hrany, ohodnocené množinou proměnných X_i, které vedou z predikátu p do všech takových predikátů těla q_i, které jsou propojeny s predikátem p právě přes proměnné z odpovídající množiny X_i. Pro každou množinu X_i platí bez výjimky tyto podmínky:

- Všechny proměnné v této množině musí být proměnné predikátů p a q_i (ale ne naopak, tedy ne každá proměnná predikátů p a q_i musí být obsažena v množině X_i).

[1] Obecně pro jediné pravidlo a řetězec ozdobení může v některých případech existovat více různých grafů. Viz Poznámky k algoritmu na str. 18.

- Všechny proměnné v této množině **musí** být prvky množiny **B**.
- **Žádná** z proměnných této množiny není prvek množiny **U**.

(5) Do množiny **B** přidáme proměnné predikátů q_i, a do množiny **U** proměnné predikátu p.

(6) Pokud jsou množiny **U** a **B** shodné, tzn. nachází se v nich stejné prvky, je graf předávání informace hotov.

(7) Z grafu vybereme predikát p takový, že všechny jeho proměnné jsou prvky množiny **B**, ale zároveň všechny nejsou prvky množiny **U**, tzn. je k dispozici ještě nejméně jedna proměnná, která dosud nebyla k přenášení informace použita. Pokračujeme krokem **(4)**.

Poznámky k algoritmu

1) V některých predikátech pravidla se mohou vyskytovat anonymní proměnné. Avšak každá anonymní proměnná je vždy pro jedno pravidlo striktně unikátní a nemůže tedy v žádném případě splnit podmínku, že se vyskytuje zároveň ve více predikátech. (Všechny proměnné v množině X_i musí být proměnné predikátů p a q_i.) Z tohoto důvodu nemají anonymní proměnné pro graf předávání informace žádný význam a lze je ignorovat. Anonymní proměnné se nevkládají do množin **B** a **U**.

2) V kroku **(7)** může často nastat situace, že lze jako predikát p vybrat více různých predikátů, čímž by pro zadané pravidlo a řetězec ozdobení vzniklo více grafů předávání informace. Výběr predikátu p v tomto kroku algoritmu může později ovlivnit další kroky metody magických množin, resp. jak kvalitně bude metoda optimalizovat vyhodnocení dotazu logického programu. My ovšem zavedeme konvenci[2], že v tomto kroku budeme vybírat vždy ten predikát, který je ve vstupním pravidle uveden nejdříve.

[2]Samozřejmě obecně není nutné ji dodržovat. Zde ji zavádíme proto, aby byla konstrukce grafu deterministická, tzn. pro jedno pravidlo a řetězec ozdobení jsme dostali vždy ten samý graf. Vliv tohoto výběru na další části metody magických množin již není součástí této práce a proto není třeba se jím hlouběji zabývat.

4.2 Ukázka aplikace algoritmu

Uvažujme pravidlo

```
child(Line1, Line2, Elem1, Elem2) :-
  self(Line1, Elem1),
  self(Line2, Elem2),
  intersection(Line1, Line2, Elem1, Elem2),
  inc(Level2, Level1),
  xml(Line2, Elem2, _, Level2).
```

a řetězec ozdobení **fbbf**.

Dle zavedené konvence změníme pravidlo do následující podoby:

```
child(Line1, Line2, Elem1, Elem2) :-
  self.1(Line1, Elem1),
  self.2(Line2, Elem2),
  intersection.1(Line1, Line2, Elem1, Elem2),
  inc.1(Level2, Level1),
  xml.1(Line2, Elem2, _, Level2).
```

Nyní můžeme přejít ke konstrukci grafu předávání informace.

Nejprve ozdobíme predikát hlavičky pravidla **(1)**[3]

```
child_fbbf(Line1, Line2, Elem1, Elem2)
```

a vytvoříme z něj speciální hlavičkový predikát **(2)**, o kterém dále řekneme, že pro tento krok je predikátem p.

```
child_h(Line2, Elem1)
```

Inicializujeme množiny **B** a **U** **(3)**.

```
B = {Line2, Elem1}

U = {}
```

[3]Tučná čísla v kulatých závorkách v této kapitole odpovídají krokům algoritmu konstrukce grafu předávání informace z kapitoly 4.1.

Nyní podle kroku **(4)** najdeme takové predikáty, které mají společné proměnné s predikátem p, a vytvoříme hrany podle formátu zavedeného v kapitole 3.2.1. Do grafu tedy přidáme tyto hrany:

```
{child_h} -- Elem1 ---------> self.1
{child_h} -- Line2 ---------> self.2
{child_h} -- Line2, Elem1 --> intersection.1
{child_h} -- Line2 ---------> xml.1
```

Příslušně upravíme množiny **B** a **U** **(5)** – tzn. do **U** přidáme všechny proměnné predikátu *child_h* a do **B** proměnné všech predikátů, kam vedou hrany přidané v předchozím kroku.

```
B = {Line2, Elem1, Elem2, Line1, Level1, Level2}

U = {Line2, Elem1}
```

Na první pohled vidíme, že podmínka ukončení algoritmu **(6)** – shodnost množin **B** a **U** – není splněna a tedy pokračujeme dalším krokem **(7)**. Nyní musíme vybrat, kterým predikátem budeme pokračovat.
V úvahu přicházejí tyto:

```
self.1, self.2, intersection.1, xml.1
```

Podle zavedené konvence (v tomto kroku **(7)** vybíráme vždy ten predikát, který je ve vstupním pravidle uveden nejdříve) zvolíme predikát *self.1* jako nový predikát p a pokračujeme krokem **(4)**.

Přidáme tedy do grafu jedinou hranu, která vyhovuje podmínce, že má s predikátem p společné proměnné, které ještě nejsou v množině **B**. Těmito proměnnými (zde pouze jedinou proměnnou) je tato nová hrana ohodnocena.

```
{child_h; self.1} -- Line1 --> intersection.1
```

Podle **(5)** změníme obsah množiny **U**.

```
U = {Line2, Elem1, Line1}
```

Obsah množiny **B** zůstane nezměněn, jelikož predikáty, kam vedou hrany (v tomto případě jen jedna hrana) přidané v tomto kroku, nemají žádné proměnné, které v **B** již nejsou.

Podmínka **(6)** opět není splněna, pokračujeme tedy krokem **(7)**, kde vybereme predikát *self.2* a pokračujeme krokem **(4)**.

Analogicky, dle stejných pravidel, kterými jsme přidávali hrany vedoucí z predikátu *intersection_h* a *self.1*, provedeme několik dalších iterací, během nichž do grafu předávání informace postupně přidáme hrany:

```
{child_h; self.2} -- Elem2 --> xml.1

{child_h, self.1; intersection.1} -- Level1 --> inc.1

{child_h, self.1, intersection.1; inc} -- Level2 --> xml.1
```

Po vytvoření poslední z uvedených hran budou množiny **B** a **U** po provedení kroku **(5)** vypadat takto:

```
B = {Line2, Elem1, Elem2, Line1, Level1, Level2}

U = {Line2, Elem1, Elem2, Line1, Level1, Level2}
```

Nyní vidíme, že jsou množiny shodné a proto podle podmínky **(6)** ukončíme konstrukci grafu předávání informace, který je tím pádem hotov. Můžeme jej (resp. všechny jeho hrany) vidět na obrázku 4.1a na str. 22.

Protože názorných příkladů není nikdy dost, ukážeme si také, jak by vypadal finální graf předávání informace za předpokladu, že predikát *self* je intenzionální[4]. Lze jej spatřit na straně 22 na obrázku 4.1b.

[4]Predikát *intersection* je intenzionální z definice, neboť se vyskytuje v hlavičce pravidla.

```
{child_h} ------------------------------ Elem1 ----------> self.1
{child_h} ------------------------------ Line2 ----------> self.2
{child_h} ------------------------------ Line2, Elem1 --> intersection.1
{child_h} ------------------------------ Line2 ----------> xml.1
{child_h; self.1} ---------------------- Line1 ----------> intersection.1
{child_h; self.2} ---------------------- Elem2 ----------> xml.1
{child_h, self.1; intersection.1} ------ Level1 ----------> inc.1
{child_h, self.1, intersection.1; inc.1} -- Level2 ----------> xml.1
```

(a) Hrany úplného grafu.

```
{child_h} -- Elem1 ----------> self.1
{child_h} -- Line2 ----------> self.2
{child_h} -- Line2, Elem1 --> intersection.1
{self.1} --- Line1 ----------> intersection.1
```

(b) Hrany finálního grafu.

Obrázek 4.1: Hrany grafu předávání informace.

5 Specifikace implementace

Konzolová aplikace

Aplikace bude spouštěna z konzole, resp. příkazové řádky – parametry (například cesta ke vstupnímu souboru) budou zadávány hned při spuštění. To znamená, že aplikaci nebude možné spustit přímo ze správce (průzkumníku) souborů, neboť při tomto způsobu spuštění nelze zadat parametry.

Dávkový a grafický mód

V dávkovém módu aplikace načte vstupní soubor (resp. logický program) a pro všechna pravidla v něm obsažená vypočte příslušné grafy, které opět uloží do textových souborů. Od uživatele nebude kromě zadání parametrů na příkazové řádce vyžadována žádná další interakce.

