Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Systém pre podporu práce s automatmi

Bakalárska práca

2019

Peter Bažík

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Systém pre podporu práce s automatmi

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	prof. RNDr. Branislav Rovan, PhD.

Bratislava, 2019
Peter Bažík

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Peter Bažík
Študijný program:	informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor:	informatika
Typ záverečnej práce:	bakalárska
Jazyk záverečnej práce:	slovenský
Sekundárny jazyk:	anglický

Názov: Systém pre podporu práce s automatmi
System supporting manipulation of automata

Anotácia: V praci je navrhnutý a implementovaný systém pre podporu práce s automatmi. Je implementovaných niekoľko algoritmov pre prácu s automatmi (napr. minimalizácia DKA, NKA, realizácia niektorých operácií, ...). Navrhnutý je jazyk pre vstup a výstup tak, aby bol systém potenciálne využiteľný ako učebná pomôcka. Systém umožní ľahko doplňovať ďalšie algoritmy.

Vedúci:	prof. RNDr. Branislav Rovan, PhD.
Katedra:	FMFI.KI - Katedra informatiky
Vedúci katedry:	prof. RNDr. Martin Škoviera, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 24.10.2018

Dátum schválenia: 24.10.2018

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.... študent

.... vedúci práce

Poďakovanie: Chcel by som sa poďakovať predovšetkým svojmu školiteľovi za vhodný námet na tému práce a za užitočné usmernenia pri návrhu systému a písaní tejto práce. Taktiež by som sa chcel poďakovať rodine a priateľom za sústavnú podporu a motiváciu.

Abstrakt

V práci navrhujeme a implementujeme systém slúžiaci pre výčbu teórie formálnych jazykov a automatov. Prvým cieľom bolo preskúmať vybrané existujúce systémy, ich funkcionalitu a nedostatky. Hlavným cieľom bolo navrhnúť systém tak, aby bol názorný, poskytoval dostatočnú funkcionalitu a eliminoval nedostatky existujúcich systémov. Systém pozostáva z užívateľského prostredia a knižnice *libfsm*, v ktorej sme implementovali štruktúry pre simuláciu konečných automatov a vybrané algoritmy na nich. Prvkami názornosti vysvetľovania sú farebná vizualizácia prechodového diagramu a detailný slovný popis algoritmov. V práci je navrhnutý jazyk pre formu vstupu a výstupu. Medzi implementované algoritmy patria operácie zjednotenia, prieniku, zreťazenia, reverzu a doplnku regulárnych jazykov, vybrané normálne tvary automatov a determinizácia nedeterministických automatov. Systém je implementovaný v programovacom jazyku *Java* a je voľne dostupný na internete.

Kľúčové slová: systém, automaty, algoritmy, výučba, vizualizácia

Abstract

In this work, we design and implement an educational system for formal languages and automata theory. The first goal of this work was to explore the functionality and shortcomings of the existing systems. The main objective was to design a system to be explanatory and sufficiently functional. It should also eliminate the deficiencies of the existing systems. The system consists of a user interface and the libfsm library, in which we implemented structures essential for finite automata and a simulation of the selected algorithms. The elements of the explanation clarity are a color visualization of the transition diagram and the detailed verbal description of the algorithms. We also designed a language for input and output. The implemented algorithms include union, intersection, concatenation, reverse, and complement of regular languages, selected normal forms of automata and determinization of nondeterministic finite automata. The system is implemented in Java programming language and open-sourced on the internet.

Keywords: system, automata, algorithms, education, visualization

Obsah

Úvod 1

1 Súčasný stav problematiky 3
1.1 Vybrané existujúce systémy . . . 3

2 Požiadavky na systém 7
2.1 Backend . . . 7
2.1.1 Štruktúry . . . 7
2.1.2 Algoritmy . . . 7
2.1.3 Vstup a výstup . . . 8
2.2 Frontend . . . 8
2.2.1 Základ . . . 8
2.2.2 Vizualizácie . . . 9
2.2.3 Priamy a krokovací mód . . . 9
2.2.4 Výpočet . . . 9
2.2.5 Konštrukčné algoritmy . . . 9

3 Návrh systému 11
3.1 Voľba paradigmy a jazyka . . . 11
3.2 Použité návrhové vzory . . . 12
3.2.1 Singleton . . . 12
3.2.2 Factory method . . . 12
3.2.3 Mediator . . . 12
3.2.4 MVC . . . 12
3.3 Použité technológie . . . 13
3.3.1 Graphviz . . . 13
3.3.2 GraphViz Java API . . . 14
3.3.3 Maven . . . 15
3.3.4 Scene Builder . . . 15
3.3.5 JavaFX a JFoenix . . . 16
3.4 Jazyk pre vstup a výstup . . . 16

3.5 Modularita 16
3.6 Knižnica libfsm 17
3.6.1 Reprezentácia základných entít 18
3.6.2 Balíček machines 18
3.6.3 Balíček conversions 20
3.6.4 Balíček normalForms 21
3.6.5 Balíček operations 21
3.6.6 Balíček helpers 22
3.7 Užívateľské prostredie 22
3.7.1 Výpočet 23
3.7.2 Operácie 23
3.7.3 Normálne tvary 24
3.7.4 Ekvivalencia 25

4 Implementácia **27**
4.1 Štruktúra aplikácie 27
4.2 Použitie návrhového vzoru MVC 28
4.3 Použitie návrhových vzorov Mediator a Singleton 30
4.4 Ukážky zaujímavých častí kódu 32
4.4.1 Vykresľovanie diagramu 32
4.4.2 Realizácia operácie s priamym výstupom 32
4.5 Zlepšenia do budúcnosti 33

5 Dokumentácia **35**
5.1 Prerekvizity 35
5.1.1 Windows 36
5.2 Inštalácia a spustenie 36
5.3 Príklad prevodu NKA na DKA 37

Záver **43**

Úvod

S teóriou formálnych jazykov a automatov sa v bežnom živote stretávame dennodenne, niekedy aj bez toho, aby sme si to uvedomili. Napríklad konečné automaty sú ekvivalentné regulárnym výrazom, ktoré sa pomerne často používajú vo vyhľadávaní či filtrovaní reťazcov znakov. Ďalším príkladom sú zásobníkové automaty, ktoré sú ekvivalentné bezkontextovým gramatikám. Tie sú používané v parseroch či kompilátoroch.

Tieto modely a ich využitie by mal každý informatik ovládať do detailov. No väčšina systémov, ktoré slúžia na výučbu teórie formálnych jazykov a automatov nepodporujú dostatočnú funkcionalitu. Ďalšie nie sú dostatočne názorné. Preto sme sa rozhodli navrhnúť a implementovať moderný a prehľadný systém, ktorý má používateľovi umožniť jednoduchšie pochopiť princípy tejto oblasti a ktorý poskytne dostatočnú funkcionalitu a názornosť.

Cieľom práce je naprogramovať systém v jazyku Java. Systém má obsahovať potrebné štruktúry, vybrané algoritmy a používateľské prostredie. Jadro systému má obsahovať prehľadné prostredie a možnosť pridania modulov s implementovanými štruktúrami a algoritmami. Prostredie treba navrhnúť tak, aby bolo používateľsky prívetivé a aby jeho komponenty názorne vysvetľovali výpočty či konštrukcie automatov.

Ďalším cieľom je systém navrhnúť tak, aby bol v budúcnosti jednoducho rozšíriteľný o ďalšiu funkcionalitu, ako napríklad pridanie ďalších modelov a algoritmov na nich. To má byť umožnené pomocou znovupoužitia vhodných štruktúr a doimplementovania komponentov, o ktoré má byť systém obohatený.

V prvej kapitole si popíšeme súčasný stav problematiky. Predstavíme si vybrané existujúce riešenia a ich výhody či nevýhody. V druhej kapitole sa budeme zaoberať špecifikáciou funkcionality a prostredia systému. V tretej kapitole si ukážeme návrh systému a predstavíme použité technológie či návrhové vzory. Vo štvrtej kapitole si ukážeme použitie technológií a návrhových vzorov. Ukážeme si v nej aj niekoľko zaujímavých častí kódu systému. Piata kapitola je zameraná na dokumentáciu. Zahŕňa zoznam prerekvizít a postup pri inštalácii a spustení systému. Zahŕňa tiež predvedenie použitia systému na konkrétnom príklade.

Kapitola 1

Súčasný stav problematiky

Existuje viacero systémov [11, 8, 2, 17] pre výučbu teórie formálnych jazykov, ktoré sa líšia v poskytovanej funkcionalite či kvalite používateľského prostredia. Niektoré sa zameriavajú na výpočet a vizualizáciu deklarovaného automatu na vstupnom slove [11, 8], ďalšie podporujú prevod nedeterministického konečného automatu na deterministický [2, 17] alebo minimalizáciu deterministického konečného automatu [17].

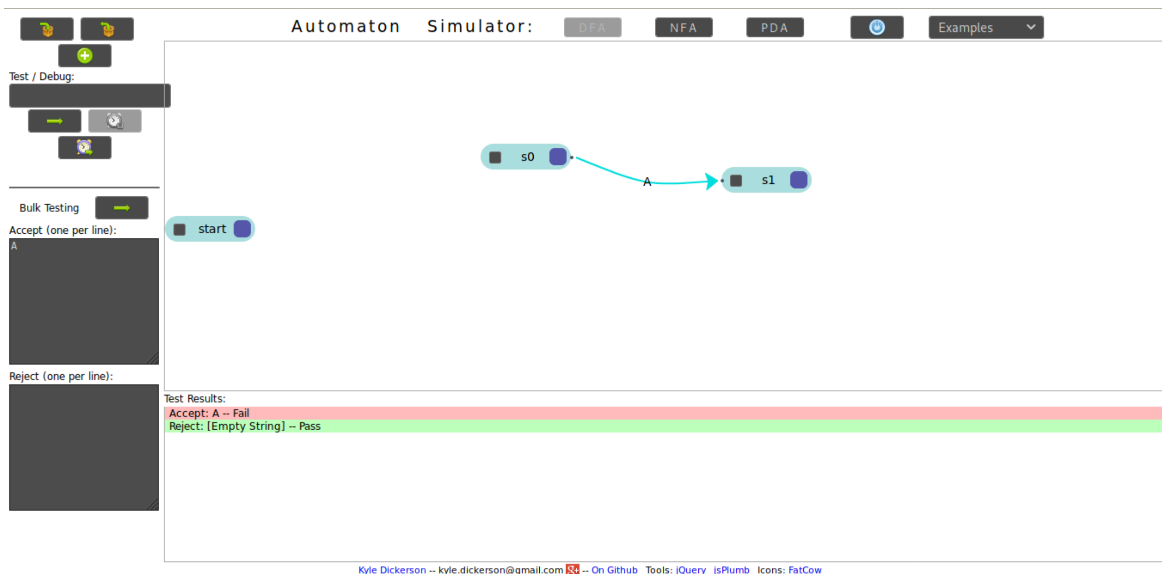
Stretneme sa aj so systémami, ktoré podporujú súčasne konštrukcie automatov, prevody, aj výpočet na slove, ale chýba im dostatočná názornosť a vysvetlenie princípov či použitých algoritmov [2, 17]. Niektoré zo spomenutých systémov využijeme pri tvorbe nášho. Dôvodom je, že napriek neduhom obsahujú funkcionalitu, ktorá môže byť pre náš systém prínosná.

1.1 Vybrané existujúce systémy

V tejto sekcii predstavíme niektoré z podobných systémov. Nahliadneme do ich poskytovanej funkcionality a porovnáme ju s očakávaniami od nášho systému. Zameriame sa aj na vizuálnu stránku a intuitívnosť prostredia.

Automaton simulator [11] je webová aplikácia, ktorá ponúka nasledovnú funkcionalitu. Automaty sa v nej dajú zostrojiť buď pomocou tlačidiel, alebo výberom jedného z príkladových. Po výbere alebo zostrojení umožní aplikácia zadať slovo a otestovať, či ho zadaný automat akceptuje, alebo zamieta. Okrem toho, táto aplikácia disponuje podporou testovania viacerých slov súčasne. K výpočtu je v prostredí k dispozícii prehľadný log. Náš systém má obsahovať podobný. Rozhranie však nie je tak používateľsky prívetivé, ako by sme chceli. Navyše nemá podporu konštrukcie operácií, ani potenciál pre extenzibilitu. Náš systém má byť jednoducho rozšíriteľný a má disponovať širšiou ponukou funkcionality. Tým by sme mali eliminovať nedostatky systému *Automaton simulator*. Ukážka prostredia *Automaton simulator* je na obrázku 1.1.

FSM simulator [8] je webová aplikácia, ktorej účelom je práca s automatmi. Do

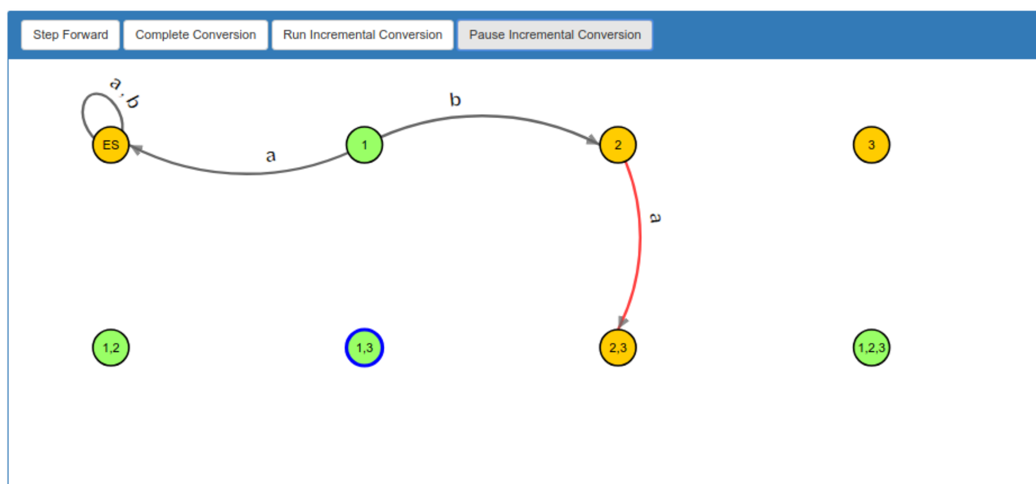


Obr. 1.1: Prostredie systému Automaton simulator

istej miery je podobná aplikácii *Automaton simulator*. Líši sa od nej v nasledovnej funkcionalite. *FSM simulator* nepodporuje testovanie viacerých slov súčasne. Podporuje ale krokovaciu formu výstupu spolu s farebnou vizualizáciou aktuálnych stavov automatu, čo môže byť pre používateľa dostatočne názorné. Tento systém, podobne ako *Automaton simulator*, nepodporuje okrem výpočtu na slove ďalšie konštrukcie či prevody. Podporuje však prácu s regulárnymi výrazmi a možnosť generovania DKA, NKA, regulárneho výrazu alebo vstupného slova. Generovanie náhodných automatov alebo slov implementovať nebudeme, pretože v tom nevidíme rozumný význam. Keby sa užívateľovi nechcelo zadávať automat, bude si v našom systéme môcť vybrať jeden z príkladových. Keby bol zo strany užívateľov veľký záujem o takéto rozšírenie, ľahko ho do systému doimplementujeme.

