VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

# KONVERZE MODELŮ REGULÁRNÍCH JAZYKŮ

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE David Navrkal
AUTHOR

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

# KONVERZE MODELŮ REGULÁRNÍCH JAZYKŮ

CONVERSION OF MODELS OF REGULAR LANGUAGES

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE David Navrkal
AUTHOR

VEDOUCÍ PRÁCE Ing. Zbyněk Křivka, Ph.D.
SUPERVISOR

BRNO 2014

# Abstrakt

Moje práce se zabývá, jak didakticky prezentovat studentům modely regulárních jazyků se zaměřením na jejich vzájemné konverze. Tyto konverze jsou převod regulárního výrazu na konečný automat (KA), odstranění epsilon pravidel z KA, determinizace KA. Cílem bylo toto implementovat aplikaci. V této práci dokumentuji vývoj této aplikace.

Tuto aplikaci jsem programoval v jazyku C++ za použití grafické knihovny Qt 5. Aplikaci jsem vyvíjel v operačním systému Linux a testoval v prostředí Microsoft Windows 7.

Všechny tyto konverze jsou implementovány. Aplikace je uživatelsky přívětivá, intuitivní a funkční. Aplikace nebyla testována na širším vzorku studentů.

Přínosem této práce je aplikace, která pomůže studentům formálních jazyků a automatů lépe pochopit a procvičit si tuto teorii na příkladech. Zdrojové kódy tohoto projektu (regularConvertor) jsou dostupné online na serveru Github.

# Abstract

My thesis deals with how to present students didactically models of regular languages focusing on their mutual conversions. These conversions are transformation of regular expression to finite automata (FA), remove epsilon rules from FA, determinization FA. Goal was implement this in application. In this thesis I documet development of this application.

This application was programmed in language C++ using graphic library Qt 5. Application was devepopped in operating systém (OS) Linux and was tested in OS Microsoft Windows 7.

All this conversions are implemented. Application is user-friendly, intuitive and functional. Application wasn't tested on wider sample of students.

Benefit of this work is application, which can help students of formal languages and finite automatas better understand this theory on examples. Source codes of this project (regularConvertor) are available online on server Github.



# Klíčová slova

Konečný automat, regulární výraz, regulární gramatika, konverze, regulární modely, formální jazyky, precedenční tabulka, syntaktická analýza, Qt5.

# Keywords

Finite automata, regular expression, regular grammar, conversions, regular models, formal languages, precedence table, syntax analysis, Qt5.

# Citace

Navrkal David: Konverze modelů regulárních jazyků, bakalářská práce, Brno, FIT VUT v Brně, 2014

# Konverze modelů regulárních jazyků

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Zbyňka Křivky, Ph.D.

Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
David Navrkal
21. 5. 2014

## Poděkování

Tímto bych chtěl poděkovat za ochotu, vřelost, lidskost, trpělivost a odborné vedení panu doktoru Křivkovi.

# Obsah

# 1 Úvod

Tato práce vznikla jako školní dílo a má pomoci studentům formálních jazyků s pochopením probírané látky a je ponejvíce průvodcem návrhu a implementace aplikace, která tyto konverze demonstruje. V této práci se budeme zabývat regulárními jazyky, jejich modely a především pak konverzemi těchto modelů. Tato aplikace se soustřeďuje na didaktickou část, takže nejenom že umí navzájem převádět modely regulárních jazyků, ale navíc se snaží pomoci uživateli pochopit, naučit a procvičit demonstrované algoritmy na konkrétních příkladech. Výsledná aplikace se snaží o co největší uživatelskou přívětivost a intuitivnost. Je určena jak pro hromadnou demonstraci probírané látky na přednáškách a cvičeních, tak i pro samostatné procvičování látky studenty.

V aplikaci je před připravené množství příkladů, které si uživatel může zkusit vyřešit, zároveň má uživatel možnost si vyzkoušet i příklady vlastní. Demonstrace převodních algoritmů na konkrétních příkladech je klíčová a umožňuje tak uživateli snáze pochopit teorii.

Aplikace umí jeden převod prezentovat třemi různými způsoby, tak že si uživatel může nejprve prohlédnout, jak výpočet probíhá krok po kroku. Tomuto způsobu reprezentace budeme říkat **Krokovací režim**. Poté co si uživatel prohlédl, jak výpočet probíhá na konkrétním příkladu, může si zkusit svůj první převod, tak že mu aplikace průběžně kontroluje, zdali postupuje správně. Tento režim budeme nazývat **Režim průběžné kontroly**. Následně pokud si uživatel myslí, že už převod umí, může si další převod zkusit samostatně a aplikace mu pouze zkontroluje výsledek (případě i řekne výsledek správný). Tomuto poslednímu režimu budeme říkat **Režim samostatné práce**.

Využití aplikace může být spousta, nejdůležitější případy užití jsou ve výuce přímo ve škole, a to na přednášce či cvičení, kde může vyučující využít Krokovací režim, kde může demonstrovat algoritmus na příkladech. Studenti tam mají možnost se v případě jakékoli nejasnosti ptát a vyučující jim může odpověď ukázat na nějakém příkladě. Zvídavý studenti mají totiž často na přednáškách a cvičeních dotazy na mezní chování algoritmů.

Po představení teorie a pár příkladů na přednášce následuje samostatné procvičování studenty. Učitel má tak možnost zadat studentům domácí úkoly. Student si může v Režimu průběžné kontroly vyzkoušet samostatně převod, v případě nejasností může využít i Krokovací režim, kde si může krokovat celý algoritmus, nebo se jen podívat na správné řešení, či si zkusit výpočet za asistence aplikace.

A na závěr, když se student připravuje na blížící se zkoušku, si může vyzkoušet vyřešit každý příklad, který ho napadne a to bez nutnosti účasti druhé osoby si nechat zkontrolovat výsledek.

V současné době existují programy, které umí převádět některé modely, neexistuje však žádný, který by uměl převádět navzájem všechny modely regulárních jazyků, zároveň je i didakticky demonstrovat a asistovat uživateli při výpočtu. Tato aplikace se o tyto cíle snaží.

Dále se chci zmínit o tom, že předpokládám, že čtenář zná základní pojmy týkající se formálních jazyků, především pak jazyků regulárních, pojmů, definic, modelů a algoritmů jejich vzájemných převodů a další věci s tím související. Přesto musím uvést alespoň základní definice, protože při studiu teorie jsem narazil na některé mírně odlišné definice a chci, aby tato práce byla srozumitelná i jiným, kteří nestudovali na naší fakultě. Méně důležité pojmy a definice jsem do hlavního těla práce neuváděl a můžete je nalézt pouze jako přílohy.

V této práci jsem se soustředil především na návrh a implementaci.

# 2 Teorie

V této části si popíšeme teorii týkající se modelů regulárních jazyků. V případě, že dané definice a konverzní algoritmy dobře znáte, můžete tuto část přeskočit a přečíst si konverzní algoritmy mezi těmito modely.

## 2.1 Konečný automat

Následující definici jsem převzal z [1].

**Definice:**

*Konečný automat (KA) je pětice:*

*$M = (Q, \Sigma, R, s, F)$, kde:*

- *$Q$ je konečná množina stavů*
- *$\Sigma$ je vstupní abeceda*
- *$R$ je konečná množina pravidel tvaru: $p\ a \rightarrow q$, kde $q, p \in Q$, $a \in \Sigma \cup \{\varepsilon\}$.*
- *$s \in Q$ je počáteční stav*
- *$F \subseteq Q$ je množina koncových stavů*

Tuto definici zde uvádím zejména proto, že jsem se setkal na internetu i v literatuře s odlišnými definicemi, které definovali například s ne jako počáteční stav, ale jako množinu počátečních stavů. Pro KA podle definice z [1] se dá nedeterministický výběr počátečního stavu simulovat ε-přechody z počátečního stavu do „simulovaných počátečních stavů“.

## 2.2 Regulární výrazy

Následující definici jsem převzal z [1].

**Definice:**

*Nechť $\Sigma$ je abeceda. Regulární výrazy nad abecedou $\Sigma$ a jazyky, které značí, jsou definovány následovně:*

- *$\emptyset$ je RV značící prázdnou množinu (prázdný jazyk)*
- *$\varepsilon$ je RV značící jazyk $\{\varepsilon\}$*
- *$a$, kde $a \in \Sigma$, je RV značící jazyk $\{a\}$*
- *Nechť $r$ a $s$ jsou regulární výrazy značící po řadě jazyky $L_r$ a $L_s$, potom:*

- *(r.s) je RV značící jazyk* $L = L_r L_s$
- *(r+s) je RV značící jazyk* $L = L_r \cup L_s$
- *(r*) je RV značící* $L = L_r^*$

V literatuře můžeme nalézt i definice $r^+$ značící *rr** nebo *r*r*. My tohle rozšíření nebudeme používat, protože vyjadřovací síla regulárních výrazů podle předešlé definice zůstává stejná. Pro zjednodušení můžeme používat *rs* na místo notace *r.s*.

Kvůli redukci počtu závorek budeme uvažovat následující priority: „* > . > +“ („*“ má větší prioritu než „.“ a to má větší prioritu než „+“).

## 2.3 Regulární gramatika

Pro jistotu si zde uvedeme i obecnou definici gramatiky a následně uvedeme i definici pravé lineární gramatiky a pravé regulární gramatiky.

**Definice gramatiky:**

*Gramatika G je čtveřice* $G = (N, \Sigma, P, S)$*, kde*

- *N je abeceda nonterminálních symbolů.*
- $\Sigma$ *je abeceda terminálních symbolů.*
- *P je konečná množina kartézského součinu* $(N \cup \Sigma)^*N.(N \cup \Sigma)^*\Sigma$*, nazývaná množinou přepisovacích pravidel.*
- $S \in N$ *je výchozí (počáteční) symbol gramatiky G.*

*Prvek* $(\alpha, \beta) \in P$ *je přepisovací pravidlo a zapisuje se ve tvaru* $\alpha \rightarrow \beta$*, kde* $\alpha$ *je levá strana,* $\beta$ *je pravá strana pravidla,* $\alpha \rightarrow \beta$*.*

Viz [2].

V předchozí části jsme si představili definici obecné gramatiky, nás však bude zajímat gramatika regulární, konkrétně nyní pak pravá lineární gramatika.

***Definice pravé lineární gramatiky:***

*Pravá lineární gramatika je taková, která má přepisovací pravidla ve tvaru:*

$A \rightarrow xB$*, nebo* $A \rightarrow x$ *kde* $A, B \in N$, $x \in \Sigma^*$ *(kde* $\Sigma^*$ *je množina řetězců nad abecedou* $\Sigma$*)*

Viz (2).

Nyní si nadefinujeme pojem pravá regulární gramatika. Jak následně uvidíte, jedná se o speciální typ

pravé lineární gramatiky kde *x* je řetězec délky jedna.

**Definice pravé regulární gramatiky:**

*Pravá regulární gramatika je taková, která má přepisovací pravidla ve tvaru:*

$A \rightarrow xB$, *nebo* $A \rightarrow x$ *kde* $A, B \in N$, $x \in \Sigma$

Viz (2).

## 2.4 Algoritmus normalizace deterministického KA

Tento algoritmus jsem čerpal z [3].

**Algoritmus:**

*Vstupní podmínky: deterministický konečný automat, úplné uspořádání abecedy KA* $\Sigma = \{a_1, a_2, ..., a_n\}$.

1. *Na počátku jsou všechny stavy neoznačené.*
2. *Počáteční stav označíme číslem 1.*
3. *Dokud nejsou všechny stavy označené, opakuje se tato činnost:*
4. *Vybere se označený nezpracovaný stav q s nejmenším číslem.*
5. *Postupně se pro* $j = 1, 2, \ldots, m$ *opakuje tato aktivita:*
6. *Jestliže stav* $q'$, *pro který platí* $q\ a_j \rightarrow q'$, *je neoznačený, potom se stav* $q'$ *označí dosud nepoužitým nejmenším číslem.*
7. *Stav q se prohlásí za zpracovaný.*
8. *Zůstaly-li na konci procesu neoznačené stavy, jsou tyto stavy nedosažitelné a dojde k jejich odstranění spolu s přechody, které z nich vedou.*

Tento algoritmus zde uvádím, protože se na naší fakultě nevyučuje a já jej používám při kontrolování uživatelských výsledků v konverzních algoritmech. Jelikož normalizuji minimální konečný automat, který je zároveň dobře specifikovaným, tak jsem poslední krok (s číslem 8) neimplementoval.