Avšak, jak bylo řečeno v kapitole 4.1, tvorba grafu předávání informace není deterministická[1] a tudíž byla zavedena jistá konvence, podle které se aplikace v dávkovém módu chová, což znamená, že vytvoří pouze jediný graf pro každé pravidlo a řetězec ozdobení. V některých případech bychom ovšem chtěli získat jiný graf, než který poskytuje dávkový mód, proto bude možno aplikaci spustit[2] s grafickým uživatelským rozhraním (dále jen GUI), kde aplikace nechá uživatele jednak načíst pravidlo ze souboru a vybrat řetězec ozdobení, pro který bude SIP vytvářen, ale především jak bude tvorba grafu pokračovat, kdykoliv bude více než jedna možnost.

Aplikace v grafickém módu bude průběžně vykreslovat již vytvořenou část grafu, aby se uživatel mohl pohodlně rozhodnout, jak je s dosavadním výsledkem spokojen a na jeho základě vybrat, jak bude tvorba grafu dále pokračovat. Navíc bude GUI kromě standardního uložení do textového souboru poskytovat také možnost uložit vytvořený graf jako obrázek ve formátu PNG. Velice vhodná by také byla možnost změnit pozice uzlů a hran grafu na plátně před jeho uložením do obrázku, kdyby se výchozí pozice generované aplikací uživateli nelíbily.

[1] Pro jeden řetězec ozdobení a pravidlo lze vytvořit více grafů.

[2] Pomocí parametru na příkazové řádce.

Rychlost generování grafů v dávkového módu

Dalším důležitým požadavkem je, aby byly grafy generovány přijatelně rychle. Bylo by vhodné, aby aplikace vygenerovala grafy pro všechna pravidla reálného logického programu[3] v řádu několika jednotek, maximálně desítek sekund. Pro programy s méně než deseti pravidly by mělo být generování hotové takřka okamžitě.

Přehlednost implementace

Aplikace by měla být implementována tak, aby se v budoucnu daly části kódu (především vlastní algoritmus generování SIP, případně také parsování vstupního logického programu) bez větších problémů dále využít, například v nějakém rozsáhlejším programu, který by implementoval celou metodu magických množin. GUI by mělo být zcela oddělitelné.

Tomuto tématu – jak a které metody a části kódu lze dále využít v jiných programech – bude věnována celá kapitola 7.

Parsování vstupního souboru

Aplikace musí umět parsovat běžné pravidlo (i s negovanými predikáty v těle), ať už rozepsané na více řádek nebo zapsané celé v jedné řádce. Například:

```
p1(a1, a2) :-
  not(q1(a3, a2)),
  q2(a3, a2, a4),
  q3(a1, a3).

p2(a5, a6) :- q4(a6, a7), q5(a7, a5).
```

kde a_n jsou proměnné, q_n predikáty těla a p_n hlavičkové predikáty.

[3] Maximálně několik set pravidel, ale spíše desítky.

Není třeba, aby aplikace uměla rozpoznat vyčíslení a porovnání, neboť ty budou nahrazeny vestavěnými predikáty (viz kap. 2.2).

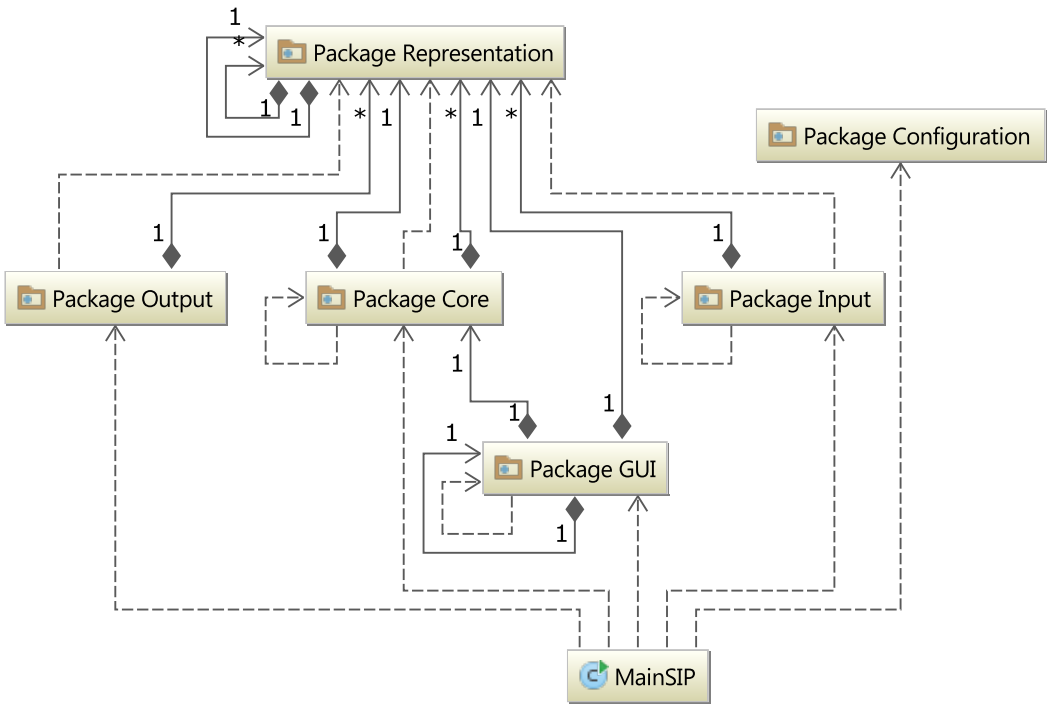
Dále také aplikace bude umět rozpoznat řádkový komentář (první znak řádky je %) a blokový komentář (blok více řádek uvozený /∗ a ∗/). Komentáře ovšem není dovoleno vnořovat.

6 Implementace

V této kapitole bude nejprve popsána struktura balíků, co který obsahuje a co dělá. Následně již bude popsána konkrétní implementace důležitých postupů a algoritmů, případně jakým způsobem byly některé z nich optimalizovány. Popisovány pochopitelně nebudou veškeré implementační detaily, ale pouze významné, či nějakým způsobem zajímavé části.

6.1 Struktura kódu aplikace

Aplikace je členěna do několika balíků podle obrázku 6.1, kde *MainSIP* je třída, která obsahuje metodu *main*.



Obrázek 6.1: UML diagram balíků projektu.

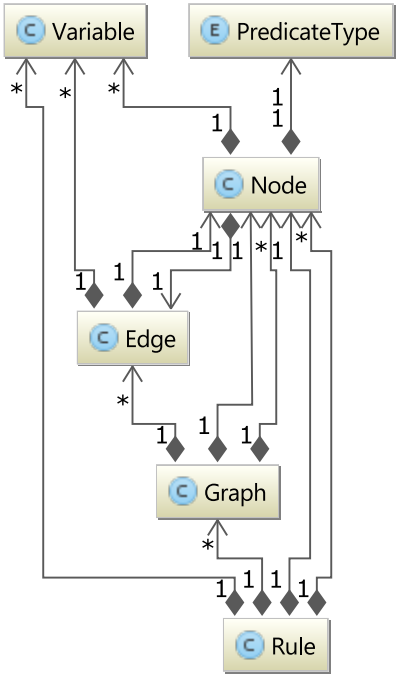
Zde pouze zevrubně nastíním, co ve kterém balíku je; podrobnější popis a některé implementační detaily se nacházejí až v další podkapitole (Vlastní implementace).

- *Configuration* – Obsahuje předně třídu s neměnnými nastaveními, jako například regulární výrazy použité při parsování vstupního souboru, ale také načítá ze souboru jazyk, ve kterém bude aplikace s uživatelem komunikovat.
- *Representation* – Reprezentace dat (grafy, predikáty, atd.) v paměti.
- *Input* – V tomto balíku se nacházejí třídy, zajišťující čtení vstupního programu ze souboru, jeho parsování a vytváření reprezentace dat v paměti.
- *Core* – Zde se nachází jádro aplikace – algoritmus generování SIP.
- *Output* – Zajišťuje převod vygenerovaných grafů do textového formátu a následně uložení do textových souborů.
- *GUI* – Tento balík obsahuje veškeré třídy, potřebné ke spuštění a správnému běhu aplikace v módu s GUI. (UML diagram na obr. A.1.)

6.2 Vlastní implementace

6.2.1 Reprezentace dat

Snaha byla taková, aby vztahy mezi jednotlivými datovými strukturami (viz UML diagram tříd na obr. 6.2) co nejvíce odpovídaly realitě. Instance třídy *Rule* odpovídá jednomu pravidlu, načtenému ze souboru; obsahuje predikáty (třída *Node*) a proměnné (třída *Variable*). Výčtový typ *PredicateType* určuje, jestli je predikát intenzionální nebo extenzionální. Instance třídy *Graph* odpovídá SIPu a vznikne z pravidla a řetězce ozdobení, tzn. jedné instanci třídy *Rule* náleží tolik instancí třídy *Graph*, kolik je možností, jak vytvořit řetězec ozdobení predikátu hlavy pravidla, které příslušná instance třídy *Rule* uchovává. Instance třídy *Edge* potom odpovídají hranám vypočtených SIPů.



Obrázek 6.2: UML diagram tříd balíku *Representation*.

6.2.2 Vstup

Načítání souboru

Průběh čtení vstupního souboru je znázorněn pseudokódem 6.1. Ze souboru jsou v cyklu načítány řádky a pro každou je zkontrolováno, jestli je řádkovým komentářem, nebo začátkem blokového komentáře. V prvním případě je řádka zahozena, ve druhém jsou v cyklu řádky zahazovány, dokud není nalezen konec blokového komentáře. Každá platná řádka je přidávána do proměnné *ruleStr*, která uchovává právě načítané pravidlo jako řetězec znaků. V momentě, kdy je nalezen znak, kterým se ukončuje pravidlo, je načtené pravidlo z proměnné zkopírováno do seznamu načtených pravidel (v pseudokódu proměnná *rulesList*) a proměnná *ruleStr* je vynulována.