FSA animate [2] je tiež webová aplikácia zameraná na prácu s automatmi. Od predchádzajúcich sa líši podporou prevodu NKA na DKA a možnosťou načítania automatu zo súboru. Vstupný súbor používa formát *Json*. *FSA animate* síce podporuje vizualizáciu priebehu konštrukcie DKA zo zadaného NKA, ale v prostredí chýba popis aktuálne vykonávaného kroku prevodu. Tento nedostatok chceme v našom systéme odstrániť umiestnením detailného popisu vykonávaného prevodu do užívateľského prostredia. Od nášho systému tiež očakávame podporu načítania zo súboru vo formáte *Json*. Vizuálne zobrazenie priebehu prevodu NKA na DKA môžeme vidieť na obrázku 1.2. Na obrázku si tiež môžeme všimnúť absenciu popisu aktuálne vykonávaného kroku.

JFLAP [17] je systém obsahujúci balíčky grafických nástrojov pre prácu s automatmi, regulárnymi výrazmi a ďalšími výpočtovými modelmi, určený na výučbu teórie formálnych jazykov.



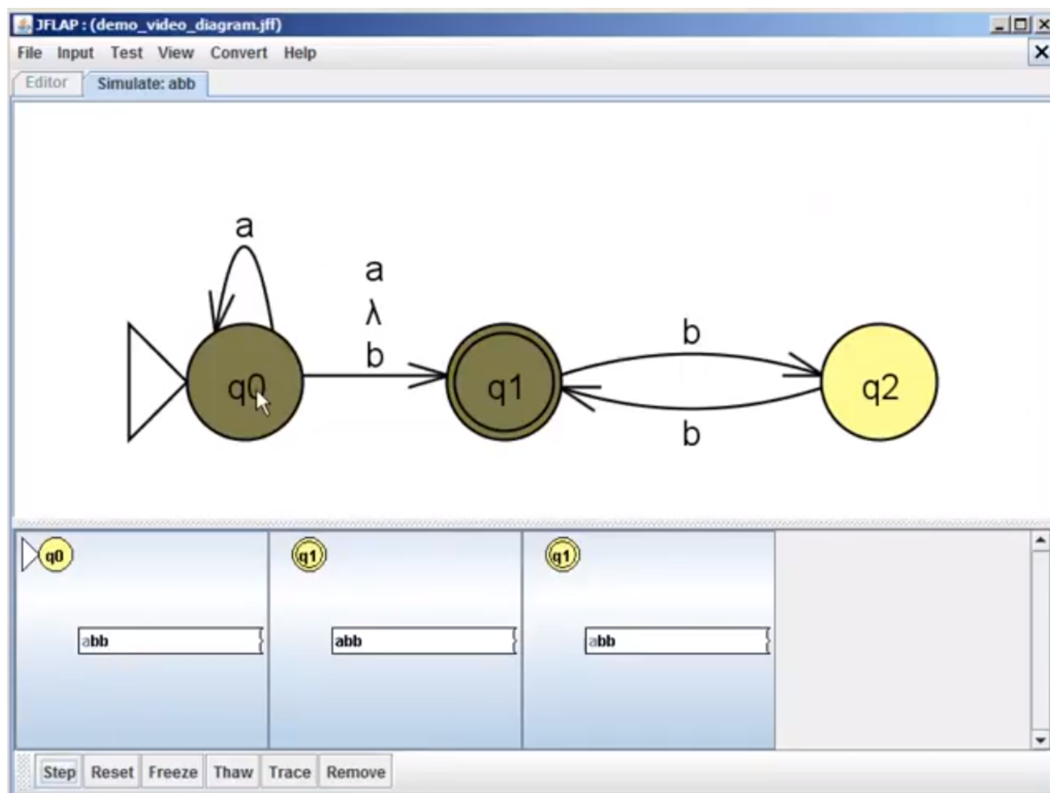
Obr. 1.2: Konštrukcia DKA z NKA v systéme FSA animate

Tento systém je zo spomenutých najpodobnejší tomu, ako má vyzerať náš. Disponuje podporou konštrukcie modelov pomocou tlačidiel, vizualizáciou zostrojovaného automatu či simuláciou výpočtu na slove. *JFLAP* podporuje vizualizáciu priebehu konštrukčných algoritmov aj výpočtu na slove. Pri výpočte na slove systém zobrazí aktuálnu konfiguráciu, v ktorej sa automat nachádza a vyznačí aktuálny (resp. aktuálne) stav (resp. stavy) automatu na prechodovom diagrame.

JFLAP poskytuje nasledovné algoritmy týkajúce sa konečných automatov alebo regulárnych jazykov:

- Odepsilónovanie NKA
- Minimalizácia DKA
- Uzáverové vlastnosti regulárnych jazykov
- Prevod DKA na regulárnu gramatiku a naopak
- Prevod DKA na regulárny výraz a naopak
- Identity regulárnych výrazov
- Vzťah DKA ku Grep-u

Avšak, náš systém bude lepšie vysvetľovať postup, ktorý sa vykonáva v priebehu algoritmu. Spôsob vysvetlenia bude taký, že pri každom kroku sa okrem farebného vyznačenia, bude aj slovne popisovať, čo sa práve vykonalo. Pri výpočte na slove sa na prechodovom diagrame farebne vyznačí, ktorý prechod prechodovej funkcie môžeme v ďalšiom kroku použiť. *JFLAP*, podobne ako *FSA animate*, nemá podporu funkcie popisu aktuálne vykonávaného kroku výpočtu či konštrukcie. Tento nedostatok je zrejmý



Obr. 1.3: Výpočet NKA na slove v systéme JFLAP

z obrázku 1.3. Na obrázku si môžeme všimnúť aj neprítomnosť farebného vyznačenia prechodu prechodovej funkcie.

Kapitola 2

Požiadavky na systém

V tejto kapitole sa budeme zaoberať požiadavkami na systém. Popíšeme si, aké požiadavky máme na funkcionalitu a užívateľské prostredie systému.

2.1 Backend

V tejto sekcii si popíšeme, akú funkcionalitu očakávame od systému. Začneme špecifikáciou modelov, ktoré majú byť v systéme zostrojiteľné. Potom popíšeme algoritmy na modeloch, ktoré chceme aby systém názorne vysvetľoval a nakoniec si ukážeme možné formy vstupu a výstupu.

2.1.1 Štruktúry

Ako hlavné výpočtové modely v našom systéme majú byť konečné automaty, menovite deterministický konečný automat (DKA) a nedeterministický konečný automat (NKA). K týmto modelom treba v systéme navrhnúť vhodnú reprezentáciu. Systém treba navrhnúť tak, aby bolo v budúcnosti možné jej jednoduché rozšírenie o ďalšie výpočtové modely, napríklad zásobníkový automat (PDA) alebo regulárne výrazy. V systéme taktiež požadujeme navrhnúť reprezentáciu jednotlivých komponentov automatov, teda stav a prechodovú funkciu.

2.1.2 Algoritmy

Pre systém treba navrhnúť balíček s funkcionalitou pre simulovanie výpočtu automatov na vstupnom slove, pre konštrukcie nových automatov podľa uzáverových vlastností triedy jazykov, ktorej jazyk daný automat rozpoznáva. Okrem konštrukcií automatov pre uzáverové vlastnosti treba navrhnúť a implementovať funkcionalitu pre konštrukcie DKA zo zadaného NKA a algoritmus na zostrojenie minimálneho DKA. Balíček s

algoritmami treba navrhnúť a implementovať tak, aby bol v budúcnosti jednoducho rozšíriteľný o ďalšie algoritmy.

2.1.3 Vstup a výstup

Teraz sa pozrime na to, akú formu vstupu a výstupu požadujeme od nášho systému. Začnime s formou pre vstup.

Používateľovi treba umožniť pri operácii výpočtu na vstupnom slove vhodným spôsobom zadať výpočtový model a vstupné slovo, ktoré má zadaný výpočtový model vypočítať. Pri operácii konštrukcie k uzáverovej operácii a ďalším algoritmom treba umožniť zadať viacero modelov a operáciu, ktorú požaduje od systému vykonať. V systéme treba navrhnúť vhodný jazyk alebo spôsob pre vstup, pomocou ktorého bude používateľ automaty zadávať. Vstup má mať dve formy. Prvou je priamo v užívateľskom prostredí a to pomocou tlačidiel. Druhou formou vstupu má byť načítanie automatov zo súboru.

Zašpecifikujme si teraz formy výstupu. Používateľovi treba počas vykonávania výpočtu či konštrukcie priebežne vracať vo vhodnej forme výstup, aby bol upovedomený, aký krok výpočtu či konštrukcie sa práve vykonal. Toto má byť umožnené vizuálne aj slovne. Pri výpočte treba používateľovi detailne znázorniť v akom stave sa automat nachádzal pred krokom výpočtu, aký prechod prechodovej funkcie bol pri kroku výpočtu použitý, a v akom stave sa automat nachádza po vykonaní daného kroku. Po celkovom vykonaní výpočtu treba dať používateľovi informáciu o tom, či automat dané slovo akceptoval, alebo zamietol. Rovnako čo sa týka algoritmov, po ich celkovom vykonaní treba dať na výstup ich výsledok. Od systému požadujeme, aby bol takýto výstup daný zobrazením prechodového diagramu výsledku v užívateľskom prostredí a tiež požadujeme, aby si používateľ mohol nechať výsledok zapísať do súboru, ktorý bude použiteľný pre ďalšie úkony.

2.2 Frontend

Teraz sa pozrieme na to, ako má vyzerať užívateľské prostredie systému. Najprv si zašpecifikujeme, čo má tvoriť jeho základ a aký má mať vzhľad. Potom si popíšeme vizualizovanie výpočtových modelov v prostredí.

2.2.1 Základ

Hlavným komponentom základu systému má byť prehľadné menu. Od menu požadujeme, aby v ňom bolo umožnené prepnutie do prostredia určeného na vytvorenie automatu. Z neho sa používateľ má vedieť dostať do nasledovných prostredí:

- výpočet na slove
- algoritmy operácií
- normálne tvary
- ekvivalencia modelov

V systéme môže byť aj úvodná obrazovka, na ktorej by bolo uvítanie používateľa a predstavenie systému.

2.2.2 Vizualizácie

Aby bol náš systém dostatočne názorný, potrebujeme vhodným spôsobom vizualizovať výpočtové modely, výpočty a konštrukcie. Treba do systému naprogramovať alebo integrovať modul, ktorý nám túto požiadavku umožní realizovať. Chceme, aby bol v systéme zobrazovaný prechodový diagram aktuálne konštruovaného či používaného automatu. Tiež chceme, aby pri kroku algoritmu alebo výpočtu boli práve vytvorené alebo použité stavy a prechody farebne vyznačené.

2.2.3 Priamy a krokovací mód

Pre prostredie požadujeme možnosť voľby formy výstupu, buď priamej alebo krokovacej. Priama forma výstupu je taká, pri ktorej zadáme vstup a systém nám vráti priamo výsledok. Krokovacia forma je taká, počas ktorej nám systém dáva priebežne výsledky spolu s vizuálnym vyznačovaním a slovným popisom.

2.2.4 Výpočet

Prostredie určené pre výpočet na slove má obsahovať možnosť výberu výpočtového modelu, ktorý chceme zadať na vstupe. Po výbere modelu sa máme dostať do prostredia určeného na zadávanie automatu, v ktorom má byť aj možnosť zahájenia výpočtu. Po zahájení výpočtu chceme mať v prostredí k dispozícii súbor tlačidiel, pomocou ktorých sa vieme vo výpočte posúvať krok vpred, prípadne krok späť. Po dokončení výpočtu treba dať používateľovi informáciu o tom, či výpočet skončil akceptáciou alebo zamietnutím.

2.2.5 Konštrukčné algoritmy

Pre prostredie určené pre prevody a konštrukcie automatov, do ktorého sa vieme dostať z úvodnej obrazovky požadujeme, aby sme si vedeli zvoliť konštrukciu, ktorú chceme, aby nám systém vykonal. Po jej zvolení nám má byť umožnené zvoliť si výpočtový

model alebo jeho zadanie zo súboru, na ktorom chceme konštrukciu vykonať. Keď si zvolíme výpočtový model, chceme mať prostredie, v ktorom si môžeme zostrojiť model alebo modely, čo závisí od druhu zvolenej konštrukcie. Po zostrojení nám má byť umožnené zahájenie koštrukcie.

Po zahájení chceme, aby nám bol konštrukčný algoritmus znázorňovaný krok po kroku na vznikajúcom prechodovom diagrame. Chceme tiež, aby bol v tomto prostredí umiestnený slovný popis výkonávaného algoritmu a súbor tlačidiel pre posun kroku vpred, prípadne kroku späť. Po dokončení konštrukcie máme dostať na výstup kompletný prechodový diagram skonštruovaného modelu a tiež možnosť uložiť výsledok do súboru pre ďalšie použitie.

Chceme, aby bol výber konštrukčných algortimov rozdelený na operácie uzáverových vlastností, normálne tvary a konštrukcie ekvivalncie modelov.

Kapitola 3

Návrh systému

V tejto kapitole navrhneme hlavné komponenty systému, teda knižnice, štruktúry a prostriedky, prostredníctvom ktorých umožníme používateľovi komunikovať so systémom. Následne navrhneme používateľské prostredie. Nakoniec zadefinujeme jazyk pre vstup a výstup a vysvetlíme možnosti komunikácie používateľa so systémom.

3.1 Voľba paradigmy a jazyka

Ako prvé sme potrebovali zvoliť programovaciu paradigmu pre náš systém. Na výber sme mali funkcionálnu, procedurálnu a objektovo orientovanú.

Funkcionálne programovanie by malo niekoľko výhod oproti procedurálnemu a objektovo orientovanému. Jednou z nich sa javila krátkosť kódu. Vedeli by sme v ňom zadefinovať krátke funkcie, ktoré by sme poskladali a zrealizovali výpočet či algoritmus. Nevýhodou by ale bolo, že funkcie by boli ťažšie zrealizovateľné pre vstup a výstup, najmä pre prácu so súbormi a programovanie užívateľského prostredia.

Procedurálne programovanie bolo taktiež dobrým kandidátom pre paradigmu nášho systému z hľadiska výpočtu backendu, avšak prostredie by sa nám implementovalo zložitejšie ako pri objektovo orientovanom.