# 3 Specifikace požadavků

Následující požadavky jsem získal opakovanými debatami s vedoucím mé bakalářské práce a mými vlastními úvahami.

Aplikace se má soustředit na grafické uživatelské rozhraní. Uživatelské rozhraní má být pokud možno co nejvíce intuitivní a přehledné. Cílem je, aby se uživatel mohl soustředit na pochopení a následné procvičení demonstrovaných konverzí a ne před použitím aplikace číst dlouhé dokumentace. Aplikace má studentovi pomoci ve studiu, a ne mu brát jeho drahocenný čas.

Aplikace má sloužit jako výukový nástroj, který má pomoci studentům formálních jazyků a teoretické informatiky pochopit a naučit se používat konverzní algoritmy modelů regulárních jazyků.

Aplikace má být určená na konverzi malých objemů dat (řádově do desítek komponent regulárních modelů), hlavní důraz se klade na didaktickou demonstraci a samostatné procvičení převodních algoritmů.

S panem doktorem Křivkou jsme se nakonec dohodli na implementaci <u>následujících převodních algoritmů</u>:

- Regulární výraz na KA.
- Obecný konečný automat (KA) na KA bez ε-pravidel.
- KA bez ε-pravidel na deterministický KA bez nedostupných stavů (DKA).

Dále by pak aplikace měla umět následující <u>módy převodů</u>:

- **Editační režim**, ve kterém si uživatel vybere jednu z konverzí a vytvoří model (regulární výraz, KA), který chce převádět. Uživatel by měl mít možnost kterýkoli z následujících módů přerušit a vrátit se do tohoto módu.
- **Krokovací režim**, ve kterém program bez zásahu uživatele převede jeden model na druhý. V tomto režimu si může uživatel algoritmus krokovat a pozorovat, jak mají být správně provedeny jednotlivé kroky. Využití má především pro uživatele, kteří si chtějí prohlédnout postup výpočtu, popřípadě srovnávají svůj postup s postupem správným a hledají, kde udělali chybu.
- **Režim průběžné kontroly**, ve kterém uživatel převádí jeden model na druhý po jednotlivých krocích algoritmu a kontinuálně se mu kontroluje výsledek práce. V tomto módu si středně pokročilí uživatelé mohou zkoušet převádět regulární modely s průběžnou informací zda postupují v převodu správně.
- **Režim samostatné práce**, ve kterém uživatel samostatně převede jeden model na druhý a program mu pouze zkontroluje, zda převod provedl správně. Tento mód bude užitečný

hlavně uživatelům, kteří převodní algoritmus již pochopili a chtějí si zkontrolovat svou samostatnou práci.

Aplikace má být využitelná pro hromadnou demonstraci (tj. na přednášce, nebo na demo cvičení), z čehož vyplývá, že mají být vidět použité fonty i ze zadních řad. Zároveň by měla být použitelná na osobních počítačích pro samostatné procvičení přednášené látky studenty.

Primárně má aplikace běžet na operačním systému Microsoft Windows 7. Vítaná by byla možnost provozovat aplikaci i na operačních systémech Unixového typu, Linux apod., vzhledem k odbornému zaměření budoucích uživatelů, mezi kterými je vysoké procento těch, kteří právě tyto operační systémy používají.

Výsledný produkt má být hlavně určen pro použití na Fakultě informačních technologii VUT v Brně, proto použité názvosloví, definice a barevné značení by mělo odpovídat tomu, které se používá v předmětech Formální jazyky a překladače a Teoretická informatika vyučovaných na naší škole.

# 4 Návrh

V kapitole návrh jsem ze zaměřil především na to, co uvidí uživatelé, tedy na návrh grafického uživatelského rozhraní. K návrhu jsem přistupoval s papírem, tužkou a mými nápady. Ne všechny věci, které jsem si naplánoval byly v prostředí Qt 5 standardní a jejich implementace nejednou odporovala zvyklostem při použití tohoto aplikačního rozhraní, přesto jsem se snažil i přes tyto problémy většinu myšlenek, nápadů a konceptů z návrhu implementovat.

## 4.1 Návrh grafického uživatelského rozhraní

Při návrhu grafického uživatelského rozhraní (GUI) jsem vycházel z klasického rozložení a vzhledu ovládacích prvků. Mým cílem bylo vytvořit aplikaci s chováním a vzhledem, na které je uživatel zvyklý z obvyklých grafických aplikací z prostředí MS Windows a ne aplikaci s experimentální GUI. Aplikace má mít intuitivní grafické uživatelské rozhraní, aby uživatele nerozptylovala od učení se nových algoritmů. Pro usnadnění práce se po najetí myši na ovládací prvek zobrazí nápověda.

Následující obrázky jsem vytvořil pomocí programu Balsamic pro tvorbu návrhu uživatelských rozhraní, tzv. „mockupů“, proto se může vzhled výsledné aplikace mírně lišit.

### 4.1.1 Návrh zobrazení základních elementů

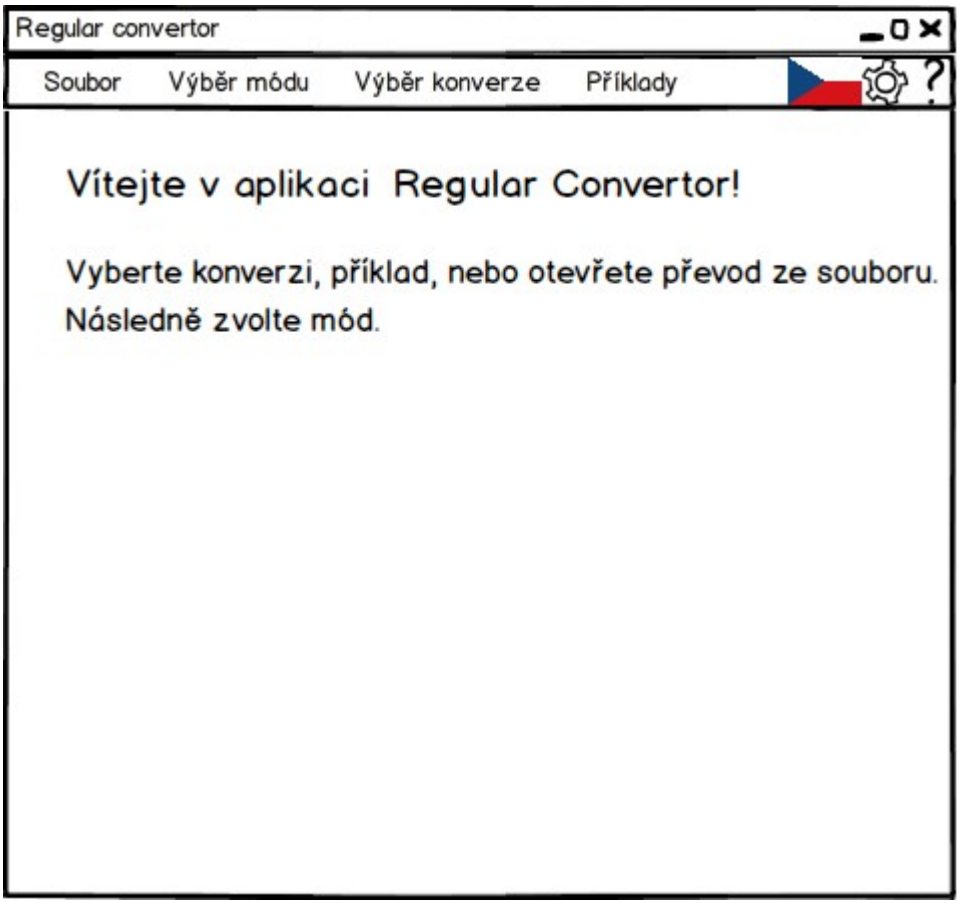
Při návrhu grafických elementů jsem postupoval od nejobecnějších ke specifičtějším.

#### 4.1.1.1 Hlavní okno

Ze všeho nejdříve se podíváme, jak bude aplikace vypadat, když ji uživatel poprvé spustí. Důležitý je totiž první dojem. Při spuštění programu, aplikace pro lepší první dojem přivítá uživatele a zároveň mu říká, jak může začít pracovat. Není totiž uživatelsky moc příjemné, když uživatel poprvé spustí program a neví jak má začít a to i přes to, že muže být aplikace jinak dobře navržená. V uživateli může zanechat dojem, že je nepřehledná.

První dojem je opravdu hodně důležitý. Jak říká známá poučka, „když uděláte špatný první dojem, musíte udělat deset dobrých věcí, abyste tento dojem zvrátili“, samozřejmě platí i naopak.

Nejen toto, ale i více o dobrém uživatelském rozhraní se můžete dočíst v [4].



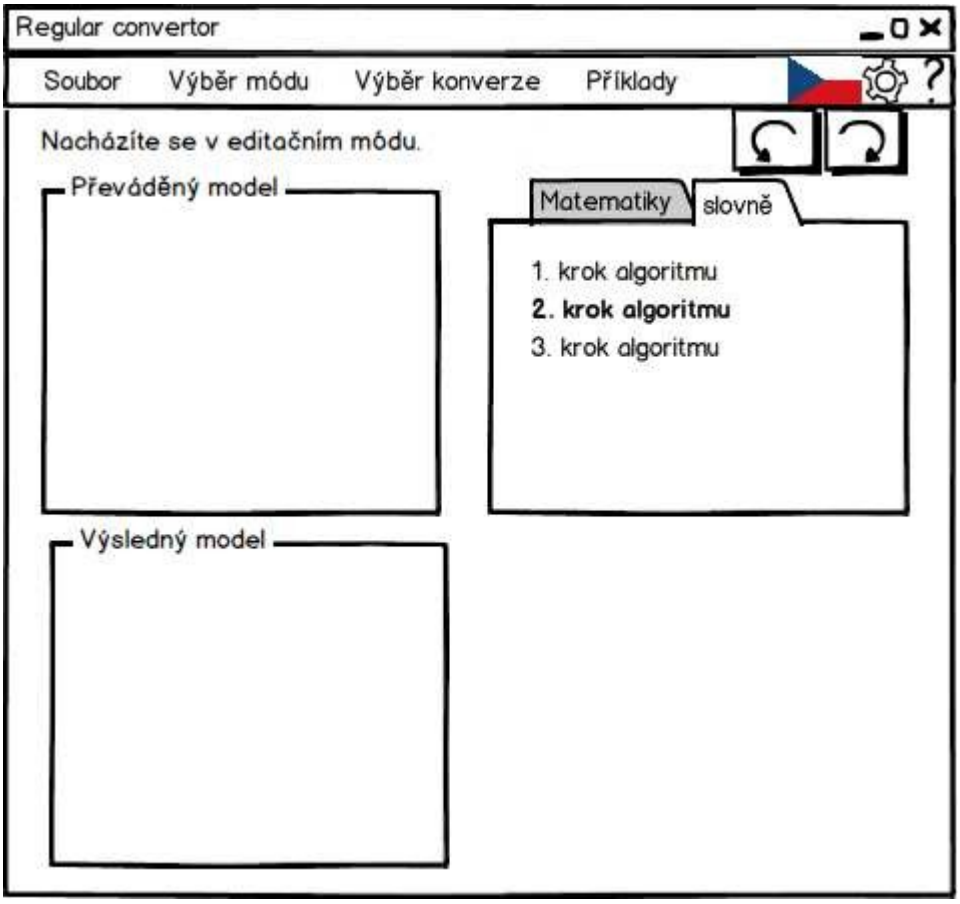

*Obrázek 1: Hlavní okno programu před výběrem některé z konverzí*

Na obrázku 1 v hlavním menu se popořadě nachází položky Soubor, pod kterou se nachází, standardní volby jako, uložení výchozího modelu pro převod do souboru a načtení modelu ze souboru. Dále pak **výběr módu** a **výběr konverze**, obojí viz kapitola Specifikace požadavků str. 8. Poslední položka menu obsahuje **příklady**, ve kterých najde uživatel předem připravené příklady modelů a jejich konverzí, které si může sám vyzkoušet. Napravo od příkladů uživatel uvidí vlajku ukazující **aktuální jazyk**, po kliku na ni se zobrazí výběr ostatních lokalizací. Napravo od ní se nachází ikony pro **nastavení** a práci s **nápovědou**. Text v okně informuje uživatele, aby si vybral konverzi, příklad, nebo načetl konverzi ze souboru a následně zvolil mód.

#### 4.1.1.2 Módy

1. V aplikaci se budou nacházet čtyři módy a to editační mód, krokovací mód, režim průběžné kontroly a režim samostatné práce. Nyní se pojďme podívat na návrh jednotlivých módů.

