Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informatiky a kvantitativních metod

Výukový nástroj pro vizualizaci a kreslení automatů

Diplomová práce

Autor: Milan Kopsa
Studijní obor: Aplikovaná informatika

Vedoucí práce: RNDr. Andrea Ševčíková

Hradec Králové srpen 2015

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 18. 8. 2015 Milan Kopsa

Děkuji RNDr. Andree Ševčíkové za poskytnutí cenných rad při vedení diplomové práce.

Anotace

Práce se zabývá návrhem a implementací aplikace pro vizualizaci a kreslení automatů, která má sloužit jako pomůcka při výuce předmětu Teoretická informatika. V teoretické části práce jsou nejprve představeny základní pojmy související s automaty a gramatikami, po kterých následuje analýza projektu. V této analýze jsou zejména stanoveny základní požadavky a také je provedena volba platformy. Po analýze jsou popsány technologie použité pro implementaci projektu, a v závěru práce je provedeno srovnání s obdobnými aplikacemi.

Annotation

Title: Learning tool for visualization and drawing automata

This diploma thesis deals with design and implementation of application for visualisation and drawing automata, which should serve as a tool to teach the subject Theoretical computer sciences. Basic terms connected with automata and grammars are introduced in the theoretical part of this work, followed by project analysis. In this analysis basic requiremets are set and platform is selected. The technologies used for implementation of the project are described after the analysis and the ending part of this thesis deals with the comparison of similar applications.

1 ÚVOD ... 1
2 ZÁKLADNÍ POJMY ... 2
2.1 ABECEDA A ŘETĚZEC ... 2
2.2 JAZYK ... 2
2.3 GRAMATIKA ... 3
2.4 CHOMSKÉHO KLASIFIKACE GRAMATIK ... 3
2.4.1 TYP 0 ... 3
2.4.2 TYP 1 ... 4
2.4.3 TYP 2 ... 4
2.4.4 TYP 3 ... 4
2.5 AUTOMATY ... 4
2.5.1 KONEČNÝ AUTOMAT ... 4
2.5.2 ZÁSOBNÍKOVÝ AUTOMAT ... 8
2.5.3 TURINGŮV STROJ ... 9
3 SOUHRN POŽADAVKŮ ... 11
4 ANALÝZA PROJEKTU ... 13
4.1 VOLBA PLATFORMY ... 13
4.2 PŘÍPADY UŽITÍ ... 14
4.2.1 VYTVOŘENÍ NOVÉHO AUTOMATU ... 14
4.2.2 VYTVOŘENÍ NOVÉHO STAVU ... 14
4.2.3 VYTVOŘENÍ NOVÉHO PŘECHODU ... 15
4.2.4 ULOŽENÍ AUTOMATU ... 15
4.2.5 OTEVŘENÍ AUTOMATU ... 15
4.2.6 SIMULACE BĚHU AUTOMATU ... 16
4.3 NÁVRH ROZHRANÍ ... 17
4.3.1 ROZHRANÍ IAUTOMATON ... 17
4.3.2 ROZHRANÍ ISTATE ... 18
4.3.3 ROZHRANÍ ITRANSITION ... 18
4.3.4 TŘÍDA POINT ... 18
4.3.5 ROZHRANÍ IRUNPARAMETER ... 18
4.3.6 ROZHRANÍ IRUNSTATE ... 18
4.3.7 VÝČET RUNRESULT ... 19
4.4 NÁVRH TŘÍD ... 19
4.5 NÁVRH UŽIVATELSKÉHO ROZHRANÍ ... 20
5 TYPESCRIPT ... 21
5.1 VÝČTOVÝ TYP ... 22
5.2 ROZHRANÍ ... 22
5.3 TŘÍDY ... 22
5.4 MODULY ... 23
5.5 FUNKCE ... 24
5.6 GENERICKÉ DEFINICE ... 24
5.7 DEFINIČNÍ SOUBORY ... 25

6 BOBRIL ... 26
6.1 VIRTUÁLNÍ DOM ... 26
6.2 INICIALIZACE APLIKACE ... 28
6.3 VIRTUÁLNÍ UZLY ... 29
6.4 KOMPONENTY ... 31
6.4.1 Kontext komponenty ... 32
6.4.2 Životní cyklus komponenty ... 33
6.5 ROZŠÍŘENÍ PRO BOBRIL ... 36
6.5.1 Focus ... 36
6.5.2 Mouse ... 38
6.5.3 OnChange ... 40
6.5.4 OnKey ... 41
6.5.5 Router ... 41
6.5.6 Další rozšíření ... 44
7 SCALABLE VECTOR GRAPHICS ... 45
7.1 CANVAS ... 46
7.2 GRAPH ... 46
7.3 VERTEX ... 47
7.4 EDGE ... 49
8 OSTATNÍ TECHNOLOGIE ... 52
8.1 BOOTSTRAP ... 52
8.2 FILESAVER.JS ... 53
8.3 JSON ... 53
8.4 FILEREADER ... 55
8.5 WEB STORAGE ... 55
9 IMPLEMENTACE APLIKACE ... 57
9.1 STRUKTURA ZDROJOVÝCH KÓDŮ ... 57
9.2 NÁSTROJE PRO KOMPILACI A SESTAVENÍ APLIKACE ... 58
10 SROVNÁNÍ S PODOBNÝMI PRODUKTY ... 59
10.1 AUTOMATON SIMULATOR ... 59
10.2 JFLAP ... 60
10.3 AUTOMATONSIMULATOR.COM ... 61
10.4 JFAST ... 63
10.5 FSM SIMULATOR ... 64
10.6 SHRNUTÍ FUNKCÍ POROVNÁVANÝCH PROGRAMŮ ... 65
11 SHRNUTÍ VÝSLEDKŮ ... 66
12 ZÁVĚRY A DOPORUČENÍ ... 67
13 SEZNAM POUŽITÉ LITERATURY ... 68
14 SEZNAM OBRÁZKŮ ... 73
15 SEZNAM TABULEK ... 75
16 SEZNAM PŘÍLOH ... 76

1 ÚVOD

Výuka teoretické informatiky patří mezi nezbytné součásti studijních programů snad na všech informatických vysokých školách. Pro studenty jde často o neoblíbený předmět, protože obsahuje značné množství teorie, ale to nic nemění na jeho důležitosti pro toho, kdo se chce informatice opravdu vážně věnovat.

Samotná teoretická informatika obsahuje několik okruhů. Tato práce ale se věnuje pouze jednomu z nich, a to automatům a gramatikám. Při výuce tohoto okruhu kreslí vyučující na tabuli velké množství automatů, na kterých vysvětluje probíranou látku.

Pro podporu výuky diskrétní matematiky, přesněji řečeno části zabývající se grafy, byl na naší škole vytvořen program GRALG [1], ale pro teoretickou informatiku takový program dosud chyběl. Cílem této práce je tedy zaplnit tuto mezeru a usnadnit tak vyučujícím část jejich pracovní činnosti.

Program by samozřejmě měl být dobře použitelný i pro studenty a díky tomu se snad trochu změní jejich postoj k tomuto předmětu.

V této práci budou nejprve představeny základní pojmy související s automaty a gramatikami, po kterých bude následovat analýza projektu. V této analýze budou stanoveny základní funkční a nefunkční požadavky a také bude provedena volba platformy. Budou zde též popsány příklady užití a provedeny návrhy rozhraní, tříd a také uživatelského rozhraní. Po analýze budou popsány technologie použité pro implementaci projektu. Půjde zejména o programovací jazyk TypeScript, framework Bobril a SVG, které bude použito pro vykreslování vektorové grafiky. V závěru práce bude provedeno srovnání s obdobnými aplikacemi, tedy zejména jak splňují požadavky stanovené v této práci a jaké mají případné další funkce, které nejsou součástí výše zmíněných požadavků.

2 ZÁKLADNÍ POJMY

2.1 Abeceda a řetězec

„Abecedou rozumíme neprázdnou množinu prvků, které nazýváme symboly abecedy. Řetězcem (také slovem nebo větou) nad danou abecedou rozumíme každou konečnou posloupnost symbolů abecedy. Prázdnou posloupnost symbolů, tj. posloupnost, která neobsahuje žádný symbol, nazýváme prázdný řetězec. Prázdný řetězec budeme označovat písmenem ε.

Formálně lze definovat řetězec nad abecedou Σ takto:

1. *prázdný řetězec ε je řetězec nad abecedou Σ*
2. *je-li x řetězec nad Σ a $a \in \Sigma$, pak xa je řetězec nad Σ*
3. *y je řetězec nad Σ, když a jen když lze y získat aplikací pravidel 1 a 2.“* [2]

Příklad: anglická abeceda obsahuje 26 různých malých písmen „a-z“. Řetězcem nad touto abecedou je libovolná (i prázdná) posloupnost těchto písmen, tedy např. „uhk“, „aaabbxz“, „ahoj“ atd. Naopak řetězce nad touto abecedou nejsou „čaj“, „123“, „Rakousko“, protože obsahují znaky, které nejsou součástí dané abecedy.

„Nechť x a y jsou řetězce nad abecedou Σ. Konkatenací (zřetězením) řetězce x s řetězcem y vznikne řetězec xy připojením řetězce y za řetězec x.“ [2]

Tedy například zřetězením řetězců „tele“ a „vize“ vznikne řetězec „televize“.

2.2 Jazyk

„Nechť Σ je abeceda. Označme symbolem Σ^* množinu všech řetězců nad abecedou Σ včetně řetězce prázdného, symbolem Σ^+ množinu všech řetězců nad Σ vyjma řetězce prázdného, tj. $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$. Množinu L, pro niž platí $L \subseteq \Sigma^*$ (případně $L \subseteq \Sigma^+$, pokud $\varepsilon \notin L$), nazýváme jazykem L nad abecedou Σ. Jazykem tedy může být libovolná podmnožina řetězců nad danou abecedou. Řetězec x, $x \in L$, nazýváme větou (také někdy slovem) jazyka L.“ [2]

Jazykem nad abecedou z minulého příkladu může být třeba množina L = {abstract, base, bool, catch, class}.

2.3 Gramatika

Gramatika představuje způsob, jak konečným způsobem reprezentovat příslušné jazyky. Pro svoji reprezentaci využívá dvě množiny symbolů:

- terminály jsou přímo symboly z abecedy daného jazyka a obvykle se reprezentují malými písmeny.
- nonterminály jsou zástupné symboly, které jsou při tvorbě slov nahrazovány jinými symboly (terminály nebo i opět nonterminály). Obvykle jsou reprezentovány velkými písmeny.

„Gramatika G je čtveřice G = (*N, Σ, P, S*)*, kde*

- *N je konečná množina nonterminálních symbolů*
- *Σ je konečná množina terminálních symbolů,* $N \cap \Sigma = \emptyset$
- *P je konečná podmnožina kartézského součinu* $(N\cup\Sigma)^*N(N\cup\Sigma)^*\times(N\cup\Sigma)^*$
- $S \in N$ *je výchozí (také počáteční) symbol gramatiky*

Prvek (*α, β*) *množiny P nazýváme přepisovacím pravidlem (krátce pravidlem) a budeme jej zapisovat ve tvaru α → β. Řetězec α resp. β nazýváme levou resp. pravou stranou přepisovacího pravidla.“* [2]

Například: gramatika G = ({S}, {a, b}, P, S), kde P = {S → aS, S → a | b} generuje všechna slova začínající písmenem „a“ následovaným libovolnou posloupností písmen „a“ a „b“.

2.4 Chomského klasifikace gramatik

„Chomského klasifikace gramatik (a jazyků), známá také pod názvem Chomského hierarchie jazyků, vymezuje čtyři typy gramatik, podle tvaru přepisovacích pravidel, jež obsahuje množina přepisovacích pravidel P. Tyto typy se označují jako typ 0*, typ* 1*, typ* 2 *a typ* 3*.“* [2]

2.4.1 Typ 0

Typ 0 nepředpisuje na gramatiku žádné omezení, proto se také gramatiky tohoto typu označují jako neomezené.

2.4.2 Typ 1

„Gramatika typu 1 *obsahuje pravidla tvaru:*

$$\alpha A\beta \rightarrow \alpha\gamma\beta,\ \ A \in N,\ \alpha,\ \beta \in (N \cup \Sigma)^*,\ \gamma \in (N \cup \Sigma)^+$$

nebo

$S \rightarrow \varepsilon$*, pokud se S neobjevuje na pravé straně žádného pravidla.“* [2]

Tyto gramatiky se také nazývají jako kontextové. Vzhledem ke tvaru pravidel nemůže během generování nikdy dojít ke zkrácení generovaných řetězců.

2.4.3 Typ 2

„Gramatika typu 2 *obsahuje pravidla tvaru:*

$$A \rightarrow \gamma,\ A \in N,\ \gamma \in (N \cup \Sigma)^*$$

Gramatiky typu 2 *se nazývají také bezkontextovými gramatikami.“* [2]

2.4.4 Typ 3

„Gramatika typu 3 *obsahuje pravidla tvaru:*

$$A \rightarrow xB \text{ nebo } A \rightarrow x;\ A,\ B \in N,\ x \in \Sigma^*.$$

Gramatika s tímto tvarem pravidel se nazývá pravá lineární gramatika (jediný možný nonterminál pravé strany pravidla stojí úplně napravo).“ [2]

Tato gramatika se také nazývá jako regulární.

2.5 Automaty

2.5.1 Konečný automat

„Konečný automat (*KA*) *je pětice* $M = (Q, \Sigma, \delta, q0, F)$*, kde*

- *Q je konečná množina stavů*
- *Σ je konečná vstupní abeceda*
- *δ je zobrazení* $Q \times \Sigma \rightarrow 2^Q$*, které nazýváme funkcí přechodu (*2^Q *je množina podmnožin množiny Q)*

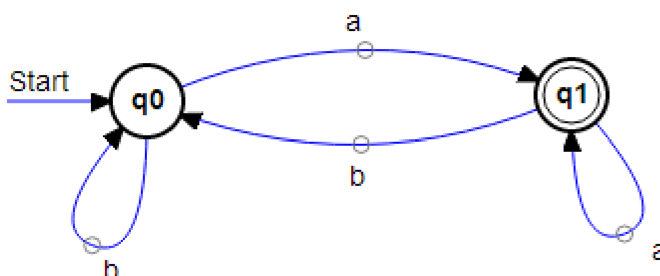
- *q0 ∈ Q je počáteční stav*
- *F ⊆ Q je množina koncových stavů."* [2]

Konečný automat může být reprezentován několika různými způsoby. Prvním je tabulka přechodů (viz Tabulka 1), kde jsou v záhlaví řádků vypsány jednotlivé stavy a v záhlaví sloupců možné vstupní symboly. Vnitřek tabulky tvoří výsledky přechodových funkcí pro daný stav a vstupní symbol.

δ	**a**	**b**
$\mathbf{q_0}$	q_1	q_0
$\mathbf{q_1}$	q_1	q_0

Tabulka 1 Tabulka přechodů automatu

Druhým způsobem může být graf (viz Obrázek 1), kde jeho vrcholy představují jednotlivé stavy, zatímco přechody jsou reprezentovány hranami. Počáteční stav bývá označen k němu mířící šipkou, koncové stavy bývají označené dvojitým okrajem. Popisek hrany představuje vstupní symboly.



Obrázek 1 Grafická reprezentace automatu, zdroj: autor

Třetím způsobem mohou být množiny možných stavů a přechodů. První a třetí způsoby se obvykle používají při počítačovém zpracování, druhý je zase dobře představitelný a uchopitelný pro člověka.

Oba výše uvedené způsoby (tabulka i obrázek) reprezentují stejný konečný automat. Tento automat má počáteční stav q_0, ze kterého se po přečtení symbolu „a" dostane do stavu q_1, zatímco po přečtení symbolu „b" zůstane ve stejném stavu. Obdobně je tomu u stavu q_1 – po přečtení symbolu „a" zůstane automat ve stejném stavu, po přečtení symbolu „b" se přesune do stavu q_0.

„Je-li $\forall q \in Q\ \forall a \in \Sigma : |\delta(q, a)| \leq 1$*, pak M nazýváme deterministickým konečným automatem* (*zkráceně DKA*)*, v případě, že* $\exists q \in Q\ \exists a \in \Sigma : |\delta(q, a)| > 1$ *pak nedeterministickým konečným automatem* (*zkráceně NKA*)*.“* [2]

To znamená, že průběh činnosti deterministického konečného automatu je možné pro dané vstupní symboly popsat jednoznačnou posloupností jeho stavů. V případě nedeterministického automatu se může takový průběh „rozvětvit“ a paralelně pokračovat ve všech těchto větvích.

„Je-li $M = (Q, \Sigma, \delta, q_0, F)$ *konečný automat, pak dvojici* $C = (q, w)$ *z* $Q \times \Sigma^*$ *nazýváme konfigurací automatu M. Konfigurace tvaru* (q_0, w) *je počáteční konfigurace, konfigurace tvaru* (q, ε), $q \in F$ *je koncová konfigurace. Přechod automatu M je reprezentován binární relací* $\vdash_M$ *na množině konfigurací C.“* [2]

Automat z předchozího příkladu má tedy počáteční konfiguraci (q_0, w), kde q_0 je počáteční stav automatu a w je vstupní řetězec. Koncová konfigurace tohoto automatu je (q_1, ε), tedy pokud se automat po vyčerpání všech vstupních symbolů nachází ve stavu q_1.

„Říkáme, že vstupní řetězec w je přijímán konečným automatem M, jestliže $(q_0, w) \vdash^* (q, \varepsilon)$*,* $q \in F$*. Jazyk přijímaný konečným automatem M označujeme symbolem L*(*M*) *a definujeme ho jako množinu všech řetězců přijímaných automatem M:*

$$L(M) = \{w \mid (q_0, w) \vdash^* (q, \varepsilon) \land q \in F\}$$ “ [2]

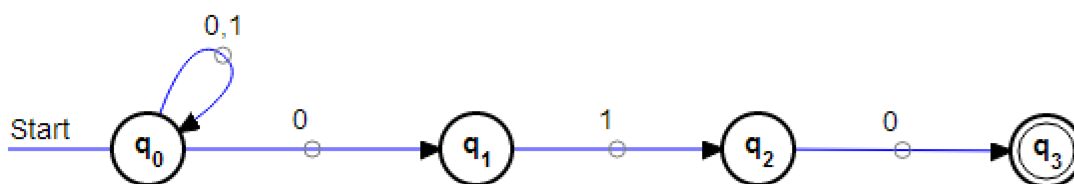
Automat tedy přijímá vstupní řetězec, pokud se po vyčerpání všech vstupních symbolů nachází v některém z koncových stavů. V případě nedeterministického automatu je dostačující, pokud se v koncovém stavu nachází alespoň jedna větev.