Rozhodli sme sa pre objektovo orientované programovanie, pretože vzhľadom k ostatným možnostiam je v tejto paradigme výhodou skutočné vytvorenie objektov, pre ktoré sa dá implementovať jednoduchá serializácia a deserializácia. Druhým dôvodom našej voľby je dostatočná podpora knižničných nástrojov pre tvorbu užívateľského rozhrania, napríklad pre štýlovanie prostredia či vykresľovanie automatov.

Existuje množstvo objektovo orientovaných programovacích jazykov. My sme si vybrali jazyk *Java*, pretože je jedným z najviac zabehnutých a disponuje množstvom knižníc a nástrojov, ktoré nám budú pri tvorbe systému nápomocné.

3.2 Použité návrhové vzory

3.2.1 Singleton

Návrhový vzor *Singleton* zaraďujeme medzi vytvárajúce návrhové vzory. Jeho hlavným účelom je uistiť sa, že trieda má iba jednu inštanciu. Okrem toho poskytuje všetkým ostatným častiam systému centralizovaný prístup k jeho inštancii. Návrhový vzor môžeme aplikovať napríklad aj vtedy, keď systém vyžaduje, aby bola inštancia rozšíriteľná dedením bez zmeny v kóde.

Predveďme si jeho význam na príklade. Majme operačný systém, napríklad *Linux*. V *Linuxe* je pre zobrazovanie okien aplikácií potrebný správca okien (angl. window manager). Ten by ale mal byť práve jeden. Keby ich bežalo súčasne viacero, vznikali by konflikty týkajúce sa správania otvorených okien aplikácií.

3.2.2 Factory method

Factory method zaraďujeme, podobne ako *Singleton*, do triedy vytvárajúcich návrhových vzorov. Jeho úlohou je vytvoriť skupinu tried so spoločným rozhraním a nechať na ďalšiu triedu, aby rozhodla, že ktorá z vytvorenej skupiny tried sa má inštancovať. *Factory method* môžeme použiť vtedy, keď trieda vopred nevie, ktorej triedy objekty bude vytvárať.

3.2.3 Mediator

Mediator zaraďujeme, na rozdiel od predchádzajúcich, do triedy návrhových vzorov týkajúcich sa správania programu. Dôležitý je najmä v situáciách, kedy by mohlo vznikať úzke prepojenie medzi triedami objektov. To ale chceme podľa princípov objektovo orientovaného návrhu eliminovať.

Nech máme definovaných niekoľko tried, z toho každá z nich obsahuje ostatné ako svoje atribúty, potom pri odstránení jednej z nich by sme ju museli odstrániť zo všetkých ostatných a aj jej prípadné správanie v metódach ostatných tried. *Mediator* rieši tento problém tak, že dá triedy objektov do spoločnej skupiny a zariaďuje komunikáciu medzi nimi tak, aby triedy o sebe nevedeli.

3.2.4 MVC

Návrhový vzor *MVC*, celým názvom *Model-View-Controller*, nemá zaradenie do triedy návrhových vzorov, ako majú napríklad *Singleton*, *Factory method* alebo *Mediator*. *MVC* špecifikuje, že keď je aplikácia navrhnutá podľa neho, tak pozostáva z modelu dát, zobrazovania informácie a kontroly informácie. Každá z týchto entít musí byť oddelená od zvyšných.

Model reprezentuje dynamické dáta. Neobsahuje logiku zobrazovania informácie, ktorú nesie. Na podnet vie obnoviť dáta, ktoré nesie. *Modelom* môže byť napríklad konečný automat, ktorý vie obnoviť svoju štruktúru.

View reprezentuje entitu, ktorej úlohou je zobraziť používateľovi informáciu obsiahnutú v *Modele*. *View* má informáciu o tom, ako pristupovať k dátam *Modelu*, ale nevie, akú informáciu dáta nesú. Príkladom *View* môže byť zobrazovanie dát pomocou grafov.

Controller reprezentuje entitu návrhového vzoru spájajúcu *Model* a *View*. Jeho úlohou je čakať na signál od používateľa, ktorý chce získať obnovené dáta od *Modelu*. Keď *Controller* príjme takýto signál, vykoná príslušnú reakciu. Reakcia je volaním určitej metódy *Modelu*. Keď *Model* vykoná volanú metódu, *Controller* podľa jej výsledku aktualizuje *View*.

3.3 Použité technológie

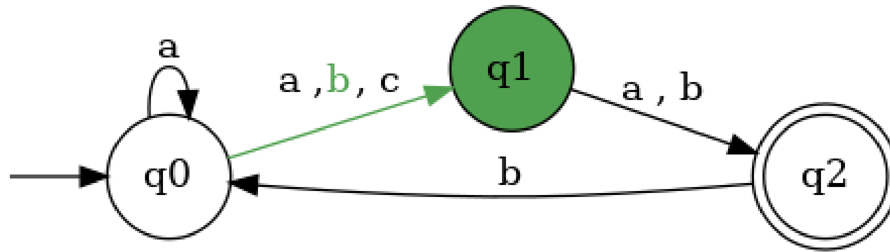
3.3.1 Graphviz

Graphviz je softvér slúžiaci na vizualizáciu grafov. Vizualizácia grafov je spôsob prezentácie informácie uloženej v grafe. Prezentácia je realizovaná pomocou diagramov, kde vrcholy sú znázorňované ako kruhy, obdĺžniky čí iné geometrické útvary a hrany sú znázorňované ako šípky spájajúce vrcholy. Vizualizácia grafov má množstvo aplikácií v praxi, najmä v sieťach, bioinformatike, softvérovom inžinierstve, strojovom učení či teórii formálnych jazykov.

Graphviz poskytuje možnosť napísať popis grafu do jednoduchého textového súboru v jazyku *Dot*. Z textového súboru následne postupnosťou úprav prevedie popis grafu na vizualizáciu diagramu. Vizualizáciu je potom možné dostať v grafických formátoch ako *PNG*, *SVG* či *GIF*, alebo vo forme dokumentu ako *PDF* či *Postscript*. *Graphviz* tiež poskytuje interaktívny prehliadač grafu, kde nám je umožnené vidieť prezentáciu diagramu už priebežne počas konštrukcie grafu.

Ukážku zakódovaného NKA v jazyku *Dot* môžeme vidieť v listingu 3.1. V prvom riadku kódovania môžeme vidieť kľúčové slovo *digraph*, ktoré znamená, že zostrojovaná štruktúra bude orientovaný graf (z angl. *directed graph*). Za kľúčovým slovom *digraph* nasleduje názov grafu, v našom prípade *fsm*.

Tým sa dostávame do tela kódovania a ako prvé si môžeme všimnúť kľúčové slovo *rankdir*, ktoré vyjadruje orientáciu konštrukcie diagramu. Hodnota *LR* znamená od ľavej do pravej strany. Nasleduje definícia vrcholov grafu, kde začíname menom vrcholu. Do hranatých zátvoriek vkladáme atribúty vrcholu tak, ako má byť znázornený vo výslednom diagrame. Pre naše účely potrebujeme definovať aj vrchol, ktorý nezobrazíme žiadnym spôsobom. Je to kvôli počiatočnému stavu automatu, do ktorého



Obr. 3.1: Výsledok príkazu dot -Tpng fsm.gv -o fsm.png

smeruje orientovaná hrana od nikadiaľ. Po definícii vrcholov nasleduje definícia hrán medzi nimi. Vieme ich v hranatých zátvorkach pomenovať a priradiť im atribúty tak, ako majú byť zobrazené. Takto definovaný graf následne uložíme do súboru s príponou *.gv*.

```
1  digraph fsm
2  {
3    rankdir=LR;
4    init [shape = none, label = ""];
5    q0 [shape = circle];
6    q1 [shape = circle, style = filled, fillcolor = "#50A14F"];
7    q2 [shape = doublecircle];
8    init -> q0
9    q0 -> q0 [label = "a"];
10   q0 -> q1 [label = <a ,<font color="#50A14F">b</font>, c>, color = "#50
         A14F"];
11   q1 -> q2 [label = "a , b"];
12   q2 -> q0 [label = "b"];
13 }
```

Listing 3.1: Ukážka kódovania NKA v jazyku Dot

Systém *Graphviz* ponúka aj jednoduchý nástroj do príkazového riadku s názvom *dot*, v ktorom vieme zo súboru s grafom v jazyku *Dot* priamo dostať zobrazený diagram. Jednoduchým príkladom takéhoto prevodu je príkaz *dot -Tpng fsm.gv -o fsm.png*, ktorý nám výsledok prevodu grafu definovaného v súbore *fsm.gv* zapíše do súboru *fsm.png*.

Výsledok prevodu grafu definovaného v listingu 3.1 pomocou príkazu popísaného vyššie môžeme vidieť na obrázku 3.1.

3.3.2 GraphViz Java API

Knižnica *GraphViz Java API* [12] poskytuje rozhranie pre jazyk *Java*. Umožňuje priamo z programu konštruovať textový súbor pre jazyk *Dot* a následne volať príkaz *dot*. Volaním príkazu *dot* sa zapíše zobrazenie grafu do súboru podľa zvoleného formátu.

Príklad použitia knižnice *GraphViz Java API* je možné vidieť v listingu 3.2. V príklade vytvoríme jednoduchý graf s tromi vrcholmi a dvomi hranami. Graf zapíšeme pomocou triedy *GraphViz* do súboru `/tmp/out.png`.

```
1   GraphViz gv = new GraphViz();
2   gv.addln(gv.start_graph());
3   gv.addln("A -> B;");
4   gv.addln("A -> C;");
5   gv.addln(gv.end_graph());
6   File out = new File("/tmp/out.png");
7   gv.writeGraphToFile(gv.getGraph(gv.getDotSource(), "png", "dot"), out);
```

Listing 3.2: Použitie knižnice GraphViz Java API

3.3.3 Maven

Maven [3] je nástroj slúžiaci na spravovanie softvérových projektov vytvorený spoločnosťou *Apache*. Nástroj je založený na spôsobe spravovania známeho pod názvom *POM* (z angl. project object model). Pomocou *Mavenu* nám je pri programovaní systému umožnené sťahovať potrebné knižnice alebo doplnky z online repozitára *MvnRepository* [6]. Celá informácia o vytváranom projekte je uložená v súbore s názvom *pom.xml*. Príklad pridania prerekvizity dostupnej v *MvnRepository* môžeme vidieť v listingu 3.3. Kód nám stačí pridať do súboru *pom.xml* a knižnica sa nám automaticky stiahne. Po stiahnutí ju môžeme ihneď používať.

```
1   <dependencies>
2           <dependency>
3               <groupId>org.openjfx</groupId>
4               <artifactId>javafx-controls</artifactId>
5               <version>11.0.1</version>
6           </dependency>
7   </dependencies>
```

Listing 3.3: Pridanie knižnice JavaFX pomocou nástroja Maven

3.3.4 Scene Builder

Scene Builder je nátroj určený na tvorbu grafického prostredia, ktorý umožňuje jednoducho dizajnovať *JavaFX* aplikácie. Nástroj je navrhnutý na štýl „drag and drop”, čo znamená, že používateľ môže pridať všetky komponenty prostredia ich potiahnutím z menu. Vytvárané prostredie je ukladané do súboru vo formáte *FXML*. Hlavnou výhodou tohoto nástroja je, že používateľ bez nutnosti programovania a spúšťania programu

vidí, aký má jeho vyvíjané prostredie vzhľad. Ďalšou výhodou je, že *FXML* súbory je možné kombinovať. Preto keď chce používateľ zmeniť za behu programu niektorú časť prostredia za inú, stačí mu iba na miesto zmeny načítať nový *FXML* súbor.

3.3.5 JavaFX a JFoenix

JavaFX [14] je platforma, ktorá obsahuje balíčky grafických prvkov a ktorej účelom je poskytnúť vývoj multiplatformových aplikácií pre rôzne operačné systémy, napríklad *Windows*, *Linux* či *Mac*. Pôvodne túto platformu vyvíjala spoločnosť *Sun Microsystems*, no momentálne je vyvíjaná spoločnosťou *Oracle* a je voľne dostupná. *JavaFX* bola dostupná v balíčku *OpenJDK*, no dnes je vyvíjaná ako samostatný balíček *OpenJFX*, ktorý je možné buď stiahnuť a nainštalovať, alebo importovať cez softvér ako napríklad *Maven* či *Gradle*.

JFoenix [10] je modul, ktorý obsahuje, podobne ako *JavaFX*, balíčky grafických prvkov pre programovanie grafických aplikácií v jazyku *Java*. Tento modul používa štýl, ktorý je známy aj ako „materiálny dizajn”, ktorý má zabezpečiť jednoduchý a moderný vzhľad aplikácie.

3.4 Jazyk pre vstup a výstup

Na to, aby mohol používateľ zadať vstupný automat a naopak prijímať výstup niektorej z konštrukcií si potrebujeme zadefinovať vhodný formát, ktorý budú používateľ a systém používať. Hlavným dôvodom poskytnutia formátu je, aby systém vedel jednoducho rozparsovať vstup a vytvoriť z neho očakávané objekty alebo naopak naformátovať výstup. Druhým dôvodom je, aby mal používateľ definované konvencie, ktorých sa má držať pri vytváraní vstupu, a aby vedel porozumieť výstupu.

Pre spôsob reprezentácie vstupu a výstupu pri práci so súbormi existujú dva známe formáty a to *XML* a *Json*. My sme zvolili formát *Json*, pretože sa javil jednoduchším a vhodnejším pre kódovanie automatov. Kódovanie automatov v jazyku *Json* je zrejmé z listingu 3.4.

3.5 Modularita

Systém sme rozdelili na knižnicu *libfsm*, ktorou sa budeme bližšie zaoberať v nasledujúcej sekcii, a na používateľské prostredie, ktorým sa budeme zaoberať v sekcii 3.7. Po implementovaní prostredia systému jednoducho integrujeme knižnicu *libfsm*, ktorá obsahuje štruktúry automatov, algoritmy na nich a pomocné štruktúry. Do systému bude možné integrovať nové knižnice vytvorené podobne ako knižnica *libfsm*. Napríklad by sme mohli vytvoriť knižnicu *libpda* pre zásobníkové automaty.

```
1  {
2   "type" : "DFA",
3   "alphabet" : ['a', 'b'],
4   "states" : ["1", "2"],
5   "transitions" : [
6    {
7     "source" : "1",
8     "symbol" : 'a',
9     "destination" : "1"
10   },
11   {
12    "source" : "2",
13    "symbol" : 'a',
14    "destination" : "2"
15   },
16   {
17    "source" : "1",
18    "symbol" : 'b',
19    "destination" : "2"
20   },
21   {
22    "source" : "2",
23    "symbol" : 'b',
24    "destination" : "2"
25   },
26  ],
27  "initial" : "1",
28  "final_states" : ["2"]
29 }
```

Listing 3.4: Ukážka kódovania DKA v jazyku Json

3.6 Knižnica libfsm

Knižnica *libfsm* tvorí prídavnú časť backendu systému. V tejto knižnici sú implementované triedy reprezentujúce konečné automaty, základné entity konečných automatov a konštrukčné algoritmy. Táto knižnica obsahuje aj podporné algoritmy pre prácu s množinami, keďže *Java* nemá ich natívnu podporu. V nasledovných podsekciách bližšie popíšeme triedy a balíčky, ktoré knižnica *libfsm* obsahuje.