Začneme **Editačním režimem**. Na obrázku 2 vpravo nahoře můžeme vidět dvě tlačítka symbolizující předchozí a následující krok v editaci. Jsou výhodná, když byste si náhodou část modelu omylem smazali, tak se můžete vrátit v editaci zpět, anebo vpřed.

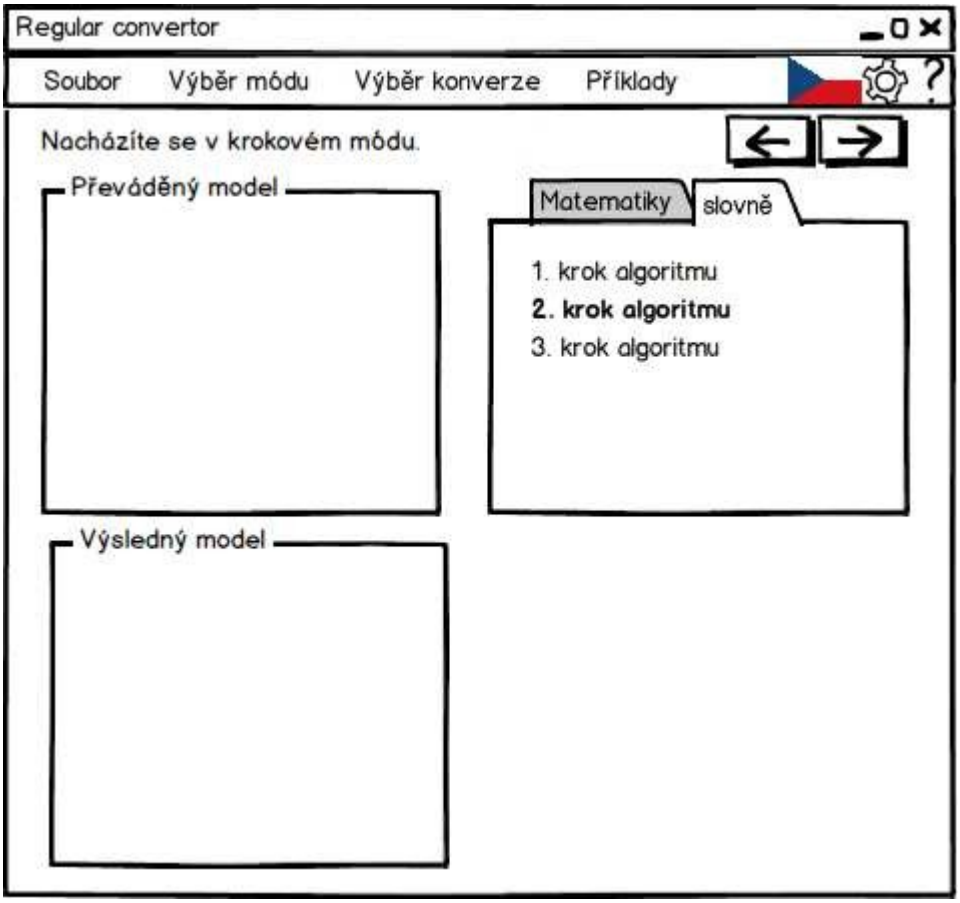



*Obrázek 2: Editační režim*

Pod těmito tlačítky můžete vidět okno s dvěma záložkami, v obou se nachází algoritmus převodu. Jediný rozdíl je, že v záložce pojmenované **matematicky** se nachází algoritmus v zapsaný ve formální podobě a druhé pojmenované **slovně** jsou instrukce ve formě vět.

Nalevo od tohoto okna je převáděný model, pod kterým se nachází okno na výsledný model. (Všechna následující okna převodů budou mít schéma rozložení takové, že vpravo nahoře je okno s algoritmem, nalevo od něj je převáděný model a v dolní části hlavního okna jsou okna potřebná pro konkrétní převod.)

Druhý v pořadí je **Režim průběžné kontroly**. Návrh rozložení hlavního okna pro tento režim




*Obrázek 3: Krokový režim*

je na obrázku 3.

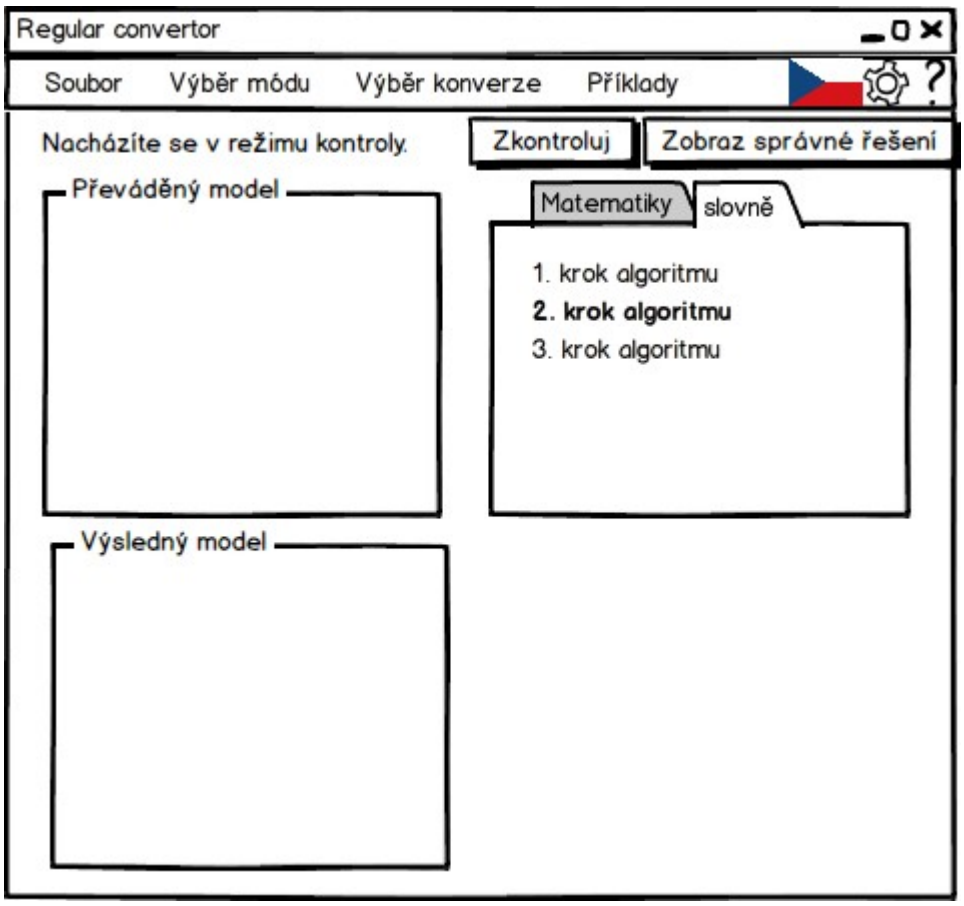
Vpravo nad oknem algoritmu můžeme stejně jako na předchozím obrázku nalézt tlačítka „vzad“ a „vpřed“ s ikonami ve tvaru šipek. Na rozdíl od předchozího režimu mají odlišný význam. Uživatel si pomocí nich může listovat v historii kroků algoritmů, které už provedl. Má tak možnost si prohlédnout celý postup výpočtu a může mu to lépe pochopit daný konverzní algoritmus.

V tomto módu uživatel postupně provádí jednotlivé kroky algoritmu. Pokud se podaří uživateli provést jeden krok, tak se v algoritmu automaticky zvýrazní další krok, informující uživatele, že se mu povedlo úspěšně provést další krok.

Původně jsem zamýšlel, že bude existovat tlačítko, které když uživatel stiskne, tak mu aplikace zkontroluje, zda provedl aktuální krok správně, pokud ano, tak se zvýrazní další krok algoritmu, který má uživatel provést. Následně jsem tento koncept zavrhl, protože by uživatel nedělal skoro nic jiného, nežli mačkal toto tlačítko a myslím, že by ho to přestalo velmi rychle bavit. Jako přirozenější se mi jeví, aby aplikace automaticky kontrolovala postup uživatele a při splnění se automaticky posunula na další krok.

Hlavní význam tohoto módu je v tom, že uživatel dostává zpětnou vazbu od programu, zda postupuje při řešení konverze správně či nikoliv a vyhne se tak pokračování hlouběji v konverzi se špatným mezivýsledkem.

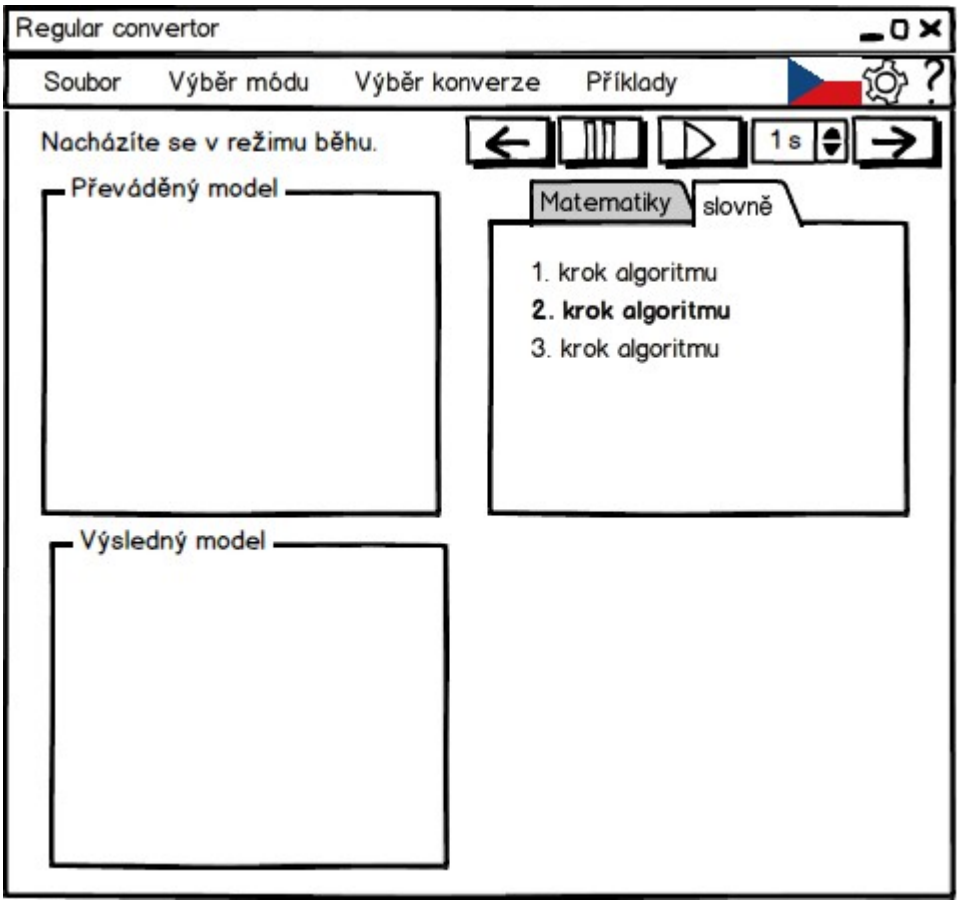
Poté co si uživatel zkusil převést daný model krok po kroku za pomoci režimu průběžné kontroly, je vhodné, aby si uživatel zkusil převést model úplně sám. Jedině tehdy se uživatel skutečně přesvědčí, zdali se převod správně naučil a rozumí všem krokům. Tohle vše umožňuje **Režim samostatné práce** jehož grafický návrh můžete vidět na obrázku 4.




*Obrázek 4: Režim kontroly*

Nad oknem pro algoritmus jsou vidět dvě tlačítka. První, „Zkontroluj“ řekne uživateli, zdali je výsledný model správný. Druhé, „Zobraz správné řešení“ ukáže uživateli, jak má správný výsledek vypadat, bylo by totiž z didaktického hlediska špatné, kdybychom uživateli řekli, že jeho řešení je nesprávné a neposkytli mu správné řešení.

Poslední a neméně důležitý je **Krokovací režim**. Jeho grafický návrh můžete nalézt na obrázku 5.