Platí následující věta, která uvádí vzájemný vztah deterministického a nedeterministického automatu:

Věta: *„Každý nedeterministický konečný automat M lze převést na deterministický konečný automat M‘ tak, že L*(*M*) *= L*(*M‘*)*.“* [2]

Důkaz věty je možno nalézt v [2]. Bude nastíněn princip převodu: pokud lze ze stavu q_0 přejít po přečtení vstupního symbolu přejít do stavů q_a a q_b, bude do deterministického automatu přidán stav $\{q_a, q_b\}$, který vznikl sjednocením obou možných stavů. Pro tento nový stav budou opět prozkoumány všechny možné přechody, tedy takové přechody, kterými je možné přejít z obou stavů. Všechny vzniklé stavy jsou následně přidány do automatu a celý tento postup je opakován tak dlouho, dokud nejsou vyčerpány všechny stavy. Koncové stavy

deterministického automatu jsou takové, které obsahují alespoň jeden koncový stav z původního nedeterministického automatu.



Obrázek 2 Nedeterministický konečný automat, zdroj: autor

Příklad: Obrázek 2 zobrazuje nedeterministický konečný automat, ve kterém je po přečtení symbolu „0“ možné přejít ze stavu q_0 do stavu q_1 či zpět do q_0. Převod na deterministický je ukázán na následující tabulce.

δ	**0**	**1**
$\{q_0\}$	$\{q_0,q_1\}$	$\{q_0\}$
$\{q_0,q_1\}$	$\{q_0,q_1\}$	$\{q_0,q_2\}$
$\{q_0,q_2\}$	$\{q_0,q_1,q_3\}$	$\{q_0\}$
$\{q_0,q_1,q_3\}$	$\{q_0,q_1\}$	$\{q_0,q_2\}$

Tabulka 2 Převod NKA na DKA

Stav $\{q_0,q_1,q_3\}$ je koncový, protože obsahuje koncový stav q_3 z původního nedeterministického automatu.

Věta: *„Nechť $L = L(M)$ pro nějaký konečný automat M. Pak existuje gramatika G typu* 3 *taková, že $L = L(G)$, tj. $L_M \subseteq L_3$.“* [2]

Důkaz je možno nalézt v [2]. Jazyky přijímané konečnými automaty jsou tedy ekvivalentní jazykům, které jsou generovány regulárními gramatikami, proto je možné regulární gramatiky převádět na konečné automaty a opačně.

Regulární gramatika se z konečného automatu vytváří pomocí dvou následujících pravidel:

- Jestliže je možné ze stavu „q“ přijít po přečtení symbolu „a“ do stavu „r“, pak gramatika obsahuje pravidlo $q \rightarrow ar$. Stavy „q“ a „r“ tedy v gramatice tvoří nonterminály a vstupní symbol „a“ je terminál.
- Jestliže stav „p“ je koncový stav, pak gramatika obsahuje pravidlo $p \rightarrow \varepsilon$.

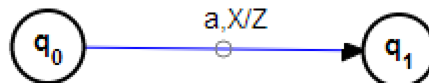
2.5.2 Zásobníkový automat

„Zásobníkový automat P je sedmice

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F), \text{ kde}$$

- *Q je konečná množina stavových symbolů reprezentujících vnitřní stavy řídicí jednotky,*
- *Σ je konečná vstupní abeceda; jejími prvky jsou vstupní symboly*
- *Γ je konečná abeceda zásobníkových symbolů*
- *δ je zobrazení z množiny $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ do konečné množiny podmnožin množiny $Q \times \Gamma^*$ popisující funkci přechodů*
- *$q_0 \in Q$ je počáteční stav řídicí jednotky*
- *$Z_0 \in \Gamma$ je symbol, který je na počátku uložen do zásobníku – tzn. startovací symbol zásobníku,*
- *$F \subseteq Q$ je množina koncových stavů."* [2]

Zásobníkový automat se podobá konečnému automatu, ale na rozdíl od něj má k dispozici ještě paměť – zásobník. Automat v průběhu činnosti nejen že čte vstupní symboly, ale také v každém kroku přečte symbol na vrcholu zásobníku a zpět na vrchol zásobníků další symbol vloží. Může samozřejmě vložit i stejný symbol, jaký přečetl – v tom případě se stav zásobníku nemění. Nebo také může na zásobník uložit prázdný řetězec – v tom případě dojde pouze k odstranění vrcholu zásobníku.



Obrázek 3 Přechod v zásobníkovém automatu, zdroj: autor

Obrázek 3 ukazuje možný přechod v zásobníkovém automatu. Pokud se automat nachází ve stavu q_0, současný vstupní symbol na pásce je „a" a na vrcholu zásobníku se nachází symbol „X", dojde k přechodu automatu do stavu q_1 a na vrchol zásobníku je vložen symbol „Z".

„Konfigurací zásobníkového automatu P nazveme trojici $(q, w, \alpha) \in Q \times \Sigma^* \times \Gamma^*$ kde

- *q je přítomný stav řídicí jednotky*
- *w je doposud nepřečtená část vstupního řetězce; první symbol řetězce w je pod čtecí hlavou. Je-li $w = \varepsilon$, pak byly všechny symboly ze vstupní pásky přečteny.*

- *α je obsah zásobníku. Pokud nebude uvedeno jinak, budeme zásobník reprezentovat řetězcem, jehož nejlevější symbol koresponduje s vrcholem zásobníku. Je-li $\alpha = \varepsilon$, pak je zásobník prázdný.“* [2]

Počáteční konfigurace zásobníkového automatu je tedy (q_0, w, Z_0). U zásobníkových automatů se obvykle uvádějí dvě jejich varianty, podle toho jakým způsobem přijímají vstupní řetězce. První způsob je podobný konečnému automatu v tom, že automat přijímá vstup, pokud se po přečtení všech vstupních symbolů nachází v koncovém stavu – koncová konfigurace je tedy (q, ε, α) pro $q \in F$.

Druhým způsobem je přijímání prázdným zásobníkem. Automat tedy přijímá vstup, pokud má prázdný zásobník, bez ohledu na to v jakém stavu se nachází. Koncová konfigurace v tomto případě je tedy (q, w, ε).

Věta: *„Nechť $R = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ je zásobníkový automat. Pak existuje $L_P \subseteq L_2$ bezkontextová gramatika G, pro kterou platí $L(G) = L(R)$.“* [2]

Stejně jako jsou jazyky přijímané konečnými automaty ekvivalentní jazykům generovaným regulární gramatikou, jsou i jazyky přijímané zásobníkovými automaty ekvivalentní jazykům generovaným bezkontextovými gramatikami.

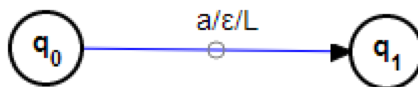
2.5.3 Turingův stroj

Turingův stroj může mít několik alternativních definic, které jsou ovšem vzájemně převoditelné [2], jedna z nich zní takto:

Turingův stroj je sedmice ve tvaru $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, kde:

- *Q je konečná množina vnitřních stavů,*
- *Σ je konečná množina symbolů nazývaná vstupní abeceda neobsahující prázdný symbol „Ø“,*
- *Γ je konečná množina symbolů, $\Sigma \subset \Gamma$, $\emptyset \in \Gamma$, nazývaná pásková abeceda,*
- *δ je přechodová funkce pro kterou platí δ: $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$,*
- *q_0 je počáteční stav, $q_0 \in Q$,*
- *q_{accept} je přijímající stav, $q_{accept} \in Q$,*
- *q_{reject} je zamítající stav, $q_{reject} \in Q$*

Turingův stroj je opět podobný zásobníkovému automatu. Stejně jako on má k dispozici paměť, ale v tomto případě nejde o zásobník, nýbrž o nekonečnou pásku. Zatímco zásobníkový automat může pracovat pouze s vrcholem zásobníku, Turingův stroj v každém kroku přečte symbol z aktuální pozice na pásce, jiný symbol na tuto pozici zapíše a pak se může po pásce přesunout vpravo či vlevo.



Obrázek 4 Přechod v Turingově stroji, zdroj: autor

Obrázek 4 opět ukazuje jeden z možných přechodů v Turingově stroji. Pokud se automat nachází ve stavu q_0 a čtecí hlava se nachází nad symbolem „a" na pásce, pak se automat přesune do stavu q_1 a čtecí hlava se posune o pozici vlevo. Stroj v tomto případě na pásku nic nezapíše.

„Konfigurace Turingova stroje je dána stavem řídicí jednotky, obsahem pásky a pozicí hlavy. Konkrétní konfiguraci stroje popíšeme trojicí (*u, q, v*)*, či zkráceně uqv, kde u, v* $\in \Gamma^*$ *a q* $\in Q$." [3]

V této definici značí „u" už přečtenou část vstupního slova, „q" je současný stav a „v" zbývající část vstupního slova. Počáteční konfigurace Turingova stroje je (q_0, w), koncové konfigurace jsou (u, q_{accept}, v) a (u, q_{reject}, v).

Na rozdíl od předcházejících typů automatů končí běh Turingova stroje okamžitě, jakmile dosáhne přijímacího či zamítajícího stavu bez ohledu na stav pásky.

Věta: *„Každý jazyk přijímaný Turingovými stroji je jazykem typu* 0.*"* [2]

Věta: *„Každý jazyk typu* 0 *je přijímán nějakým Turingovým strojem."* [2]

Z výše uvedených vět vyplývá, že jazyky přijímané Turingovými stroji jsou ekvivalentní jazykům, které jsou generovány gramatikami typu 0 – tedy neomezenými gramatikami. Důkazy těchto vět se opět nachází v [2].

3 SOUHRN POŽADAVKŮ

Cílem této práce je vytvořit aplikaci, která bude moci být využita k podpoře výuky předmětu Teoretická informatika, konkrétněji k části zabývající se automaty. Jako základní byly stanoveny tyto funkční požadavky:

- V aplikaci bude možné sestrojovat modely konečných automatů, zásobníkových automatů a Turingových strojů.
- Nedeterministické konečné automaty bude možno převádět na deterministické. Tento převod bude proveden pouze v tabulkové podobě. Převod do grafické podoby není vyžadován, a to zejména z toho důvodu, že není vyžadován ani při výuce.
- Konečné a zásobníkové automaty bude možno převést na odpovídající gramatiky. Převod Turingových strojů na gramatiku není vyžadován, stejně tak nejsou vyžadovány převody z gramatik na automaty.
- Modely automatů budou navrhovány v běžně používané grafické podobě, tedy jako orientované grafy, kde vrcholy grafu představují jednotlivé stavy automatů a hrany grafu zase přechody mezi nimi.
- Stavy automatů budou mít automaticky generované názvy, které ale bude moci uživatel změnit. Stavy bude také možno obarvit. Jejich barva má čistě kosmetický význam bez vlivu na ostatní funkce aplikace.
- U přechodů bude možno definovat vstupní symboly či další parametry závislé na typu automatu. Stejně jako u stavů jim bude možno změnit barvu.
- Kromě obyčejných přechodů mezi dvěma různými stavy, bude moci být definována i smyčka, tedy přechod ze stavu zpět do stejného stavu.
- Přechod mezi dvěma stavy bude možné zakřivit, aby se zamezilo nežádoucímu překřížení či překrytí různých přechodů.
- Navržený automat bude možné uložit do souboru a naopak zase ze souboru načíst.
- Navržený automat bude také možné exportovat do obrázku.
- Chod automatu bude možné simulovat zadáním vstupu. Uživatel si bude moci krokovat činnost automatu vpřed i vzad. Při každém pohybu bude zvýrazněn aktuální vstupní symbol společně s přechodem (či přechody), který se v daném kroku použije. Při ukončení činnosti bude uživatel informován, zda byl běh automatu úspěšný či nikoliv.

Kromě funkčních požadavků byly stanoveny ještě tyto nefunkční požadavky:

- Aplikace musí mít velmi snadnou instalaci, ideálně by se nemusela instalovat vůbec.
- Aplikace by neměla být závislá na žádných knihovnách či jiných aplikacích, pokud se jejich přítomnost nedá očekávat na drtivé většině strojů, kde bude potenciálně provozována. A pokud už takové knihovny či aplikace vyžaduje, musí být součástí instalačního balíčku aplikace.
- Aplikace musí být nezávislá na připojení k internetu.
- Aplikace nesmí mít výrazné požadavky na hardware počítače, musí být dobře použitelná i na podprůměrně vybavených strojích.

4 ANALÝZA PROJEKTU

4.1 Volba platformy

Zatímco ze seznamu funkčních požadavků nevyplývá žádné omezení na programovací jazyk či platformu, na které by aplikace měla běžet, nefunkční požadavky už mají na tuto volbu podstatně výraznější vliv. Na výběr připadají zejména tyto platformy:

- Nativní Windows aplikace: splňuje v zásadě všechny požadavky, nicméně by pak tato aplikace nebyla použitelná pro uživatele Linuxu či počítačů od Applu.
- .NET aplikace: stejné platformové omezení jako v předchozím případě, navíc vyžaduje nainstalovaný .NET Framework, což by mohl být problém v případě Windows XP, které standardně .NET Framework neobsahují.
- Mono aplikace: Mono je open source alternativa k .NET Frameworku dostupná i pro Linux a Mac. Nicméně aplikace by pro svůj běh vyžadovala přechozí instalaci Mono frameworku.
- Java aplikace: podobná situace jako v případě Mono frameworku. Java je dostupná pro všechny tři hlavní platformy, ale není standardní součástí instalace systému.
- Webová aplikace: má nejmenší požadavky na vybavení počítače, vyžaduje pouze internetový prohlížeč, u kterého se lze prakticky s jistotou spolehnout, že bude všude dostupný. Nicméně taková aplikace vyžaduje připojení k internetu, což odporuje stanoveným požadavkům.
- Single Page aplikace (SPA): v zásadě jde o typ webové aplikace, která ale může být navržena tak, aby minimalizovala či přímo eliminovala potřebu webového serveru. Veškerá aplikační logika tak může být prováděna přímo v prohlížeči bez potřeby internetového připojení.

Z tohoto seznamu byla jako nejméně omezující zvolena poslední možnost, tedy Single Page aplikace. Jak už z jejího názvu vyplývá, celá aplikace je tvořena jedinou stránkou, která je za běhu aplikace podle potřeby upravována pomocí JavaScriptu. Taková aplikace má pak prakticky okamžitou odezvu na činnost uživatele a tím se blíží k běžným desktopovým aplikacím.

Komunikace se serverem, pokud je vyžadována, se tak často omezuje pouze na načítání a ukládání dat. Tím se liší od klasických webových aplikací, kdy server vrací kompletní webovou stránku, kterou prohlížeč pouze zobrazí.

Pro tvorbu Single Page aplikací existuje několik frameworků, které usnadňují jejich tvorbu. Populární jsou zejména AngularJS [4] a React [5]. Aplikace, která je výsledkem této práce, byla ze začátku vyvíjena pomocí AngularJS, nicméně v průběhu implementace se ukázalo, že to nebyla příliš dobrá volba. V aplikaci totiž docházelo k výrazným výkonnostním propadům zejména na pomalejších strojích a při použití Internet Exploreru.

Bylo tedy rozhodnuto o změně použitého frameworku a jako nejvhodnější se jevily frameworky využívající ke své práci virtuální DOM (viz str. 26), tedy např. React. Volba nakonec padla na Bobril (viz str. 26) a to zejména kvůli jeho vysokému výkonu [6].

4.2 Případy užití

Ze zadání projektu plyne, že aplikace bude mít pouze jednoho aktéra, který bude využívat typové úlohy s následujícím scénáři:

4.2.1 Vytvoření nového automatu

1. Uživatel iniciuje vytvoření nového automatu

2. Pokud už v systému existuje nějaký automat:

2.1. Systém zobrazí dotaz, zda má nový automat vytvořit v nové či stávající záložce prohlížeče

2.2. Uživatel si zvolí jednu z možností

2.3. Systém vytvoří nový automat ve vybrané záložce

3. Pokud automat neexistuje:

3.1. Systém vytvoří nový automat ve stávající záložce

4.2.2 Vytvoření nového stavu

1. Uživatel iniciuje vytvoření nové stavu:

2.1. Systém vytvoří nový stav

2.2. Systém novému stavu vygeneruje název

2.3. Systém zobrazí dialog pro editaci stavu

3. Uživatel vyplní informace o stavu

4. Systém ověří zadané informace:

4.1. Pokud jsou informace o stavu v pořádku, uloží je

4.2. Jinak zobrazí chybové hlášení

4.2.3 Vytvoření nového přechodu

1. Uživatel iniciuje vytvoření nového přechodu

2.1. Systém vytvoří nový přechod

2.2. Systém zobrazí dialog pro editaci přechodu

3. Uživatel vyplní informace o přechodu

4. Systém ověří zadané informace:

4.1. Pokud jsou informace o přechodu v pořádku, uloží je

4.2. Jinak zobrazí chybové hlášení

4.2.4 Uložení automatu

1. Uživatel iniciuje uložení automatu

2.1. Systém vytvoří soubor pro uložení

2.2. Systém vyvolá uložení souboru v prohlížeči

4.2.5 Otevření automatu

1. Uživatel iniciuje otevření automatu

2. Systém zobrazí dialog pro výběr otevíraného souboru

3. Uživatel vybere příslušný soubor

4.1. Systém načte vybraný soubor

4.2. Systém zahrne typovou úlohu „Vytvoření nového automatu“

4.3 Systém do nového automatu přidá načtené stavy a přechody

4.2.6 Simulace běhu automatu

1. Uživatel iniciuje simulaci běhu automatu

2. Systém vyhodnotí zadaný vstup

3. Pokud vstup odpovídá nějakému přechodu:

3.1. Systém zvýrazní zjištěný přechod a příslušný vstupní symbol

4. Pokud vstup neodpovídá žádnému přechodu:

4.1. Systém zobrazí zprávu o neúspěšné simulaci

4.2 Systém ukončí simulaci

5. Uživatel vybere krok vpřed či vzad

6. Pokud zbývají nějaké vstupní symboly:

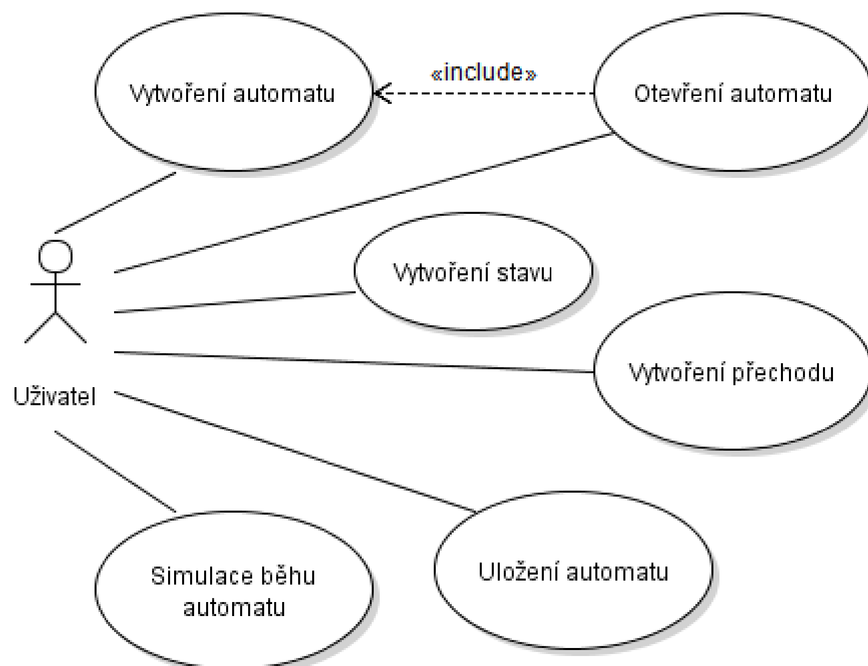
6.1. Systém pokračuje krokem 2

7. Pokud už žádné vstupní symboly nezbývají:

7.1. Systém vyhodnotí simulaci

7.2. Systém zobrazí zprávu o úspěchu či neúspěchu

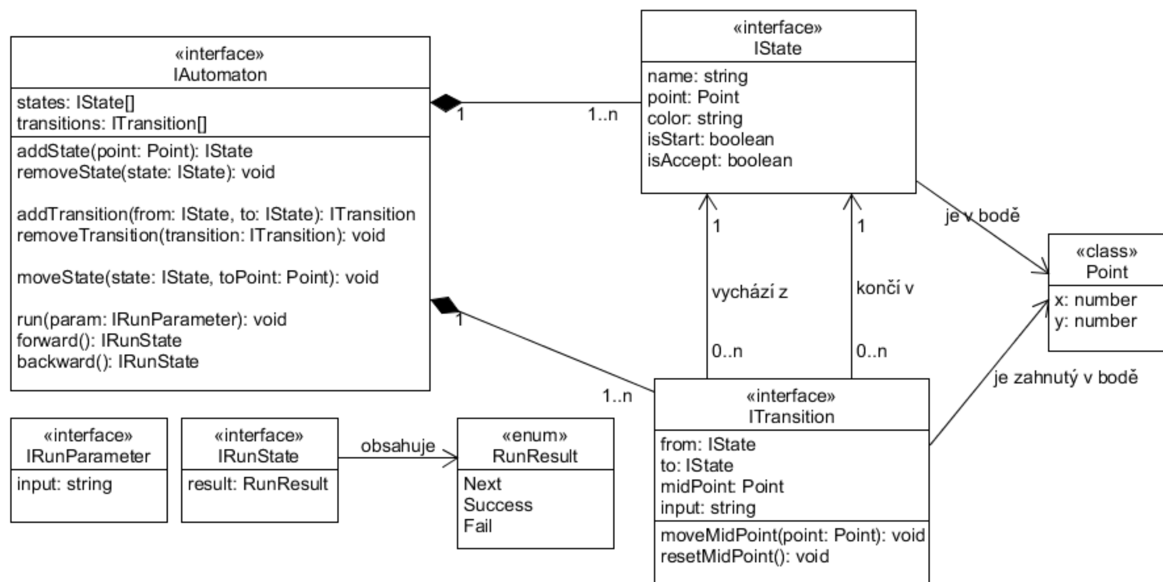
Všechny tyto typové úlohy zobrazuje Obrázek 5.



Obrázek 5 Diagram typových úloh, zdroj: autor

4.3 Návrh rozhraní

Obrázek 6 zobrazuje rozhraní, která tvoří základ modelu příslušných automatů.



Obrázek 6 Návrh rozhraní, zdroj: autor

4.3.1 Rozhraní IAutomaton

Rozhraní `IAutomaton` reprezentuje samotný automat. Má následující vlastnosti:

- `states`: kolekce stavů
- `transitions`: kolekce přechodů

Dále předepisuje následující metody:

- `addState` – přidání nového stavu na zadaných souřadnicích
- `removeState` – odstranění stavu
- `addTransition` – přidání nového přechodu mezi zadanými stavy
- `removeTransition` – odstranění přechodu
- `moveState` – zajišťuje přepočítání polohy přechodů při změně polohy stavu
- `run` – spustí simulaci automatu podle zadaných vstupních parametrů
- `forward` – posun na další vstupní symbol při simulaci
- `backward` – návrat na předchozí vstupní symbol při simulaci

4.3.2 Rozhraní IState

Rozhraní `IState` reprezentuje použité stavy. Má tyto vlastnosti:

- `name` – název stavu
- `point` – souřadnice stavu
- `color` – název barvy stavu
- `isStart` – informace, zda jde o počáteční stav
- `isAccept` – informace, zda je o přijímající stav

4.3.3 Rozhraní ITransition

Rozhraní `ITransition` reprezentuje jednotlivé přechody. Musí implementovat následující vlastnosti:

- `from` – počáteční stav přechodu
- `to` – koncový stav přechodu
- `midPoint` – souřadnice středového bodu přechodu
- `input` – vstupní symbol přechodu

Dále musí implementovat následující metody:

- `moveMidPoint` – zajišťuje přepočítání tvaru přechodu při pohybu středového bodu
- `resetMidPoint` – zajišťuje narovnání přechodu

4.3.4 Třída Point

Třída `Point` má vlastnosti `x` a `y`, které představují souřadnice bodu na obrazovce.

4.3.5 Rozhraní IRunParameter

Rozhraní `IRunParameter` reprezentuje vstupní parametry pro simulaci běhu automatu. Třídy implementující toto rozhraní musí obsahovat vlastnosti `input`, což je posloupnost vstupních symbolů.

4.3.6 Rozhraní IRunState

Rozhraní `IRunState` reprezentuje informace o jednom kroku simulace automatu. Má vlastnost `result`, reprezentující výsledek tohoto kroku.

4.3.7 Výčet RunResult

Tento výčet obsahuje možné výsledky kroku simulace automatu.

4.4 Návrh tříd

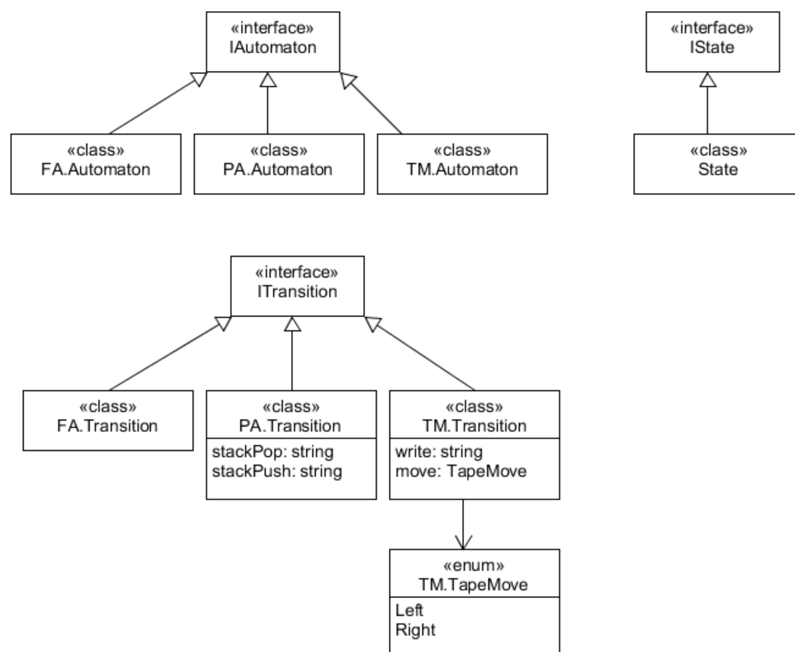
Výše uvedená rozhraní jsou implementována následujícími třídami:

Rozhraní `IAutomaton` je implementováno třídami `FA.Automaton`, `PA.Automaton` a `TM.Automaton`, které reprezentují konečné automaty, zásobníkové automaty, resp. Turingovy stroje. Implementace se liší zejména v metodách `forward` a `backward`, které vyhodnocují jednotlivé kroky simulace běhu automatu.

Rozhraní `ITransition` je implementováno třídami `FA.Transition`, `PA.Transition` a `TM.Transition`, které reprezentují přechody jednotlivých typů automatů. Model přechodu zásobníkového automatu obsahuje, kromě vstupního symbolu, také ještě požadovaný symbol na vrcholu zásobníku a symbol, který se má uložit na vrchol zásobníku. Model přechodu Turingova stroje obsahuje symbol, který má být zapsán na pásku a směr, kterým se má posunout čtecí hlava.

Rozhraní `IState` je implementováno pouze třídou `State`.

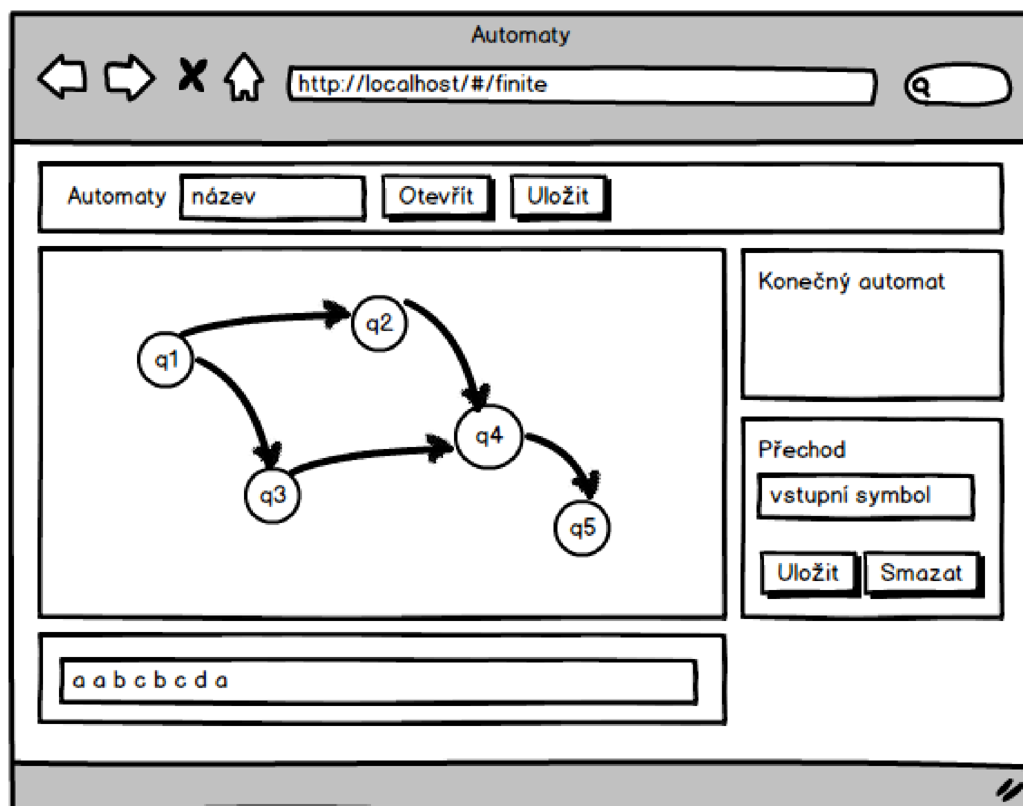
Obrázek 7 zobrazuje implementované třídy.



Obrázek 7 Návrh tříd, zdroj: autor

4.5 Návrh uživatelského rozhraní

Obrázek 8 ukazuje náhled uživatelského rozhraní, které bylo pro aplikaci navrženo.



Obrázek 8 Návrh uživatelského rozhraní, zdroj: autor

V horní části stránky se nachází navigační panel. Ten slouží k otevření či uložení souboru a také k vytvoření nového automatu.

Pod ním se nachází plocha sloužící ke grafické editaci automatu. Základní tvorba by měla být maximálně jednoduchá a rychlá, takže nový stav by se měl vytvářet dvojklikem na plochu, nový přechod tažením myši mezi počátečním a koncovým stavem. Se všemi částmi automatu by se mělo dát pohybovat pomocí myši.

Vpravo od kreslicí plochy se nachází panel s informacemi o automatu. Měl by zobrazovat typ automatu a případné další relevantní informace. Pod tímto panelem se nachází další panel sloužící k editaci aktuálně vybraného stavu resp. přechodu. Je zde také tlačítko pro smazání aktuálního prvku.

Pod kreslicí plochou se nachází panel, do kterého se zadávají vstupy potřebné pro simulaci běhu automatu.

5 TYPESCRIPT

TypeScript [7] je programovací jazyk vyvinutý firmou Microsoft, který slouží jako nadstavba nad klasickým JavaScriptem. Jakýkoliv kód v JavaScriptu je tak zároveň i validním TypeScriptovým kódem.

Obrázek 9 Logo TypeScriptu [7]

Jak už název naznačuje, jeho hlavní rozdíl oproti JavaScriptu spočívá v typovosti, tedy možnosti definovat typ proměnných, vstupních parametrů funkcí, jejich návratových hodnot atd. Tyto typy jsou kontrolovány při kompilaci a v podporovaných vývojových prostředích i přímo během psaní kódu. Výsledkem kompilace TypeScriptového kódu už je čistý JavaScript a za běhu už správnost použitých typů není kontrolována.

Příklad definice typu u proměnné:

```
var count: number = 5;
```

Jak je z příkladu vidět, typ se definuje pomocí dvojtečky, za kterou následuje uvedení typu proměnné a také případné uvedení konkrétní výchozí hodnoty, která musí odpovídat udanému typu, jinak dojde k chybě při kompilaci. Není samozřejmě nutné definovat pouze primitivní typy, použitelná jsou třeba i pole, třídy, rozhraní atd.

Uvedení typu je zcela dobrovolné. Pokud není uveden, není dodržování typu kontrolováno. Díky tomu je možné libovolně kombinovat TypeScriptový a JavaScriptový kód. Další možností, jak se vyhnout typové kontrole, je použít typ `any`.

Obdobným způsobem se typy definují u parametrů a návratových hodnot funkcí či metod:

```
function log(text: string): void { ... }
```

Návratový typ se definuje za parametry funkce. Klíčové slovo `void` označuje funkce bez návratové hodnoty.

Na následujících stránkách budou popsány další vlastnosti jazyka TypeScript s důrazem na ty, které byly použity v aplikaci. Podrobný popis těchto vlastností je možné najít v oficiální příručce [8] či v jazykových specifikacích [9].

5.1 Výčtový typ

TypeScript umožňuje definovat výčtové typy v podobné formě, jakou je možné použít v jazycích C# či Java, tedy například:

```
enum AutomatonType {
    Finite,
    Pushdown,
    Turing
}
var type: AutomatonType = AutomatonType.Turing;
```

5.2 Rozhraní

Definice rozhraní patří mezi základní vlastnosti objektově orientovaných jazyků a TypeScript v tomto ohledu není výjimkou. I díky tomu je možné snadno programovat proti rozhraní a ne konkrétní implementaci. Rozhraní se definuje následujícím způsobem:

```
interface ITransition {
    from: IState;
    to: IState;
    midPoint?: Point;
    resetMidPoint(): void;
    isLoop(): boolean;
}
```

Jak je v příkladu vidět, rozhraní může definovat jak metody, tak i vlastnosti, které musí implementující třídy obsahovat. Tím se TypeScriptové rozhraní liší např. od rozhraní v Javě, kde je možné definovat pouze metody [10].

Otazník „`?`" mezi názvem metody a jejím typem označuje nepovinné členy daného rozhraní. Implementující třídy je tedy nemusí nutně obsahovat. Pokud je ale už obsahují, musí dodržet stanovený typ.

Definovaná rozhraní je možné rozšiřovat pomocí klíčového slova `extends`. Třída implementující takto rozšířené rozhraní musí obsahovat všechny prvky z rozšiřujícího i rozšiřovaného rozhraní. Jak bývá v objektově orientovaných jazycích obvyklé, třídy mohou implementovat libovolný počet rozhraní.

5.3 Třídy

Klasický JavaScript neumožňuje definovat třídy způsobem, který je obvyklý v Javě, C# či jiných jazycích. Třídy v JavaScriptu se definují jako funkce [11], zatímco TypeScript umožňuje tuto definici provádět způsobem mnohem bližším výše zmíněným jazykům.

Konstruktor tříd je definovaný klíčovým slovem `constructor`. Parametry konstruktoru mohou obsahovat modifikátory přístupu `public`/`private`. Při použití těchto modifikátorů konstruktor automaticky vytváří ve třídě stejnojmenné vlastnosti s příslušnými modifikátory. Předané hodnoty jsou tak dostupné ve všech metodách a ne pouze v konstruktoru.

```
class Point {
    constructor(public x: number, public y: number) {
    }
    setCoords(x: number, y: number): void {
        this.x = x;
        this.y = y;
    }
}
```

Na rozdíl od mnoha jiných jazyků jsou metody a vlastnosti třídy standardně veřejně přístupné a není nutné uvádět klíčové slovo `public`. Je naopak nutné uvádět klíčové slovo `private` u členů, které takto přístupné být nemají.

Třídy v TypeScriptu podporují dědičnost pomocí klíčového slova `extends` podobně, jako je možné rozšiřovat rozhraní. Pokud děděná třída obsahuje definici konstruktoru, musí konstruktor obsahovat i dědící třída. V konstruktoru této třídy je nutné jako první zavolat konstruktor rodičovské třídy pomocí metody `super()`.

5.4 Moduly

Pomocí modulů je v TypeScriptu možné vytvořit jmenné prostory a tím kód aplikace dělit na logické celky. Moduly jsou definované klíčovým slovem `module`:

```
module Automata.Models {
    export class Point {
        ...
    }
}
```

Standardně nejsou rozhraní a třídy definované v modulu přístupné mimo daný modul. Pokud přístupné být mají, je nutné je označit klíčovým slovem `export`. Jeden modul může být definován v různých souborech, při kompilaci jsou pak všechny spojeny do jednoho celku.