3.6.1 Reprezentácia základných entít

Predtým, než sme definovali reprezentáciu konečných automatov, bolo nutné zvoliť reprezentáciu entít, s ktorými automaty pracujú.

Prvou základnou entitou automatu je stav. Jednou z možností, ako reprezentovať bolo definovať triedu, ktorá by obsahovala meno stavu a informáciu o tom, či je stav akceptačný, alebo nie. Výhodou by bolo jednoduché rozšírenie tejto triedy v prípade, že by sme si v stave chceli v budúcnosti pamätať aj ďalšie informácie. Pre jednoduchosť sme ale zvolili reprezentáciu stavu ako reťazec znakov. Informácia o tom, či je stav akceptačný, je obsiahnutá v automate. V reťazci znakov máme možnosť zakomponovať meno stavu aj prípadnú doplňujúcu informáciu. Na stav máme špeciálnu požiadavku — meno stavu môže byť ľubovoľný neprázdny reťacez znakov, ale nesmie mať prefix „<init" ani „<final". Tie vyhradzujeme z praktických dôvodov ako špeciálne.

Ďalšiou základnou časťou konečného automatu je abeceda. V teórii formálnych jazykov je abeceda definovaná ako neprázdna množina symbolov. Pre praktické účely je ale táto definícia moc voľná, preto sme ju mierne sprísnili. Abeceda je v našom systéme zúžená na alfanumerické symboly.

Hlavným dôvodom tejto voľby bolo, aby sa neplietli symboly abecedy so symbolom prázdneho slova a hlavne kvôli jednoduchosti systému. Za symbol pre prázdne slovo sme nezvolili ϵ, pretože sa nenachádza na bežnej klávesnici. Používateľ by si musel buď pamätať symbolické číslo v sústave *Unicode*, alebo ho vyhľadať na internete. Namiesto epsilónu sme pre symbol prázdneho slova zvolili znak '$'.

3.6.2 Balíček machines

V systéme treba vhodným spôsobom reprezentovať konečný automat. Keďže sme zvolili objektovo orientované programovanie, mali sme na výber dva rozumné spôsoby reprezentácie. Prvou možnosťou bolo reprezentovať automat pomocou rozhrania, ktoré by definovalo spoločné metódy. Tie by implementovali triedy *DFA* a *NFA*. Tieto triedy popíšeme neskôr v tejto podsekcii. Druhou možnosťou bolo použiť abstraktnú triedu, od ktorej by triedy *DFA* a *NFA* dedili spoločné premenné a metódy. Abstraktná trieda by tiež definovala abstraktné metódy, ktoré by bolo nutné implementovať v podtriedach *DFA* a *NFA*.

My sme zvolili druhý prístup, zadefinovali sme abstraktnú triedu *FSM*. *FSM* tvorí spoločný základ pre deterministické a nedeterministické konečné automaty. Obsahuje ich spoločné entity, ktorými sú vstupná abeceda, množina stavov, počiatočný stav a množina akceptačných stavov. Prechodovú funkciu majú triedy *DFA* a *NFA* rôznu, preto je v triede *FSM* vynechaná. Trieda *FSM* obsahuje aj statickú konštantu pre definíciu symbolu prázdneho slova, ktorý sme definovali v podsekcii 3.6.1. Balíček *machines* obsahuje triedy reprezentácie konečných automatov, ktoré teraz popíšeme bližšie.

Trieda FSM

Definujme abstraktnú triedu *FSM* (z angl. finite state machine), ktorá obsahuje nasledovné premenné a metódy:

- *EPSILON = '$'* - konštanta reprezentujúca symbol prázdneho slova
- *fsmType* - typ automatu
- *name* - meno automatu
- *alphabet* - množina vstupných symbolov, teda vstupná abeceda
- *states* - množina stavov automatu
- *initial* - počiatočný stav automatu
- *finalStates* - množina akceptačných stavov automatu
- *abstract addState(state)* - pridá stav *state* do množiny stavov
- *abstract removeState(state)* - odstráni stav *state* z množiny stavov
- *setName(name)* - nastaví meno automatu na *name*.
- *getName()* - vráti meno automatu
- *addStateFinal(state)* - pridá stav *state* do množiny akceptačných stavov
- *removeStateFinal(state)* - odstráni stav *state* z množiny akceptačných stavov
- *abstract renameState(oldName, newName)* - premenuje stav *oldName* na stav *newName*
- *abstract addTransition(source, symbol, destination)* - pridá do prechodovej funkcie prechod zo stavu *source* na symbol *symbol* do stavu *destination*
- *abstract removeTransition(source, symbol, destination)* - odstráni z prechodovej funkcie prechod zo stavu *source* na symbol *symbol* do stavu *destination*
- *abstract getAllTransitionsFrom(source)* - vráti množinu dvojíc *(symbol, destination)* takých, že automat prejde zo stavu *source* na symbol *symbol* do stavu *destination*.
- *abstract computeStep(states, symbol)* - vykoná krok výpočtu na slove zo stavov *states* na symbol *symbol* a vráti stavy v ktorých sa nachádza po jeho vykonaní
- *abstract compute(word)* - vykoná výpočet na slove *word* a vráti výsledok výpočtu

- *abstract epsilonClosure(states)* - vráti epsilonový chvost pre každý stav v *states*
- *abstract isCorrectlyDefined()* - vráti informáciu o tom, či je inštancia *DFA* alebo *NFA* dobre definovaná
- *abstract setAlphabet(alphabet)* - nastaví abecedu automatu na *alphabet*
- *abstract writeToFile(file)* - zapíše svoju inštanciu do súboru podľa definovaného formátu
- *readFromFile(file)* - načíta automat zo súboru podľa definovaného formátu a vráti načítaný automat

Trieda DFA

Deterministický konečný automat reprezentujeme ako potomka abstraktnej triedy *FSM*. Táto trieda sa nazýva *DFA* (z angl. deterministic finite automaton). *DFA* má oproti *FSM* skutočne definovanú prechodovú funkciu *transitions*, ktorú reprezentujeme ako slovník, pričom kľúčom je dvojica *(stav, symbol)* a hodnotou je *stav*. *DFA* implementuje všetky abstraktné metódy triedy *FSM*.

Trieda NFA

Nedeterministický konečný automat reprezentujeme, podobne ako deterministický, ako potomka triedy *FSM*. Trieda sa nazýva *NFA* (z angl. nondeterministic finite automaton). Oproti *DFA* sa *NFA* líši v definícii prechodovej funkcie a implementácii abstraktných metód triedy *FSM*. Prechodovú funkciu definujeme ako slovník, kde kľúčom je dvojica *(stav, symbol)* a hodnotou je množina stavov, do ktorých sa vieme dostať na jeden krok výpočtu zo stavu *stav* na symbol *symbol*.

3.6.3 Balíček conversions

Balíček *conversions* obsahuje algortimy prevodu medzi výpočtovými modelmi regulárnych jazykov. Každý algoritmus má svoju triedu, ktorá obsahuje:

- *steps* - obsahuje dvojice (číslo kroku, popis kroku)
- *getSteps()* - vráti premennú *steps*
- *apply(fsm)* - aplikuje algoritmus na *fsm* a vráti výsledok aplikácie

V našom systéme je cieľom práce pracovať s konečnými automatmi, preto balíček *conversions* obsahuje iba triedu *NFAtoDFA*.

3.6.4 Balíček normalForms

V tomto balíčku sú obsiahnuté algoritmy pre prevody automatov do ich normálnych tvarov. Je rozdelený do dvoch podbalíčkov, jeden pre *DFA* a druhý pre *NFA*. Každý algoritmus prevodu do normálneho tvaru obsahuje rovnaké premenné a metódy ako triedy v balíčku *conversions*. Systém obsahuje nasledovné triedy pre algoritmy prevodu do normálnych tvarov:

- *Reachable* - obsahuje algoritmus, ktorý zo zadaného *NFA* alebo *DFA* odstráni nedosiahnuteľné stavy
- *EpsilonFree* - obsahuje algoritmus odepsilónovania *NFA*
- *Piggy* - obsahuje algoritmus prevodu *NFA* do „prasiatkového”[1] normálneho tvaru

3.6.5 Balíček operations

Balíček *operations* obsahuje konštrukčné algoritmy uzáverových operácií triedy regulárnych jazykov. Balíček je rozdelený na unárne a binárne operácie. Každá operácia má svoju vlastnú triedu.

Triedy pre unárne operácie obsahujú rovnaké premenné a metódu ako triedy v balíčkoch *conversions* a *normalForms*. Avšak triedy pre binárne operácie sa líšia v metóde *apply(fsm)*, ktorá je v nich definovaná nasledovne:

- *apply(first, second)* - aplikuje binárnu operáciu na automaty *first* a *second* a vráti výsledok aplikácie

Balíček obsahuje tieto triedy pre konštrukčné algoritmy uzáverových operácií:

- *binary/Concatecation* - algoritmus pre zreťazenie
- *binary/Intersection* - algoritmus pre prienik
- *binary/Union* - algoritmus pre zjednotenie
- *unary/Complement* - algoritmus pre doplnok
- *unary/Reverse* - algoritmus pre reverz

[1] Podľa [5] je NKA v prasiatkovom normálnom tvare, keď „má práve jeden akceptačný stav, počas výpočtu sa nikdy nevráti do začiatočného stavu a zastane okamžite po dosiahnutí akceptačného stavu”.

3.6.6 Balíček helpers

Balíček obsahuje pomocné triedy, ktoré zjednodušujú možnosti dostupné v *Jave*. V balíčku nájdeme nasledovné triedy:

- *Pair* - trieda reprezentujúca dvojicu
- *Triple* - trieda reprezentujúca trojicu
- *Sets* - trieda obsahujúca statické metódy vykonávajúce množinové operácie na generickej triede *Set<T>*

3.7 Užívateľské prostredie

Prostredie je rozdelené na štyri sekcie, každé vo vlastnej karte. Prvé je určené pre výpočet na slove, druhé pre konštrukcie uzáverových operácii, tretie pre konštrukciu normálnych tvarov a štvrté pre ekvivalenciu výpočtových modelov. Každá sekcia obsahuje úvodné menu, prostredie pre zostrojenie vstupného automatu a prostredie pre realizáciu výpočtu alebo konštrukčného algoritmu. V prostredí pre zadávanie môže užívateľ zostrojiť automat pomocou súboru tlačidiel, ktoré sú nasledovné:

- *Add S* - umožňuje pridať stav do automatu tak, že užívateľovi otvorí dialógové okno, kde zadá meno stavu a informáciu, či je stav akceptačný
- *Remove S* - umožňuje odobrať stav z automatu tak, že otvorí dialógové okno, kde užívateľ zadá meno stavu, ktorý chce z automatu odstrániť
- *Rename S* - umožňuje premenovať stav automatu tak, že otvorí dialógové okno, kde užívateľ zadá meno stavu, ktorý chce premenovať a nové meno stavu
- *Add T* - umožňuje pridať prechod z automatu podobne cez dialógové okno, kde užívateľ zadá zdroj, symbol a cieľ prechodu, ktorý sa má pridať do prechodovej funkcie
- *Remove T* - umožňuje odstrániť prechod z prechodovej funkcie automatu cez dialógové okno, v ktorom užívateľ zadá zdroj, symbol a cieľ prechodu, ktorý má byť odstránený
- *Alphabet* - umožňuje nastaviť abecedu automatu cez dialógové okno, v ktorom užívateľ zadá alfanumerické znaky oddelené čiarkou
- *Initial* - umožňuje nastaviť počiatočný stav automatu cez dialógové okno, v ktorom užívateľ zadá meno stavu

- *Clear* - odstráni celý aktuálne zostrojený automat a nechá prázdny panel pre možnosť zostrojenia nového automatu
- *Export* - umožňuje export aktuálneho automatu do súboru, po kliknutí dá možnosť voľby súboru, do ktorého sa má automat uložiť
- *Rename* - umožňuje premenovať zadávaný automat

V nasledovných podsekciách si bližšie popíšeme návrh sekcií systému.

3.7.1 Výpočet

V menu pre výpočet si môžeme zvoliť, či chceme vykonať výpočet na slove pomocou *NFA* alebo *DFA*. Taktiež máme možnosť zvoliť si import automatu zo súboru alebo si vybrať z vopred zostrojených príkladov.

Po výbere automatu pre výpočet sa dostávame do prostredia, v ktorom je možné zostrojiť nový či modifikovať načítaný automat. V tomto prostredí je zobrazovaný prechodový diagram prislúchajúci práve zostrojovanému automatu, možnosť výberu priamej alebo krokovacej formy výstupu a pole pre zadanie vstupného slova pre výpočet. Z prostredia sa vieme ľahko dostať do úvodného menu pomocou tlačidla *Back*. Po výbere formy výstupu a vyplnení poľa pre vstupné slovo môžeme zahájiť výpočet na zadanom slove pomocou tlačidla *Run*.

Zahájenie výpočtu nás presunie do nového prostredia, v ktorom je nainicializovaný náš skoštruovaný či načítaný automat s vyznačenou počiatočnou konfiguráciou. Prvý znak ešte neprečítanej časti vstupného slova je vyznačený červenou farbou. Počiatočný stav, špeciálne pri *NFA* aj s jeho epsilónovým chvostom, je vyznačený zelenou farbou. Rovnako zelenou farbou sú vyznačené tie prechody delta funkcie, kde z aktuálnych stavov na prvý symbol neprečítanej časti slova existuje prechod v delta funkcii. Pomocou tlačidla *Next* sa posunieme pri priamej forme výstupu na koniec výpočtu a pri krokovacej forme výpočtu sa posunieme o krok výpočtu. Po ukončení výpočtu dostaneme informáciu o tom, či automat slovo akceptoval, alebo zamietol. Navyše nám ostanú zvýraznené tie stavy, v ktorých sa automat nachádzal po skončení výpočtu.

3.7.2 Operácie

Úvodné menu pre operácie obsahuje možnosť výberu všetkých implementovaných algoritmov pre operácie, ktoré sme vymenovali v podsekcii 3.6.5.