Pseudokód 6.1 Načítání souboru

Vstup: Vstupní soubor

Výstup: Seznam řetězců znaků, odpovídajících jednotlivým pravidlům

```
 1: procedure readFile
 2:     initialize rulesList and ruleStr
 3: loop:
 4:     while file has next line do
 5:         line ← read next line from file.
 6:         if line is start of block comment then
 7:             while true do
 8:                 line ← read next line from file.
 9:                 if line is end of block comment then
10:                     goto loop.
11:                 end if
12:             end while
13:         end if
14:         if line is line comment then
15:             goto loop.
16:         end if
17:         ruleStr ← ruleStr + line.
18:         if line is end of rule then
19:             add ruleStr to rulesList.
20:             ruleStr ← null.
21:         end if
22:     end while
23: end procedure
```

Parsování pravidla

Pravidlo, načtené jako řetězec znaků, je pomocí regulárního výrazu a knihovní metody *String.split* rozděleno na jednotlivé predikáty, které jsou stejným způsobem děleny na predikátový symbol a jednotlivé proměnné. Následně jsou z těchto znakových řetězců vytvořeny příslušné instance tříd *Variable*, *Node* a *Rule*.

6.2.3 Graf předávání informace

Tvorba instancí třídy *Graph*

Protože vstupem algoritmu generování SIPu je instance třídy *Graph*, je nutné pro každé pravidlo (*Rule*) nejdříve vytvořit pro každý možný řetězec ozdobení instanci této třídy. Řetězce ozdobení jsou generovány rekurzivní metodou, která na vstupu dostává řetězec znaků (v první iteraci prázdný), vytvoří z něj dva nové přidáním znaků **b** a **f** (např. z **bf** vznikne **bfb** a **bff**) a poté zavolá sama sebe pro každý z těchto dvou nových řetězců. Jakmile se takto dosáhne požadované délky řetězce ozdobení, je tento řetězec přidán do seznamu a rekurzivní volání končí. Pro každý z řetězců ozdobení ze seznamu je poté vytvořena instance třídy *Graph*.

Algoritmus generování grafu předávání informace

Graf je vytvářen rekurzivní[1] metodou *createEdgesOneStep* (viz pseudokód 6.2 nebo zdrojový kód v příloze C.1), kdy jeden její průchod (respektive jedno volání) odpovídá jedné iteraci algoritmu generování SIPu (kroky **(4)** až **(7)**, viz kap. 4.1). Před voláním této metody jsou inicializovány seznamy vázaných a použitých proměnných (B a U) a do seznamu vázaných proměnných jsou přidány argumenty predikátu hlavy. Dále je také inicializován důležitý seznam (v pseudokódu *possibleNextSourceNodes*), který uchovává ty predikáty (instance třídy *Node* – dále také **uzel**), které vyhovují podmínce z kroku **(7)**.

Metoda *createEdgesOneStep* jako argumenty dostává uzel *srcNode*, ze kterého budou vycházet hrany, tvořené v této iteraci (tzn. v první iteraci dostává predikát hlavy), a hranu *prevEdge* (instance třídy *Edge*), což je hrana, která vede do uzlu *srcNode* (v první iteraci je tedy **null**, neboť do predikátu hlavy žádné hrany nevedou). Postupně projde každý uzel (v pseudokódu *bodyNode*) těla pravidla (s výjimkou *srcNode*) a zkoumá, jestli má proměnné, které nebyly ještě použity k přenášení informace. Vyhovující proměnné (viz krok **(4)** algoritmu generování SIPu v kap. 4.1) jsou přidány do seznamu (*edgeVars*).

[1] Rekurze byla zvolena místo jednoduchého cyklu proto, že v módu s GUI je potřeba tvorbu SIPu krokovat, což by u cyklu bylo zbytečně komplikované.

Pseudokód 6.2 Generování SIPu

Vstup: Graf (*Graph*) bez hran

Výstup: Hrany grafu

```
1: initialize B and U.                                  ▷ Bound and Used variables
2: add head node's variables to B.
3: initialize possibleNextNodesList.
4:
5: procedure createEdgesOneStep(srcNode, prevEdge)
6:     initialize edgeVars.                             ▷ List of edge's variables
7:     declare chosenNextNode.
8:     for every bodyNode of Graph do                   ▷ All nodes except head node
9:         empty edgeVars.
10:        for every variable in bodyNode do
11:            if variable satisfies conditions then    ▷ See chapter 4.1
12:                add variable to edgeVars.
13:            end if
14:        end for
15:        if edgeVars is not empty then
16:            if possibleNextNodesList does not contain bodyNode then
17:                add bodyNode to possibleNextNodesList.
18:            end if
19:            add content od edgeVars to B.            ▷ Without duplications
20:            Edge ← (srcNode, bodyNode, edgeVars, prevEdge).   ▷ New edge
21:            add Edge to Graph.
22:            bodyNode's input edge ← Edge.
23:        end if
24:    end for
25:    add srcNode's variables to U.                    ▷ Without duplications
26:    if B and U do not contain same variables then
27:        chosenNextNode ← chooseNextNode().
28:        if chosenNextNode ≠ null then
29:            inputEdge ← input edge of chosenNextNode.
30:            createEdgesOneStep(chosenNextNode, inputEdge).
31:        end if
32:    end if
33: end procedure
```

Pokud takové proměnné byly nalezeny (tzn. *edgeVars* není prázdný), je provedeno následující:

- Všechny proměnné uzlu *bodyNode* jsou přidány do seznamu vázaných proměnných (B).
- Pokud *bodyNode* není obsažen v seznamu *possibleNextSourceNodes*, je do něj přidán.
- Je vytvořená nová hrana (instance třídy *Edge*), vedoucí ze *srcNode* do *bodyNode*, a je ohodnocena množinou proměnných *edgeVars*. Tato hrana je přidána do grafu (*Graph*).
- Uzlu *bodyNode* je nastavena vytvořená hrana jako vstupní hrana. Toto provázání je nezbytné[2], aby bylo možno pro každý uzel (predikát) grafu rekonstruovat cestu, kudy byla informace předávána, až do predikátu hlavy.

Poté je do seznamu použitých proměnných přidána každá proměnná uzlu, ze kterého vedou hrany, vytvořené v této iteraci (*srcNode*). Nyní, jestliže není splněna podmínka ukončení generování SIPu (shodnost seznamů U a B), je ze seznamu *possibleNextSourceNodes* vybrán metodou *chooseNextNode* (viz zdrojový kód v příloze C.2) uzel (*chosenNextNode*) podle konvence zavedené v kapitole 4.1 (Poznámka **2** na str. 18) a metoda *createEdgesOneStep* je zavolána znovu. Jako argumenty dostává vybraný uzel a hranu, která do něj vede.

6.2.4 Výstup

Před vlastním zápisem souborů je vytvořen nový adresář, který má v názvu časovou známku (opakované spuštění programu tedy nepřepisuje soubory a adresáře z předchozích spuštění). V tomto adresáři je dále pro každé pravidlo ze vstupního souboru založen další adresář, aby byly grafy předávání informace pro konkrétní pravidlo dobře k nalezení a nebylo je třeba hledat v jediném adresáři, obsahujícím stovky souborů.

Každý textový soubor obsahuje původní pravidlo, upravené pravidlo (predikát hlavy upraven na speciální predikát hlavy podle řetězce ozdobení a predikáty těla očíslované podle konvence zavedené v kap. 3.2.1), hrany SIPu a hrany FSIPu. Ukázku výstupního textového souboru lze najít v příloze D.

[2]Bude potřeba při uvádění grafu do textového formátu.

Pro každý jeden vygenerovaný graf předávání informace (tzn. jeden soubor) je výstupní text nejprve postupně složen opakovaným voláním metody a*ppend* knihovní třídy *StringBuilder* (viz zdrojový kód v příloze C.3) a následně najednou zapsán do souboru jediným voláním metody *write* knihovní třídy *FileWriter*. Každá třída balíku *Representation* překrývá metodu *toString()*, která vrací textovou reprezentaci příslušné instance třídy tak, jak má být uložena ve výstupním souboru (příloha D).

Optimalizace

Výstup do textových souborů doznal největší optimalizace z hlediska časové náročnosti. V prvotních verzích aplikace bylo do souboru zapisováno postupně (tj. téměř každé nynější volání metody *append* třídy *StringBuilder* bylo původně voláním metody *write* třídy *FileWriter*) a formátování výstupu probíhalo pomocí knihovní metody *String.format*. Tyto dvě praktiky, především metoda *format*[3], zbytečně výstup zpomalovaly a proto byly nahrazeny současným způsobem.

Celková časová úspora, které takto bylo dosaženo, je významná – průměrný čas, potřebný na načtení jednoho pravidla, vytvoření příslušných SIPů a jejich uložení do souborů, klesl přibližně na polovinu (viz kap. 8.1).

6.2.5 GUI

Veškeré třídy, náležící GUI, se nachází v balíku *GUI* (viz UML diagram tříd v příloze A.1), který je od aplikace zcela oddělitelný (lze jej smazat a aplikace bude po smazání jedné řádky, kterou se okno GUI vytváří, nadále spustitelná v dávkovém módu). Okno GUI lze vidět v příloze na obr. A.2.

Základními kameny GUI jsou třídy *MainPanel*, *NodeGui*, *EdgeGui*, *NodeGuiListener* a *MainPanelListener*. Ostatní třídy menšího významu (triviální funkčnost jako je např. vytvoření okna, přidání položek do menu atd.) není třeba zmiňovat jmenovitě.