K prvkům definovaným v jiných modulech je nutné přistupovat pomocí celého jejich názvu včetně názvu příslušných modulů. Je ale možné jim vytvořit alias a přistupovat k nim pomocí tohoto aliasu. Alias se vytváří pomocí klíčového slova `import`:

```
import Point = Automata.Models.Point;
var point: Point = new Point(10, 10);
```

5.5 Funkce

Funkce je možné v TypeScriptu definovat stejným způsobem jako v JavaScriptu. Navíc je možné použít i zkrácený zápis. Kromě zkrácení výsledného kódu má tento zápis výhodu i v tom, že klíčové slovo `this` stále odkazuje na objekt, ve kterém je tato funkce volána. Při použití klasického zápisu se na dostupnost tohoto objektu není možné spolehnout [12].

```
var add = (first: number, second: number): number => first + second;
```

5.6 Generické definice

V TypeScriptu je možné definovat typy také genericky. To znamená, že je přípustné definovat typy zástupným symbolem a na tento symbol se pak dále odkazovat. Definice generického rozhraní či třídy pak funguje jako jakási šablona, kterou je možné použít pro různé typy. Konkrétní definice daného typu je pak v šabloně na všech příslušných místech nahrazena a kompilátor dodržování tohoto typu kontroluje.

```
interface IAutomaton<TTransition extends ITransition> {
    states: IState[];
    transitions: TTransition[];
    ...
}
class Transition1 implements ITransition {
    ...
}
class Transition2 implements ITransition {
    ...
}
var automaton: IAutomaton<Transition1>;
automaton.transitions.push(new Transition1());
automaton.transitions.push(new Transition2()); // chyba při kompilaci
```

Ve výše uvedeném příkladu je takovým generickým parametrem `TTransition`. Klíčové slovo `extends` v definici typu omezuje použití generického typu pouze na třídy implementující rozhraní `ITransition`.

Je nutné zdůraznit, že všechny typy, včetně generických, se kontrolují pouze během vývoje a při kompilaci, ale už ne později. Informace o použitých typech jsou při převodu do JavaScriptu ztraceny. V případě silně typových jazyků, jako je např. C#, bývá poměrně obvyklé se na použitý typ dotazovat i za běhu aplikace. To v případě TypeScriptu není možné.

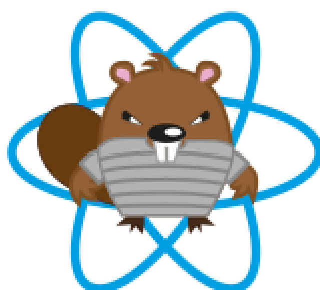
5.7 Definiční soubory

TypeScriptové definiční soubory jsou jakousi obdobou hlavičkových souborů z jazyků C/C++ [13]. Obsahují tedy pouze definice modulů, tříd a rozhraní bez konkrétní implementace. Použití těchto souborů umožní kompilátoru kontrolovat typovou správnost i v případě, že aplikace používá knihovny, u nichž nejsou typy definovány. Jde zejména o knihovny psané přímo v JavaScriptu nebo i ty psané pomocí TypeScriptu, ale distribuované ve zkompilované podobě.

V případě druhých jmenovaných se typové soubory obvykle distribuují společně se samotným kódem, protože je možné je vytvářet automaticky při kompilaci. Jiná situace nastává u knihoven napsaných v JavaScriptu, kde je nutné tyto soubory vytvořit ručně. U nejčastěji používaných knihoven, jako je např. jQuery, AngularJS či D3, je ale možné použít typové soubory z kolekce DefinitelyTyped [14].

6 BOBRIL

Bobril [15] je komponentově orientovaný framework pro tvoru webových aplikací inspirovaný podobně zaměřenými frameworky React [5] a Mithril [16]. Pracuje na principu virtuálního DOMu a je zaměřený zejména na rychlost i za cenu o něco vyšší složitosti programování. Je volně dostupný v podobě zdrojových kódů pod MIT licencí.

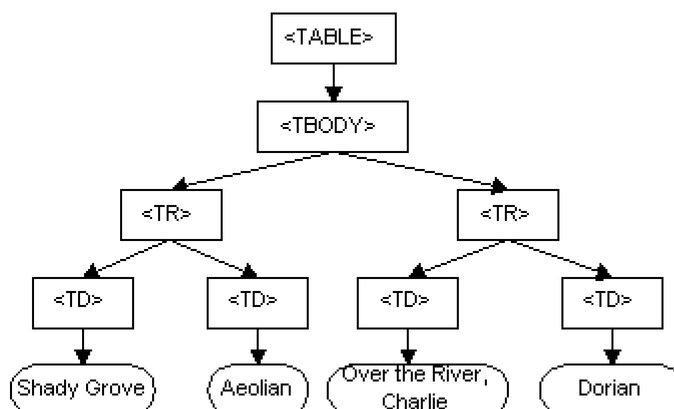


Obrázek 10 Logo Bobrilu [15]

6.1 Virtuální DOM

„DOM neboli Document Object Model je platformově a jazykově neutrální rozhraní, které umožňuje programům a skriptům dynamicky přistupovat a upravovat obsah, strukturu a styl dokumentů." [17]

Document Object Model je reprezentován pomocí stromu, jehož jednotlivé uzly představují elementy HTML stránky (či XML dokumentu) nebo jejich textový obsah. Obrázek 11 zobrazuje příklad takového stromu pro jednoduchou tabulku.



Obrázek 11 Příklad DOM stromu [18]

Každý uzel má ještě velké množství vlastností, které jsou nutné pro správné zobrazení výsledné stránky. Např. obyčejný element `div` má více než 200 vlastností, viz Obrázek 12. Při každé změně DOMu je nutné tyto vlastnosti ověřit a případně změnit, což může být u rozsáhlé stránky časově náročné. A to zejména pokud jde o vlastnosti určující velikost jednotlivých prvků, jejich viditelnost, překrývání apod.

```
> var div = document.createElement('div');
< undefined
> var i = 0;
< undefined
> for (var element in div) { i++; }
< 205
```

Obrázek 12 Počet vlastností DIV elementu, zdroj: autor

Pokud je použit virtuální DOM, změny nejsou aplikovány přímo do DOMu dané stránky, ale do jeho zjednodušeného modelu, kde jednotlivé uzly mají mnohem méně vlastností. V případě Bobrilu jich je pouze 13 a nejsou v nich obsaženy zejména ty, jejichž výpočet je nejvíce náročný, např. již zmíněné vlastnosti určující velikost na stránce.

Teprve v okamžiku, kdy si programátor vyžádá vykreslení provedených změn, Bobril porovná poslední vykreslený virtuální DOM s jeho aktuální podobou a do skutečného DOMu zapíše pouze provedené změny. Není tak nutné překreslovat celou stránku, ale pouze ty části, které byly změněny.

Někdy ale bývá problematické zjistit, které části to byly, což může vést k neočekávaným výsledkům. Málo problematické bývá, pokud je virtuální uzel přidán až za existující uzly (viz Obrázek 13). V tom případě jsou původní uzly opravdu ponechány nezměněné, a je pouze přidán nový.

```
<div>                           <div>
  <span>A</span>  ------------>   <span>A</span>
  <span>B</span>  ------------>   <span>B</span>
  <span>C</span>  ------------>   <span>C</span>
                                  <span>D</span>
</div>                          </div>
```

Obrázek 13 Přidání uzlu za už existující uzly, zdroj: autor

Horší je to v případě, že je uzel přidán před existující uzly (viz Obrázek 14), či dokonce mezi ně. V tom případě se může stát, že Bobril tuto úpravu posoudí jako změnu obsahu existujících uzlů a přidání nového na konec, což neodpovídá skutečnosti.

Obrázek 14 Přidání nového uzlu před existující uzly, zdroj: autor

V tomto případě to, kromě menšího výkonnostního propadu, nebude mít žádné závažné následky. Nicméně mnohé uzly mají svůj vnitřní stav, který je potřeba zachovat a který by byl za podobné situace ztracen. Takovým vnitřním stavem může být například pozice kurzoru uvnitř vstupního pole nebo pozice vertikálního či horizontálního posuvníku.

Chybné identifikaci změn je možné zabránit přiřazením unikátní hodnoty k daným uzlům, která slouží jako jejich identifikátor (viz kapitola 6.4). Při porovnávání předchozího a aktuálního virtuálního DOMu jsou na sebe uzly s těmito identifikátory namapovány přednostně a teprve poté dojde k porovnání ostatních uzlů.

Porovnávání skutečného a virtuálního DOMu neprobíhá okamžitě, ale dávkově při následujícím překreslení obrazovky. Pokud by tedy krátce za sebou bylo provedeno několik změn, jsou tyto změny do skutečného DOMu zapsány najednou.

6.2 Inicializace aplikace

Vstupním bodem aplikace v Bobrilu je metoda **init**, která má následující definici:

```
init(factory: () => IBobrilChildren, element?: HTMLElement): void;
```

Tato metoda přijímá dva parametry:

- **factory** je funkce vracející pole virtuálních uzlů, které budou vykresleny do kořenového uzlu
- **element** je nepovinný parametr určující, který z elementů stránky bude použit jako kořenový uzel. Pokud tento parametr není specifikován, je použit element **<body>**.

Bobril definuje globální objekt `b`, pomocí kterého jsou volány všechny jeho metody. Metoda `init`, tedy může být zavolána například takto:

```
b.init(() => [
      { tag: 'h1', children: 'Ahoj světe!' },
      { tag: 'div', children: 'Aplikace v Bobrilu' }
], document.getElementById('root'));
```

Po tomto zavolání budou do elementu s id „root“ přidány následující dva uzly:

```
<h1>Ahoj světe!</h1>
<div>Aplikace v Bobrilu</div>
```

V aplikaci může být definováno i více kořenových uzlů. Další takové uzly je možné přidávat zavoláním metody `b.addRoot()` a odstraňovat metodou `b.removeRoot()`.

6.3 Virtuální uzly

V předcházejícím příkladu byly jako virtuální uzly použity TypeScriptové objekty, nicméně virtuální uzel může být i pole objektů, pouhý text, či booleovská proměnná. Booleovské proměnné či hodnoty `null` a `undefined` jsou automaticky odstraňovány, vnořená pole jsou zplošťována.

Pokud je použit TypeScriptový objekt, musí implementovat rozhraní `IBobrilChild`. Toto rozhraní má všechny vlastnosti nepovinné, nicméně by vždy měla být použita jedna z vlastností `tag` či `component`.

Vlastnost `tag` udává, jaký tag bude vykreslen na stránce. Tento typ uzlu se používá, pokud je potřeba přidat na stránku přidat element bez nějaké vlastní logiky. Pokud je jako tag uvedeno lomítko `/`, text uvedený ve vlastnosti `children` se bere jako `innerHTML` [19]. V žádném případě by se tímto způsobem nemělo na stránku vkládat nic, co do aplikace přišlo jako vstup od uživatele. Aplikace by v takovém případě mohla být zranitelná Cross-site Scripting (XSS) útokem [20].

Vlastnost `component` specifikuje použitou komponentu. Protože komponenty představují základní stavební kameny aplikace v Bobrilu, bude jim věnována větší pozornost v následující části této práce.

Další často používané vlastnosti definované rozhraním `IBobrilChild` jsou:

- `className` definuje atribut class HTML tagu. Název `class` není možné použít, protože jde o klíčové slovo TypeScriptu.

- `style` definuje inline styl HTML tagu. Styl může být určen pomocí TypeScriptového objektu (doporučená varianta), či přímo jako text.
- `attrs` definuje další atributy HTML tagu. Opět je zde použit TypeScriptový objekt.
- `data` je vlastnost používaná společně s komponentami. Představuje data předávaná z rodičovské komponenty do potomků.

Obrázek 15 zobrazuje příklady, jakým způsobem jsou některé virtuální uzly převedeny do HTML.

HTML	Bobril
	{ tag:"div" }
Hello	{ tag:"span", children:"Hello" }
Ex	{ tag:"a", attrs: { href:"url" }, children:"Ex" }
Big	{ tag:"span", style: { textSize:"20px" }, children:"Big" }
<h1 class="head"><</h1>	{ tag:"h1", className:"head", children:"<" }
style="float:left"	style: { cssFloat:"left" }
 	{ tag:"div", children: { tag:"br" } }
AB	{ tag:"div", children: [{ tag:"span", children: "A" }, "B"] }

Obrázek 15 Příklady převodu virtuálních uzlů do skutečných [21]

V tabulce jsou uvedeny i určité zvláštnosti či výjimky:

- pokud je ve stylu použita vlastnosti `text-size` musí být zapsána jako `textSize`. Stejný princip platí i u ostatních vlastností, které mají v názvu pomlčku.
- CSS vlastnost `float` musí být zapsána jako `cssFloat`. Opět je to z toho důvodu, že `float` je klíčové slovo TypeScriptu.

Problém s klíčovými slovy TypeScriptu se objeví také při použití HTML atributu `for`, který musí být zapsán jako `htmlFor`.

Pokud je styl uzlu definován jako objekt a ne jako textový řetězec, provádí Bobril určité úpravy, které zadávání stylů zjednodušují, a je tedy vhodné tuto formu preferovat. Například vlastnosti `left` a `top`, které musí být v CSS definovány s postfixem `px`, stačí v Bobrilu zapsat pouze číslem. Bobril dále usnadňuje používání CSS vlastností, které je nutné pro různé prohlížeče definovat zvlášť, například:

```
style: { userSelect: 'none' }
```

Je do CSS převedeno takto:

```
{
  -webkit-user-select: none;
  -ms-user-select: none;
  -moz-user-select: none;
}
```

V případě, že Bobril správně detektuje typ prohlížeče, nejsou použity všechny tyto vlastnosti, ale pouze ta, která odpovídá použitému prohlížeči.

6.4 Komponenty

Jak již bylo napsáno, komponenty představují základní stavební kameny aplikace v Bobrilu. Od obyčejných virtuálních uzlů se liší zejména tím, že mohou obsahovat i logiku. Mezi jejich další výhody patří zejména znovupoužitelnost a možnost skládat z nich větší celky, které nejsou ničím jiným, než opět komponentami. Vytvoření aplikace pak může připomínat skládání Lega, kde jednotlivé kostičky jsou základní komponenty (tlačítka, vstupní pole atd.), které po složení vytvoří další větší komponenty (menu, formulář atd.), z nichž je pak složena celá aplikace.

Komponenta v Bobrilu musí implementovat rozhraní `IBobrilComponent`, které definuje metody používané v jednotlivých částech životního cyklu komponenty (viz str. 33). Všechny tyto metody jsou nepovinné, nicméně zejména metoda `render` je používaná téměř v každé komponentě. Rozhraní je definované následovně:

```
interface IBobrilComponent {
    id?: string;
    init?(ctx: IBobrilCtx, me: IBobrilCacheNode): void;
    render?(ctx: IBobrilCtx, me: IBobrilNode, oldMe?: IBobrilCacheNode): void;
    postRender?(ctx: IBobrilCtx, me: IBobrilNode, oldMe?: IBobrilCacheNode): void;
    shouldChange?(ctx: IBobrilCtx, me: IBobrilNode, oldMe: IBobrilCacheNode):
boolean;
    postInitDom?(ctx: IBobrilCtx, me: IBobrilCacheNode, element: HTMLElement):
void;
    postUpdateDom?(ctx: IBobrilCtx, me: IBobrilCacheNode, element: HTMLElement):
void;
    destroy?(ctx: IBobrilCtx, me: IBobrilNode, element: HTMLElement): void;
    shouldStopBubble?(ctx: IBobrilCtx, name: string, param: Object): boolean;
    shouldStopBroadcast?(ctx: IBobrilCtx, name: string, param: Object): boolean;
}
```

Všechny metody životního cyklu přijímají nejméně dva parametry. Prvním z nich je kontext dané komponenty, druhým pak virtuální uzel, na který je komponenta navěšena. Komponenty v Bobrilu nemohou existovat samostatně, musí vždy patřit do nějakého virtuálního uzlu. Tento

uzel ale nutně nemusí odpovídat jednomu uzlu ve skutečném DOMu. Ve skutečnosti může jeden virtuální uzel komponenty vytvářet libovolný počet skutečných uzlů.

Komponenty mají volitelnou vlastnost `id`, která slouží jako identifikátor při porovnávání předchozího a současného stavu DOMu, viz kapitola 6.1.

6.4.1 Kontext komponenty

Kontext komponenty je objekt udržující informace o jejím vnitřním stavu. Přestože je vhodné co nejvíce používat bezestavové komponenty, ne vždy je to možné. K udržování tohoto stavu je určena proměnná `ctx`, která je předávána do všech metod jako první parametr. Je nutné zdůraznit, že se jedná opravdu o vnitřní stav komponenty, neměl by tedy být dostupný či ovlivnitelný zvenčí.

Z tohoto pravidla existují v dvě výjimky. Tou první je nastavení počáteční hodnoty stavu komponenty při jejím vytvoření, a tou druhou je proměnná `ctx.data`, která představuje data předávaná z nadřazené komponenty. U ní naopak platí pravidlo, že komponenta by tato data nikdy neměla přímo měnit. Může pouze oznámit nadřazené komponentě, že se stala nějaká událost (kliknutí na tlačítko, stisk klávesy atd.), a je pak pouze na nadřazené komponentě, jak a zda vůbec na tuto událost nějakým způsobem zareaguje.

Příklad předání dat z jedné komponenty do druhé:

```
var parent: IBobrilComponent = {
    render(ctx: IBobrilCtx, me: IBobrilNode): void {
        me.tag = 'div';
        me.children = { component: child, data: { name: 'Milan', age: 35 } };
    }
}

var child: IBobrilComponent = {
    render(ctx: IBobrilCtx, me: IBobrilNode): void {
        me.tag = 'span';
        me.children = ctx.data.name + ' (' + ctx.data.age + ')';
    }
}
```

V tomto příkladu byly použity dvě komponenty, z nichž jedna (`parent`) předává data do druhé (`child`). Data jsou reprezentována objektem s vlastnostmi `name` a `age`, která jsou pak v podřízené komponentě dostupné pomocí objektu `ctx.data`, konkrétně tedy `ctx.data.name` a `ctx.data.age`. Použití komponenty `parent` pak vede k následujícímu HTML kódu:

```
<div><span>Milan (35)</span></div>
```

6.4.2 Životní cyklus komponenty

Během životního cyklu komponenty jsou volány různé její metody, v závislosti na tom, v jaké fázi tohoto cyklu se komponenta právě nachází. Komponenta nemusí implementovat všechny tyto metody, ale měla by implementovat alespoň jednu, jinak by nebyla použitelná.

```
init(ctx: IBobrilCtx, me: IBobrilCacheNode): void;
```

První metoda je `init`, která se pro každou komponentu volá pouze jednou a to právě před tím, než je komponenta přidána do virtuálního DOMu. Jak už její název naznačuje, slouží hlavně k inicializaci vnitřního stavu komponenty.

```
shouldChange(ctx: IBobrilCtx, me: IBobrilNode, oldMe: IBobrilCacheNode): boolean;
```

Metoda `shouldChange` je volána před každým voláním metody `render`, kromě zcela prvního pro danou komponentu. Její výsledek určuje, zda se komponenta nějakým způsobem od posledního volání změnila a zda tedy má být překreslena. Pokud tato metoda vrátí `false`, žádné následující metody této komponenty ani jejích podřízených komponent se už neprovádějí.