*Obrázek 5: Režim běhu*

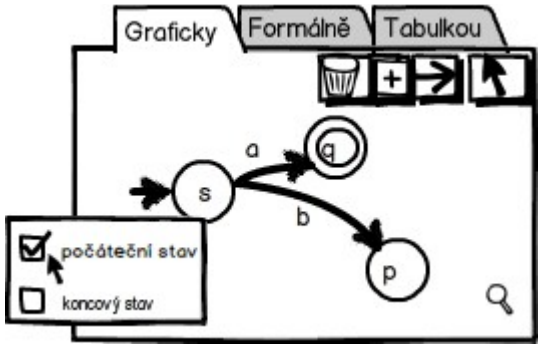
V tomto módu si uživatel může krokovat algoritmus a sledovat tak jak funguje na konkrétních příkladech.

Na rozdíl od ostatních má nejvíce tlačítek na ovládání. Uživatel si může krokovat algoritmus stiskem tlačítek s ikonou šipek nacházejících se úplně vlevo a úplně vpravo. Mezi nimi jsou tlačítka s ikonami pauzy a play. Po navolení prodlevy mezi kroky si pomocí těchto dvou tlačítek můžeme buď algoritmus zastavit, anebo nechat volně běžet s přednastavenou dobou prodlevy mezi jednotlivými kroky až do té doby dokud nenarazí algoritmus na případný breakpoint (místo kde se krokování algoritmu zastaví při použití tlačítka „play“).

Funkci „breakpointu“ jsem původně nezamýšlel, ale spolu s vedoucím mé bakalářské práce se nám tato funkce zalíbila u jiné podobné bakalářské práce zaměřené na demonstraci grafových algoritmů, načež jsem se ji rozhodl také implementovat. Tato funkce bude užitečná zvláště tehdy pokud bude chtít vyučující ukázat nějakou zajímavou část algoritmu a nebude se k ní chtít doklikat přes mnoho kroků. Jednoduše zvolí přehrávání a algoritmus se vždy zastaví tam, kde je zvolený breakpoint.

### 4.1.1.3 Reprezentace konečného automatu

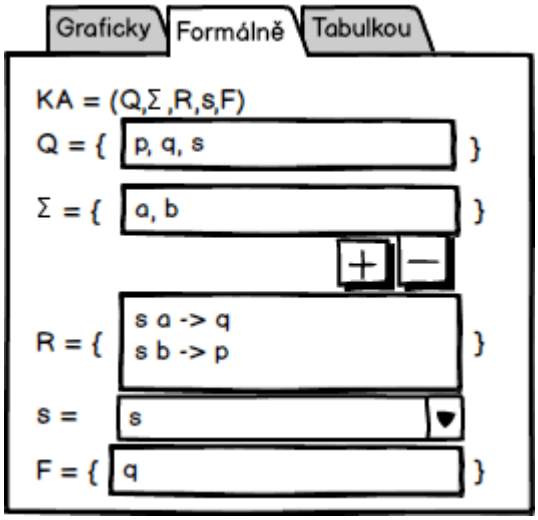
Konečný automat můžeme reprezentovat třemi různými způsoby, první a to **grafický pohled** můžete vidět na obrázku 6 (ostatní dva si představíme následně).




*Obrázek 6: Grafický pohled na KA*
*po kliknutí pravým tlačítkem myši*
*na uzel s*

V grafickém pohledu se nachází v pravém horním rohu tlačítka na **mazání** uzlů a hran, dále pak tlačítko na **přidávání uzlů**, **přidávaní hran** a na **přemisťování** hran a uzlů. Pokud chceme zároveň přesouvat více uzlů, můžeme je označit klikem myši spolu s držením klávesy „Ctrl“. Po kliknutí pravím tlačítkem na uzel, jak je vidět na obrázku 6, má uživatel možnost nastavit uzel jako koncový a jako počáteční. (Aplikace dovolí uživateli nastavit jako počáteční právě jeden uzel, viz kapitola Teorie str. 5.) Vpravo dole se nachází ikona pro přiblížení, nebo oddálení objektů v grafickém pohledu, stejného efektu se dá dosáhnout i stiskem klávesy „Ctrl“ a pohybem kolečka myši.

Na následujícím obrázku 7 můžete vidět druhý a to **formální pohled** na stejný KA. Kde $Q$ je konečná množina stavů, $\Sigma$ je abeceda, $R$ je množina přechodů, s je počáteční stav a $F$ je konečná množina koncových stavů.




*Obrázek 7: Formální popis KA*
*z obrázku 2*

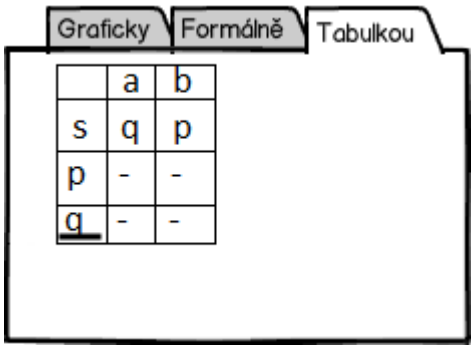
Uživatel bude moci pomocí textového řádkového editoru upravovat množiny $Q$ a $\Sigma$ a $F$. Jak symboly abecedy, tak názvy stavů musí být odděleny čárkou a libovolným množstvím mezer. Řádkový editor pro symboly abecedy akceptuje pouze jednoznakové symboly s výjimkou znaku mezera, která slouží spolu s čárkami jako oddělovač. Jiné než výše uvedené způsoby zápisů nebudou akceptovány. Po dokončení editace (stisk klávesy „enter“, nebo ztráty zaměření, anglicky focus) jedné z množin $Q$, $\Sigma$ nebo $F$ jsou duplicitní prvky odstraněny a ostatní se automaticky seřadí podle abecedy a jako oddělovač se použije čárka následovaná právě jednou mezerou, s výjimkou posledního prvku, za který se čárka nedoplní. Při editaci množiny $F$ se budou navíc při psaní automaticky nabízet prvky z množiny $Q$, prvky, které nejsou z množiny $Q$, se po dokončení editace automaticky odstraní.

Prvky množiny $R$ se přidávají pomocí tlačítka „+“, po jehož stisknutí se objeví dialogové okno, v němž může uživatel nastavit nový přechod. Pokud chce uživatel smazat prvky z množiny $R$, pak si je označí a s pomocí tlačítka „-“ je odstraní. Dialog pro editaci se zobrazí pomocí dvojkliku na položku z množiny $R$.

Jako počáteční stav si uživatel vybere jeden prvek z množiny $Q$ pomocí rozbalovacího seznamu (anglicky „combo box“). Pokud uživatel nezvolí vlastní počáteční stav, tak se automaticky vybere abecedně první stav.

#### 4.1.1.4 Tabulková reprezentace konečného automatu

Třetím možným zápisem je zapsat KA **tabulkou**. Tohoto zápisu nejvíce využijeme při algoritmu minimalizace KA.




*Obrázek 8: Tabulkový popis*

*obrázku 2*

Na obrázku 8 můžete vidět zápis KA tabulkou. V prvním sloupci je jméno uzlu při čemž na prvním řádku je počáteční uzel a všechny uzly, které jsou podtržené, jsou uzly koncové. V záhlaví tabulky jsou uvedeny symboly abecedy a pod nimi je naznačena množina přechodů. Pokud z místa

nevede hrana, se symbolem abecedy, pak je v tabulce uveden znak pomlčky. V tabulce je zapsán opět stejný KA jako z předchozích obrázků.

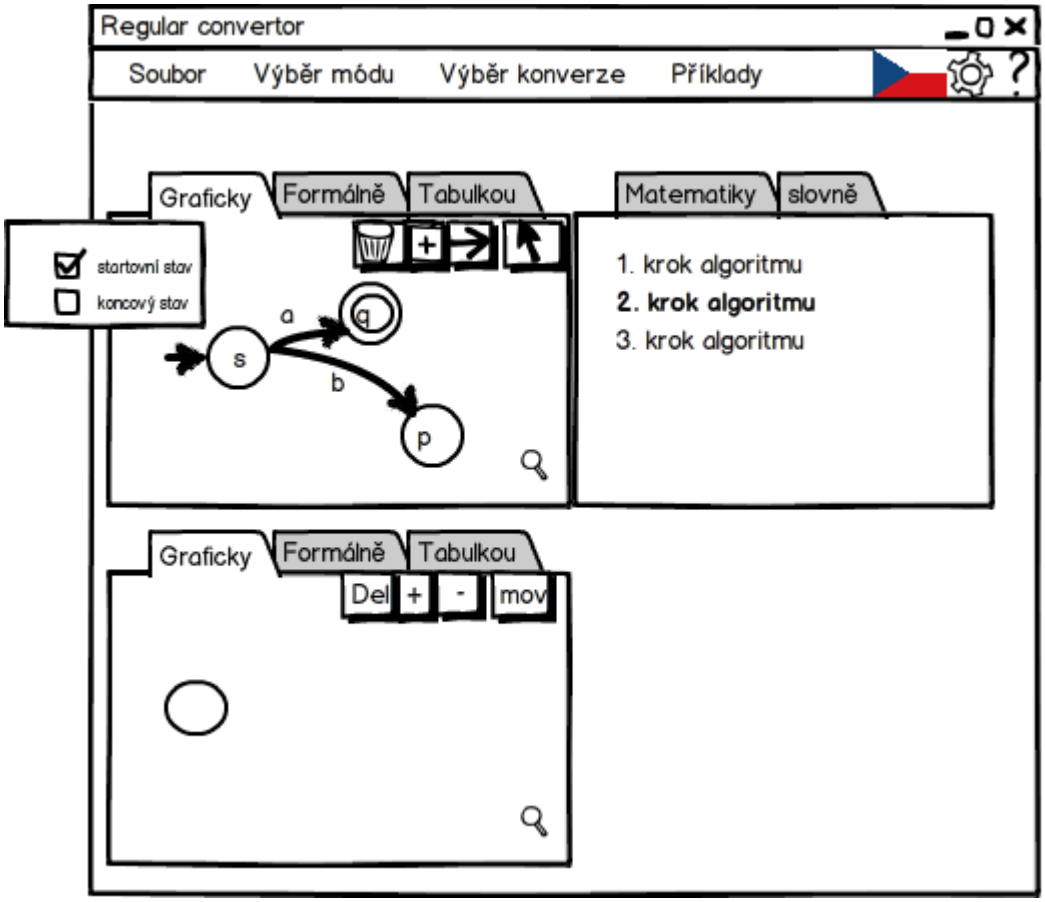
### 4.1.2 Návrh zobrazení převodů

Při návrhu GUI pro převody jsem se snažil držet schématu, že vždy vpravo nahoře bude převodní algoritmus, nalevo od něj bude výchozí model a v dolní části obrazovky okna potřebná k získání výsledného modelu.

#### 4.1.2.1 Konverze konečných automatů

Následující schéma zobrazené na obrázku 9 znázorňuje převod konečného automatu na jeho speciální typy a to:

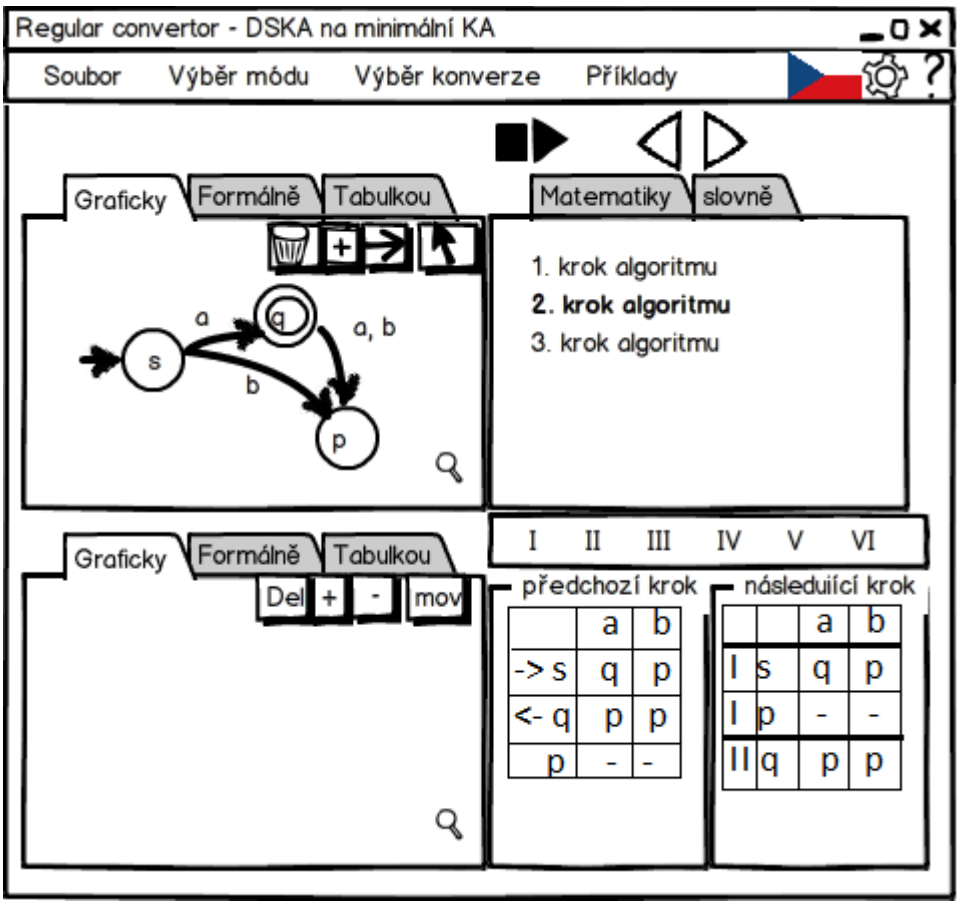
- Obecný KA na KA bez ε-pravidel.
- KA bez ε-pravidel na deterministický KA bez nedostupných stavů (DKA).