[3]Pro představu: starý úryvek kódu v příloze C.4.1 zabere 20-krát více času, než optimalizovaný kód (příloha C.4.2) se stejným výsledkem.

- *Mainpanel* – Reprezentuje plátno na kterém je vytvářen (a vykreslován) SIP.
- *NodeGui* & *EdgeGui* – Tyto třídy reprezentují predikát a hranu na plátně. Rozšiřují původní třídy *Node* a *Edge* z balíku *Representation*[4] o informace, potřebné ke správnému vykreslení (pozice uzlů, u hran počet a pozice zlomů atd.).
- *NodeGuiListener* & *MainPanelListener* – Nejdůležitější třídy GUI – detekce a vyhodnocení akcí myši na plátně. Zajišťují posuny objektů po plátně, úpravy hran a výběr zdrojového predikátu do další iterace konstrukce SIPu.

V příloze a na obr. A.3 lze vidět rozpracovaný SIP s pozicemi uzlů a hran, jak je generuje aplikace automaticky. Na automatické rozmisťování nebyl kladen důraz (kromě jediného požadavku – vykreslené hrany musí být okem rozlišitelné od ostatních), neboť uživatel může snadno kdykoliv změnit pozice hran i uzlů a navíc hranám přidávat a ubírat zlomy, se kterými lze také libovolně pohybovat po plátně. Hotový a poté upravený graf lze vidět na obr. A.4.

[4]Nejde o dědění, třídy *NodeGui* a *EdgeGui* obsahují *Node* a *Edge* jako proměnnou.

7 Další využitelnost

V této kapitole bude uvedeno a popsáno, jak a které metody lze dobře opětovně využít, resp. celé vyjmout z kódu a přenést do jiného programu. Jejich vstupem a/nebo výstupem ovšem často jsou instance tříd z balíku *Representation*, které by případně musel být přenášeny také. To samé platí o třídě *Constants*, která uchovává důležité konstanty (regulární výrazy pro parsování pravidla atd.), využívané ostatními třídami.

Metody budou uváděny postupně v takovém pořadí, jaké vede od vstupu k výstupu. Popisovány pochopitelně nebudou metody, provozující nějakou triviální činnost, jako například otevření souboru pro čtení (včetně ošetření výjimek). Veškeré metody, náležící balíku *GUI*, budou vynechány úplně.

Nadpis podkapitoly třetí úrovně (např. **7.1.1 InputFileReader**) určuje třídu, nad jejíž instancí (nebo staticky) jsou metody v příslušné podkapitole volány. Nadpisy podkapitol druhé úrovně odpovídají balíkům, ve kterých se tyto třídy nacházejí.

7.1 Input

7.1.1 InputFileReader

public ArrayList<String> readFile(Scanner sc)

Metoda přečte soubor, kterému náleží zaslaný *Scanner*, a vrátí seznam řetězců znaků, kde každý řetězec odpovídá jednomu pravidlu načtenému ze souboru.

7.1.2 RuleParser

public Rule parseRule(String ruleStr)

Tato metoda naparsuje řetězec znaků *ruleStr*, odpovídající jednomu pravidlu, načtenému ze souboru, a vrátí vytvořenou instanci třídy *Rule*. Využívá metodu (ta tedy není volána odjinud, než z této metody) *parseNode*.

private Node parseNode(String nodeStr, int order)

Metoda vrací instanci třídy *Node*, vytvořenou z naparsovaného řetězce znaku *nodeStr*. Parametr *order* je pořadové číslo predikátu, který tato instance třídy *Node* zastupuje, ve vstupním pravidle.

7.2 Core

7.2.1 CoreController

public void createGraphs(Rule[] rules)

Tato metoda pro každé pravidlo ze seznamu vytvoří graf (instance třídy *Graph*) pro každý řetězec ozdobení, získaný metodou *createAdornmentVariations*.

public static void createAdornmentVariations(ArrayList<String> adornments, int requestedLength, String previous)

Rekurzivně vytváří veškeré možné řetězce ozdobení zadané délky (*requestedLength*). První parametr je seznam, do kterého budou vyhovující řetězce ozdobení přidávány a třetí parametr je řetězec znaků, vytvořený v předchozí iteraci této metody (při prvním volání je tedy tento řetězec prázdný).

7.2.2 SIP

public SIP(Graph g)

Konstruktor třídy *SIP* inicializuje seznamy vázaných a použitých proměnných a seznam uzlů, ze kterých bude vybírán uzel, ze kterého povedou hrany v další iteraci algoritmu generování SIPu.

public void createEdgesOneStep(Node srcNode, Edge prevEdge, boolean gui)

Tato metoda provede jednu iteraci přidání hran do grafu. Je rekurzivní, takže za předpokladu že parametr *gui* je **false** ji stačí zavolat pouze jednou tak, že *srcNode* bude predikát hlavy a *prevEdge* **null**. Je-li parametr *gui* **true**, zastaví se, kdykoliv je více možností, jaký uzel zvolit jako zdrojový do další iterace (tzn. velikost seznamu, který je uchovává, je větší než 1), a musí být volána znovu, jakmile bude uživatelem zdrojový uzel vybrán.

private Node chooseNextNode()

Vybere (a odstraní) ze seznamu uzel, ze kterého povedou hrany v další iteraci. Pokud žádný takový není (tzn. seznam je prázdný), vrací **null**.

7.3 Output

7.3.1 AllOutput

Tato třída zajišťuje veškerý výstup (pro všechna pravidla) do textových souborů[1] interně po vytvoření instance konstruktorem bez volání jakýchkoliv dalších metod „zvenku“. Všechny metody mají tedy v hlavičce modifikátor přístupové úrovně private (kromě jedné statické, která je volána z GUI bez vytváření instance této třídy). Banální záležitosti, jako vytvoření složky, kam budou soubory zapisovány, nebo vytvoření časové známky, opět nebudou zmiňovány.

private void writeOneRule(FileWriter fw, Rule r, Graph g, boolean gui)

Metoda zapíše do souboru, na který ukazuje zaslaný *FileWriter*, původní načtené pravidlo, pravidlo upravené podle příslušného řetězce ozdobení, hrany SIPu, náležící grafu *g*, a po transormaci SIPu na FSIP také hrany FSIPu.

[1] Včetně vytváření potřebných složek.

private void transformSIPtoFSIP(Graph g)

Pro graf *g* transformuje tato metoda SIP na FSIP. Původní SIP není smazán, instance třídy *Graph* uchovává seznam hran SIPu i FSIPu.

private String writeEdges(Graph g, boolean FSIP)

Tato metoda vrátí textový řetězec, který obsahuje (ve správném formátu) hrany grafu *g*. Jestli se jedná o hrany SIPu nebo FSIPu určuje **boolean** *FSIP*.

8 Testování

Aplikace byla testována především v dávkovém módu podle dvou kritérií – správnosti generovaných SIPů a časové náročnosti (viz kap. 8.1). Co se týče správnosti, nebyly v konečné verzi aplikace nalezeny žádné nedostatky; grafy předávání informace, poskytované aplikací, se zcela shodují s ručně vytvářenými grafy[1].

V módu s GUI byla kromě správnosti testována především robustnost (a nalezené chyby odstraněny) – tzn. aplikace nespadne při špatném vstupu (např. vstupní soubor není ve správném formátu, nebo uživatel omylem (či schválně) klikne na něco, co se od něj neočekává), ale varuje uživatele, že něco provedl špatně.

Pozornost byla věnována také intuitivnosti ovládání GUI, v první řadě snadnému přemisťování uzlů a hran na plátně.

8.1 Časová náročnost

Doby běhu aplikace byly testovány na notebooku s OS Windows 7, čtyřjádrovým procesorem s taktem 1.9 GHz a RAM 8GB pro logický program z přílohy B (19 pravidel – 18 se 4 proměnnými v hlavičce, 1 pravidlo se 2 proměnnými) a soubor, ve kterém je tento program 20× zkopírován (tzn. 380 pravidel). Počet vygenerovaných grafů předávání informace pro tyto vstupní soubory činí 273, respektive 5460.

V tabulce 8.1 se nachází průměr (zaokrouhleno na celé milisekundy) z 20ti spuštění před optimalizací formátování výstupu a zápisu do souboru, v tabulce 8.2 po optimalizaci.

Počet pravidel	Příslušný počet SIPů	Doba běhu celkově [ms]	Doba běhu na jedno pravidlo [ms]
19	273	2066	109
380	5640	31095	82

Obrázek 8.1: Doba běhu před optimalizací.

[1]S výstupem aplikace bylo porovnáno nejméně 20 ručně vytvořených grafů.

Počet pravidel	Příslušný počet SIPů	Doba běhu celkově [ms]	Doba běhu na jedno pravidlo [ms]
19	273	989	52
380	5640	14831	39

Obrázek 8.2: Doba běhu po optimalizaci.

Z tabulek lze vidět, že optimalizací klesl čas, potřebný na zpracování jednoho pravidla, přibližně na polovinu původní hodnoty. Tento výsledný čas je myslím uspokojivý – na zpracování logického programu, obsahujícího několik desítek pravidel, není třeba čekat více než několik sekund.

9 Závěr

Hlavním cílem práce bylo jednak navrhnout formát, v jakém budou grafy předávání informace uloženy v textových souborech, ale především implementovat algoritmus konstrukce grafu předávání informace. Oba cíle se podařilo splnit.