Tuto metodu není za normálních okolností potřeba implementovat. Slouží zejména k optimalizaci rychlosti v případě použití velmi rozsáhlého virtuálního DOMu.

```
render(ctx: IBobrilCtx, me: IBobrilNode, oldMe?: IBobrilCacheNode): void;
```

Hlavní a nejčastěji používaná metoda všech komponent. Zajišťuje samotné vykreslení komponenty a jejích podřízených komponent, kterým také předává data.

```
postRender(ctx: IBobrilCtx, me: IBobrilNode, oldMe?: IBobrilCacheNode): void;
```

Tato metoda je volána poté, co jsou zavolány metody `render` dané komponenty a všech jejích podřízených komponent. Je to zároveň i poslední metoda, která se volá před vykreslením virtuálního DOMu do skutečného.

```
postInitDom(ctx: IBobrilCtx, me: IBobrilCacheNode, element: HTMLElement): void;
```

Po prvním vykreslení dané komponenty do skutečného DOMu je zavolána metoda `postInitDom`. Tato metoda už by neměla pracovat s virtuálním DOMem, zato má k dispozici objekt `element`, který představuje skutečný element na stránce. Toto se hodí v případě, že je potřeba znát informace o uzlu, které nejsou ve virtuálním DOMu dostupné, například jeho rozměry. Nebo v případě, že je potřeba použít nějaké funkce či knihovny, které nejsou součástí Bobrilu, například jQuery.

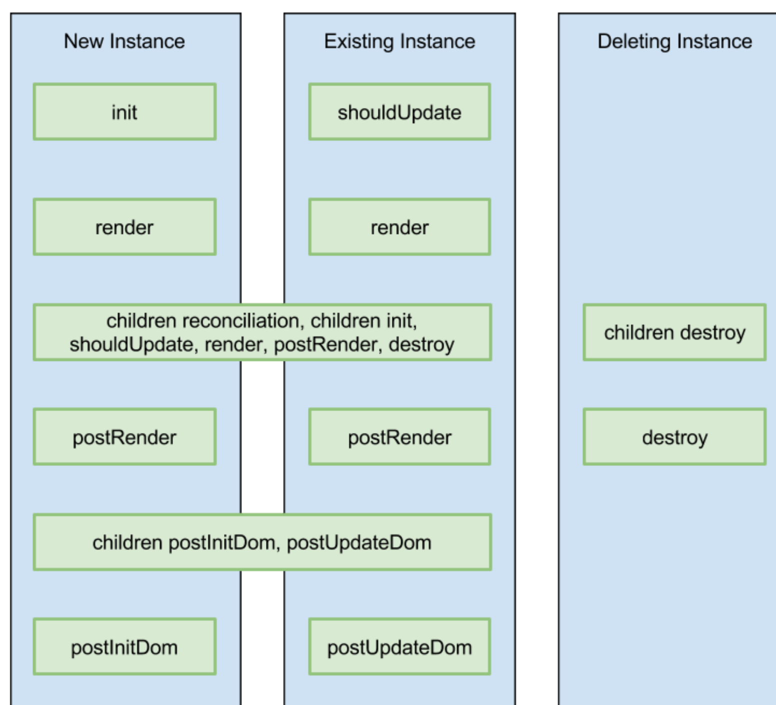
Objekt `element` je dostupný pouze v případě, že komponenta vytváří právě jeden uzel ve skutečném DOMu. Pokud vytváří jiný počet, není možné namapovat virtuální uzel komponenty na skutečný. Stále je ovšem možné manipulovat se skutečným DOMem, např. pomocí volání `document.getElementById`.

```
postUpdateDom(ctx: IBobrilCtx, me: IBobrilCacheNode, element: HTMLElement): void;
```

Tato metoda má v zásadě stejné použití a význam jako předchozí. Liší se pouze v tom, že není nikdy volána po prvním vykreslení komponenty do skutečného DOMu, ale až při všech ostatních.

```
destroy(ctx: IBobrilCtx, me: IBobrilNode, element: HTMLElement): void;
```

Metoda `destroy` je volána pouze jednou, a to právě před tím, než je element komponenty odstraněn z DOMu.



Obrázek 16 Životní cyklus komponenty [21]

Obrázek 16 zobrazuje celý životní cyklus komponenty. První sloupec popisuje cyklus nově vytvořené komponenty, zatímco druhý už existující komponenty. V zásadě jsou velmi podobné, liší se pouze voláním metod `init` resp. `shouldUpdate` a `postInitDom` resp. `postUpdateDom`. Poslední sloupec pak ukazuje, že při odstraňování komponenty jsou nejdříve odstraněny všechny její podřízené komponenty.

Za zmínku stojí ještě to, že první dvě metody (tedy `init`/`shouldUpdate` a `render`) jsou provedeny nejdříve pro nadřazenou komponentu a teprve poté pro podřízené, zatímco u metody `postRender` je to naopak. A naopak je to také u metod `postInitDom`/`postUpdateDom`. Díky tomu jsou ve všech metodách dostupné podřízené komponenty, či jimi vytvořené uzly, pro případné další úpravy.

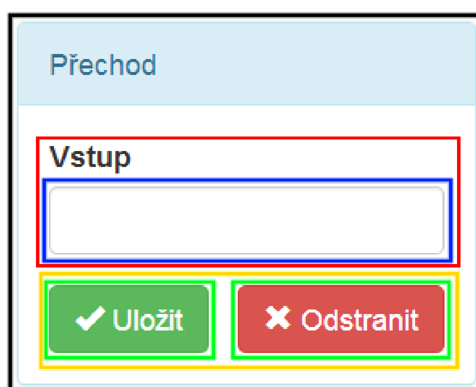
Zatímco metody `init` a `postInitDom` a `destroy` jsou pro každou komponentu zavolány pouze jednou, metody `shouldUpdate` až `postUpdateDom` jsou volány při každém vykreslení komponenty. Kdy k tomuto vykreslení dojde, je pouze na programátorovi, přesněji řečeno na tom, kdy vyvolá metodu `invalidate`.

Zavolání této metody bez parametrů způsobí překreslení celého virtuálního DOMu, pokud nebude v nějaké komponentě zastaveno metodou `shouldUpdate`. Metoda `invalidate` přijímá dva nepovinné parametry. Prvním z nich je kontext komponenty, která má být překreslena, a druhým je hloubka do jaké má Bobril jít při vykreslování podřízených komponent.

Příklady použití:

```
b.invalidate();
b.invalidate(ctx);
b.invalidate(ctx, 1);
b.invalidate(ctx, 0);
```

První volání způsobí překreslení celého virtuálního DOMu, druhé pouze aktuální komponenty a jejích podřízených komponent. Třetí vykreslí komponentu a její podřízené komponenty, ale už ne podřízené komponenty těchto komponent. Poslední volání vykreslí pouze samotnou komponentu.



Obrázek 17 Skládání komponent do větších celků, zdroj: autor

Obrázek 17 naznačuje, jak se v aplikaci skládají základní komponenty do větších celků. Jsou zde použity dva typy základních komponent: tlačítko (`button` - zelený rámeček) a vstupní pole

(`input` - modrý rámeček). Tlačítka jsou seskupena do komponenty určené pro uložení provedených změn či pro odstranění daného prvku (`saveAndDelete` - žlutý rámeček). V aplikaci se totiž podobná dvojice vyskytuje dvakrát.

Vstupní pole je spojeno s virtuálním uzlem reprezentujícím jeho nadpis do komponenty `inputWithLabel` (červený rámeček). Komponenty `inputWithLabel` a `saveAndDelete` jsou vloženy do komponenty `panel` (černý rámeček) a to celé tvoří komponentu `transitionInfo`, která se používá ve všech třech typech automatů.

6.5 Rozšíření pro Bobril

Aby byl výsledný kód Bobrilu co nejmenší, byly jeho funkce rozděleny do několika modulů či rozšíření a je na programátorovi, která z nich použije. Na předchozích stránkách bylo popsáno pouze společné jádro. Další část této práce se bude věnovat právě těmto rozšířením.

6.5.1 Focus

Rozšíření Focus slouží k detekci, zda je komponenta aktivním prvkem (tj. má „focus“). Typicky se používá ve spojení s formulářovými prvky (tlačítko, vstupní pole atd.) Do komponent přidává následující metody:

```
onFocus(ctx: Object): void;
```

Tato metoda je vyvolána poté, co komponenta získá focus. Tedy například v případě, že uživatel klikne do vstupního pole, aby do něj mohl psát.

```
onBlur(ctx: Object): void;
```

Metoda `onBlur` je vyvolána poté, co komponenta ztratí focus. Například když se uživatel posune na další vstupní pole ve formuláři.

```
onFocusIn(ctx: Object): void;
```

Tato metoda je podobná metodě `onFocus`, ale je vyvolána poté, co nějaká podřízená komponenta získá focus. Není ale vyvolána v případě, že předchozí aktivní komponenta byla komponentou, které patří tato metoda, či některá z jejích podřízených komponent.

```
onFocusOut(ctx: Object): void;
```

Jde v podstatě o opak předchozí metody. Je vyvolána v případě, že podřízená komponenta ztratí focus a zároveň není aktivní aktuální komponenta či některá z jejích podřízených komponent.

Kromě těchto čtyř metod ještě toto rozšíření přidává následující dvě metody do globálního objektu `b`.

```
focused(): IBobrilCacheNode;
```

Vrací aktuální aktivní komponentu.

```
focus(node: IBobrilCacheNode): boolean;
```

Nastaví komponentu jako aktivní. Návratová hodnota určuje, zda bylo nastavení úspěšné.

Příklad použití rozšíření Focus v komponentě `input`:

```
interface IData {
    value: string;
    shouldFocus?: boolean;
    onBlur?(): void;
}

interface ICtx extends IBobrilCtx {
    data: IData;
}

var inputComponent: IBobrilComponent = {
    render(ctx: ICtx, me: IBobrilNode): void {
        me.tag = 'input';
        me.attrs = { type: 'text', value: ctx.data.value };
    },
    postInitDom(ctx: IBobrilCtx, me: IBobrilCacheNode, element:HTMLElement):void {
        if (ctx.data.shouldFocus) {
            b.focus(me);
        }
    },
    onBlur(ctx: ICtx): void {
        if (ctx.data.onBlur) {
            ctx.data.onBlur();
        }
    }
};

function input(data: IData): IBobrilNode {
    return { component: inputComponent, data: data };
}

b.init(() => input({
    value: 'Ahoj světe',
    shouldFocus: true,
    onBlur: () => alert('Zavoláno onBlur')
}));
```

V příkladu bylo nejprve definováno rozhraní `IData`, které má vlastnosti `value` (hodnota zobrazená ve vstupním poli), `shouldFocus` (určuje, zda má být vstupní pole aktivní) a `onBlur`

což je tzv. callback [22], tedy funkce, která se má zavolat poté, co vstupní pole přestane být aktivní.

Následuje definice rozhraní `ICtx` (kontext komponenty `input`) s vlastností `data` typu `IData`. A poté definice vlastní komponenty `inputComponent`. V komponentě jsou definovány metody `render`, `postInitDom` a `onBlur`. V metodě `render` je vytvořen element input typu text s hodnotou určenou vstupními daty. Metoda `postInitDom` zajišťuje, že komponenta bude po vytvoření aktivní (pokud je to tak nastaveno ve vstupních datech) a nakonec metoda `onBlur` volá již zmíněný callback.

Následuje funkce `input`, která přijímá data a vrací virtuální uzel s komponentou `inputComponent`, do které jsou tato data předána. V poslední části kódu je inicializace samotné aplikace, která zajistí vykreslení výše uvedené komponenty.

6.5.2 Mouse

Rozšíření Mouse slouží k tomu, aby komponenty mohly reagovat na události spojené s používáním myši či dotykového displeje. Při použití tohoto rozšíření mohou komponenty používat následující metody:

```
onClick(ctx: Object, event: IBobrilMouseEvent): boolean;
```

Tato metoda je vyvolána po kliknutí myši na komponentu. Návratová hodnota určuje, zda komponenta tuto událost zachytila. Pokud metoda vrátí `false`, mohou tuto událost zachytit komponenty, které leží na stránce pod ní. Jde o tzv. „event bubbling“ [23]. Tento typ návratového hodnoty vrací většina metod z tohoto rozšíření.

V objektu `event`, který je předáván jako druhý parametr, jsou souřadnice dané události, dále které tlačítko myši ji způsobilo, a zda při ní byly stisknuty klávesy Ctrl, Shift či Alt. Tento parametr je opět společný všem metodám. Hodnoty v tomto objektu jsou nezávislé na použitém prohlížeči, protože Bobril provádí jejich normalizaci.

```
onDoubleClick(ctx: Object, event: IBobrilMouseEvent): boolean;
```

Metoda je vyvolána po dvojitém kliku na komponentu.

```
onMouseDown(ctx: Object, event: IBobrilMouseEvent): boolean;
onMouseUp(ctx: Object, event: IBobrilMouseEvent): boolean;
```

Metody jsou vyvolány po stisknutí resp. uvolnění tlačítka myši.

```
onMouseEnter(ctx: Object, event: IBobrilMouseEvent): void;
onMouseLeave(ctx: Object, event: IBobrilMouseEvent): void;
```

Metody jsou vyvolány po přejetí myší nad komponentu resp. po přejetí myši mimo komponentu.

```
onMouseMove(ctx: Object, event: IBobrilMouseEvent): boolean;
```

Metoda je vyvolána při každém pohybu myši po komponentě.

```
onPointerDown(ctx: Object, event: IBobrilPointerEvent): boolean;
onPointerUp(ctx: Object, event: IBobrilPointerEvent): boolean;
onPointerMove(ctx: Object, event: IBobrilPointerEvent): boolean;
```

Tyto metody významově odpovídají metodám `onMouseDown`, `onMouseUp` a `onPointerMove`, ale na rozdíl od nich dokáží reagovat nejen na události myši, ale i pohybu prstu po dotykovém displeji. V aplikaci jsou použity výhradně tyto metody.

Druhý parametr (`event`) těchto tří metod je mírně odlišný od předchozích. Kromě výše zmíněných vlastností nese i informaci o tom, zda k události došlo po použití myši, dotyku prstu či dotykového pera.

Rozšíření Mouse poskytuje ještě metody rozšiřující globální objekt `b`. Jde zejména o tyto tři:

```
registerMouseOwner(ctx: any): void;
```

Umožňuje zaregistrovat komponentu jako „vlastníka“ myši. Komponenta pak dokáže reagovat na události i v případě, že se myš nenachází nad ní. V aplikaci je tohoto principu použito při pohybování s prvky automatu. Při stisku tlačítka (tedy v metodě `onPointerDown`) se komponenta zaregistruje jako vlastník, v metodě `onPointerMove` si kontroluje, zda je stále vlastníkem a v metodě `onPointerUp` pak toto vlastnictví uvolňuje. Pokud by k této registraci nedošlo, komponenta by při rychlém pohybu myši, kdy by se kurzor dostal mimo prostor komponenty, přestala reagovat na potřebné události.

```
isMouseOwner(ctx: any): boolean;
```

Vrací, zda je komponenta vlastník myši.

```
releaseMouseOwner(): void;
```

Uvolňuje vlastnictví myši.

Následuje ukázka použití těchto metod z komponenty `midPoint`:

```
var midPoint: IBobrilComponent = {
    onPointerDown(ctx: ICtx, event: IBobrilPointerEvent): boolean {
        b.registerMouseOwner(ctx);
```

```
        ...
        return true;
    },
    onPointerUp(ctx: ICtx, event: IBobrilPointerEvent): boolean {
        if (b.isMouseOwner(ctx)) {
            b.releaseMouseOwner();
        }
        return true;
    },
    onPointerMove(ctx: ICtx, event: IBobrilPointerEvent): boolean {
        if (!b.isMouseOwner(ctx)) {
            return false;
        }
        ...
        b.invalidate();
        return true;
    }
};
```

6.5.3 OnChange

Rozšíření OnChange umožňuje komponentě reagovat na změnu její hodnoty. Používá se výhradně ve spojení s formulářovými prvky. Komponenty rozšiřuje o jedinou metodu, která je vyvolána při změně hodnoty komponenty:

```
onChange(ctx: Object, value: any): void;
```

Druhý parametr (`value`) je aktuální hodnota komponenty. V případě textového pole jde o text v tomto poli, v případě zaškrtávacího políčka informace, zda je zaškrtnuto apod.

V následujícím příkladě je použita komponenta z kapitoly 6.5.1, která je rozšířena o metodu `onChange`.

```
interface IData {
    ...
    onChange(value: string): void;
}

var inputComponent: IBobrilComponent = {
    render(ctx: ICtx, me: IBobrilNode): void {
        ...
    },
    postInitDom(ctx: IBobrilCtx, me: IBobrilCacheNode, element:HTMLElement):void {
        ...
    },
    onBlur(ctx: ICtx): void {
        ...
    },
    onChange(ctx: Object, value: any): void {
        ctx.data.onChange(value);
    }
};
```

Na změnu hodnoty komponenty je opět reagováno pomocí callbacku. Tím je zodpovědnost za zpracování změny přesunuta na nadřazenou komponentu. To odpovídá pravidlu, že komponenta by neměla měnit data, která jsou jí předána z nadřazené komponenty. Tato nadřazená komponenta pak může vyvolat překreslení DOMu (zavoláním `b.invalidate()`), pokud usoudí, že je to potřeba.

6.5.4 OnKey

Toto rozšíření umožňuje komponentám reagovat na stisk klávesy. V komponentě je možné použít následující metody:

```
onKeyDown(ctx: Object, event: IKeyDownUpEvent): boolean;
onKeyUp(ctx: Object, event: IKeyDownUpEvent): boolean;
onKeyPress(ctx: Object, event: IKeyPressEvent): boolean;
```

Tyto metody jsou vyvolány po přidržení (`onKeyDown`), uvolnění (`onKeyUp`) a kompletním stisku klávesy (`onKeyPress`). Druhý parametr (`event`) obsahuje informace o tom, která klávesa byla stisknuta a zda byla stisknuta společně s klávesami Ctrl, Shift či Alt.