*Obrázek 9: Hlavní okno pro převod konečných automatů*

Uživatel bude postupovat podle převodního algoritmu a postupně bude v levém dolním okně tvořit výsledný KA. Pokud bude uživatel převádět v krokovém režimu, pak nebude moci po dobu jeho spuštění moci upravovat výchozí KA.

Speciálním typem převodu konečných automatů je převod **DSKA** na **minimální konečný automat**. Při tomto převodu potřebujeme dvě pomocná okna, ve kterých bude uživatel hledat rozlišitelné stavy.

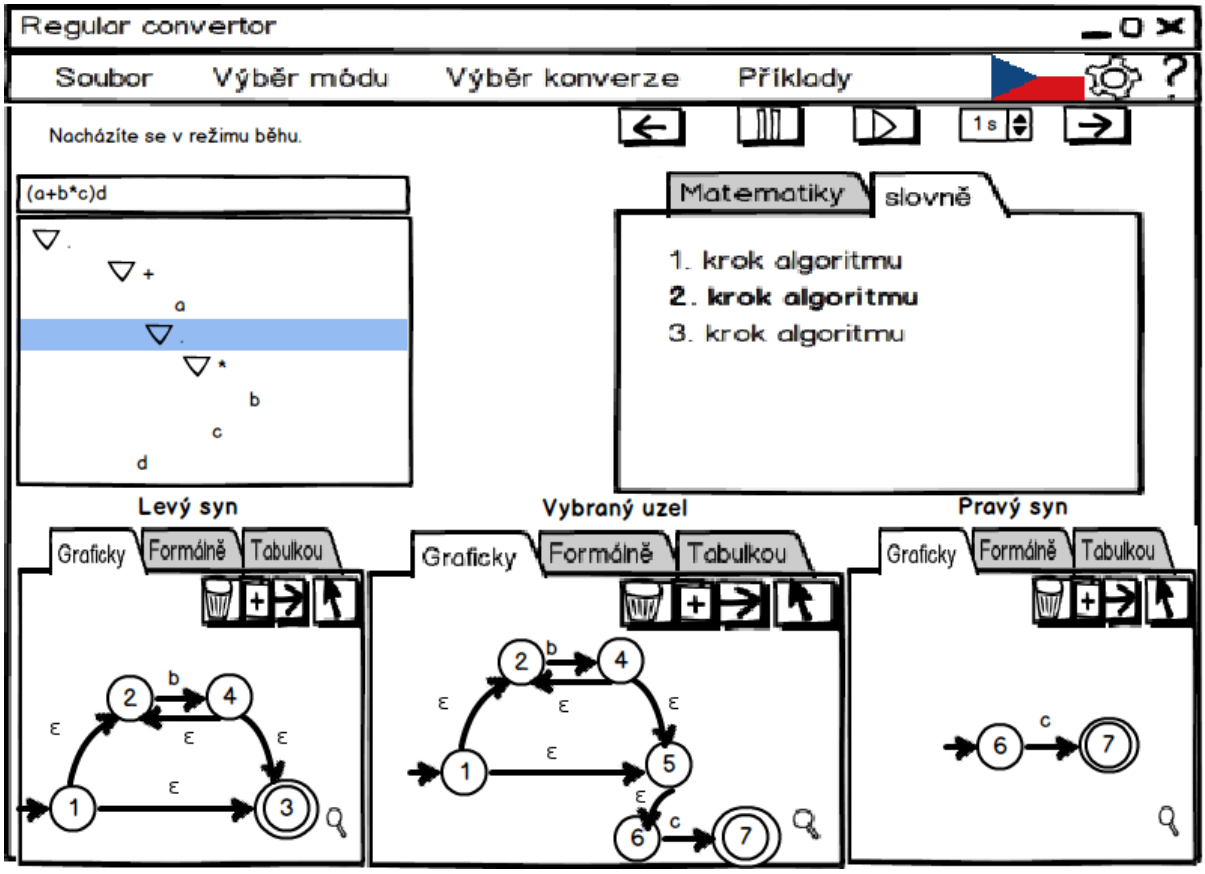



*Obrázek 10: Obrazovka pro převod DSKA na minimální FA*

Jak můžete vidět na obrázku 10, tyto dvě okna jsem umístil do volného prostoru vpravo dole pod okno s algoritmem. V levém okně pojmenovaném **předchozí krok** uživatel vidí předchozí krok algoritmu. Nový krok, který se následně stane krokem předchozím, tvoří uživatel v levém okně pojmenovaném **následující krok**. Jelikož je konvence v převodu na minimální KA označovat jednotlivé skupiny uzlů římskými číslicemi, je nad těmito dvěma okny prvek, ve kterém si může uživatel vybrat, kterou římskou číslicí chce označit kterou skupinu. Po rozdělení uzlů KA do finálních skupin musí uživatel výsledný automat překreslit do výstupního okna pro minimální automat. Z tohohle pohledu lze okna předchozí a následující krok brát jako okna pomocné.

### 4.1.2.2 Konverze regulárního výrazu na konečný automat

Návrh okna pro převod regulárního výrazu můžete nalézt na obrázku 11.




*Obrázek 11: Převod regulárního výrazu na konečný automat*

V levé horní části okna se nachází řádkový editor pro regulární výraz. Když uživatel edituje regulární výraz, tak se automaticky vytváří v okně pod řádkovým editorem derivační strom. Pokud se nachází v řádkovém editoru neplatný regulární výraz (definici validního regulárního výrazu můžete nalézt na straně 5, tak se jeho barva změní na červenou, informujíc uživatele, že udělal, chybu a zmizí i derivační strom.

V dolní části obrazovky vidíme tři editory konečných automatů. Prostřední z nich je konečný automat odpovídající regulárnímu výrazu v označeném uzlu. (V našem případě se jedná o regulární výraz „b*c“.) Levý konečný automat odpovídá levému synu (v našem horizontálním zobrazení derivačního stromu vrchnějšímu potomkovy) a pravý KA je analogický levému KA.

Převod v Krokovacím režimu probíhá tak, že uživatel postupuje od listů derivačního stromu směrem ke kořeni. V nekoncových uzlech se nachází operátory a v jejich synovských uzlech se nacházejí operandy. Takto uživatel pomocí atomických operací skládá výsledný konečný automat, který se nachází v kořeni stromu. Všechny uzly, ke kterým není doplněný KA, jsou označeny žlutě –

čekají na vyplnění. Pokud uživatel vyplní konečný automat pro jeden uzel syntaktického stromu správně, změní se jeho barva na zelenou – signalizující úspěch, v opačném případě na červenou – signalizující neúspěch.

Tento přístup je velmi vhodný i pro Režim samostatné práce, kdy uživatel nemusí vyplňovat všechny uzly, ale jenom některé, například může vynechat triviální reprezentaci KA pro listové uzly derivačního stromu, anebo i u jednoduššího regulárního výrazu napsat rovnou správný výsledek do kořenového uzlu. Navíc pokud uživatel udělá chybu, tak se mu červeně zvýrazní všechny uzly stromu, pro které vytvořil nesprávný KA.

# 5 Implementace

Celý projekt jsem implementoval v C++ za použití grafické knihovny Qt 5. Zdrojové kódy byly vyvíjeny pod linuxovým operačním systémem a testovány v MS Windows 7.

Vlastní program se nachází v projektu RegularConvertor a dále jsem si vytvořil pro jednotkové testování (anglicky Unit tests) druhý projekt RegularConvertorUnitTests. V tomto druhém projektu jsem si napsal testy pro klíčové metody hlavních tříd.

## 5.1 Poznámky k návrhu aplikace

V rámci zjednodušení a přirozenější práci s programem namísto explicitního přepínání do editačního módu jsem se rozhodl implementovat tento mód jako implicitní a to tak, že může uživatel v kterékoli chvíli editovat jak výstupní a vstupní model. Při změně vstupního modelu dojde k internímu restartu konverze se vymaže výstupní model. (Po změně výstupního modelu se vstupní model nemění.) Tento restart je především z důvodu změny vstupního modelu v průběhu krokovacího režimu, kde by nastávaly inkonzistence. Implementace historie změn editací modelů, pro přepínání předchozí a následující změny se ukázala z hlediska výukových účelů jako nepotřebná a v rámci zjednodušení jsem ji neimplementoval.

Místo implementace změny fontu a měřítka obsahu jednotlivých widgetů mi přišlo vhodnější, takové zobrazení, které je kompromisem mezi extra velkými fonty a naopak co největším množství informací v jednom okně, proto zobrazuji jen nejpodstatnější detaily, tak abych využil celou obrazovku.

Dále jsem rozpracoval změnu lokalizace a zvolil mezinárodní jazyk angličtinu. Rozpracované také zůstalo i ukládání a načítání konverze ze souboru. Jako nepotřebné jsem zvážil implementaci nastavení. Aplikace je natolik jednoduchá a intuitivní a navíc obsahuje kontextovou nápovědu (, která se zobrazí po najetí myši na ovládací prvek), že jsem se nakonec rozhodl nápovědu neimplementovat.

Odlišností od návrhu a implementace je, že původně jsem měl za to, že formálně zapsané algoritmy jsou pro studenty těžko stravitelné a proto jsem zamýšlel i druhou záložku neformální popis. Pak jsem se rozhodl, že k náročným krokům algoritmu přidám obrázky, aby bylo uživatelovi intuitivně jasné co se daným krokem algoritmu myslí.

Reprezentace KA tabulkou je nejužitečnější v algoritmu minimalizace KA. Jelikož v mnou prezentovaných konverzích nemá valný smysl, tak jsem tento pohled na KA neimplementoval.

Velkou předností je to, že zdrojové kódy s celou jejich historii jsou veřejně přístupné na GitHubu na adrese <https://github.com/navrkald/regularConvertor> a tak v případě zájmu

se mohou do vývoje aplikace zapojit i ostatní zájemci o zlepšení. Pokud bude o aplikaci větší zájem, budu se dál podílet na jejím vývoji i po ukončení bakalářské práce.

## 5.2 Implementace konečného automatu

Implementace KA sestává ze tří částí. První je třída *FiniteAutomata* reprezentující **vlastní KA**. Najdeme v ní data týkající se konečného automatu, to jest množinu uzlu, hran, koncových uzlů, počáteční stav, abecedu. Hranu KA jsem implementoval ve třídě ComputationalRules. V třídě *FiniteAutomata* jsem také ukládal informaci i rozmístění uzlů ve scéně. Tuto informaci jsem dodal do této třídy až později, protože se ukázalo, že při každém vložení KA do scény pomocí náhodného rozmístění není zrovna uživatelsky přívětivé. Tato nová informace mi také umožnila, při vytváření příkladů do aplikace, předem rozmístit uzly KA tak, aby to bylo co nejpřehlednější. Poslední informaci ukládanou v této třídě je proměná *nextId*, která slouží pro automatické vytváření jmen nových uzlů, protože uchovává poslední použité unikátní číslo pro vytvoření dalšího stavu. Kromě samotných dat KA třída *FiniteAutomata* obsahuje všechny možné operace nad konečným automatem, od jeho editace v podobě například přidáváním a odebíráním uzlů přes metody, které odpovídají na otázky zda má automat nějaké epsilon přechody, nebo zda je deterministický a podobně. Také obsahuje metody na determinaci, odstranění epsilon pravidel, minimalizaci a celou řadu dalších.