Byla vytvořena aplikace v jazyce Java, které implementuje algoritmus konstrukce grafu předávání informace. Dále byla zavedena drobná úprava algoritmu konstrukce grafu předávání informace, díky které je konstrukce deterministická.

Aplikace běží ve dvou módech – dávkovém a s grafickým uživatelským rozhraním. V dávkovém módu, kde se chová podle zavedené úpravy algoritmu, aplikace vygeneruje pro každé pravidlo logického programu příslušné grafy předávání informace a uloží je do textových souborů v navrženém formátu. V módu s grafickým rozhraním se zavedená úprava ignoruje a konstrukci grafu předávání informace lze uživatelsky řídit.

V budoucnu by mohla být implementována celá metoda magických množin, proto byla aplikace navržena a implementována tak, aby byla bez větších úprav rozšiřitelná o další části metody magických množin, nebo se případně daly jinde použít některé její metody a třídy.

Použité zkratky

- **FSIP** – finální graf předávání informace (angl. final sideways information passing)
- **GUI** – grafické uživatelské rozhraní (angl. graphical user interface)
- **PNG** – Portable Network Graphics
- **SIP** – graf předávání informace (angl. sideways information passing)

Literatura

[1] ZÍMA, Martin. *Experimentální deduktivní databázový systém s neurčitostí.* Plzeň, 2002. Disertační práce. Západočeská univerzita v Plzni.

[2] BEERI, Catriel a Raghu RAMAKRISHNAN. On the power of magic. *The Journal of Logic Programming* [online]. 1991, vol. 10, 3-4, s. 255-299 [cit. 2015-04-16]. DOI: 10.1016/0743-1066(91)90038-Q. Dostupné z: http://linkinghub.elsevier.com/retrieve/pii/074310669190038Q

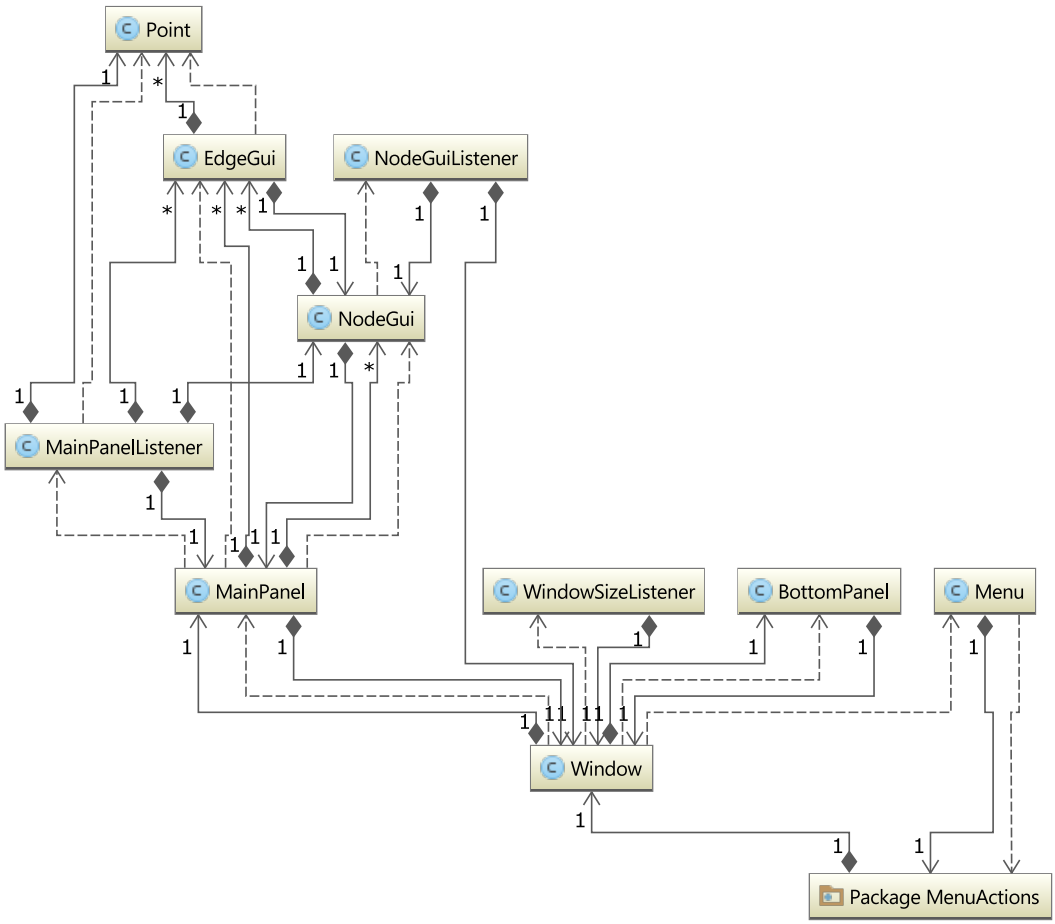
[3] BANCILHON, Francois, David MAIER, Yehoshua SAGIV a Jeffrey D ULLMAN. Magic sets and other strange ways to implement logic programs. *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems* [online]. New York, New York, USA: ACM Press, 1986, s. 1-15 [cit. 2015-04-17]. DOI: 10.1145/6012.15399. Dostupné z: http://portal.acm.org/citation.cfm?doid=6012.15399

[4] Prolog (programovací jazyk). In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2015-04-21]. Dostupné z: cs.wikipedia.org/wiki/Prolog_(programovací_jazyk)

[5] ČADA, Roman, Tomáš KAISER a Zdeněk RYJÁČEK. *Diskrétní matematika.* 1. vyd. Plzeň: Západočeská univerzita v Plzni, 2004, s. 129 [cit. 2015-04-22]. ISBN 80-7082-939-7. Dostupné z: http://www.cam.zcu.cz/řyjacek/students/DMA/skripta/12.pdf