Tyto události reagují pouze u aktivních prvků, tedy u těch, které mají „focus“. Standardně mohou mít focus pouze formulářové prvky, a proto pokud je potřeba, aby focus přijímaly i jiné prvky (např. tag `<div>`), je nutné jim přidat atribut `tabindex` [24]. Takto je to v aplikaci řešeno v komponentě `canvas`, která potřebuje reagovat na stisk klávesy Ctrl, protože se pomocí ní přepíná, zda pohyb myši hýbe se stavem automatu, nebo vytváří nový přechod mezi stavy. Musí také reagovat na stisk klávesy Delete, pomocí které se maže označený stav či přechod.

```
var canvas: IBobrilComponent = {
    render(ctx: ICtx, me: IBobrilNode): void {
        me.tag = 'div';
        me.className = 'canvas';
        me.attrs = { tabindex: 0 };
        ...
    }
};
```

6.5.5 Router

Zatímco přecházející rozšíření sloužila zejména k přidávání nových vlastností do komponent, rozšíření Router ovlivňuje chování celé aplikace. Umožňuje aplikaci reagovat na změnu URL adresy, resp. její části nazývané „hash“ [25]. Hash je část URL adresy, která se nachází za znakem `#` (včetně), v následujícím případě jde tedy o `#/hash`:

```
http://www.domain.com/path/#/hash
```

Tato část adresy se v Bobrilu a i jiných Single Page aplikacích používá k tomu, aby různé části aplikace měly různý vzhled, přestože fakticky bude v prohlížeči zobrazena stále ta samá stránka, ze které se nějaké komponenty odstraní a další zase přidají. Podstatné je, že nedojde ke znovunačtení celé stránky a aplikace si tak stále v paměti drží svůj stav.

V aplikaci je možné pracovat se třemi různými druhy automatů a těm odpovídají tři různé virtuální stránky neboli „routy". Jde konkrétně o tyto routy: `#/finite` (konečný automat), `#/pushdown` (zásobníkový automat) a `#/turing` (Turingův stroj). Přestože tyto routy definují různé stránky, mají společné aplikační menu, které zůstává stejné po celou dobu běhu aplikace.

Použití rozšíření Router mění způsob inicializace aplikace. Nepoužívá se při tom metoda `init`, nýbrž metoda `routes`:

```
b.routes(
    b.route({ handler: NavBar, data: routeData }, [
        b.route({
            name: 'finite',
            handler: Finite,
            data: routeData
        }),
        b.route({
            name: 'pushdown',
            handler: Pushdown,
            data: routeData
        }),
        b.route({
            name: 'turing',
            handler: Turing,
            data: routeData
        }),
        b.routeDefault({
            handler: Finite,
            data: routeData
        })
    ])
);
```

Uvnitř metody `routes` se na rozdíl o metody `init` nedefinují jednotlivé virtuální uzly, ale právě jednotlivé routy pomocí metody `route`, která má následující předpis:

```
route(config: IRouteConfig, nestedRoutes?: Array<IRoute>): IRoute;
```

První parametr této metody je konfigurace dané routy, tedy zejména její jméno (vlastnost `name`), URL (pokud není uvedeno, použije se jméno), dále komponenta, která se má vykreslit (vlastnost `handler`) a data, která jsou komponentě předána pomocí jejího kontextu (vlastnost `data`).

Druhý parametr metody je nepovinný a umožňuje, aby jedna routa obsahovala zanořené routy, které jsou opět definovány metodami `route`. Díky použití zanořených rout je možné, aby při

změně routy zůstala část aplikace stejná a změnil se pouze zbytek. Jak je v příkladu vidět, všechny routy pro různé typy automatů jsou zanořeny do routy s navigačním menu.

Pokud jsou definovány zanořené routy, musí být v nadřazené komponentě definováno místo, kam se má příslušná vnořená routa vykreslit. Toto místo se definuje zavoláním metody `activeRouteHandler`, viz následující příklad:

```
var navBar: IBobrilComponent = {
    render(ctx: ICtx, me: IBobrilNode): void {
        me.children = [
            ...
            me.data.activeRouteHandler()
        ];
    }
};
```

Za povšimnutí jistě stojí, že tato metoda je do komponenty předána pomocí jejích dat (`me.data.activeRouteHandler()`) a není tedy volána pomocí globálního objektu `b` jako v případě většiny ostatních metod. Je to z toho důvodu, že aplikace může obsahovat více komponent s vnořenými routami, a musí tak být zřejmé, které komponenty se mají vykreslit.

Rozšíření Router kromě již zmíněných metod `routes` a `route` přidává do objektu `b` ještě několik dalších metod, z nichž nejdůležitější jsou metody `routeDefault`, `routeNotFound` a `link`.

```
routeDefault(config: IRouteConfig): IRoute;
```

Tato metoda umožňuje definovat routu, která bude použita, pokud je hash část URL prázdná. Uživateli se tak obvykle zobrazí úvodní obrazovka aplikace.

```
routeNotFound(config: IRouteConfig): IRoute;
```

Tato metoda definuje routu, která se použije v případě, že URL sice obsahuje hash, ale neodpovídá žádné známé routě. Používá se tedy zejména pro zobrazení chybové hlášky.

```
link(node: IBobrilNode, name: string, params?: Params): IBobrilNode;
```

Metoda `link` slouží k vytvoření odkazu na jinou routu. První parametr této metody je virtuální uzel, který má sloužit jako odkaz. Typicky jde o nějaký text či tlačítko. Druhým parametrem je název routy, na kterou se má aplikace přesměrovat. A pomocí třetího je možné do routy předat parametry.

Příklad vytvoření odkazu s parametrem:

```
b.link({ tag: 'div', children: 'Otevři dokument'}, 'document', { id: 157 });
```

Toto zavolání vede k vytvoření následujícího elementu na stránce:

```
<a href='#/document/157'><div>Otevři dokument</div></a>
```

Komponenta obsluhující routu `document` si pak může hodnotu předaného parametru přečíst v jí předaných datech v objektu `routeParams`: `ctx.data.routeParams.id`.

6.5.6 Další rozšíření

Bobril má v současné době ještě více rozšíření, která ale v aplikaci nejsou použita, proto zde budou popsána pouze velmi stručně.

Smysl rozšíření DnD (Drag and Drop) [26] je zcela zřejmý už z jeho názvu. Slouží k podpoře operace „táhni a pusť".

Rozšíření L10n (Localization [27]) slouží k podpoře více jazyků v aplikaci.

Pomocí rozšíření Media [28] je možné získat informace o používaném zařízení (typ zařízení, rozlišení a orientace obrazovky).

Rozšíření Promise slouží k podpoře definování závislostí asynchronních volání [29].

Rozšíření Scroll [30] umožňuje komponentám reagovat na scrollování.

Pomocí rozšíření Swipe mohou komponenty reagovat na horizontální pohyby prstu po dotykovém displeji.

Rozšíření VG [31] slouží ke zjednodušení práce s vektorovou grafikou SVG. Do verze 2.0 umožňoval Bobril pomocí tohoto rozšíření simulovat použití SVG i v Internet Exploreru verze 8, která SVG nepodporuje. Ve verzi 3.0 byla ale podpora IE8 z Bobrilu odstraněna. V aplikaci jsou použity i SVG prvky, které toto rozšíření nepodporuje, a proto v ní nakonec nebylo použito.

Poslední rozšíření, které tu bude zmíněno, je rozšíření Style, které slouží k definování kaskádových stylů a které tak umožňuje, aby aplikace nemusela používat CSS soubory. Ve výsledku tak celá aplikace může být tvořena pouze JavaScriptovým kódem na rozdíl od klasické trojice HTML + CSS + JS.

Toto rozšíření nebylo v době tvorby aplikace dostupné, a proto v ní není použito, nicméně v dalších verzích už jistě použito bude.

7 SCALABLE VECTOR GRAPHICS

„Scalable Vector Graphics (SVG) je značkovací jazyk, který je určen pro popis dvourozměrné vektorové grafiky. SVG je podporováno všemi moderními prohlížeči jak na počítačích, tak i na mobilních zařízeních.“ [32]

Obrázek 18 Logo SVG [32]

Z běžně používaných prohlížečů není SVG podporováno pouze v Internet Exploreru ve verzi 8 a starších [33].

SVG je definováno pomocí XML dokumentů, které mohou být vloženy přímo do zdrojového kódu webové stránky nebo i použity samostatně. V případě vložení přímo do zdrojového kódu stránky je nutné definovat správný jmenný prostor použitých tagů [34]:

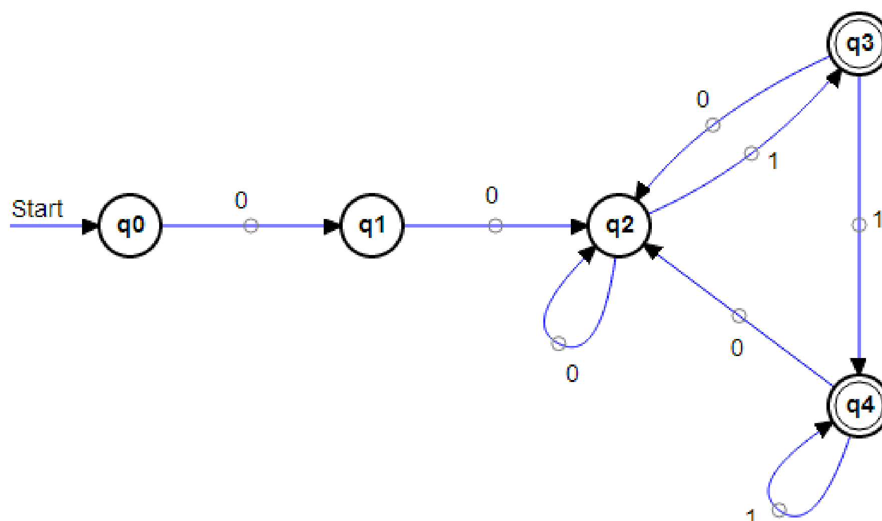
```
<svg xmlns="http://www.w3.org/2000/svg"></svg>
```

Tento jmenný prostor je také potřeba použít v případě dynamického vkládání elementů na stránku, není tedy možné je přidávat zavoláním metody `document.createElement()`. Místo toho je potřeba použít metodu `document.createElementNS()`:

```
var circle = document.createElementNS('http://www.w3.org/2000/svg', 'circle');
```

V případě použití Bobrilu je tento jmenný prostor automaticky přidáván k `svg` uzlům a také všem jejich pořízeným elementům.

V aplikaci je SVG použito k samotnému vykreslení modelu automatu, viz Obrázek 19.



Obrázek 19 Příklad použití SVG v aplikaci, zdroj: autor

Jak je z obrázku vidět, jsou v aplikaci použity pouze základní prvky, nicméně SVG může být použito i k vykreslení podstatně složitějších obrázků. Ostatně i samotné logo SVG (viz Obrázek 18) je vytvořeno pomocí SVG.

Za správné vykreslení automatu jsou zodpovědné komponenty `canvas`, `graph`, `vertex` a `edge`, které budou popsány na následujících stránkách.

7.1 Canvas

Komponenta `canvas` vytváří na stránce samotný uzel `<svg>` a reaguje na stisky klávesnice. V případě, že byla spuštěna simulace automatu, překryje tato komponenta automat průhledným elementem `div`, což zajišťuje, že automat nemůže být v tomto okamžiku změněn.

7.2 Graph

Komponenta `graph` je vytvářena komponentou `canvas` a sama vytváří virtuální uzly s komponentami `vertex` a `egde`, které reprezentují jednotlivé stavy resp. přechody. Kromě toho ještě vytváří definice stylů pro přechody, přesněji řečeno pouze tu jejich část, která zajišťuje vykreslení šipky na konci přechodu. Celá definice těchto stylů vypadá takto:

```
<defs>
  <marker id="arrow" markerWidth="30" markerHeight="30" refX="28" refY="4"
orient="auto" markerUnits="userSpaceOnUse">
    <path d="M0,0L0,9L11,4L0,0"></path>
  </marker>
  <marker id="arrow-new-line" markerWidth="30" markerHeight="30" refX="10" refY="4"
orient="auto" markerUnits="userSpaceOnUse">
```

```
    <path d="M0,0L0,9L11,4L0,0"></path>
  </marker>
</defs>
```

Definice je uvozena elementem `defs`, který obsahuje dva elementy `marker`. V aplikaci jsou totiž použity dva typy šipek. Jeden pro použití v již existujících přechodech, druhý v přechodu, který je právě vytvářen. Jediná odlišnost mezi nimi je ale v tom, že šipka u existujícího přechodu není umístěna na jeho konci, ale je o 18 pixelů posunutá (`refX=28` vs `refX=10`). Proč to tak je, bude vysvětleno v části věnující se komponentě `edge`, viz str. 49.

Element `marker` má definován atribut `id`, pomocí kterého je na něj v ostatních elementech odkazováno, a důležitý je také atribut `orient` s hodnotou `auto`, která zajišťuje, že šipka bude zobrazena ve správné orientaci, ať už bude přechod vykreslen v jakémkoliv směru.

Uvnitř elementu `marker` se nachází element `path`, který definuje samotnou podobu šipky. V tomto případě jde o malý trojúhelník. V této definici jsou uvedeny pouze rozměry, samotná podoba je zajištěna pomocí CSS.

V souvislosti s těmito šipkami se projevil bug v Internet Exploreru, který způsobuje, že při pohybu těchto přechodů nedochází k jejich překreslování [35]. Microsoft bohužel nepovažuje tuto chybu za příliš závažnou, a proto se rozhodl, že ji nebude opravovat [36].

Je tedy nutné překreslování zajistit vlastními silami. Jedna z možností, jak toho dosáhnout, je vykreslovat tyto šipky jako samostatné elementy. Druhá řeší problém hrubou silou, kdy nejprve z DOMu odstraní příslušné uzly definující styly šipek a poté je tam opět vloží. Tím prohlížeč donutí k jejich překreslení. Před tím je ale ověřeno, že použitý prohlížeč je opravdu Internet Explorer, aby k této zbytečné činnosti nedocházelo i v prohlížečích, které touto chybou netrpí.

V současné době je v aplikaci použita druhá metoda, zejména kvůli její implementační jednoduchosti. V dalších verzích ale pravděpodobně bude použita první, která kromě větší elegance také umožní mít nad vykreslením šipek větší kontrolu.

7.3 Vertex

Komponenta `vertex` (neboli vrchol) je zodpovědná za vykreslení stavu automatu. Je tvořena několika SVG elementy, z nichž některé jsou nepovinné a jsou zobrazeny pouze v určitých případech. Definice vrcholu může vypadat například takto:

```
<g class="vertex">
  <path d="M252,110L322,110" style="marker-end: url(#arrow);" class="vertex-start-line"></path>
```

```
  <text x="253" y="105" class="vertex-start-line-text">Start</text>
  <circle cx="322" cy="110" r="18" class="vertex-circle"
fill="lightcoral"></circle>
  <circle cx="322" cy="110" r="18" class="vertex-circle-invisible"></circle>
  <circle cx="322" cy="110" r="14" class="vertex-circle-accept"></circle>
  <text x="322" y="110" dy="4" class="vertex-text">q1</text>
</g>
```

Celá definice je zabalena do elementu `g`, který slouží k definici skupiny souvisejících elementů. Je to zejména z toho důvodu, aby jakákoliv část komponenty reagovala správně na klik myši.

Červeně označená část slouží k vykreslení šipky s nápisem „Start“, která označuje počáteční stav automatu a u ostatních stavů se tak neobjevuje. Element `path` pomocí atributu `d` definuje samotnou šipku. Zápis Mx_1,y_1Lx_2,y_2 znamená: Posuň se (Move) na pozici x_1,y_1 a nakresli čáru (Line) do pozice x_2,y_2. V atributu `style` je pak řečeno, že má být na konci čáry (`marker-end`) vykreslena šipka podle určeného identifikátoru, zde je to `arrow`.

Element `text` zajišťuje vykreslení textu „Start“ na pozici určené pomocí atributu `x` a `y`.

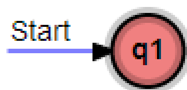
Modře označená část kódu obsahuje dva elementy `circle` a slouží k vykreslení kruhu se středem na souřadnicích určených atributy `cx` a `cy` a o poloměru určeném atributem `r`. Dva kruhy na stejném místě a se stejnými rozměry jsou použity ze dvou důvodů. První je ten, že zatímco první kruh je vykreslen 2 pixely silnou černou čarou, druhý má čáru širokou 8 pixelů, ale je vykreslen s maximální průhledností a tedy prakticky neviditelně. Širší čára ovšem zajišťuje, že komponenta zareaguje na myš o něco dříve, než při najetí až nad samotný viditelný kruh.

Tento efekt je použit i u ostatních částí zobrazeného automatu a zejména u přechodů velmi přispívá k uživatelské přívětivosti. Umožňuje totiž vybrat přechod, aniž by na něj musel uživatel kliknout s přesností jednoho pixelu.

Druhý důvod použití dvou kruhů je ten, že vybraný stav je zobrazen s malým stínem. Tento stín je tvořen právě zmíněnou druhou kružnicí, která je v tomto případě viditelná, byť stále částečně průhledná.

Zelená část kódu zajišťuje vykreslení menší kružnice uvnitř přijímajících stavů automatu, a nakonec černě vyznačená část zajišťuje zobrazení názvu stavu.

Obrázek 20 zobrazuje celý stav určený výše uvedenou definicí.



Obrázek 20 Příklad vykreslení stavu, zdroj: autor

7.4 Edge

Komponenta `edge` (neboli hrana) je zodpovědná za vykreslení přechodu mezi stavy. Opět je tvořena několika elementy a využívá i dvě další menší komponenty, čím se liší od komponenty `vertex`, která je soběstačná. Definice hrany může vypadat například takto:

```
<g class="edge">
  <path d="M268.5,128L437.5,175" class="edge-line" style="marker-end:
url(#arrow);"></path>
  <path d="M268.5,128L437.5,175" class="edge-invisible" style="marker-end:
url(#arrow);"></path>
  <circle cx="353" cy="151.5" r="4" class="edge-mid-point"></circle>
  <g>
    <text x="361" y="143.5" class="edge-text">a</text>
    <path d="M353,151.5L361,141" class="edge-text-line"></path>
  </g>
</g>
```

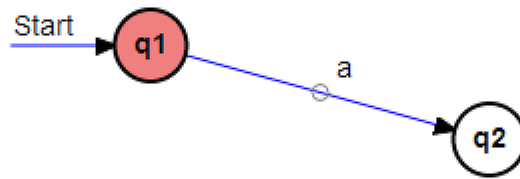
Červeně označená část zajišťuje vykreslení čáry mezi dvěma stavy. V tomto případě jde o úsečku mezi středy jednotlivých stavů. Přestože se na obrazovce může zdát, že úsečka začíná a končí na kružnici ohraničující daný stav, ve skutečnosti tomu tak není. Hrany jsou totiž v grafu vykreslovány před zobrazením vrcholů a přebytečné části hran jsou tak překryty.