Druhá část, **grafický editor KA** jsem implementoval ve vlastní režii pomocí Qt třídy *QGraphicsView* a QgraphicsScene. (Všechny zde prezentované názvy tříd začínající na velké „Q“, jsou třídami grafické knihovny Qt) Uzly jsem implementoval třídou *QGraphicsitem*. Přechody mezi uzly (šipky) jsem implementoval pomocí vlastní třídy, která dědí od *QgraphicLineItem.* Při implementaci grafického editoru konečného automatu jsem se inspiroval projektem v Qt příkladech základních aplikací a je dostupný online viz [5]. Tento grafický pohled se skládá ze tříd *Arrow* reprezentující spojnici mezi uzly konečného automatu, které jsou implementovány ve třídě *StateNode*.

Po dvojkliku na uzel je jej možno editovat, přípustné jsou jenom takové názvy uzlů, které již nejsou v grafu použity. Po kliknutí na uzel pravým tlačítkem, se zobrazí kontextová nabídka pro editaci, zda se jedná o počáteční stav a nebo koncový stav. Jelikož KA musí mít alespoň jeden uzel počáteční, není dovoleno odznačení příznaku počátečního stavu. Je možné odznačení pouze tak, že označíme jako počáteční stav, stav jiný. Při přidání prvního uzlu do scény se tento uzel automaticky nastaví na počáteční stav s názvem „0“. Pro editaci hrany slouží widget *SymbolInputDialog* ve, kterém se editují symboly ležící na dané hraně. Tato třída kontroluje validitu znaků abecedy, kontroluje, zdali jsou to řetězce o délce jedna a zda jsou odděleny čárkami s libovolným množstvím mezer. Pokud se seznam symbolů v tomto formátu nenachází tak nedovolí jejich přijmutí. Přebytečný počet mezer automaticky maže. Tyto dva druhy grafických primitiv jsou

uloženy ve scéně reprezentované třídou *DiagramScene*. Tato scéna reprezentuje grafický popis KA. Tento a formální pohled v sobě konsoliduje třída *FA_widget*.

Po výpočet automatického rozmístění uzlů jsem chtěl původně použit knihovnu pro práci s grafy Graphviz. Avšak díky nekvalitní dokumentaci a nedostatku času jsem byl nucen naimplementovat vlastní rozmísťování uzlů. Můj způsob řešení spočívá v tom, že pro každý nový uzel si nechám náhodně vygenerovat pozici a následně zkontroluji, jestli na tomto místě nekoliduje s již umístěným uzlem. Takto má algoritmus *n* pokusů (v mé implementaci jsem stanovil *n* rovno 1000) na to, aby umístil uzel na své místo. Pokud ani po *n* náhodných pokusech algoritmus nenalezne místo, kde by mohl umístit nový uzel bez kolize s ostatními, tak jej umístí na takovou pozici z *n* pokusů, kde by kolidoval s co nejmenším počtem uzlů. Tento způsob funguje rychle s počtem uzlů v řádu desítek, dokud již není scéna zcela zaplněná. Jelikož se jedná o výukový program, neshledávám toto omezení za zcela zásadní. Když se už jednou KA vygeneruje, pak si pamatuje pozici svých uzlů.

Třetí část je **editor formálního popisu KA**, který je součástí třídy *FA_widget*. V tomto editoru je KA popsán textově. Pro pohodlnější editaci počátečního uzlu a množiny koncových stavů a editaci přechodů jsem napsal třídu MultiSelectCompleter, která uživateli při vyplňování těchto hodnot nabízí funkci automatického doplňování. Také se v *FA_widget* kontroluje, zda uživatel nastavil korektní hodnoty a nedovolí mu nastavit hodnoty nesprávně, například nemůže přidat přechod, který vede do stavu který neexistuje a podobně.

## 5.3 Implementace editoru regulárních výrazů

Při implementaci editoru RV jsem řešil hlavní dva problémy a to kontrolu jeho validity a jak jej vhodně zobrazit uživateli. Pro účely prezentace a jeho následného převodu jsem zvolil řádkový editor RV, implementovaný pomocí mé třídy *RegExpTextEdit*, která dědí od *QTextEdit* společně s jeho derivačním stromem, který jsem zobrazoval pomocí widgetu *QTreeView*. Abych mohl získat derivační strom, rozhodl jsem se použít syntaktickou analýzu zdola nahoru, při které jsem konstruoval derivační strom, který se následně graficky prezentoval uživateli. Pokud se při vytváření stromu narazilo na chybu, derivační strom se nevytvořil a uživateli se červeně zbarvil řádkový editor signalizující, že vložil nesprávný regulární výraz. Precedenční tabulku můžete vidět v tabulce číslo Tabulka 1.

| | + | . | * | ( | ) | i | $ |
|---|---|---|---|---|---|---|---|
| + | > | < | < | < | > | < | > |
| . | > | > | < | < | > | < | > |
| * | > | > | > | | > | < | > |
| ( | < | < | < | < | = | < | |
| ) | > | > | > | | > | | > |
| i | > | > | > | | > | | > |
| $ | < | < | < | < | | < | |

*Tabulka 1: Precedenční tabulka*

V prvním řádku se nachází další čtený symbol a v levém sloupci se nachází terminál nejblíže vrcholu zásobníku. Uvnitř tabulky se nacházejí symboly „<“, „>“, „=“ a „ “ (prázdná buňka tabulky), které popořadě znamenají „shift”, „redukce“, „rovno“ a „chyba“.

Jelikož se v zápisu regulárního výrazu obvykle vynechává operátor „.“, tak jej bylo nutné pro vytvoření derivačního stromu interně doplnit. Znak tečky se doplňoval, pokud byl aktuální znak znakem abecedy, levá závorka, nebo hvězdička a zároveň pokud následující znak byl znakem abecedy a nebo levá závorka.

Interní třída, kterou jsem reprezentoval regulární výraz je *RegExp*, takto třída využívá k parsování regulárního výrazu třídu *RegExpParser*, která řídí celý proces syntaktické analýzy zdola nahoru. Dále třída *ParserStack* reprezentuje zásobník, na kterém se provádějí jednotlivé redukce.

Samotný derivační strom je interně reprezentován pomocí stromové struktury, která je implementována pomocí třídy *RegExpNode. Jelikož třída QtreeView* vyžaduje k zobrazení svých dat model-view přístup, musel jsem implementovat ještě třídu *RegExpTreeModel*, která dědí od *QAbstractItemModel* a vystupuje jako prostředník mezi třídami *RegExpNode* a *QtreeView.*

## 5.4 Kontrola správných výsledků

1. Ve všech třech implementovaných konverzích je výstupem konečný automat. V rámci komfortnosti se nekontroluje slepě uživatelův KA a správný KA, ale kontroluje se zda výstupní KA má požadovanou vlastnost (nemá epsilon pravidla, je deterministický) a zda je ekvivalentní se správným řešením. Kontrola ekvivalence se provádí tak, že se jak správný výsledek, tak výsledek od uživatele převede na KA bez epsilon pravidel, následně se determinizuje, převede na dobře specifikovaný KA, minimalizuje, pak se oba normalizují (viz kapitola Teorie a podkapitola „Algoritmus normalizace deterministického KA“) a až na závěr se porovnávají jedna ku jedné.

## 5.5 Implementace konverzí

Při implementaci jednotlivých konverzí jsem vycházel ze snímků z přednášek předmětu Formální jazyky a překladače, viz [6] a [7]. Použité obrázky v konverzích v programu jsem se svolením pana profesora Meduny použil odtud taktéž.

Nyní si přiblížíme, jak jsem konverze implementoval. Základ tvoří můj vlastní widget (grafická komponenta) *AlgorithmWidget*, která se skládá z ovládacích tlačítek, které se při měnění módů skrývají nebo objevují. Dále obsahuje můj widget *Algorithmview* na zobrazení kroků konverze a dědí od Qt třídy *QlistView*. Jedná se o model-view architekturu, kdy máme model, který celý proces řídí, v mém případě je to moje třída *Algorithm*, která dědí of třídy *QStandardModel*. Od bázové třídy *Algorithm* pak dědí všechny moje implementace konkrétních algoritmů. V těchto třídách je pak implementován vlastní algoritmus převodu a také řízení grafického zobrazení v třídě *Algorithmview*. *Algorithmview* používá k zobrazení jednotlivých řádků algoritmu moji vlastní třídu *HTMLDelegate*, která zde vystupuje jako delegát. V této třídě je implementováno zobrazení break pointů zobrazování textu ve formátu *html*, obrázků a zvýrazňování aktuálního provedeného kroku. Jak jsem již uvedl třída *Algorithm* a její potomci dostanou přes parametry widgety na zobrazení a ovládání algoritmu, widgety s daty pro vstupní a výstupní modely konverze a všechny tyto modely pak řídí.

Všechny módy jsou implementovány ve třídě *Algorithm*. Výpočet dalšího kroku v Krokovacím módu zajišťuje vždy metoda *nextStep()* která je implementovaná pomoci struktury podobné case-switch, kde se vyhodnocuje pomocí sady podmínek, který krok algoritmu se má vykonat. V případě algoritmů na odstranění epsilon pravidel a determinizace KA se výpočet prování na základě zjištění předchozího kroku a pomocí sady podmínek se zjistí, který krok se má následně provést.

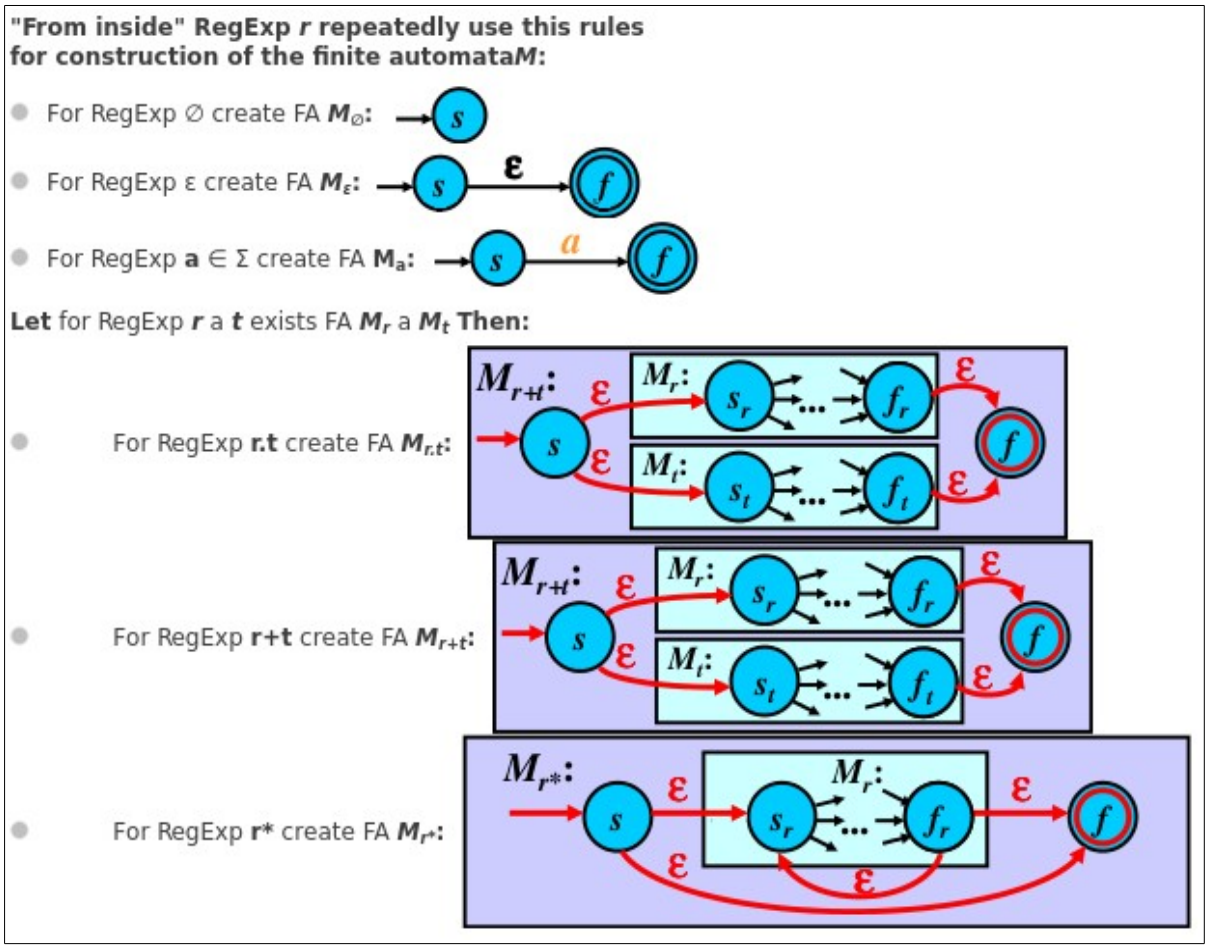
Pro účely přehrávání algoritmu si algoritmus vždy pamatuje jenom minulé kroky a nepamatuje si budoucnost. V případě, že jsme se vracely o krok zpět, tak posunem na následující krok se vždy provádí jeho novým výpočtem. Proto je nutné vždy po každém kroku budoucí historii smazat. Tento přístup má výhodu v tom, že nemusíme sledovat, zda jsme už následující krok provedly nebo ne a nedochází pak ani k inkonzistencím v případě, že by uživatel v průběhu krokování výstupní nebo vstupní model změnil.