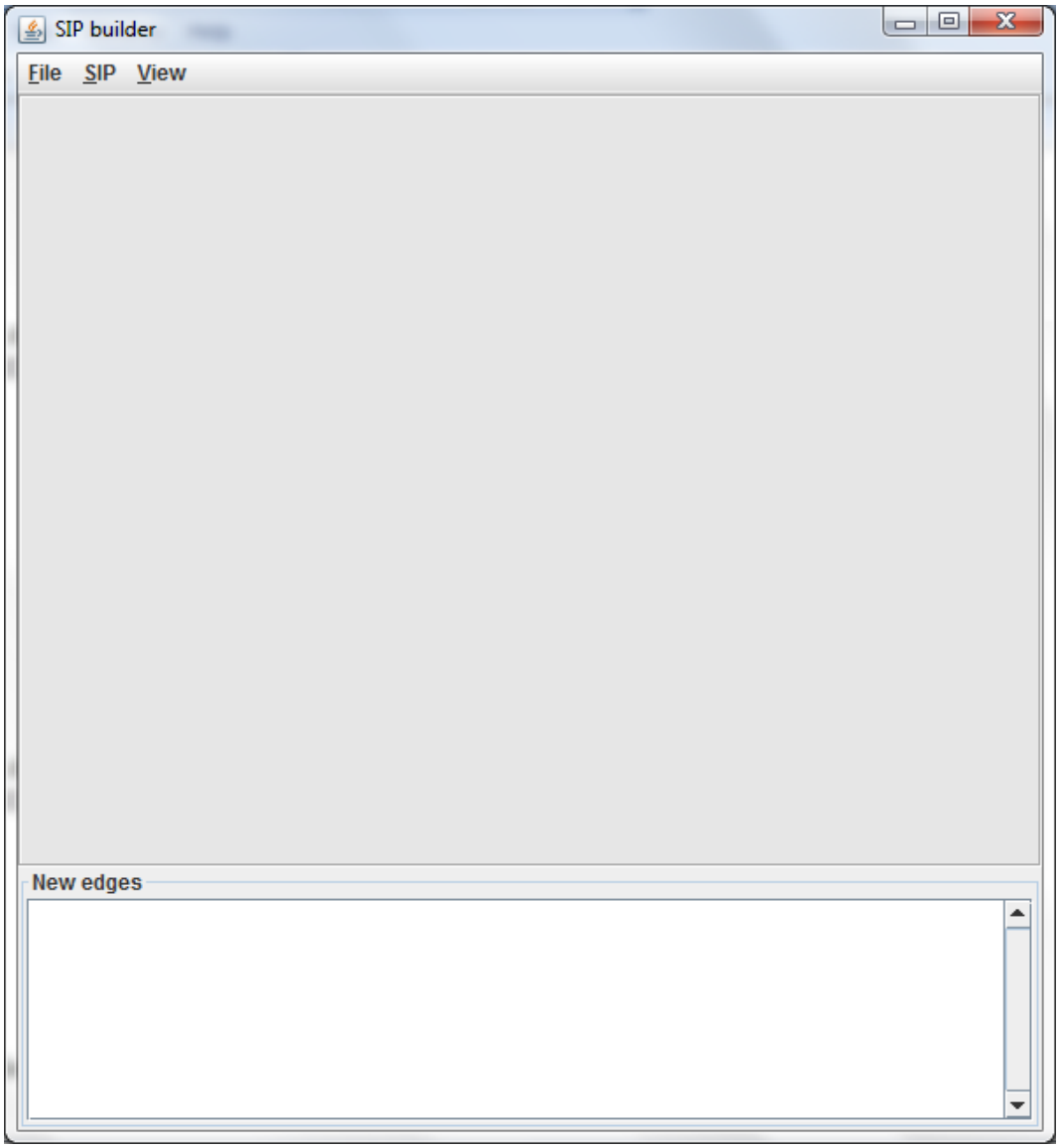
A Obrázky

A.1 UML diagramy



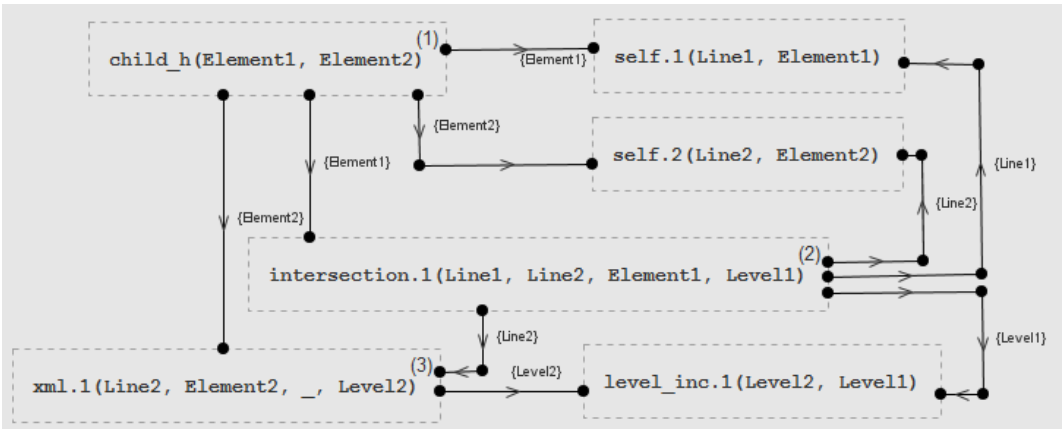
Obrázek A.1: UML diagram tříd balíku GUI.

A.2 GUI



Obrázek A.2: GUI aplikace po spuštění.

Obrázek A.3: Rozpracovaný SIP s automatický generovanými pozicemi objektů na plátně.



Obrázek A.4: Hotový SIP s upravenými pozicemi a hranami.

B Testovací program

```
intersection(Line1, Line2, Element, 1) :-
  xml(Line1, Element, Order, 1),
  xml(Line2, Element, Order, 1),
  lt(Line1, Line2).

intersection(Line1, Line2, Element, Level1) :-
  xml(Line1, Element, Order, Level1),
  xml(Line2, Element, Order, Level1),
  dec(Level2, Level1),
  intersection(Line1, Line2, _, Level2).

self(Line, Element) :-
  xml(Line, Element, _, Level1),
  inc(Level2, Level1),
  not (xml(Line, _, _, Level2)).

child(Line1, Line2, Element1, Element2) :-
  self(Line1, Element1),
  self(Line2, Element2),
  intersection(Line1, Line2, Element1, Level1),
  inc(Level2, Level1),
  xml(Line2, Element2, _, Level2).

parent(Line1, Line2, Element1, Element2) :-
  self(Line1, Element1),
  self(Line2, Element2),
  intersection(Line2, Line1, Element2, Level1),
  inc(Level2, Level1),
  xml(Line1, Element1, _, Level2).

descendant(Line1, Line2, Element1, Element2) :-
  child(Line1, Line2, Element1, Element2).

descendant(Line1, Line3, Element1, Element3) :-
  child(Line1, Line2, Element1, Element2),
  descendant(Line2, Line3, Element2, Element3).

ancestor(Line1, Line2, Element1, Element2) :-
```

```
  parent(Line1, Line2, Element1, Element2).

ancestor(Line1, Line3, Element1, Element3) :-
  parent(Line1, Line2, Element1, Element2),
  ancestor(Line2, Line3, Element2, Element3).

descendant_or_self(Line, Line, Element, Element) :-
  self(Line, Element).

descendant_or_self(Line1, Line2, Element1, Element2) :-
  descendant(Line1, Line2, Element1, Element2).

ancestor_or_self(Line, Line, Element, Element) :-
  self(Line, Element).

ancestor_or_self(Line1, Line2, Element1, Element2) :-
  ancestor(Line1, Line2, Element1, Element2).

following_sibling(Line1, Line2, Element1, Element2) :-
  self(Line1, Element1),
  xml(Line1, Element1, Order1, Level),
  self(Line2, Element2),
  xml(Line2, Element2, Order2, Level),
  gt(Order2, Order1),
  parent(Line1, Line3, Element1, Element3),
  parent(Line2, Line3, Element2, Element3).

preceding_sibling(Line1, Line2, Element1, Element2) :-
  self(Line1, Element1),
  xml(Line1, Element1, Order1, Level),
  self(Line2, Element2),
  xml(Line2, Element2, Order2, Level),
  lt(Order2, Order1),
  parent(Line1, Line3, Element1, Element3),
  parent(Line2, Line3, Element2, Element3).

following(Line1, Line3, Element1, Element3) :-
  following_sibling(Line1, Line2, Element1, Element2),
  descendant_or_self(Line2, Line3, Element2, Element3).

following(Line1, Line4, Element1, Element4) :-
```

```
    ancestor(Line1, Line2, Element1, Element2),
    following_sibling(Line2, Line3, Element2, Element3),
    descendant_or_self(Line3, Line4, Element3, Element4).

preceding(Line1, Line3, Element1, Element3) :-
    preceding_sibling(Line1, Line2, Element1, Element2),
    descendant_or_self(Line2, Line3, Element2, Element3).

preceding(Line1, Line4, Element1, Element4) :-
    ancestor(Line1, Line2, Element1, Element2),
    preceding_sibling(Line2, Line3, Element2, Element3),
    descendant_or_self(Line3, Line4, Element3, Element4).
```

C Zdrojové kódy

C.1 Tvorba hran grafu

```
//SIP step 4 - finding reachable nodes & creating
    corresponding edges
public void createEdgesOneStep(Node srcNode, Edge
    prevEdge, boolean gui) {

  this.lastNode = srcNode;
  ArrayList<Variable> edgeVars;
  Node chosenNextNode;

  //let's try to reach every body node
  for(Node bodyNode : this.graph.getBodyNodes()) {

      //possible target node is source node
      if(srcNode.equals(bodyNode))
          continue;

      edgeVars = new ArrayList<Variable>();

      //check every target node's variable
      for(Variable bodyVar : bodyNode.getVariables()) {

          //SIP step 4
          if(bodyVar.isValid() //variable bodyVar is
              valid (= not constant or anonymous variable)
                  && srcNode.getVariables().contains(
                      bodyVar) //srcNode's variables
                      contain bodyVar
                  && this.boundVariables.contains(bodyVar)
                       //bound variables contain bodyVar
                  && !this.usedVariables.contains(bodyVar)
                      ) //used variables do NOT contain
                      bodyVar, this prevents using the same
                       variable again
          {
```

```
                edgeVars.add(bodyVar);

                //add reached node to list (if it isn't
                    already there)
                if(!this.possibleNextSourceNodes.contains(
                    bodyNode))
                    this.possibleNextSourceNodes.add(
                        bodyNode);
            }
        }

        //does this node have any variables that can be
            used for passing information further?
        if(!edgeVars.isEmpty()) {
            //it indeed does

            //SIP step 5 (B) - add variables from reached
                new node to bound variables
            for(Variable v : bodyNode.getVariables())
                if(!Variable.isAnonymous(v) && !this.
                    boundVariables.contains(v)) {
                    this.boundVariables.add(v);
                }

            //create new edge and add it graph
            Edge newEdge = new Edge(srcNode, bodyNode,
                prevEdge, edgeVars);
            this.graph.addEdgeSIP(newEdge);
            bodyNode.setInputEdge(newEdge);
        }
    }

    //SIP step 5 (U) - add variables from source node to
        used varibles
    for(Variable v : srcNode.getVariables())
        if (!Variable.isAnonymous(v) && !this.usedVariables
            .contains(v)) {
            this.usedVariables.add(v);
        }

    //SIP step 6 - graph constructing continues, if all
```

```
        variables haven't been used yet
    if(!isDoneFoulproof() &&
            // but we also don't want to continue, if app is
                run with gui and there is more than one
               possible node to go to in the next iteration
            (!gui || possibleNextSourceNodes.size() <= 1))
    {
        //SIP step 7
        if((chosenNextNode = this.choseNextNode()) == null)
            return;

        //SIP step 4 (again)
        this.createEdgesOneStep(chosenNextNode,
            chosenNextNode.getInputEdge(), gui);
    }
}
```

C.2 Výběr uzlu

```
private Node choseNextNode() {

  int choseIndex, order;
  boolean isChosenOK = false;
  Node chosen = null;

  //until suitable node is found
  while(!isChosenOK) {

      choseIndex = 0;
      order = Integer.MAX_VALUE;

      //let's find which reached node has the lowest
         order in rule
      for(int i = 0; i < possibleNextSourceNodes.size();
         i++) {
        if(possibleNextSourceNodes.get(i).
           getOrderInRule() < order) {
          order = possibleNextSourceNodes.get(i).
             getOrderInRule();
          choseIndex = i;
        }
      }

      if(possibleNextSourceNodes.size() < 1)
        return null;

      chosen = possibleNextSourceNodes.remove(choseIndex)
         ;
      //at least one of chosen node's variables mustn't
         be in used variables
      //if none is found, then we try to choose another
         reached node
      isChosenOK = checkNode(chosen);

  }

  return chosen;
}
```