Toto je ovšem právě ten důvod, proč musí být šipky na konci hran posunuty o 18 pixelů, tedy o poloměr kružnice ohraničující stav (viz str. 46).

Modře označená část je tvořena komponentou `midPoint`, která vykresluje malý kruh uprostřed hrany. Ten je možné uchopit myší a tím danou hranu zakřivit. Hrana tedy nemusí vždy nutně být vykreslena jako úsečka.

Zeleně označená část je pro změnu tvořena komponentou `edgeText`, která zobrazuje popisek daného přechodu. Kromě toho ještě při najetí myši zobrazuje čáru spojující tento text s kruhem uprostřed přechodu. Tato čára slouží k lepší orientaci v tom, ke kterému přechodu tento popisek patří.

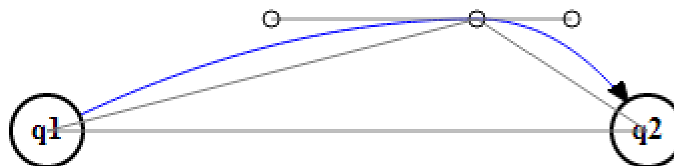
Obrázek 21 zobrazuje výše definovanou hranu včetně počátečního a koncového stavu.



Obrázek 21 Příklad zobrazení přechodu, zdroj: autor

Jak již bylo zmíněno, přechod mezi stavy nemusí být vždy pouze úsečka, ale může být i zakřivený. V tom případě ji není možné definovat ve tvaru Mx_1y_1,bLx_2,y_2, který je určený pro vykreslení úsečky. Místo toho je zobrazen jako dvojice křivek. Jedna vede od počátečního stavu ke „středu“ přechodu a druhá od „středu“ ke koncovému stavu.

Tyto křivky jsou definovány ve tvaru $Mx_1,y_1Qx_2,y_2,x_3,y_3Qx_4,y_4,x_5,y_5$, kde x_1,y_1 jsou souřadnice počátečního stavu, x_5,y_5 souřadnice koncového stavu, x_3,y_3 souřadnice středového kruhu a nakonec x_2,y_2 a x_4,y_4 jsou souřadnice pomocných bodů. Písmena `Q` určují, že k vykreslení budou použity Bézierovy kvadratické křivky, viz Obrázek 22.



Obrázek 22 Příklad zobrazení přechodu pomocí Bézierových křivek, zdroj: autor

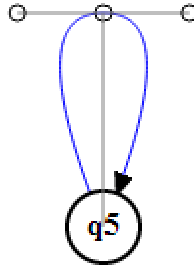
Pozice pomocných bodů je vypočítána tak, aby tvořily úsečku procházející „středem“ přechodu o délce rovnající se polovině vzdálenosti mezi středy obou vrcholů. Poměr vzdáleností těchto bodů a středového kruhu je stejný jako poměr vzdáleností mezi středy vrcholů a středovým kruhem.

Reálná definice pak může vypadat například takto:

```
<path d="M100,400Q212.1300120717398,344,315,344Q362.13001207173977,344,400,400"
class="edge-line" style="marker-end: url(#arrow);"></path>
```

Červeně jsou označené souřadnice stavů, modře středového kruhu a zeleně pomocných bodů.

Podobného principu je použito v případě, že je přechod zobrazený jako smyčka, tedy když se počáteční stav rovná koncovému, viz Obrázek 23.



Obrázek 23 Příklad zobrazení přechodu jako smyčky, zdroj: autor

Opět je zde použita dvojice Bézierových kvadratických křivek, ale výpočet pomocných bodů je poněkud odlišný. Pomocné body opět ohraničují úsečku procházející středovým kruhem, ale v tomto případě je tato úsečka kolmá na přímku určenou středovým kruhem a středem vrcholu. Vzdálenosti pomocných bodů od středového kruhu jsou stejné a rovnají se 0,4násobku vzdálenosti mezi středovým kruhem a středem vrcholu.

8 OSTATNÍ TECHNOLOGIE

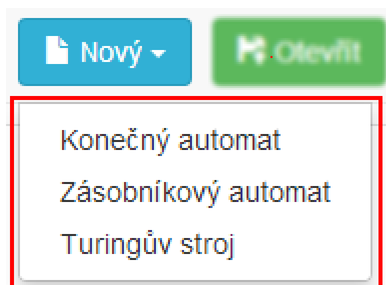
Přestože hlavní část aplikace je vytvořena pomocí TypeScriptu, Bobrilu a SVG, není výčet použitých technologií kompletní. Na následujících stránkách budou popsány ostatní použité technologie či knihovny.

8.1 Bootstrap

„Bootstrap je nejpopulárnější HTML, CSS a JS framework pro vývoj responsivních webových projektů.“ [37]

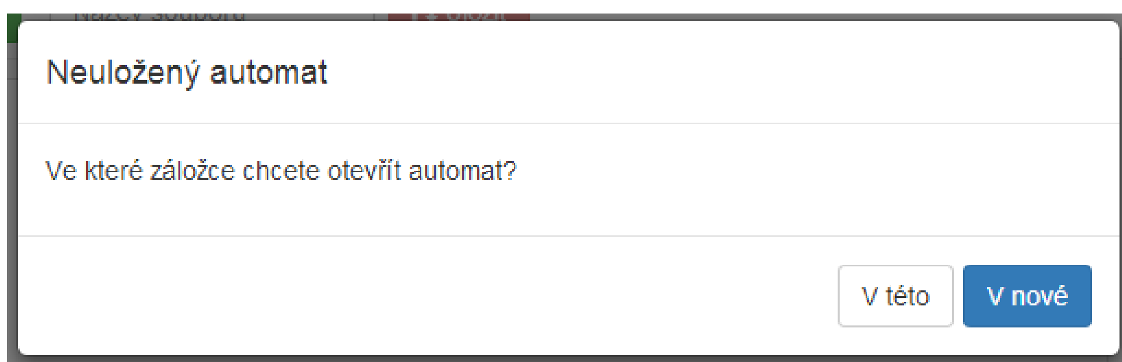
Bootstrap je vyvíjen firmou Twitter a je dostupný pod MIT licencí. V aplikaci je použit pro definování veškerého vzhled vyjma podoby samotného automatu. Kromě samotného CSS souboru definujícího vzhled jsou použity ještě dvě komponenty `dropdown` a `modal`.

Komponenta `dropdown` je použita v nabídce jednotlivých typů automatů při vytváření nového automatu, viz Obrázek 24.



Obrázek 24 Použití komponenty dropdown, zdroj: autor

Komponenta `modal` je použita k vytvoření modálního dialogového okna, viz Obrázek 25.



Obrázek 25 Použití komponenty modal, zdroj: autor

Obě komponenty pro svoji činnost vyžadují přítomnost knihovny jQuery.

8.2 FileSaver.js

FileSave.js [38] je knihovna umožňující webovým aplikacím, aby mohly ukládat soubory bez toho, aby bylo toto uložení vynuceno webovým serverem. Tato funkce je dostupná ve všech moderních prohlížečích, v případě Internet Exploreru je podporována od verze 10, viz Obrázek 26.

Browser	Constructs as	Filenames	Max Blob Size	Dependencies
Firefox 20+	Blob	Yes	800 MiB	None
Firefox < 20	data: URI	No	n/a	Blob.js
Chrome	Blob	Yes	500 MiB	None
Chrome for Android	Blob	Yes	500 MiB	None
IE 10+	Blob	Yes	600 MiB	None
Opera 15+	Blob	Yes	500 MiB	None
Opera < 15	data: URI	No	n/a	Blob.js
Safari 6.1+*	Blob	No	?	None
Safari < 6	data: URI	No	n/a	Blob.js

Obrázek 26 Podpora FileSaver.js v prohlížečích [38]

V aplikaci je tato knihovna použita pro ukládání vytvořených automatů. Použití knihovny je velmi jednoduché, jde pouze o zavolání metody `saveAs()`:

```
var json = Services.serialization.serialize(ctx.data.automaton);
var blob = new Blob([json], { type: 'text/plain;charset=utf-8' });
saveAs(blob, ctx.fileName + '.aut');
```

Metoda `saveAs` přijímá dva parametry, z nichž první je `Blob` [39], což je objekt reprezentující obsah ukládaného souboru a druhým parametrem je název samotného souboru.

Soubor je ukládán v textovém formátu JSON.

8.3 JSON

„JSON neboli JavaScript Object Notation je úsporný formát pro výměnu dat, který je pro člověka i stroj snadné číst i vytvářet. Jde o textový formát, který je jazykově zcela nezávislý, ale který využívá konvence blízké programátorů, kteří znají jazyky odvozené od syntaxe jazyka C.“ [40]

Obrázek 27 Logo JSONu [40]

JSON má dvě základní datové struktury – objekt a pole. Objekt je ohraničen složenými závorkami `{}`, uvnitř kterých se nachází čárkou oddělené dvojice ve tvaru `"název": hodnota`. Pole je ohraničené hranatými závorkami `[]`, v nich se nacházejí čárkou oddělené hodnoty. Hodnotou může být číslo, textový řetězec, booleovská hodnota, nebo i opět objekt či pole.

Zjednodušený zápis automatu ve formátu JSON může vypadat například takto:

```
{
  "transitions": [
    {
      "from": 0,
      "to": 1
    }
  ],
  "states": [
    {
      "index": 0,
      "name": "q0",
      "point": {
        "x": 110,
        "y": 196
      }
    },
    {
      "index": 1,
      "name": "q1",
      "point": {
        "x": 322,
        "y": 193
      }
    }
  ]
}
```

Celý objekt má dvě vlastnosti `transitions` a `states`, které obsahují pole jednotlivých přechodů resp. stavů. Objekt přechodu má vlastnosti `from` a `to` odkazující na počáteční resp. koncové stavy. Objekt stavu má vlastnosti `index`, `name` a `point`, což je opět objekt s vlastnostmi `x` a `y`, které určují souřadnice daného stavu.

8.4 FileReader

FileReader [41] umožňuje webovým aplikacím číst obsah souboru bez toho, aby byl tento soubor nejprve zpracován webovým serverem. V aplikaci je toto rozhraní použito k načtení dříve vytvořených a uložených souborů.

FileReader je dostupný ve všech moderních prohlížečích, v případě Internet Exploreru až od verze 10 [42].

V aplikaci je FileReader použit v komponentě `openFileInput`:

```
var reader = new FileReader();
reader.onload = (loadEvent: any) => {
    ctx.data.onFileOpen(loadEvent.target.result, fileName);
};
var target = <HTMLInputElement>event.target;
var file = target.files[0];
fileName = file.name;
reader.readAsText(file);
```

Nejprve je vytvořený samotný `reader`, kterému je potom nastavena vlastnost `onload`, což je funkce, která se má provést po načtení souboru. Potom je z elementu `input` získán odkaz na vybraný soubor, který je následně načten metodou `readAsText`.

8.5 Web Storage

Web Storage [43] někdy nazývané také jako „Local Storage" či „DOM Storage" je rozhraní, které umožňuje webovým aplikacím, aby si na klientovi mohly uložit svá data. Tím se podobá známým cookies, na rozdíl od nich ale nejsou tato data odesílána při každém dotazu na webový server.

Další významný rozdíl je v maximální velikosti uložených dat. Zatímco cookies jsou omezeny řádově na jednotky KB [44], Web Storage je v závislosti na použitém prohlížeči omezeno nejčastěji na 5-10 MB [45].

Toto rozhraní je opět použitelné ve všech moderních prohlížečích. V případě Internet Exploreru už dokonce od verze 8 [46]. Implementace v Internet Exploreru ale není použitelná, pokud není aplikace načtena z webového serveru [47].

Web Storage je rozdělené na dvě části, první z nich je `localStorage` a druhou `sessionStorage`. Rozdíl mezi nimi je takový, že data uložená v `sessionStorage` jsou platná pouze pro jednu relaci a jsou tak automaticky mazána po zavření okna prohlížeče.

V aplikaci je rozhraní Web Storage použito k předávání dat mezi stávající a nově otevřenou záložkou prohlížeče. Aplikace totiž běží v kontextu jedné záložky prohlížeče a nemůže žádným způsobem přímo komunikovat s jinou záložkou. V případě běžných webových aplikací je předání dat řešeno pomocí webového serveru. Tento způsob ale není možné v aplikaci použít.

Web Storage definuje metody `setItem`, `getItem` a `removeItem`. Metoda `setItem` slouží k uložení hodnoty a přijímá dva parametry – název a samotnou hodnotu. Metoda `getItem` čte uložené hodnoty podle parametru – jména hodnoty. A nakonec metoda `removeItem` odstraňuje uloženou hodnotu podle názvu daného parametrem.

9 IMPLEMENTACE APLIKACE

Výsledná aplikace se nachází na přiloženém CD, které obsahuje čtyři adresáře:

- adresář `src` obsahuje zdrojové kódy aplikace
- v adresáři `build_tools` se nacházejí nástroje používané ke kompilaci a sestavení aplikace
- adresáře `build` a `publish` obsahují sestavenou aplikaci. Rozdíl mezi nimi je zejména v tom, že v adresáři `build` je aplikace rozdělena do mnoha JavaScriptových souborů, které odpovídají původním souborům v TypeScriptu. Toto rozdělení je vhodnější pro vývoj a testování aplikace. Pro samotný provoz je ale naopak výhodnější mít jeden výsledný soubor, což odpovídá stavu adresáře `publish`.

9.1 Struktura zdrojových kódů

Přímo v adresáři `src` se nacházejí zejména dva soubory:

- `index.html` je jediný HTML soubor aplikace, obsahuje pouze odkazy na ostatní soubory aplikace, tedy JavaScriptové kódy a definice CSS stylů. Samotný obsah tohoto souboru je generován až za běhu aplikace, viz kapitola 4.1.
- soubor `app.ts` obsahuje vstupní bod aplikace a definici routování, viz kapitola 6.5.5

Zbytek aplikace je rozčleněn do následujících adresářů:

- adresář `components` obsahuje jednotlivé komponenty aplikace, viz kapitola 6.4
- v adresáři `definitions` jsou uloženy definiční soubory popsané v kapitole 5.7
- externí knihovny, na kterých je aplikace závislá, se nacházejí v adresáři `libs`. Popis těchto knihoven se nachází v kapitolách 6 a 8.
- zdrojové kódy zodpovědné za aplikační logiku jsou uloženy v adresáři `models` a jeho podadresářích. Tyto soubory jsou výsledkem implementace analýzy popsané v kapitole 4.
- v adresáři `routes` jsou definovány jednotlivé virtuální stránky aplikace, který odpovídají příslušným typům automatů. V zásadě jde opět o komponenty, ale jejich význam v aplikaci se výrazně liší od ostatních, proto jsou uloženy v samostatném adresáři.
- adresář `services` obsahuje soubor `serializationService.ts`, který je zodpovědný za serializaci automatů, tedy o jejich převod do formátu JSON (viz kapitola 8.3) a zpět.

Druhým souborem v tomto adresáři je `exporter.ts`, který se zajišťuje export automatu do obrázku.

- posledními adresáři jsou `style`, který obsahuje definice CSS stylů, a `fonts` obsahující fonty použité ve frameworku Bootstrap, viz kapitola 8.1

9.2 Nástroje pro kompilaci a sestavení aplikace

Jak již bylo zmíněno, v adresáři `build_tools` se nacházejí nástroje používané pro kompilaci a sestavení aplikace. Jde zejména o Gulp [48], který slouží k automatizaci opakujících se činností. Jeho nastavení se nachází v souboru `gulpfile.js`, kde jsou definovány tyto úkoly:

- spuštění příkazu `gulp` v příkazové řádce iniciuje sestavení aplikace do adresáře `build`. Po dokončení sestavení běh úkolu pokračuje a Gulp sleduje změny TypeScriptových souborů v adresáři `src`. Pokud se nějaký soubor změní, je ihned zkompilován do JavaScriptu. Není tak nutné kompilaci spouštět ručně a vývoj aplikace je tak snazší.
- po spuštění příkazu `gulp publish` dojde k sestavení aplikace v adresáři `publish`, po kterém běh úkolu končí.

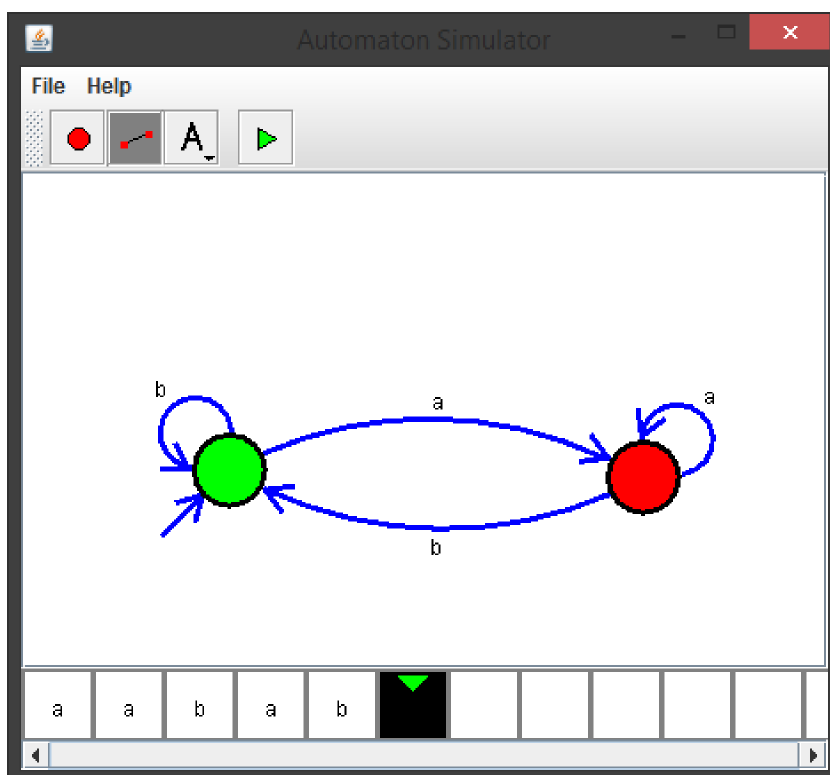
Samotný Gulp ke své činnosti používá platformu Node.js [49], která umožňuje spouštět aplikace napsané v JavaScriptu. Pro jejich běh používá stejné výkonné jádro, jaké je použito v prohlížeči Google Chrome.