### 5.5.1 Implementace převodu regulárního výrazu na KA

Při implementaci regulárního výrazu na konečný automat jsem vycházel z algoritmu uvedením v [6]. Algoritmus, který vidí uživatel je v podstatě „jedna ku jedné“ s tím co je v [6] plus jsem k jednotlivým krokům přidal obrázky taktéž odtamtud.

Následuje snímek obrazovky tohoto algoritmu implementovaný v programu. Stejně jako ve všech následujících algoritmech šedé puntíky před každým krokem označují breakpointy. (Šedé

jsou protože jsou neoznačené, aktivní breakpointy mají červenou barvu, jejíž výběrem jsem se inspiroval v nejednom vývojovém prostředí.)




*Obrázek 12: Snímek obrazovky algoritmu na převod RV na KA*

### 5.5.2 Implementace odstranění epsilon pravidel z obecného KA

Implementace algoritmu na odstranění epsilon pravidel se skládá ze dvou částí, první je algoritmus na získání epsilon uzávěru. Jakmile máme epsilon uzávěr, můžeme ho použít ve vlastním algoritmu na odstranění epsilon pravidel.

Obrázky a inspirace při vytváření krokovatelného algoritmu jsem čerpal z [7].

Následující algoritmus je cictací z [7] a nemohl jsem jej použít tak jak byl, ale musel jsem ho upravit do krokovatelné formy.

**Algoritmus odstranění epsilon pravidel:**

*Vstup: KA M = (Q, Σ, R, s, F)*

*Výstup: KA bez epsilon pravidel* $M'=(Q, \Sigma, R', s, F')$

$R' := \emptyset;$

<u>*for each*</u> $p \in Q$ <u>*do*</u>

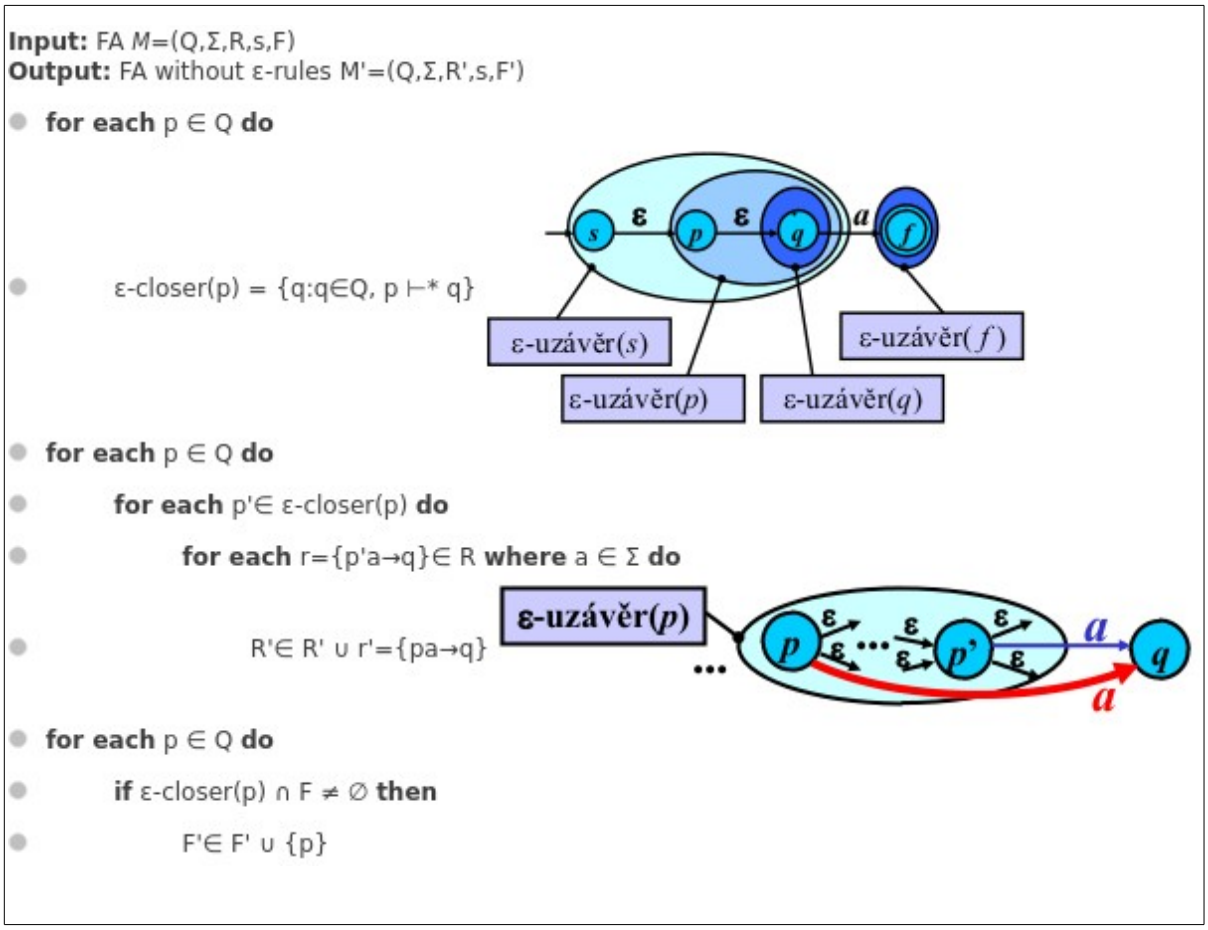
$R' := R' \cup \{pa \rightarrow q, p'a \rightarrow q \in R, a \in \Sigma, p' \in \varepsilon\text{-uzávěr}(p), q \in Q\};$

$F' := \{p: p \in Q, \varepsilon\text{-uzávěr}(p) \cap F \neq \emptyset\}$

Výsledný algoritmus, jehož snímek obrazovky můžete vidět na obrázku 13. Jak je vidno spojil jsem oba algoritmy do jednoho. Prvně jsem si v cyklu prošel všechny stavy a pro každý z nich vytvořil jeho epsilon uzávěr. Následně jsem ve třech zanořených cyklech prošel prvně všechny stavy, pak pro každý jsem prošel jeho epsilon uzávěr a nakonec pro každý jsem hledal pravidla taková, která vycházejí z těchto uzlů spolu s nějakým symbolem abecedy. A následně jsem do nového automatu přidával takto upravená pravidla, že jsem v nich nahradil původní vycházející místo, uzlem pro jehož jsme epsilon uzávěr počítaly.

V posledním cyklu se v novém KA označují koncové stavy všechny takové, které mají ve svém epsilon uzávěru alespoň jeden koncový stav.

Na závěr jsem algoritmus doplnil obrázky, aby byl více srozumitelný uživateli.




*Obrázek 13: Snímek obrazovky odstranění epsilon pravidel*

### 5.5.3 Implementace determinizace KA bez epsilon přechodů

Implementací determinizace KA jsem se inspiroval z [7]. Dále uvádím citaci algoritmu, ze kterého jsem vycházel. (V [7] je uveden ještě jeden algoritmus na determinizaci, ale po jeho aplikaci vznikne KA s velkým množstvím nedostupných stavů, proto jsem jej nepoužil.)

**Algoritmus determinizace KA:**

*Vstup: KA bez epsilon přechodů* $M = (Q, \sum, R, s, F)$

*Výstup: DKA* $M_d' = (Q_d, \sum, R_d, s_d, F_d)$

1. $s_d := \{s\}$; $Q_{new} := \{s_d\}$; $R_d := \emptyset$; $Q_d := \emptyset$; $F_d := \emptyset$;
2. <u>*repeat*</u>
3. *nechť* $Q' \in Q_{new}$; $Q_{new} := Q_{new} - \{Q'\}$; $Q_d := Q_d \cup \{Q'\}$;
4. <u>*for each*</u> a ∈ Σ *do* <u>*begin*</u>
5. $Q'' := \{q: p \in Q', pa \rightarrow q \in R\}$;
6. <u>*if*</u> $Q'' \neq \emptyset$ <u>*then*</u> $R_d := R_d \cup \{Q'a \rightarrow Q''\}$;

7. <u>if</u> Q" $\notin Q_d \cup \{\emptyset\}$ *then* $Q_{new} := Q_{new} \cup \{Q''\}$

8. *end;*

9. *<u>if</u>* $Q' \cap F \neq \emptyset$ *then* $F_d := F_d \cup \{Q'\}$

10. *<u>until</u>* $Q_{new} = \emptyset$

Když jsem se snažil uvedený algoritmus implementovat, tak jsem v něm narazil na praktický problém, který nastává na řádku 6, kdy se má do výstupního konečného automatu přidat hrana „$Q'\ a \rightarrow Q''$“, jenže uzel $Q''$ v tu chvíli ještě ve výstupním automatu neexistuje a vytváří se až po doběhnutí iterace na řádku 3 výše uvedeného algoritmu. V mém formálním popisu, ani v mé interní reprezentaci by to tolik nevadilo, jako v grafickém popisu, kde by šipka reprezentující přechod KA vysela tak říkajíc „ve vzduchu“. Abych tento problém vyřešil, musel jsem upravil tento algoritmus na:

**Upravený algoritmus determinizace KA:**

*Vstup: KA bez epsilon přechodů* $M = (Q, \sum, R, s, F)$

*Výstup: DKA* $M_d' = (Q_d, \sum, R_d, s_d, F_d)$

1. $s_d := \{s\}$; $Q_{new} := \{s_d\}$; $R_d := \emptyset$; $Q_d := Q_{new}$; $F_d := \emptyset$;

2. *<u>repeat</u>*

3. *nechť* $Q' \in Q_{new}$; $Q_{new} := Q_{new} - \{Q'\}$;

4. *<u>for each</u>* a $\in \Sigma$ *do <u>begin</u>*

5. $Q'' := \{q: p \in Q', pa \rightarrow q \in R\}$;

6. <u>if</u> Q" $\notin Q_d \cup \{\emptyset\}$ *then* $Q_{new} := Q_{new} \cup \{Q''\}$; $Q_d := Q_d \cup \{Q''\}$

7. *<u>if</u>* $Q'' \neq \emptyset$ *<u>then</u>* $R_d := R_d \cup \{Q'\ a \rightarrow Q''\}$;

8. *end*

9. *<u>if</u>* $Q' \cap F \neq \emptyset$ *then* $F_d := F_d \cup \{Q'\}$

10. *<u>until</u>* $Q_{new} = \emptyset$

Úpravy začínají na řádku 1, kde se přidá do $Q_d$ $Q_{new}$, potažmo počáteční stav. Dalé na řádku 3. je odstraněna inicializace $Q_d$. Pak oproti předcházejícím algoritmu jsou prohozeny řádky 6 a 7 a na řádku 7 se do $Q_d$ přidá nově objevený stav Q" a následně na řádku 7 se přidá nová hrana do DKA s novým stavem Q". Tento změněný algoritmus jsem testoval na řadě příkladů a proto i bez provedení matematického důkazu se domnívám, že je správný.

Výsledný algoritmus použitý v aplikaci můžete vidět na obrázku 14.