C.3 Vytváření výstupního souboru

```
    private void writeOneRule(FileWriter fw, Rule r,
       Graph g, boolean gui) throws IOException {

          outputBuffer = new StringBuilder();
          final String ls = LINE_SEPARATOR +
             LINE_SEPARATOR;
          final String lss = ls + LINE_SEPARATOR;

          outputBuffer.append(Language.getValue("
             original_rule", "Original␣rule:"))
                   .append(ls)
                   .append(r.toString())
                   .append(lss)
                   .append(Language.getValue("modified_rule
                      ", "Modified␣rule:"))
                   .append("␣(")
                   .append(g.getAdornment())
                   .append(")")
                   .append(ls)
                   .append(g.toString()).append(lss)
                   .append(Language.getValue("file_sip", "
                      SIP:␣"))
                   .append(ls);
          outputBuffer.append(writeEdges(g, false));
          outputBuffer.append(ls)
                   .append(Language.getValue("file_fsip", "
                      FSIP:␣"))
                   .append(ls);
          if(!gui) {
              transformSIPtoFSIP(g);
              outputBuffer.append(writeEdges(g, true));
          }
          else
              outputBuffer.append(Language.getValue("
                 no_fsip", "(not␣created)"));

          fw.write(outputBuffer.toString());
          fw.close();
}
```

C.4 Ukázka optimalizace kódu

C.4.1 Původní kód

```
outStr += String.format("{%s} %s — %s %s —> %s%s",
          visitedNodes.get(i), separatorN,
          variables.get(i),separatorV,
          FSIP ? g.getEdgesFSIP().get(i).getDstNode().
              getNodeName() :
               g.getEdgesSIP().get(i).getDstNode().
                   getNodeName(), LINE_SEPARATOR);
```

C.4.2 Optimalizovaný kód

```
outputBuffer.append("{")
          .append(visitedNodes.get(i))
          .append("} ")
          .append(separatorN)
          .append("— ")
          .append(variables.get(i))
          .append(" ")
          .append(separatorV)
          .append("—> ")
          .append((FSIP ? g.getEdgesFSIP().get(i).
              getDstNode().getNodeName() :
               g.getEdgesSIP().get(i).getDstNode().
                   getNodeName()))
          .append(LINE_SEPARATOR);
```

D Ukázka výstupního souboru

```
Original rule:

intersection(Line1, Line2, Element, Level1) :-
  xml(Line1, Element, Order, Level1),
  xml(Line2, Element, Order, Level1),
  dec(Level2, Level1),
  intersection(Line1, Line2, _, Level2).


Modified rule: (fbfb)

intersection_h(Line2, Level1) :-
  xml.1(Line1, Element, Order, Level1),
  xml.2(Line2, Element, Order, Level1),
  dec.1(Level2, Level1),
  intersection.1(Line1, Line2, _, Level2).


SIP:

{intersection_h} --------- Level1 ----------> xml.1
{intersection_h} --------- Line2, Level1 ---> xml.2
{intersection_h} --------- Level1 ----------> dec.1
{intersection_h} --------- Line2 -----------> intersection.1
{intersection_h; xml.1} -- Element, Order --> xml.2
{intersection_h; xml.1} -- Line1 -----------> intersection.1
{intersection_h; dec.1} -- Level2 ----------> intersection.1


FSIP:

{intersection_h} --------- Line2 ---> intersection.1
{intersection_h; xml.1} -- Line1 ---> intersection.1
{intersection_h; dec.1} -- Level2 --> intersection.1
```

E Uživatelská příručka

E.1 Překlad

Pro překlad je nutné mít nainstalovanou Javu verze 1.7 (nebo novější) a nástroj Apache Ant.

Překlad aplikace se provádí pomocí příkazu `ant build.xml` v příkazové řádce. Soubor `build.xml` musí být ve stejném adresáři, kde se nachází adresář `src` a soubory s definovanými lokalizacemi (soubory s příponou properties[1]).

Výsledkem překladu je jediný spustitelný Java archiv `SIPBuilder.jar`.

E.2 Spuštění

Přeloženou aplikaci je nutné spustit z příkazové řádky, neboť vyžaduje parametry, které by nebylo možné zadat při spuštění rovnou z průzkumníků souborů.

Aplikace se spouští takto formátovaným příkazem (bez uvozovek):

```
java -jar SIPBuilder.jar '-par1=val1 -par2=val2 ...'
```

kde `par` je název parametru a `val` jeho hodnota. Jednotlivé dvojice `par=val` (parametr a hodnota) jsou odděleny mezerou a v názvu parametru, hodnotě parametru a okolo znaků = (rovnítko) a - (pomlčka) nesmí být mezery. Možné parametry lze vidět v tabulce E.1 na straně 58.

Parametr **lang** určuje jazyk aplikace a není povinný. Parametr **file** je povinný, pokud hodnota parametru **gui** je **no**. Pokud ovšem parametr **file** zadán nebyl, musí být zadán parametr **gui=yes**. Jiné názvy parametrů, než které jsou uvedené v tabulce, aplikace ignoruje.

[1] Název těchto souboru se v žádném případě nesmí změnit, jinak z nich aplikace nebude schopna načíst jazyk.

Název parametru	**Povolené hodnoty (oddělené čárkou)**	**Výchozí hodnota (nebyl-li parametr zadán)**
gui	yes, no	no
lang	en, cs	en
file	*cesta k souboru*	–

Obrázek E.1: Parametry spuštění aplikace.

Příklady spuštění (poslední příklad je sice dobře, ale parametr **abc** aplikace nezná a proto jej jednoduše ignoruje):

```
java -jar SIPBuilder.jar -gui=no -file=myProgram.pro

java -jar SIPBuilder.jar -gui=yes -lang=cs

java -jar SIPBuilder.jar -gui=yes -abc=cde
```

Pokud budou parametry zadány špatně (chybí povinný parametr, nějaký je zapsán ve špatném formátu, atd.), aplikace uživatele upozorní výpisem do příkazové řádky a ukončí se.

Příklady chybného zadání parametrů (v posledním příkladu je sice název parametru **lang** známý, ale parametr je ve špatném formátu):

```
java -jar SIPBuilder.jar -gui=no -lang=cs

java -jar SIPBuilder.jar -gui = yes

java -jar SIPBuilder.jar -file=myProgram.pro -lang
```

E.3 Ovládání

E.3.1 Dávkový mód

Aplikace v dávkovém módu nevyžaduje po spuštění již žádnou další interakci. Pouze do příkazové řádky vypíše, jak dlouho běžela, a do adresáře `output`[2]

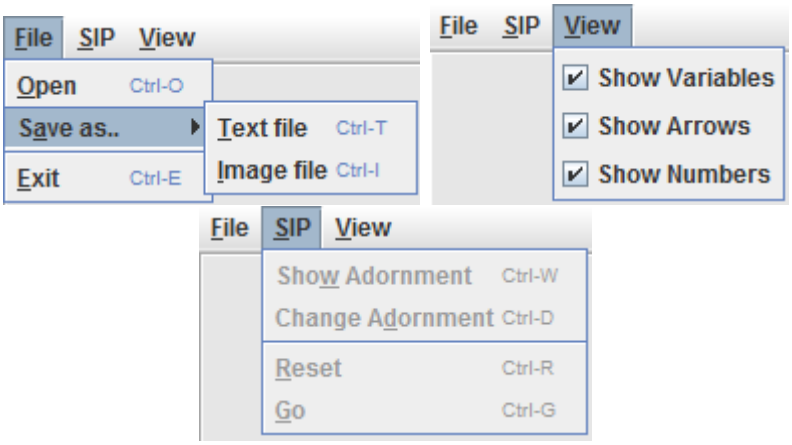
[2]Který vytvoří, pokud neexistuje.

přidá adresář, jehož název je složený z `out_` a časové značky[3]. V tomto adresáři jsou uložené vypočtené grafy.

E.3.2 GUI

Po spuštění s GUI (parametr **gui=yes**) se zobrazí okno z obrázku A.2. Má tři části:

- Textové okno ve spodní části, kam se vypisují doposud přidané hrany.
- Plátno - veliký panel s šedým pozadím v prostřední části, ve kterém se vytváří a upravuje SIP.
- Horní lišta s menu, které slouží k ovládání aplikace. Má tři podmenu – viz obr. E.2. Pro ovládání lze použít také klávesové zkratky uvedené na pravé straně položky podmenu.



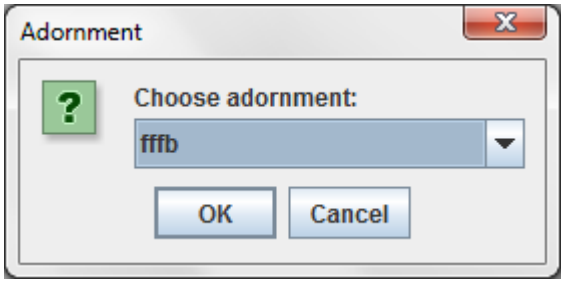
Obrázek E.2: Jednotlivá podmenu aplikace.