Po zvolení operácie sa dostávame do menu pre zvolenie automatu, na ktorom chceme operáciu vykonať. Na výber máme buď iba *DFA*, alebo *DFA* aj *NFA*, podľa toho, pomocou ktorého automatu sme algoritmus implementovali. Okrem toho máme možnosť importu zo súboru a možnosť výberu z vopred zostrojených príkladov.

Keď si zvolíme alebo načítame automat, tak sa dostaneme do prostredia pre konštrukciu automatu rovnakého, ako pri výpočte. Keď máme automat skonštruovaný, tak pri voľbe unárnej operácie môžeme rovno zahájiť konštrukciu pomocou tlačidla *Run*. Pri voľbe binárnej operácie máme jedinú možnosť pre pokračovanie a to dostať sa do prostredia výberu druhého automatu pomocou tlačidla *Next*. Výber a konštrukcia druhého automatu prebieha rovnakým spôsobom ako pri prvom.

Po skonštruovaní automatu (resp. automatov) si vyberieme formu výstupu a zahájime konštrukciu pomocu tlačidla *Run*. Po zahájení konštrukcie nás systém presunie do prostredia v ktorom prebieha samotný algoritmus. Toto prostredie pozostáva pri unárnej operácii z dvoch panelov pre vizualizáciu prechodových diagramov, z toho jeden pre pôvodný automat a druhý pre konštruovaný. Pri voľbe binárnej operácie prostredie pozostáva z troch panelov pre vizualizáciu, z toho prvé dva sú určené pre pôvodné automaty a tretí je určený pre konštruovaný. V prostredí je aj panel so slovným popisom vykonávania algoritmu. Každý krok algoritmu sa najprv vypíše zhrnutý v bode.

Na začiatku konštrukcie sa zobrazia pôvodné automaty. Klikaním na tlačidlo *Next* sa vždy vykoná jeden krok konštrukcie, ktorý sa znázorní na diagramoch a slovne sa popíše. Krok konštrukcie môže byť jedným z nasledovných typov a na prechodovom diagrame je znázorňovaný nasledovne:

- Pridanie symbolu do abecedy
- Pridanie stavu do množiny stavov - pridaný stav sa vyznačí zelenou farbou
- Pridanie prechodu do prechodovej funkcie - zdroj prechodu sa vyznačí zelenou farbou, hrana aj s jej ohodnotením sa vyznačí červenou farbou, cieľ prechodu sa vyznačí modrou farbou a špeciálne, ak sa zdroj rovná cieľu, vrchol sa vyznačí hnedou farbou
- Nastavenie počiatočného stavu - nastavený stav sa vyznačí zelenou farbou
- Pridanie stavu do množiny akceptačných stavov - pridaný stav sa vyznačí zelenou farbou

Navyše každá binárna operácia predpokladá disjunktné množiny stavov a odlišné mená vstupných automatov.

3.7.3 Normálne tvary

Menu v úvode karty pre normálne tvary máme, rovnako ako v úvodnom menu pre výpočet na slove, možnosť na výber *DFA*, *NFA*, import zo súboru alebo vopred vytvorený príklad.

Po zvolení v úvodnom menu sa dostávame do prostredia zadávania automatu na prevod, ktoré je rovnaké ako pri ostatných možnostiach. Tiež je prístupná možnosť voľby formy výstupu. Zahájenie konštrukcie prebieha tak, že klikneme na tlačidlo *Choose NF*, kde si vyberieme normálny tvar, na ktorý chceme zadaný automat previesť. Svoju voľbu potvrdíme a zároveň spustíme algoritmus prevodu pomocou tlačidla *Run*.

Po zahájení algoritmu prevodu sa nám zobrazí prostredie rovnaké ako pri vykonávaní unárnej operácie. To znamená dva panely. Jeden pre pôvodný automat a druhý pre konštruovaný. Prostredie tiež obsahuje textový popis vykonávania prevodu. Krok konštrukcie môže byť rovnakého typu a znázorňovaný rovnakým spôsobom ako pri uzáverových operáciách.

3.7.4 Ekvivalencia

V systéme je zatiaľ jediný prevod a tým je prevod *NFA* na *DFA*. V úvodnom menu pre ekvivalenciu modelov preto nájdeme v možnostiach výberu iba *NFA*, importu zo súboru alebo výber niektorého z príkladov.

Po výbere sa dostaneme do prostredia pre zadávanie automatu. Toto prostredie je rovnaké ako pri predchádzajúcich možnostiach. Keď máme automat zadaný, pomocou tlačidla *Choose model* dostaneme možnosť voľby modelu, na ktorý chceme náš automat previesť. Po zvolení možnosti zahájime prevod tlačidlom *Run*.

Keď zahájime prevod, dostaneme sa do prostredia výkonu algoritmu, ktoré je rovnaké, ako pri unárnych operáciach. Máme teda k dispozícii panel, na ktorom sa nám zobrazí vstupný automat, panel na ktorom sa postupne zobrazuje konštrukcia nového automatu. Máme k dispozícii aj logovací panel so slovným popisom algoritmu prevodu. Farebné označovanie diagramu je identické ako pri operáciách a ďalší krok prevodu dostaneme tlačidlom *Next*.

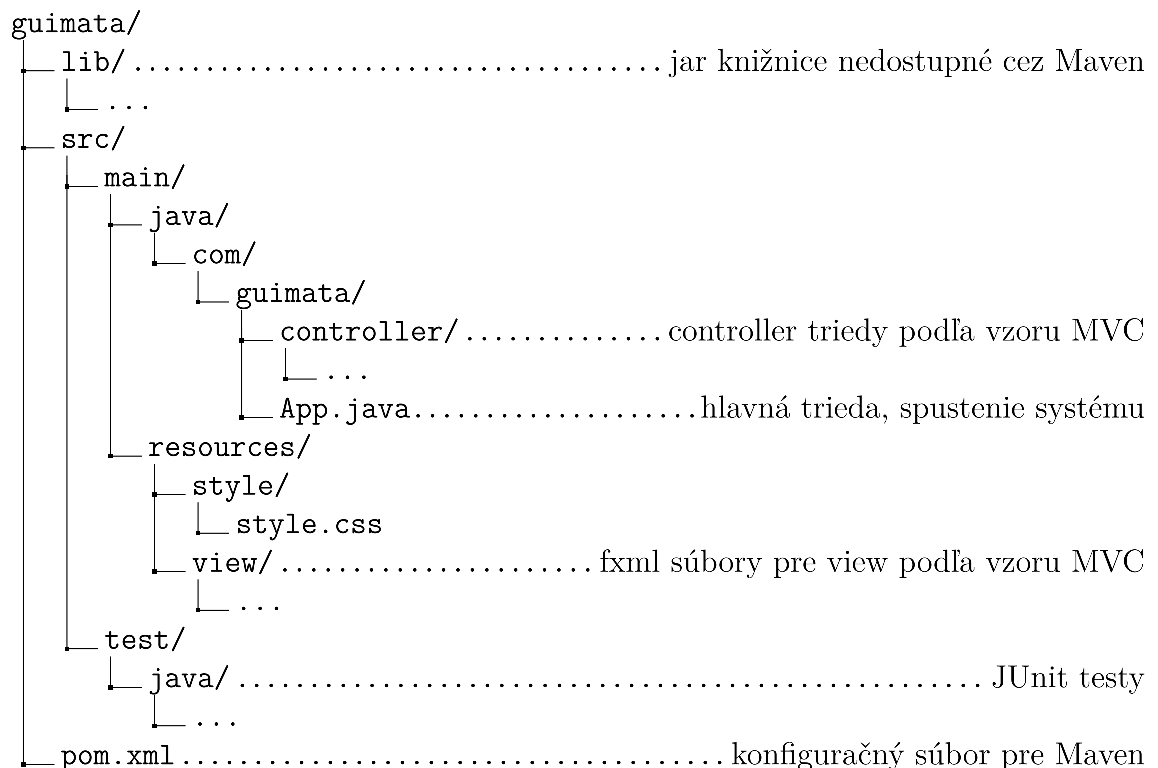
Kapitola 4

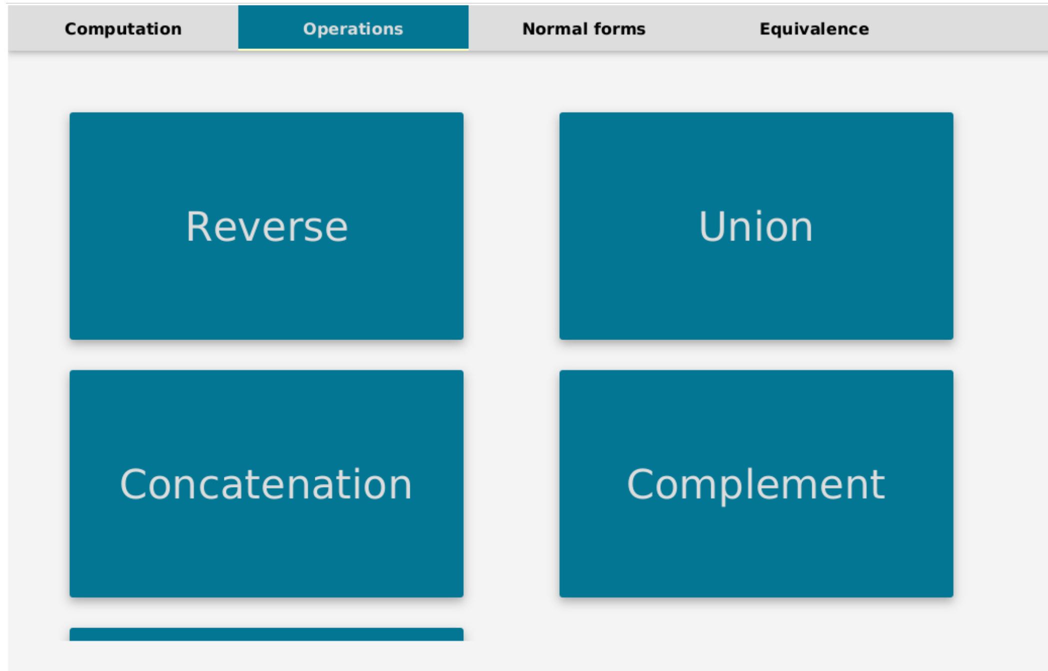
Implementácia

V tejto kapitole sa budeme zaoberať významnými časťami implementácie systému. Najprv si popíšeme súborovú štruktúru systému. Potom si ukážeme, ako sme použili návrhové vzory predstavené v sekcii 3.2. Pozrieme sa aj na niektoré zaujímavé časti kódu systému a nakoniec si povieme, ako by sa dal systém vylepšiť.

4.1 Štruktúra aplikácie

Organizácia súborov v systéme je urobená podľa vzoru aplikácie konfigurovanej cez nástroj *Maven* a tiež podľa návrhového vzoru *MVC*. Knižnice pre automaty, napríklad *libfsm*, pridávame do priečinku `guimata/lib/`. Kostra štruktúry je v nasledovnej stromovej štruktúre:





Obr. 4.1: Ukážka Main View a OpHome View

4.2 Použitie návrhového vzoru MVC

Systém je rozdelený na štyri sekcie: výpočet na slove, operácie, normálne tvary a ekvivalenciu. Tieto sekcie sú obsiahnuté v hlavnej sekcii s názvom *Main*. Pre každú z týchto sekcií sme osobitne použili návrhový vzor *MVC*. *Model* v našom systéme predstavujú automaty, konkrétne triedy *DFA* a *NFA*. Keď niektorý *Controller* dostane podnet od používateľa, tak naň zareaguje príslušnou metódou, pričom táto metóda je zameraná práve na určitú akciu, ktorú dá *Controller* vykonať automatu. Keď automat akciu vykoná, úlohou *Controllera* je podľa výsledku akcie vykonanej automatom aktualizovať *View*.

Pozrime sa teraz, ako sme implementovali návrhový vzor pre sekciu *Main*. V ďalšom texte budeme cesty k súborom písať relatívne pre *Controller* k zložke `guimata/src/main/java/com/guimata/controller` a pre *View* relatívne k zložke `guimata/src/main/resources/view`.

View pre sekciu *Main* sme zostrojili pomocou nástroja *Scene Builder* a po zostrojení sme ho uložili do súboru *common/Main.fxml*. *Main View* obsahuje triedu z knižnice *JFoenix* s názvom *JFXTabPane*. Ten obsahuje štyri prvky triedy *Tab*, nazývané aj ako karty. Každá z nich prislúcha jednej sekcii rozdelenia. *Main View* aj s kartami pre sekcie môžeme vidieť na obrázku 4.1.

Pre *Main View* sme implementovali *Controller* s názvom *MainController*, ktorý je uložený v súbore *common/MainController.java*. *MainController* sa v našom systéme

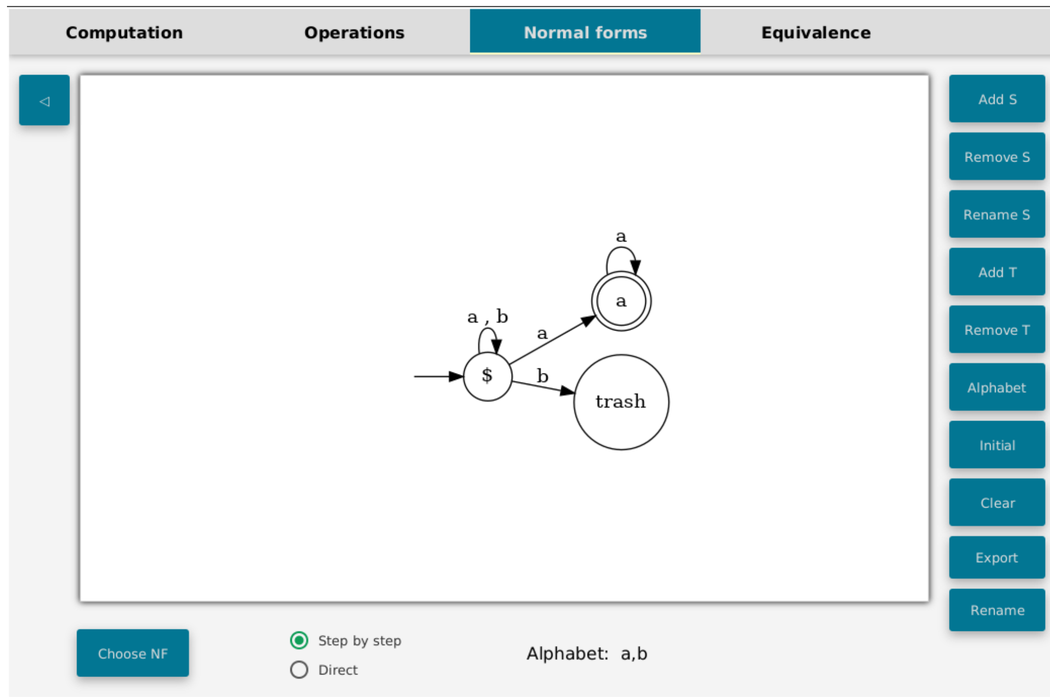
View	Controller
computation/CHome.fxml	computation/CHomeController.java
computation/CBuildFSM.fxml	computation/CBuildFSMController.java
computation/CRunFSM.fxml	computation/CRunFSMController.java
computation/CExamples.fxml	computation/CExamplesController.java
equivalence/EqHome.fxml	equivalence/EqHomeController.java
equivalence/EqBuildFSM.fxml	equivalence/EqBuildFSMController.java
equivalence/EqRunFSM.fxml	equivalence/EqRunFSMController.java
equivalence/EqExamples.fxml	equivalence/EqExamplesController.java
normalForms/NFHome.fxml	normalForms/NFHomeController.java
normalForms/NFBuildFSM.fxml	normalForms/NFBuildFSMController.java
normalForms/NFRunFSM.fxml	normalForms/NFRunFSMController.java
normalForms/NFExamples.fxml	normalForms/NFExamplesController.java
operations/OpHome.fxml	operations/OpHomeController.java
operations/OpChoiceFirstFSM.fxml	operations/OpChoiceFirstFSMController.java
operations/OpChoiceSecondFSM.fxml	operations/OpChoiceSecondFSMController.java
operations/OpBuildFirstFSM.fxml	operations/OpBuildFirstFSMController.java
operations/OpBuildSecondFSM.fxml	operations/OpBuildSecondFSMController.java
operations/OpRunUnaryFSM.fxml	operations/OpRunUnaryFSMController.java
operations/OpRunBinaryFSM.fxml	operations/OpRunBinaryFSMController.java
operations/OpExamples.fxml	operations/OpExamplesController.java

Tabuľka 4.1: Dvojice súborov podľa návrhového vzoru MVC

vie iba inicializovať a nereaguje priamo na žiadne podnety od používateľa. Jeho funkciou je inicializovať domovské menu pre každú sekciu rozdelenia a poskytovať základné panely kariet *Controllerom* určeným pre spravovanie im prislúchajúcim *Viewom*.