**Input:** FA without ε-rules $M=(Q, \Sigma, R, s, F)$
**Output:** DKA $M'=(Q_d, \Sigma, R_d, s_d, F_d)$

- $s_d = \{s\}$; $Q_{new} = \{s_d\}$; $R_d = \emptyset$; $Q_d = Q_{new}$; $F_d = \emptyset$
- **do**
- $\quad Q' \in Q_{new}$; $Q_{new} = Q_{new} - \{Q'\}$;
- $\quad$ **for each** $a \in \Sigma$ **do**
- $\quad\quad Q'' = \emptyset$
- $\quad\quad$ **for each** $r = \{p\ a \rightarrow q\}$ **where** $p \in Q'$
- $\quad\quad\quad Q'' \in Q'' \cup \{q\}$
- $\quad\quad$ **if** $Q'' \notin Q_d \cup \{\emptyset\}$ **then** $Q_{new} = Q_{new} \cup \{Q''\}$; $Q_d = Q_d \cup \{Q''\}$;
- $\quad\quad$ **if** $Q'' \neq \emptyset$ **then** $R_d = R_d \cup r'=\{Q'\ a \rightarrow Q''\}$;
- $\quad$ **if** $Q' \cap F \neq \emptyset$ **then** $F_d = F_d \cup \{Q'\}$
- **until** $Q_{new} = \emptyset$

*Obrázek 14: Snímek obrazovky algoritmu determinizace KA*

## 5.6 Ukládání a načítání konverzí

Konverze a prvky, ze kterých se skládají ukládám do souboru pomocí standardní Qt technologie takovým způsobem, že se zapisují do třídy *QDataStream*, která se následně zapisuje do souboru. Prostředí Qt nabízí pro většinu standardních tříd již implementované operátory „<<“ pro zápis a „>>“ pro čtení z tohoto proudu. Pro svoje vlastní třídy a objekty z nichž se skládají jsem si musel implementovat vlastní operátory pro čtení a zápis do proudu *QDataStream*. Tento postup byl víceméně přímočarý, až na třídu *RegExpNode*, která reprezentuje derivační strom regulárního výrazu, kde každý uzel v sobě uchovává množství vlastností, přičemž nejvýznamnější z nich je KA. U této třídy jsem musel jak pro ukládání, tak načítání použít rekurzivní algoritmus ve kterém jsem kromě vlastností každého uzlu musel ukládat i počet synovských uzlů.

V režimech Samostatné práce a Průběžné kontroly se kromě vstupního modelu ukládá i model výstupní. Naproti tomu se v Krokovacím režimu ukládá jenom vstupní model. Nemá totiž valný smysl ukládat rozpracované krokování, protože se můžu snadno vrátit v krokování tam, kde jsem krokoval předtím. Na rozdíl od předchozích dvou módů v tomto módu výstupní model netvoří uživatel, ale program a tak nepřichází uživatel o svou práci. V případě, že by se měl ukládat

i výstupní model v Krokovacím režimu znamenalo by to, že by se musela serializovat a ukládat do souboru i celá historie všech provedených kroků.

## 5.7 Metriky kódu

Na závěr se podíváme na metriky kódu a neb kolik práce bylo odvedeno. V následující tabulce můžete vidět, kolik zdrojových souborů jsem měl a kolik v nich bylo prázdných řádku, komentářů a řádků kódu. Tyto údaje jsem zjistil pomocí programu CLOC za použití příkazu „cloc <adresář se zdrojovými kódy>“.

| | **Typ** | **Počet souborů** | **Prázdné řádky** | **Komentáře** | **Řádku kódu** |
|---|---|---|---|---|---|
| | Cpp soubory | 30 | 956 | 590 | 5912 |
| | Hlavičkové soubory | 30 | 272 | 148 | 1217 |
| **Suma** | 60 | 60 | 1228 | 738 | 7129 |

*Tabulka 2: Metrika projektu RegularConvertor*

| | **Typ** | **Počet souborů** | **Prázdné řádky** | **Komentáře** | **Řádku kódu** |
|---|---|---|---|---|---|
| | Cpp soubory | 6 | 84 | 22 | 608 |
| | Hlavičkové soubory | 5 | 32 | 1 | 130 |
| **Suma** | 60 | 60 | 116 | 23 | 738 |

*Tabulka 3: Metrika kódu projektu RegularConvertorUnitTests*

# 6 Testování

Tento projekt jsem testoval jak ručně, tak pomocí unit testů, na které jsem si napsal samostatný Qt projekt jménem *RegularConvertorUnitTests*. V tomto projektu jsem si napsal „stínové“ testovací třídy pro hlavní třídy projektu *RegularConvertor*. Tyto třídy jsou:

- *FiniteAutomata_test*, zde se testuje: konkatenace, alternace, iterace konečných automatů, dále epsilon uzávěr, odstranění epsilon pravidel, determinizace, odstranění nedostupných stavů, odstranění neukončujících stavů, vytvoření dobře specifikovaného automatu, minimalizaci a normalizaci konečných automatů.
- *RegExpToFA_test*, zde se testuje převod regulárního výrazu na KA.
- *RemoveEpsilon_test*, zde se testuje odstranění epsilon pravidel z KA.
- *FaToDFA_test*, zde se testuje determinizace KA.

- *DFAtoMinFA_test*, zde se testuje minimalizace KA.

Tyto unit testy mi pomáhaly, kontroloval zda vše funguje správně v hlavním projektu i po změnách a zda jsem náhodou při úpravách nezavlekl do zdrojových souborů nové chyby, takže tyto testy sloužili zároveň jako regresní testy.

Unit testy se v Qt vytvářejí docela rychle, do testovací třídy je nutné vložil hlavičkový soubor „<QtTest/QtTest>“, vytvořit jí jako potomka *QObject* a jako soukromé sloty definovat minimálně metody *initTestCase()*, která je určená pro inicializaci proměnných, dále pak následují samotné testovací metody-sloty a na závěr metodu *cleanupTestCase()* ve které můžeme po doběhnutí testů po sobě uklidit. Aby vše správně fungovalo musí se v každé testovací metodě vyskytovat Qt makro *QCOMPARE* ve kterém se porovnává výsledek testu s referenčním výsledkem. Toto makro používá k porovnání výsledků c++ operátor „==“ a ten musí být pro daný typ/třídu definován. Abychom mohli testovat třídy a metody musíme je samozřejmě do testovacího projektu správně přidat.

Samotné testy se spouštějí ze souboru *main_test.cpp*, tak že se zavolá statická metoda *qExec(&<testovací_třída>)*, kde jako parametr použije reference na třídu která se má testovat. Toto volání provede všechny metody testované třídy. Po spuštění celého projektu se vypíše přehledná statistika, které metody uspěly a pokud neuspěly, ukáže nám, na kterém řádku ve zdrojovém kódu neuspělo makro *QCOMPARE*.

# 7 Zhodnocení

Na začátku jsem si dal předsevzetí, že mým cílem je uživatelská přívětivost a myslím, že se mi toto předsevzetí povedlo.

Dále je, v zadání, že v aplikaci má být obsaženo deset příkladů, já jich mám 26, z toho 10 příkladů je ke konverzi z regulárního výrazu na konečný automat, dále pak po 8 k odstranění epsilon pravidel a u determinizace KA. Příklady jsem vkládal od nejjednodušších po nejsložitější. Aby měl uživatel informaci jaký vstupní KA se skrývá pod kterým příkladem, tak jsem u konverzí konečných automatů jsem navíc vložil ke každému příkladu tooltip (anglicky nápověda zobrazovaná po najetí myši na ovládací prvek) ve kterém je obrázek převáděného konečného automatu. (Tohoto jsem musel docílit nestandardním způsobem, tak že jsem si musel vytvořit vlastní třídu *MyTooltipQMenu*, která umožňuje zobrazení tooltipu, protože standardní Qt třída *QMenu* funkci tooltipu neumožňuje.)

Množství příkladů je prakticky neomezené díky tomu, že moje aplikace umí ukládat a načítat, nejen předpřipravené, ale i rozpracované konverze ze souboru. Je tam možné, že učitel může připravit a studentům poslat sadu zajímavých a poučných příkladů, které si mají za domácí úkol vyřešit. Také může vyučující zveřejnit uložené konverze, které prezentoval na přednáškách.

Aplikace je lokalizovaná v mezinárodním jazyce angličtina, proto tuto aplikaci mohou používat nejenom čeští studenti naší fakulty, ale i hosté z výměnného programu studentů Erasmus.

Dále cíl, že má být aplikace použitelná na didaktickou demonstraci konverzí je plně splněna Krokovacím módem, který podporuje přetočení celého algoritmu na začátek, na konec, přehrání o jeden krok v před a vzad, automatické přehrávání spolu s breakpouinty a prodlevy mezi jednotlivými kroky.

Zcela nad rámec zadání jsem v módech Samostatná práce a režimu Průběžné kontroly implementoval to, že si mohou studenti samostatně zkoušet převodní algoritmy a ne jen pasivně sledovat postup výpočtu. Však i Jan Ámos Komenský říkal, že by se měli studenti učit na příkladech a naučit se již naučené samostatně aktivně používat a aplikovat.

Grafické uživatelské rozhraní jsem udělal takových způsobem, aby bylo využitelné ve výuce ve škole tak samostatně studenty doma takovým způsobem, že v každý okamžik se na obrazovce zobrazují jen nejdůležitější informace v rozumné míře, takže mohou být všechny dost veliké, aby byly vidět i ze zadních řad na přednášce a zároveň nebyli tak velké, aby obtěžovali studenta při práci doma.

# 8 Závěr

V této technické zprávě jste se dočetly, jak se postupem času vyvíjela moje bakalářské práce. V úvodu jste se dočetli, co bylo mým záměrem. Ve specifikaci požadavků, na čem jsem se dohodl s vedoucím mé bakalářské práce. V návrhu jsem popisoval, jak bych si představoval svou aplikaci. Z hlediska funkcionality, kterou má program podporovat, jak má vypadat a jak se má chovat se mi všechny hlavní cíle podařilo splnit. Některé funkcionality a ovládací prvky z návrhu se při implementaci ukázaly nepotřebné, nebo nepraktické, tak jsem od jejich realizace upustil. Na druhou stranu jsem implementoval některé funkce, které mě původně nenapadli, ale které přidávají uživatelský komfort a přívětivost. Při implementování konverzí z předmětu Formální jazyky a překladače jsem musel některé části algoritmů popsané matematicky přetransformovat do „krokovatelné“ formy a v jednom algoritmu jsem našel a opravil i menší chybu. Při implementaci jsem musel řešit řadu problémů. Například implementace regulárních výrazů si nakonec vyžádala implementaci syntaktického analyzátoru, sestrojení precedenční tabulky a vytvoření derivačního stromu.

Jako možná rozšíření se nabízí implementace dalšího konverzních algoritmu modelů regulárních jazyků, například regulárních gramatik a převodu konečného automatu zpět na regulární výraz. Tématem na diplomovou práci by mohlo byt implementace konverzních algoritmů pro modely jazyků složitější než regulární.

Při implementaci této práce jsem se hodně naučil o grafické knihovně Qt 5 a vývojem aplikací pro operační systémy rodin Microsoft Windows a Unix. Kdybych měl zhodnotit práci jako celek, domnívám se, že všechny hlavní cíle byly splněny s tím, že vždy je co vylepšovat.

# Literatura


[1] Meduna, Alexandr: Automata and Languages.. London, Springer, 2000, 81-8128-333-3

[2] Smrčka, Aleš, Vojnar, Tomáš, Češka, Milan: Teoretická informatika - studijní opora, Brno, [online]. 16.5.2014 [cit. 16.5.2014]. Dostupné na WWW: <http://www.fit.vutbr.cz/study/courses/TIN/public/Texty/oporaTIN.pdf>

[3] Dostál, Hubert: Normovaný tvar konečného automatu [online]. 16.5.2014 [cit. 16.5.2014]. Dostupné na WWW: <http://iris.uhk.cz/tein/teorie/normalizace.html>

[4] Krug, Steve: Nenuťte uživatele přemýšlet. Brno, Computer Press, 2006, 80-251-1291-8

[5] Qt dokumentace: Diagram Scene Example [online]. 19.5.2014 [cit. 19.5.2014]. Dostupné na WWW: <http://qt-project.org/doc/qt-4.8/graphicsview-diagramscene.html>

[6] Meduna, Alexandr: Modely regulárních jazyků - snímky k přednáškám do předmětu Formální jazyky a překladače [online]. 12.5.2014 [cit. 12.5.2014]. Dostupné na WWW: <https://www.fit.vutbr.cz/study/courses/IFJ/private/prednesy/Ifj03-cz.pdf>

[7] Meduna, Alexandr: Speciální konečné automaty - snímky k přednáškám do předmětu Formální jazyky a překladače [online]. 12.5.2014 [cit. 12.5.2014]. Dostupné na WWW: <https://www.fit.vutbr.cz/study/courses/IFJ/private/prednesy/Ifj04-cz.pdf>

# Seznam obrázků

# Seznam příloh na CD

1. Zdrojové kódy.
2. Technická zpráva ve formátu *otd* a *pdf*.
3. Obrázky použité v této technické zprávě.
4. Prezentace z obhajoby seminárního projektu.
5. Elektronické zdroje, ze kterých jsem čerpal.