Nejprve je nutné načíst pravidlo ze souboru[4], což lze provést z nabídky **File** → **Open** – tato položka menu vyvolá standardní dialog pro výběr souboru. Po otevření souboru je uživatel dialogem (viz obr. E.3) vyzván k výběru řetězce ozdobení, pro který bude graf předávání informace vytvářen. Jakmile uživatel

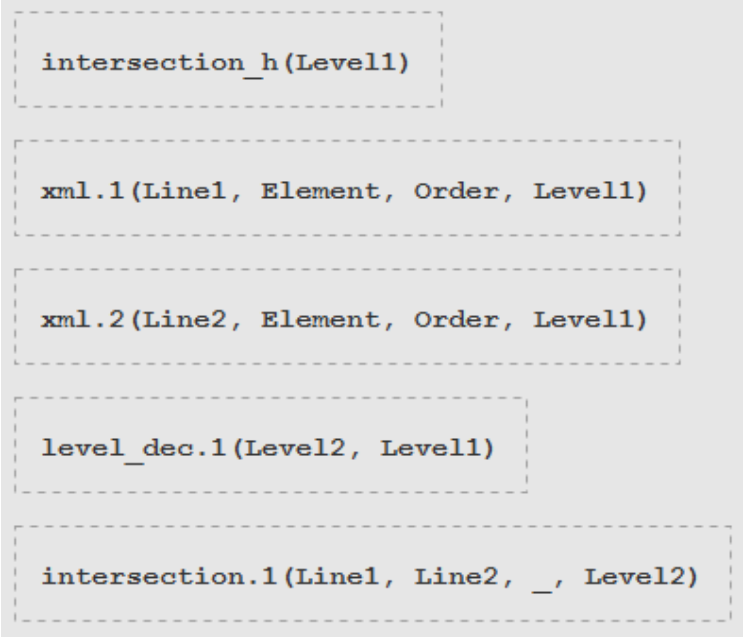
[3]Např. `out_2015-04-27_02-07-39`

[4]Pokud se ve vybraném souboru nachází více pravidel, je načteno pouze první a ostatní jsou ignorována.

řetězec ozdobení vybere, zobrazí se načtené pravidlo (se speciálním predikátem hlavy) na plátně - viz obr. E.4.



Obrázek E.3: Výběr řetězce ozdobení.



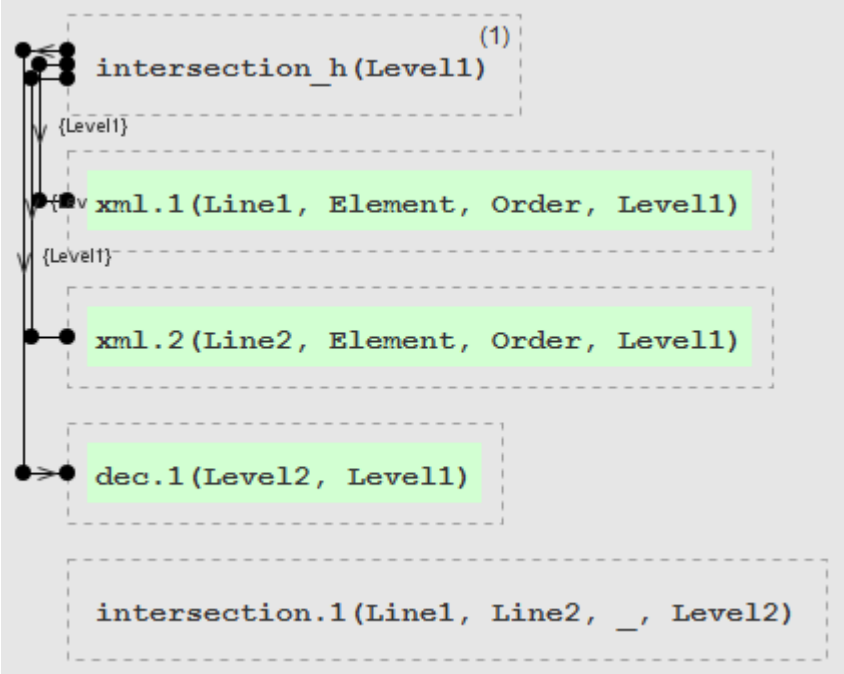
Obrázek E.4: Pravidlo zobrazené na plátně po výběru řetězce ozdobení.

Vybraný řetězec ozdobení lze kdykoliv změnit (což pochopitelně vyvolá i restartovaní konstrukce SIPu) položkou menu **SIP** → **Change Adornment** nebo zobrazit pomocí **SIP** → **Show Adornment**.

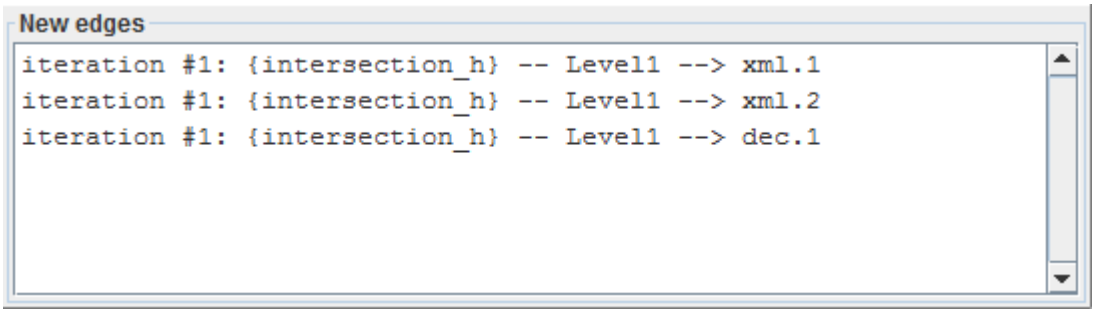
Konstrukce SIPu se spustí položkou menu **SIP** → **Go** a lze ji kdykoliv restartovat (se stejným řetězcem ozdobení) volbou **SIP** → **Reset**. Po spuštění provede aplikace jednu iteraci konstrukce SIPu[5] a poté nechá uživatele vybrat,

[5]Nebo více, pokud je možné do další iterace vybrat jako zdrojový pouze jediný predikát.

jakým predikátem chce pokračovat. Vybrat lze zeleně podbarvené predikáty (viz obr. E.5 nebo po provedení několika iterací obr. A.3) levým nebo pravým tlačítkem myši. Do spodního panelu jsou průběžně vypisovány nově vytvořené hrany (obr. E.6). Číslo u pravého horního rohu predikátu udává pořadí, v jakém byly predikáty vybírány jako zdrojové (ať už aplikací, nebo uživatelem). Text vedle hrany je její ohodnocení – zobrazuje se vždy mezi dvěma prostředními zlomy této hrany.



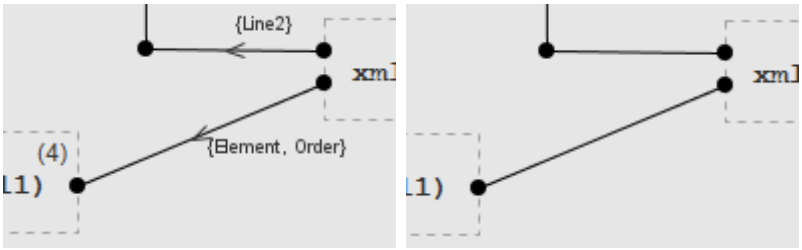
Obrázek E.5: Plátno po spuštění konstrukce SIPu.



Obrázek E.6: Výpis hran do textového okna spodního panelu.

Jakmile bude konstrukce grafu dokončena, je možné jej uložit do textového souboru (**File** → **Save as..** → **Text file**) ve stejném formátu, v jakém grafy ukládá dávkový mód. Volbou **File** → **Save as..** → **Image file** lze také uložit plátno jako obrázek ve formátu PNG. Pro obě tyto možnosti bude soubor uložen do složky `output_GUI` pod názvem složeným z predikátového symbolu speciálního predikátu hlavy, použitého řetězce ozdobení a časové značky[6].

Check boxy v podmenu **View** slouží ke změně zobrazení grafu na plátně. Položka menu **View** → **Show Variables** schová nebo zobrazí ohodnocení vykreslených hran, **View** → **Show Arrows** skryje nebo zobrazí šipky na hranách, udávající směr přenášení informace. Položkou **View** → **Show Numbers** lze skrýt pořadová čísla v pravých horních rozích predikátů. Na obr. E.7 vlevo lze vidět fragment plátna se všemi zobrazeními zapnutými a vpravo se všemi vypnutými.



Obrázek E.7: Změna zobrazení grafu na plátně.

Posledními ovládacími prvky aplikace jsou posuny objektů po plátně, posun plátnem a přidávání a odebírání zlomů hran (velké černé tečky na hranách). Tyto akce lze provádět kdykoliv, ale kvůli přehlednosti je doporučeno nejprve dokončit konstrukci SIPu.

Celým grafem, resp. plátnem, lze posunout pohybem myši za současného držení levého tlačítka myši a klávesy `Ctrl`. Predikátem nebo zlomem hrany se posunuje pohybem myši za současného držení levého tlačítka myši a klávesy `Shift`, přičemž krajním zlomem hrany lze pohybovat jen po hranici predikátu (přerušovaná čára), na které je tento zlom umístěn. Kliknutím pravým tlačítkem myši na zlom, resp. čáru hrany, lze zlomy rušit (neplatí pro krajní zlomy), resp. přidávat. Graf po přidání a odebrání některých zlomů a posunech zlomů a predikátů lze vidět na obr. A.4.

[6]Např. `child_fffb_2015-05-01_16-10-17.png`

F Obsah přiloženého CD

- adresář `aplikace`
  - adresář `javadoc` – vygenerovaná dokumentace aplikace
  - adresář `src` – zdrojové soubory aplikace
  - adresář `test` – obsahuje používaný testovací soubor (`program.pro`) a složku s příslušným výstupem dávkového módu
  - soubor `build.xml` – Ant skript sloužící k překladu
  - soubor `language_cs_CZ.properties` – česká lokalizace
  - soubor `language_en_US.properties` – anglická lokalizace
  - soubor `SIPBuilder.jar` – spustitelná aplikace
- adresář `text`
  - adresář `src` – zdrojové soubory textu
  - soubor `text.pdf` – text bakalářské práce
- soubor `README.txt` – popis adresářové struktury CD