Pre sekciu výpočtu na slove máme dvojice (*View*, *Controller*) pre úvodné menu, prostredie zadávania automatu a prostredie realizácie výpočtu popísaných v sekcii 3.7. Tieto dvojice sú uložené v tabuľke 4.1. Úvodné menu je obsiahnuté v *CHome View*, konštrukčné prostredie v *CBuildFSM View* a prostredie realizácie výpočtu v *CRunFSM View*.

Pre sekcie ekvivalencie a normálnych tvarov máme rovnako ako pri komponente výpočtu dvojice (*View*, *Controller*) pre úvodné menu, zadávacie prostredie a prostredie realizácie prevodu. Všetky tieto prostredia sme popísali v sekcii 3.7. Na obrázku 4.2 môžeme vidieť *NFBuildFSM View*, kde si na pravom kraji môžeme všimnúť tlačidlá pre konštruovanie automatu, dole výber formy výstupu a tlačidlo výberu normálneho tvaru. Dvojice súborov prislúchajúce týmto dvojiciam môžeme vidieť v tabuľke 4.1.



Obr. 4.2: Ukážka NFBuildFSM view

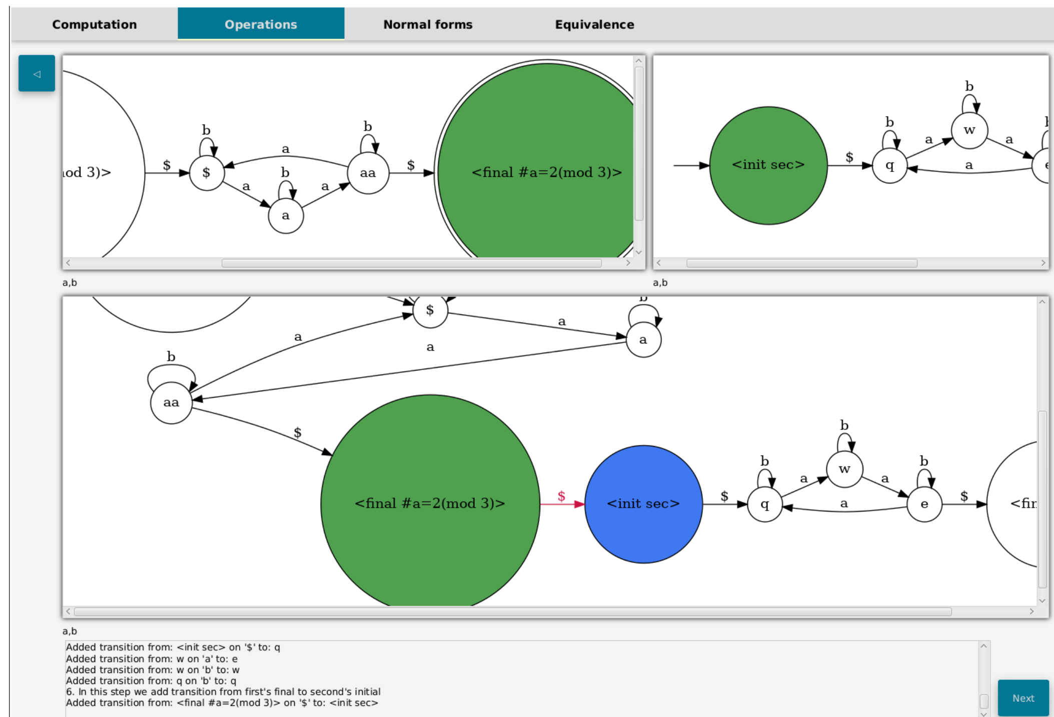
Sekcia pre operácie sa líši od predchádzajúcich v tom, že používateľ si v úvodnom menu nevyberá automat, ale operáciu, ktorú chce vykonať. *View* pre úvodné menu operácií sa nazýva *OpHome*. Líši sa aj v tom, že si volíme buď jeden alebo dva automaty, podľa toho, či sme si zvolili buď unárnu, alebo binárnu operáciu. *OpHome View* môžeme vidieť na obrázku 4.1.

Ak sme si zvolili unárnu operáciu, tak zadanie automatu si vyberáme v *OpChoiceSecondFSM View*, z ktorého sa dostaneme do *OpBuildSecond View*, v ktorom zadávame vstupný automat. Po zhotovení automatu a zahájení algoritmu konštrukcie unárnej operácie sa presunieme do *OpRunUnaryFSM View*, v ktorom prebieha samotný algoritmus.

Pri voľbe binárnej operácie si volíme dva automaty. Prvý v *OpChoiceFirstFSM View*, ktorý konštruujeme v *OpBuildFirstFSM View* a druhý v *OpChoiceSecondFSM View*, ktorý konštruujeme v *OpChoiceSecondFSM View*. Algoritmus konštrukcie nového automatu prebieha v *OpRunBinaryFSM View*, ktorý môžeme vidieť na obrázku 4.3. Zoznam dvojíc (*View*, *Controller*) je v tabuľke 4.1.

4.3 Použitie návrhových vzorov Mediator a Singleton

Každá zo sekcií je riadená niekoľkými *Controllermi*, jedným pre každé *View* v sekcii. V určitých prípadoch potrebujeme, aby *Controllery* vnútri sekcie medzi sebou komuni-



Obr. 4.3: Ukážka OpRunBinaryOperation view

kovali. Chceli sme to ale dosiahnúť bez vytvorenia úzkeho prepojenia tried. Preto sme pre každú sekciu vytvorili *ControllerMediator*. Pre výpočet je to *CControllerMediator*, pre ekvivalenciu *EqControllerMediator*, pre normálne tvary *NFControllerMediator* a pre operácie *OpControllerMediator*. Týmto môžu *Controllery* rovnakej sekcie spolu komunikovať bez toho, aby vedeli o vzájomnej existencii. Výhodou tohoto prístupu je jednoduchá extenzibilita systému. Navyše, každý *ControllerMediator* je navyše *Singleton*, aby sme predišli prípadným konfliktom.

V každej sekcii obsahuje *ControllerMediator* všetky *Controllery* danej sekcie a *MainController*. *MainController* je v každom *ControllerMediatore* preto, že mu poskytuje základné panely každej sekcie. V listingu 4.1 môžeme vidieť príklad použitia *CControllerMediatora* pri kliknutí v *CRunFSM View* na tlačidlo *Back*, pomocou ktorého sa vieme vrátiť do *CBuildFSM View*.

```
1   @FXML
2   void changeToFSMBuild(ActionEvent event) {
3       CControllerMediator.getInstance().loadCBuildFSM(machine);
4   }
```

Listing 4.1: Zavolanie CControllerMediatora z CRunFSM

4.4 Ukážky zaujímavých častí kódu

4.4.1 Vykresľovanie diagramu

Jednou z častí kódu systému, ktoré stoja za povšimnutie je použitie knižnice *GraphViz Java API* na vykreslenie diagramu počas zadávania, konštrukcie alebo výpočtu. Systém má pre tento účel definovanú statickú metódu *BuildFSM.updateDiagram(GraphViz graphViz, ScrollPane scrollPane)*, ktorej implementáciu môžeme vidieť v listingu 4.2. Metóda berie ako parametre už vytvorené objekty triedy *GraphViz* a *ScrollPane*. Do *scrollPane* vloží obrázok vykresleného diagramu. Knižnica *GraphViz Java API* je uložená v priečinku pathguimata/lib/.

```
1   public static void updateDiagram(GraphViz graphViz, ScrollPane
        scrollPane) {
2     String type = "png";
3     String represantionType = "dot";
4     File out = new File("/tmp/machine."+type);
5     graphViz.writeGraphToFile(graphViz.getGraph(graphViz.getDotSource(),
          type, represantionType), out);
6
7     Image diagramImage = new Image(out.toURI().toString());
8     ImageView diagramView = new ImageView(diagramImage);
9     StackPane imageHolder = new StackPane(diagramView);
10    imageHolder.setStyle("-fx-background-color: #ffffff");
11    imageHolder.setMinWidth(scrollPane.getPrefWidth());
12    imageHolder.setMinHeight(scrollPane.getPrefHeight());
13    scrollPane.setContent(imageHolder);
14  }
```

Listing 4.2: Implementácia metódy vykreslenia diagramu

4.4.2 Realizácia operácie s priamym výstupom

Druhou časťou kódu, ktorú spomenieme, je realizácia priameho výstupu pri vykonávaní unárnej operácie. Pre operácie máme v knižnici *libfsm* definovaný *enum OperationType*, ktorý po zvolení operácie v úvodnom menu *OpHomeController* zapíšeme do premennej triedy *OpControllerMediator*. Pri realizovaní operácie si od neho tento typ vypýtame, a podľa neho sa *UnaryOperationFactory* rozhodne, ktorá operácia sa má zo vstupného automatu skonštruovať. Časť popísaného kódu môžeme vidieť v listingu 4.3. V prípade nečakanej chyby vytvoríme prázdnu inštanciu triedy *NFA*. Volaný kód predpokladá korektne skonštruovaný vstupný automat.

```
1       void runOperation(FSM machine, boolean direct) {
2           this.machine = machine.copy();
3           this.direct = direct;
4           operation = new UnaryOperationFactory().get(OpControllerMediator.
                getInstance().getType());
5           steps = operation.getSteps();
6           resultMachine = operation.apply(machine);
7           ...
8       }
9   }
```

Listing 4.3: Realizácia unárnej operácie s priamym výstupom

Podobným spôsobom realizujeme spustenie algoritmu binárnej operácie a spustenie prevodu do normálneho tvaru. Po implementácii novej operácie vieme jednoducho vložiť nový *case* s typom danej operácie.

4.5 Zlepšenia do budúcnosti

Systém je zatiaľ pomerne málo rozsiahly a vieme si predstaviť množstvo rozšírení, o ktoré by sa dal obohatiť. Prvou možnosťou je pridať tlačidlo pre krok späť pri realizácii výpočtu a konštrukčných algoritmov. Druhým rozšírením je implementovanie algoritmu, ktorý spája nerozlíšiteľné stavy DKA. Jeho spojením s algoritmom odstraňovania nedosiahnuteľných stavov DKA by bola umožnená minimalizácia DKA. Ďalšiou možnosťou je implementovať moduly knižnice pre výpočtové modely regulárnych jazykov — regulárne gramatiky, regulárne výrazy alebo dvojsmerné automaty a integrovať ich do systému.

Nad systémom je možné vytvoriť nadstavbu zochovávajúcu návrhový vzor *MVC*, kde by si používateľ mohol zvoliť triedu jazykov s ktorými chce pracovať. Do nadstavby by bolo následne možné pridať možnosť voľby pre bezkontextovú, kontextovú či inú triedu jazykov a implementovať a integrovať knižnice s príslušnými výpočtovými modelmi a algoritmami na nich.

Kapitola 5

Dokumentácia

V tejto kapitole najprv uvedieme softvérové prerekvizity, potom prejdeme na inštaláciu a spustenie systému. Na záver kapitoly si ukážeme predvedenie použitia systému na príklade prevodu zadaného NKA na ekvivalentný DKA.

5.1 Prerekvizity

Java 11

Prvou prerekvizitou systému je platforma *Java*. Systém používa verziu *Java 11.0.3*. Je potrebné mať nainštalovanú buď tú verziu, alebo novšiu. Platforma je dostupná na stránke `https://www.oracle.com/technetwork/java/javase/downloads/index.html` pre *Windows*, *Linux* a *Mac*.

GraphViz

Pre inštaláciu systému je potrebné mať v operačnom systéme nainštalovaný nástroj *GraphViz* popísaný v podsekcii 3.3.1, ktorý nájdeme na webovej stránke `http://www.graphviz.org/download/`. Nástroj je dostupný pre *Windows*, *Linux* a *Mac*.

Maven

Poslednou prerekvizitou nášho systému, ktorú treba inštalovať, je nástroj *Maven* popísaný v podsekcii 3.3.3. *Maven* je dostupný na stránke `https://maven.apache.org/download.cgi` pre *Windows*, *Linux* a *Mac*.

Lokálne prerekvizity ako *libfsm* či *GraphViz Java API* netreba inštalovať — sú už importované v systéme. Ostatné sa automatický stiahnú cez *Maven* počas vytvárania *.jar* balíčku.

5.1.1 Windows

Pred inštaláciou v operačnom systéme *Windows* je potrebné nastaviť nasledovné premenné prostredia:

- JAVA_HOME = C:\Cesta\k\jdk-<verzia>
- GRAPHVIZ_HOME = C:\Cesta\k\Graphviz<verzia>
- MAVEN_HOME = C:\Cesta\k\apache-maven-<verzia>
- TEMP = C:\Lubovolna\cesta

Následne treba pridať do premennej *Path* nasledovné riadky:

- %JAVA_HOME%\bin
- %GRAPHVIZ_HOME%\bin
- %MAVEN_HOME%\bin

Dôvodmi týchto nastavení sú použitie príkazového riadku pri inštalácii a použitie príkazu *dot* v našom systéme. Pre *Linux* a *Mac* nie sú potrebné žiadne špeciálne nastavenia. Keď už máme prerekvizity nainštalované a nastavené, môžeme sa presunúť k samotnej inštalácii.

5.2 Inštalácia a spustenie

Systém je voľne dostupný na web stránke https://gitlab.com/pbazik/guimata, odkiaľ ho je možné stiahnúť. Pre stiahnutie máme dva spôsoby. Prvým je stiahnúť a rozbaliť *zip* archív *guimata-master.zip*. Druhým spôsobom je použiť príkaz:

```
$ git clone https://gitlab.com/pbazik/guimata
```

Po stiahnutí a prípadnom rozbalení *zip* súboru sa presunieme v príkazovom riadku do priečinku systému:

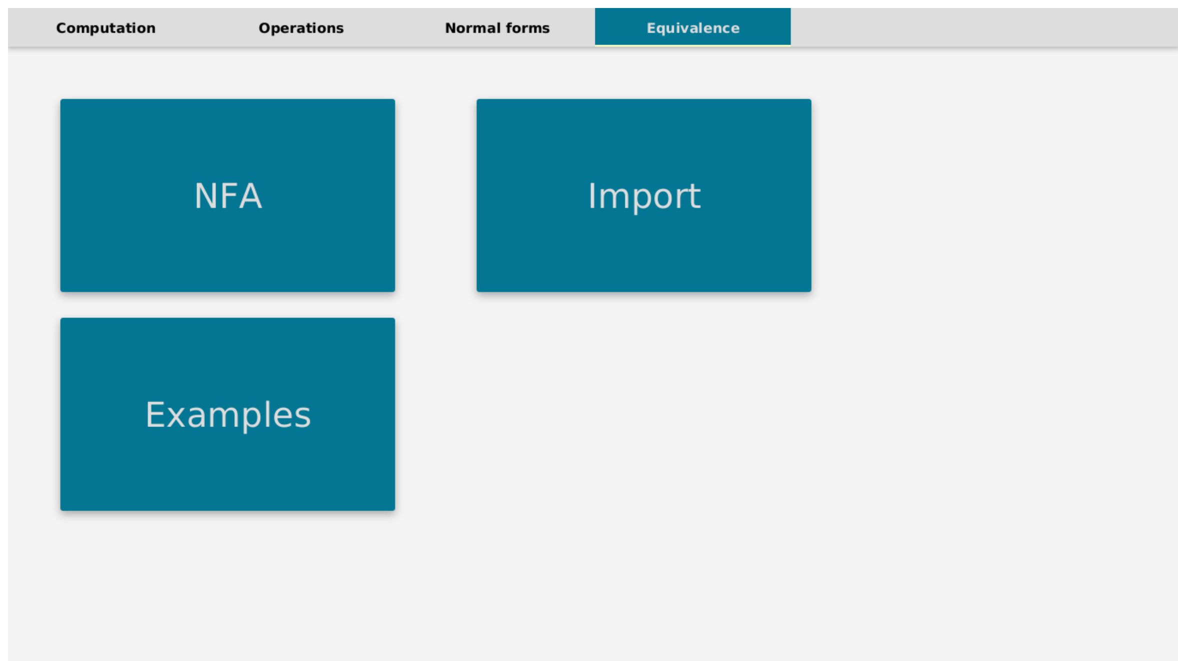
```
$ cd guimata-master
```

V tomto priečinku spustíme inštaláciu systému príkazom:

```
$ mvn clean install
```

Po nainštalovaní spustíme systém príkazom:

```
$ java -jar target/guimata-1.0.jar
```



Obr. 5.1: Úvodné menu ekvivalencie

5.3 Príklad prevodu NKA na DKA

Na záver tejto kapitoly uvedieme príklad použitia nášho systému, na ktorom predvádzame prevod zadaného NKA na ekvivalentný DKA. Pri realizácii sme použili krokovaciu formu výstupu.

Ako prvé sme si v úvodnom menu v sekcii ekvivalencie zvolili NKA. Menu môžeme vidieť na obrázku 5.1.

Po zvolení NKA sme sa dostali do prostredia pre zadávanie vstupného NKA. Najprv sme pridali stavy pomocou tlačidla *Add S*. Pridanie stavu *1* môžeme vidieť na obrázku 5.2. Z obrázku je tiež zrejmé, ako pridáme akceptačný stav.

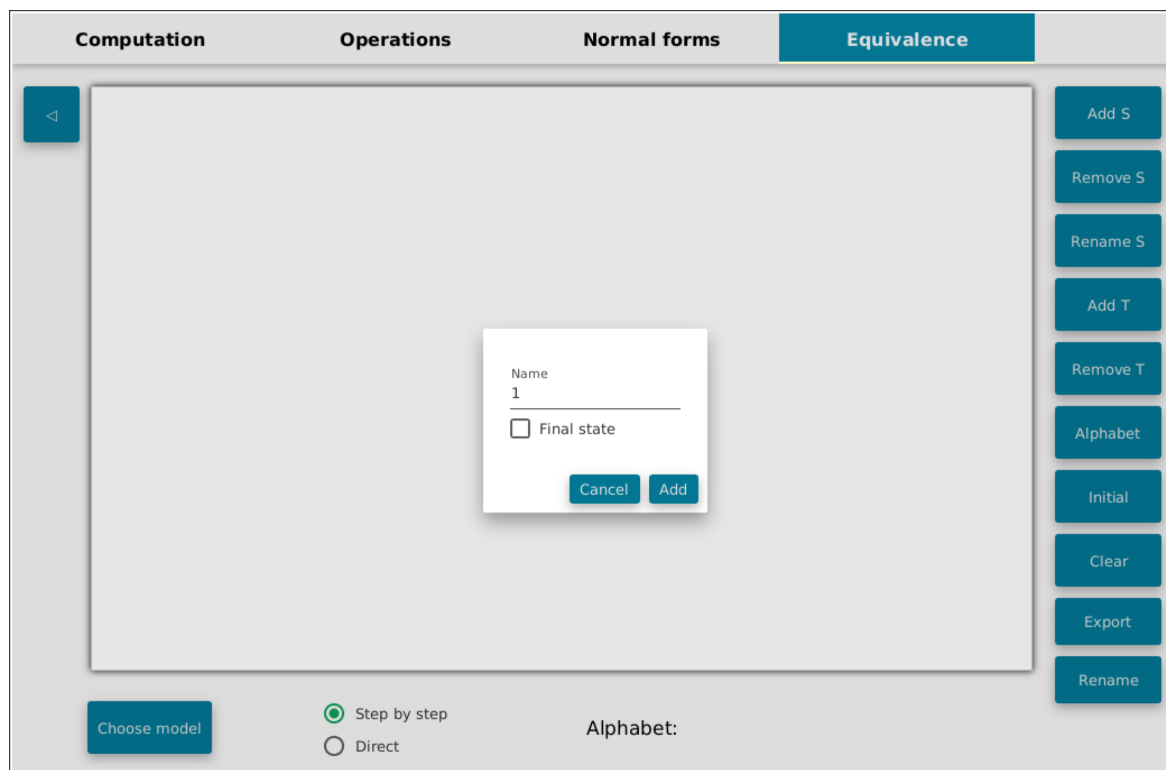
Následne sme zvolili stav *1* ako počiatočný stav automatu pomocou tlačidla *Initial*. Zvolenie počiatočného stavu môžeme vidieť na obrázku 5.3.

Aby sme mohli začať pridávať prechody do prechodovej funkcie, musíme najprv nastaviť abecedu automatu. Toto je umožnené pomocou tlačidla *Alphabet*. Nastavenie abecedy je na obrázku 5.4.

Po pridaní abecedy môžeme pridávať prechody pomocou tlačidla *Add T*. Na obrázku 5.5 je znázornené pridanie prechodu zo stavu *1* na symbol '$' do stavu *2*.

Keď máme automat zadaný, môžeme zvoliť krokovaciu formu výstupu pomocou radiobuttonu *Step by step*. Model, na ktorý chceme automat previesť, sme zvolili pomocou tlačidla *Choose model*. Po zobrazení dialógového okna s možnosťou voľby sme zvolili DKA a následne zahájili prevod tlačidlom *Run*. Voľba modelu je na obrázku 5.6.

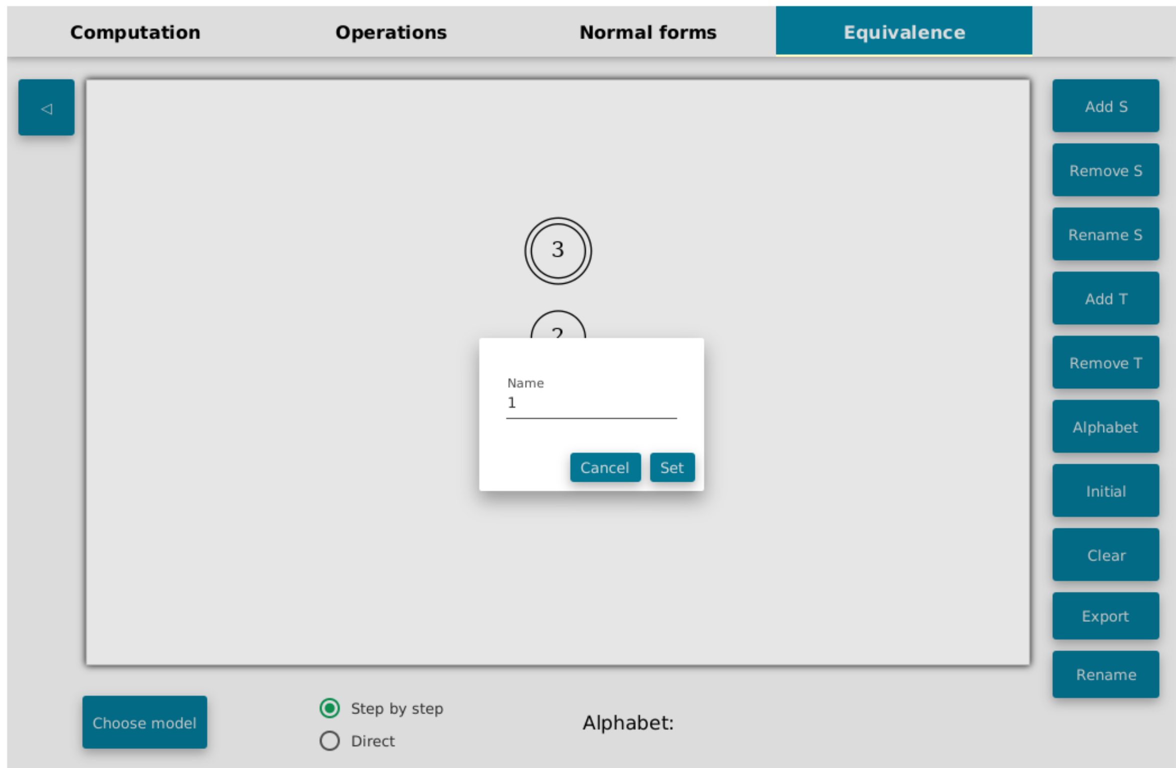
Tým sa dostávame do prostredia priebehu prevodu. V tomto prostredí už len klikáme tlačidlo *Next*, ktoré nám zobrazí ďalší krok prevodu. Ukážka priebehu prevodu je



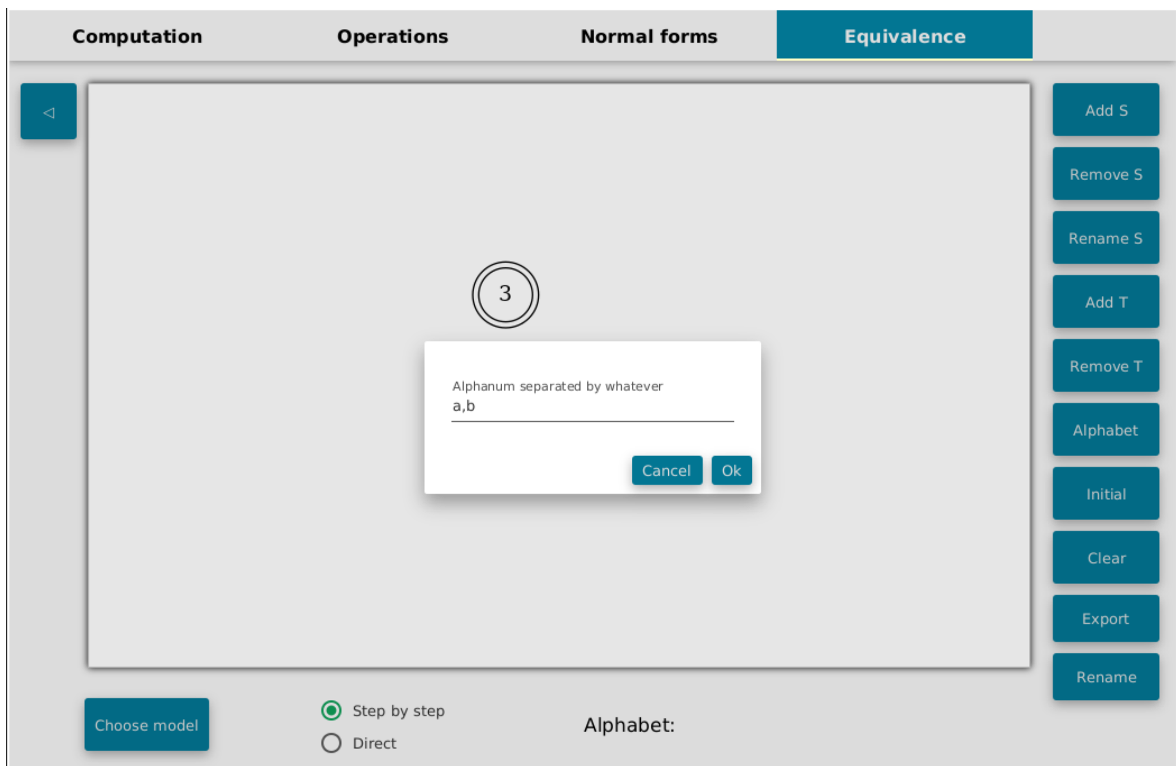
Obr. 5.2: Pridanie stavu 1

na obrázku 5.7. Z obrázku je tiež zrejmé, že keď je NKA v stavoch *2* a *3* a chce spraviť krok na symbol b, tak skončí v stave *3*. Stav *3* je v NKA hnedou farbou, pretože je zdrojom aj cieľom prechodu na b. Môžeme si tiež všimnúť slovný popis prevodu.

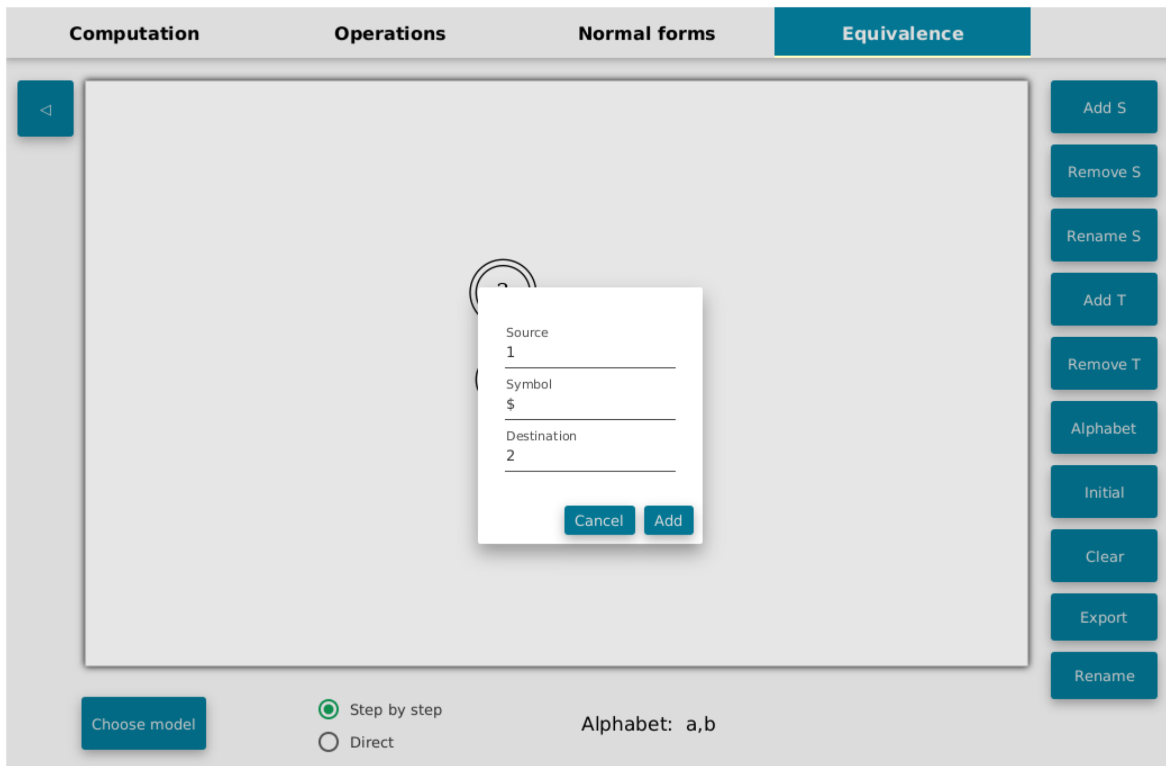
Po dokončení prevodu sme dostali oznámenie o dokončení prevodu. Môžeme ho vidieť na obrázku 5.8.



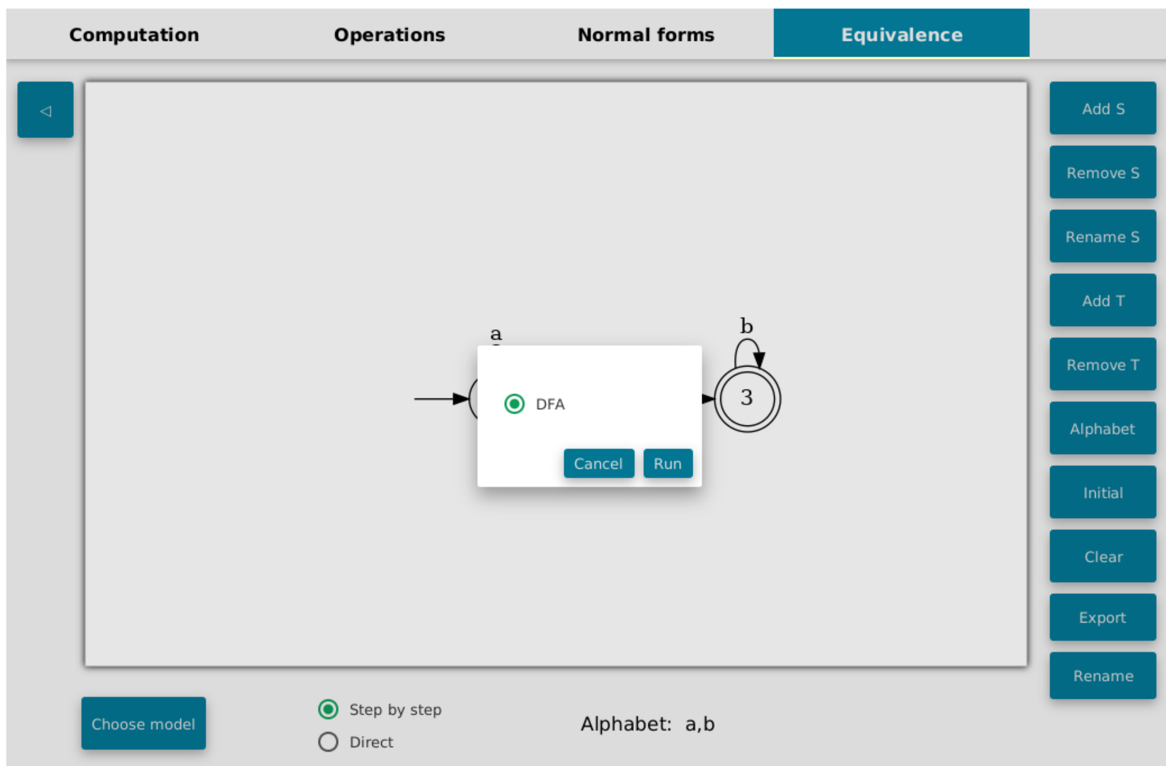
Obr. 5.3: Zvolenie stavu 1 ako počiatočného stavu



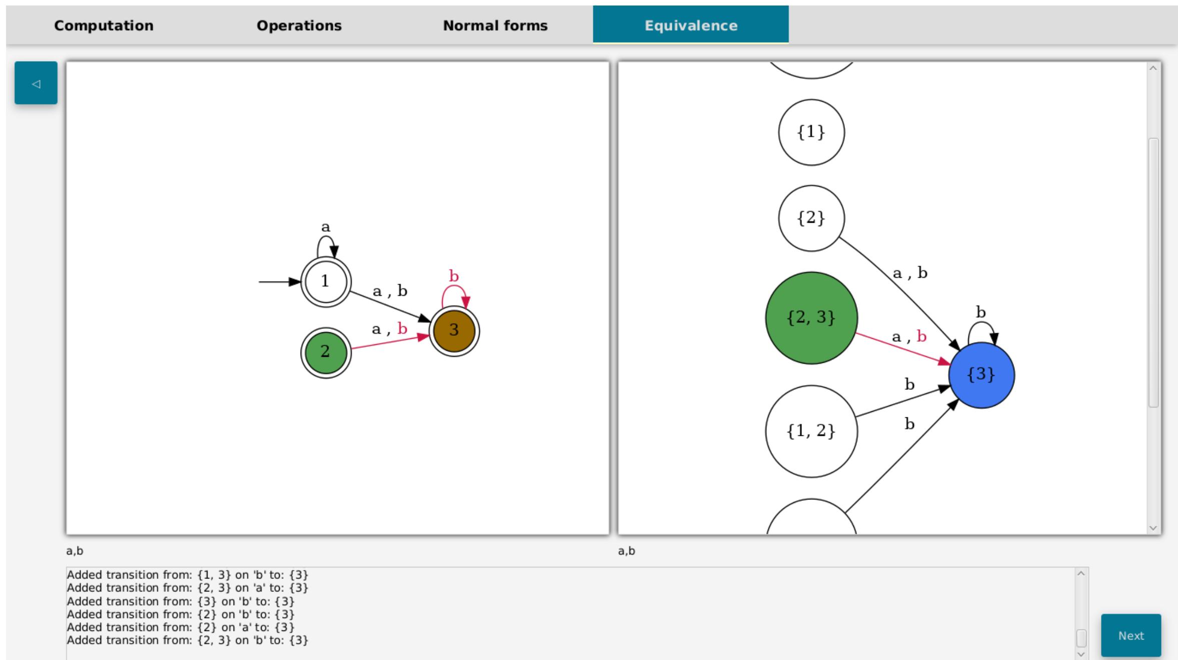
Obr. 5.4: Nastavenie abecedy a,b



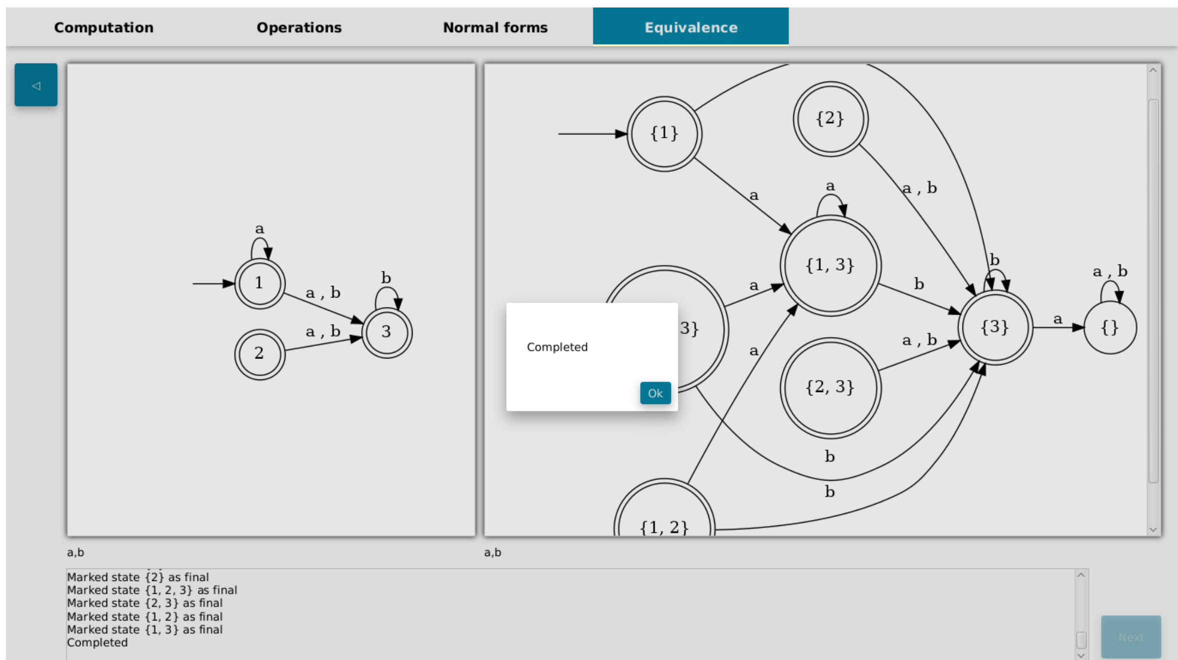
Obr. 5.5: Pridanie prechodu z 1 na $ do 2



Obr. 5.6: Voľba modelu na prevod



Obr. 5.7: Priebeh prevodu NKA na DKA



Obr. 5.8: Oznámenie o dokončení prevodu

Záver

V práci sa nám podarilo navrhnúť a implementovať systém vhodný pre výučbu teórie formálnych jazykov a automatov. Systém sme naprogramovali v jazyku *Java*, pričom sme dbali na konvencie nástroja *Maven* a návrhového vzoru *MVC*. Systém je voľne dostupný pod licenciou GNU GPL v3 na web stránke `https://gitlab.com/pbazik/guimata`.

Implementovali sme knižnicu *libfsm*, ktorá obsahuje potrebné štruktúry a vybrané algoritmy pre konečné automaty. Knižnicu sme úspešne integrovali do systému. Prostredie systému je používateľsky prívetivé a poskytuje názorné vysvetlenie algoritmov pomocou farebnej vizualizácie prechodového diagramu a detailného slovného popisu. V systéme je umožnené zadávať automaty pomocou tlačidiel alebo načítaním zo súboru. Forma výstupu je dostupná krokovacia aj priama. Čiastočne sa nám podarilo implementovať minimalizáciu DKA — implementovali sme algoritmus odstraňovania nedosiahnuteľných stavov.

Do systému bude možné jednoduchým spôsobom integrovať ďalšie knižnice. Za účelom zlepšenia by sme v budúcnosti chceli do systému pridať ďalšiu funkcionalitu definovanú v sekcii 4.5.

Literatúra

[1] Erich Gamma a kol. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[2] Alex Klibisz, Connor Minton. Fsa animate. `http://alexklibisz.github.io/FSA-Animate/`. [Online; navštívené 24-Január-2019].

[3] Apache. Apache maven. `https://maven.apache.org/`. [Online; navštívené 12-Apríl-2019].

[4] Apache. Maven in 5 minutes. `https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html`. [Online; navštívené 20-December-2018].

[5] Branislav Rovan, Michal Forišek. Formálne jazyky a automaty. Skriptá, Univerzita Komenského v Bratislave, 2013.

[6] frodriguez. Mvnrepository. `https://mvnrepository.com/`. [Online; navštívené 15-Apríl-2019].

[7] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 2000.

[8] Ivan Zuzak. Fsm simulator. `http://ivanzuzak.info/noam/webapps/fsm_simulator/`. [Online; navštívené 24-Január-2019].

[9] Jenkov Aps. Java json tutorial. `http://tutorials.jenkov.com/java-json/index.html`. [Online; navštívené 19-Február-2019].

[10] JFoenix. Jfoenix. `http://www.jfoenix.com/`. [Online; navštívené 10-Máj-2019].

[11] Kyle Dickerson. Automaton simulator. `http://automatonsimulator.com/`. [Online; navštívené 24-Január-2019].

[12] Laszlo Szathmary. Graphviz java api. `https://github.com/jabbalaci/graphviz-java-api`. [Online; navštívené 11-Apríl-2019].

[13] Marco Jakob. Javafx tutorial. `https://code.makery.ch/library/javafx-tutorial/`. [Online; navštívené 13-Február-2019].

[14] Oracle. Javafx. `https://openjfx.io/`. [Online; navštívené 27-Apríl-2019].

[15] Herbert Schildt. *Introducing JavaFX 8 Programming*. McGraw-Hill Education, 2015.

[16] Kshatriya Jagannath Rajini Singh. Visualizing the minimization of a deterministic finite state automaton. Diplomová práca, MONTANA STATE UNIVERSITY, Bozeman, Montana, 2007.

[17] Susan H. Rodger. Jflap. `http://www.jflap.org/`. [Online; navštívené 24-Január-2019].

[18] Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft. *Java 8 in Action: Lambdas, Streams, and functional-style programming*. Manning Publications, 2014.