10 SROVNÁNÍ S PODOBNÝMI PRODUKTY

Na následujících stránkách budou porovnány některé podobné produkty. Ke každému produktu bude uveden snímek obrazovky znázorňující stejný konečný automat.

10.1 Automaton Simulator

Autorem programu Automaton Simulator [50] je Carl Burch, který svůj produkt uvolnil pod GPLv2 licencí. Program je napsaný v Javě 1.3 a proto pro svoji činnost potřebuje běhové prostředí Java Virtual Machine.



Obrázek 28 Okno programu Automaton Simulator, zdroj: autor

Program umožňuje tvorbu deterministických a nedeterministických konečných automatů, deterministických zásobníkových automatů a Turingových strojů. Stavy těchto automatů není možné nijak pojmenovat či barevně odlišit.

Přechodům je možné nastavit vstupní symboly, ale standardně je možné vybrat pouze symboly a, b, c, d a „else“. Toto omezení je možné obejít přímou editací uložených souborů, ale jde o nedokumentovanou funkci, takže je možné, že nebude vždy použitelná. I tak by se ale jednalo o poměrně nepohodlný způsob editace.

Do kreslicí plochy automatu je možné vložit libovolný text, čímž se dá částečně obejít nemožnost pojmenovávat stavy, ale tento popisek není nijak svázán s polohou daného stavu, takže se s ním nepohybuje.

Vytvořené automaty je možné uložit a opět načíst ze souboru. Soubory jsou ukládány v textovém formátu, je tedy možná jejich přímá editace.

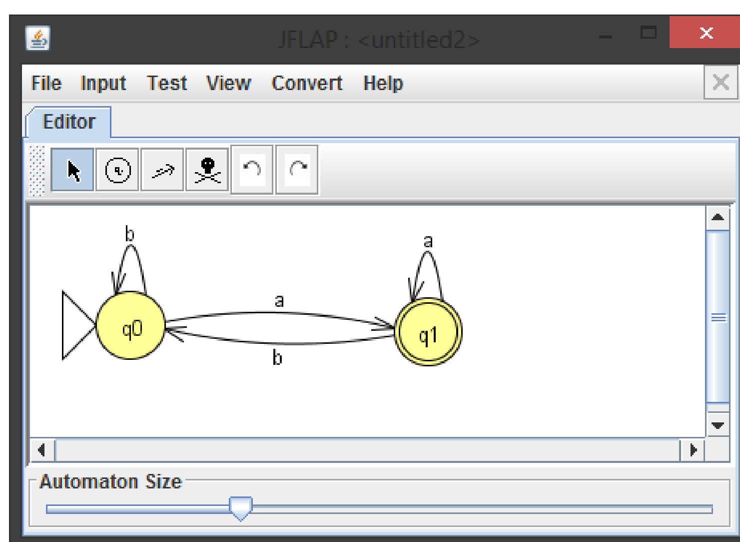
U všech typů automatů je možné spustit simulaci, avšak způsob simulace se liší podle typu automatu. U Turingových strojů je možné předem definovat vstupní symboly a pak spustit kompletní simulaci, nebo ji spustit po krocích vpřed i vzad. Ostatní typy automatů okamžitě reagují na vstupní symboly zadávané z klávesnice. Není tedy možné všechny zadat předem. Není také možné simulaci krokovat.

Kromě výše zmíněných vlastností nemá program žádné další funkce. Není tedy možné převádět automaty na gramatiky ani nedeterministické konečné automaty na deterministické, či exportovat vytvořené automaty do obrázku.

10.2 JFLAP

JFLAP (Java Formal Languages and Automata Package) [51] je program vytvořený pro podporu výuky na Duke University. V současné době je dostupný ve verzi 7. Licence umožňuje téměř libovolné využití s výjimkou placené distribuce.

Je naprogramovaný v Javě 1.6 a tedy vyžaduje Java Virtual Machine. Program je možné získat včetně zdrojových kódů.



Obrázek 29 Okno programu JFLAP, zdroj: autor

Aplikace umožňuje tvorbu deterministických i nedeterministických konečných automatů, zásobníkových automatů i Turingových strojů. Kromě toho ale i Mealyho [52] či Moorovy [53] stroje.

Vytvořené stavy jsou automaticky pojmenovávány. Toto jméno je možné změnit. Kromě toho je možné přidat ke stavům štítek s bližším popiskem. Není ale možné je obarvit. Myší je možné označit celou skupinu stavů a s celou takovou skupinou pohybovat po kreslicí ploše.

Přechodům je možné přiřadit libovolné vstupní symboly a je možné je libovolně zakřivovat či vytvářet smyčky. Při editaci je možné používat funkce zpět/znovu.

Automaty je možné ukládat i načítat ze souboru. Soubory jsou ukládány v XML formátu, jsou tedy opět přímo editovatelné. Kromě toho je možné je exportovat do běžných rastrových formátů (GIF, JPEG, PNG, BMP) a také do vektorového SVG.

Běh všech typů automatů je možné simulovat několika způsoby. Je možné je krokovat, nebo i spustit celou simulaci najednou. Kromě toho je možné dávkově simulovat více různých vstupů, a tak automat důkladně otestovat.

V případě nedeterministických konečných automatů je možné zvýraznit nedeterministické stavy či celý automat převést na deterministický. Je možné je také převést na gramatiku nebo regulární výraz. Na gramatiku je možné převést i zásobníkové automaty a Turingovy stroje.

Program umožňuje i převod opačným směrem, tedy z gramatiky či regulárního výrazu na příslušný typ automatu. Tyto automaty jsou vytvořeny v grafické podobě a pro rozmístění jejich stavů je možné použít několik různých algoritmů (např. rozmístění do kruhu, spirály či do stromu).

10.3 Automatonsimulator.com

Dalším podobným produktem je webová aplikace s názvem rovněž Automaton Simulator [54], jejímž autorem je Kyle Dickerson. Aplikace je napsaná v JavaScriptu s použitím knihoven jQuery UI a jsPlumb.

V aplikaci je možné vytvořit deterministické i nedeterministické konečné automaty a také zásobníkové automaty. Není možné vytvářet Turingovy stroje ani jiné typy automatů.

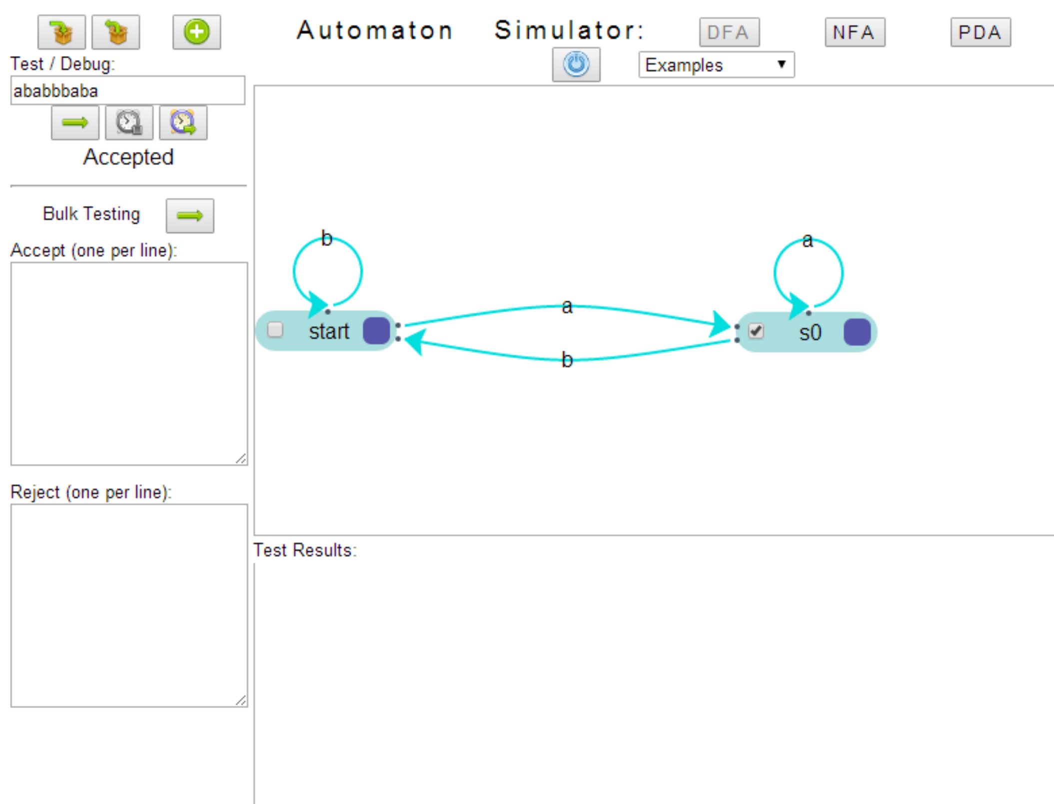
Názvy stavu jsou automaticky generovány a nelze je měnit. Přechodům je možné přiřadit libovolné vstupní symboly, ale vždy pouze jeden. Pokud je nutné, aby bylo možné přecházet mezi dvěma stavy pomocí různých symbolů, je nutné vytvořit více přechodů.

Je možné vytvářet smyčky, ale měnit tvar přechodů možné není. Jejich tvar si řídí sama aplikace.

Automaty není možné přímo ukládat do souborů, je možné je pouze ukládat do Local Storage prohlížeče (viz kapitola 8.5), či vyexportovat v textové podobě ve formátu JSON (viz kapitola 8.3). Stejným způsobem je možné automaty znovu načíst.

Stejně jako v předchozím případě lze běh automatů simulovat najednou, po krocích i dávkově. U dávkové simulace se zvlášť nastavují vstupy, u kterých se předpokládá přijetí, a ty u kterých se předpokládá nepřijetí.

Nedeterministické konečné automaty není možné převádět na deterministické. A žádné typy automatů nelze převádět na gramatiky.



Obrázek 30 Okno programu Automaton Simulator, zdroj: autor

10.4 jFAST

Java Finite Automaton Simulation Tool [55] je program vytvořený Timothy Whitem jako závěrečná práce k jeho bakalářskému studiu na Villanova University. Je dostupný pod GPL licencí včetně zdrojových kódů. Jak už název naznačuje, jde o program napsaný v Javě.

V aplikaci je možné vytvářet deterministické i nedeterministické konečné automaty, zásobníkové automaty i Turingovy stroje.

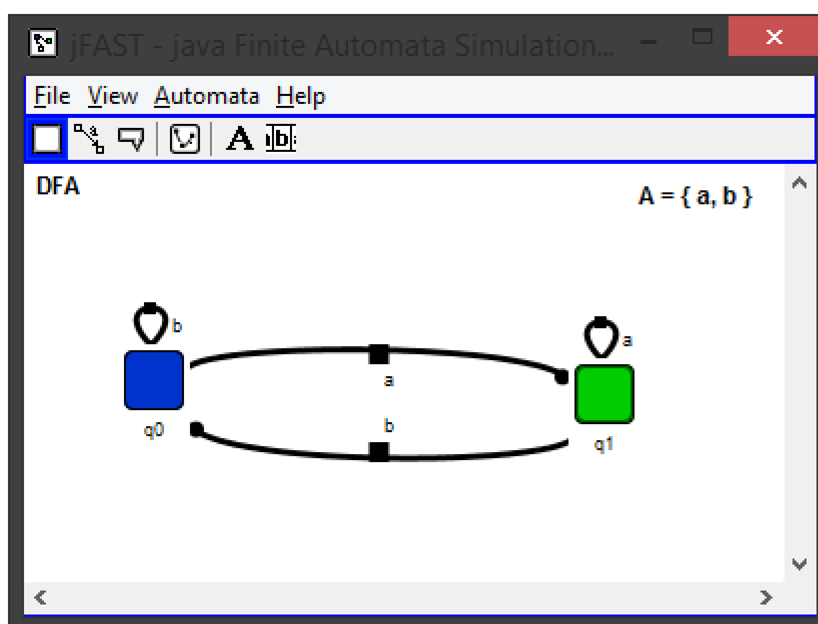
Názvy stavů jsou generovány automaticky a je možné je měnit. Stavy jsou neobvykle znázorněny jako čtverce a není možné je barevně odlišit. Barevně jsou odlišeny pouze počáteční a přijímající stavy.

Přechodům je možné nastavit libovolné vstupní symboly, ale tvary přechodů není možné ovlivnit. Jsou opět řízeny aplikací.

Automaty je možné ukládat do souborů ve formátu XML a také exportovat jako obrázky v JPEG formátu.

Běh automatu je možné simulovat dvěma způsoby. První je krokování podle předem zadaného vstupu a při druhém způsobu (nazývaným jako interaktivní) reaguje automat na právě zadávané vstupní symboly.

Vytvořené automaty není možné nijak převádět na jiné typy či na gramatiky.



Obrázek 31 Okno programu jFAST, zdroj: autor

10.5 FSM Simulator

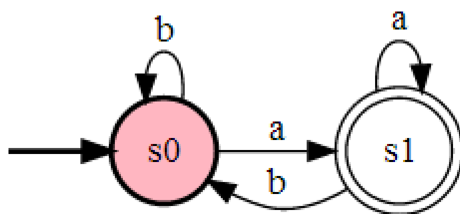
Posledním porovnávaným programem je FSM Simulator [56] autorů Ivana Zuzaka a Vedrany Jankovic. Jde o webovou aplikaci využívající knihovny noam, zajišťující logiku aplikace, a Viz.js, pomocí které vykresluje graf s využitím SVG.

Aplikace je určená pouze pro vytváření konečných automatů, které jsou zadávány v textové podobě. Tato textová podoba je následně převedena do grafické, kterou ale uživatel nemůže přímo měnit. Jediná možnost je upravit textovou podobu a znovu vygenerovat graf.

Automat je také možné vygenerovat z regulárního výrazu. Převod opačným směrem ale možný není.

Automaty není možné přímo ukládat ani načítat ze souboru, ale vzhledem k textovému zadávání je možné jejich definice zkopírovat a uložit pomocí jiného programu.

Běh automatu je opět možné simulovat pomocí krokování. Aplikace umí automaticky vytvářet vstupy, které automat přijímá či nepřijímá.



Obrázek 32 Příklad automatu vytvořeného programem FSM simulator, zdroj: autor

10.6 Shrnutí funkcí porovnávaných programů

Žádná z výše uvedených aplikací nesplňuje všechny požadavky stanovené v kapitole 3, přestože každá z nich má i funkce, které původně nebyly požadovány, a které mohou sloužit jako inspirace pro další vývoj. Zdaleka nejvíce funkcí poskytuje program JFLAP, který splňuje téměř všechny požadavky s výjimkou závislosti na Javě a nemožnosti obarvit prvky automatu. Žádný z uvedených programů také není distribuován v českém jazyce, což sice nebylo uvedeno v požadavcích, ale i tak může jít o vlastnost, která by mohla studentům pomoci v pochopení probíraného učiva. Splnění či nesplnění základních požadavků je shrnuto v následující tabulce:

	Automaton Simulator	JFLAP	Automatonsimulator.com	jFAST	FSM Simulator
Podpora všech typů automatů	X	X		X	
Převod KA na deterministický		X			
Převod na gramatiky		X			
Návrh automatu v grafické podobě	X	X	X	X	
Obarvení prvků automatu					
Automaticky generované názvy stavů		X	X	X	
Uložení a načtení automatu ze souboru	X	X		X	
Export automatu do obrázku		X		X	
Simulace běhu automatu	X	X	X	X	X
Nezávislost na externích knihovnách			X		X
Nezávislost na připojení k internetu	X	X		X	

Tabulka 3 Srovnání funkcí jednotlivých programů

11 SHRNUTÍ VÝSLEDKŮ

V úvodu této práce byla zmíněna motivace k vytvoření tohoto projektu, tedy zjednodušení činnosti vyučujících teoretické informatiky a i možná změna postoje studentů k tomuto předmětu.

V následující části této práce byly shrnuty základní pojmy, které je nutné znát pro pochopení smyslu práce. Byly definovány pojmy abeceda, řetězec a jazyk. Dále byl definován pojem gramatika a byla představena Chomského klasifikace gramatik. Následovaly definice různých druhů automatů, možné způsoby jejich reprezentace a také jaké jazyky dané automaty přijímají.

Ve třetí části byla provedena analýza projektu. Nejprve byly shrnuty funkční i nefunkční požadavky a byla provedena volba cílové platformy. Pak byly uvedeny příklady užití projektu, jak v podobě scénářů, tak i jako shrnující diagram. Následoval návrh programových rozhraní a z nich vycházející návrh tříd. Jako poslední část analýzy projektu byl zmíněn návrh uživatelského rozhraní.

V několika dalších částech byly představeny použité technologie. První z nich byl programovací jazyk TypeScript, pomocí kterého byl vyvinut nejen tento projekt, ale také framework Bobril, který je v projektu použit a jehož popis následuje. V tomto popisu je nejprve vysvětlen pojem Virtuální DOM, dále jsou popsány komponenty a jejich životní cyklus v aplikaci. Poslední část věnovaná Bobrilu, se zabývá jeho rozšířeními.

Další použitou technologií je SVG, které v projektu slouží k samotnému vykreslování vytvořených automatů. Jsou zde také popsány bobrilovské komponenty, které vytvářejí jednotlivé části SVG obrázku. V poslední části popisu použitých technologií jsou zmíněny Bootstrap, FileSaver.js, JSON, FileReader a WebStorage.

Závěrečná část této práce se věnuje popisu podobně zaměřených aplikací. Pro každou aplikaci je uvedeno, jak splňuje požadavky stanovené na začátku této práce. Jsou zde také zmíněny vlastnosti, které nejsou součástí požadavků na projekt, ale které by se mohly ukázat jako užitečné pro jeho další vývoj. U každé aplikace je pro srovnání vzhledu uživatelského rozhraní umístěn obrázek se stejným automatem. Celé srovnání je shrnuto v tabulce.

12 ZÁVĚRY A DOPORUČENÍ

Životní cyklus softwaru by samozřejmě neměl končit v okamžiku jeho vydání a ani tato aplikace by neměla být výjimkou. Při jejím vývoji a při psaní této práce se objevilo několik poznatků, které by v budoucnu mohly aplikaci udělat ještě lepší.

Dva z nich už byly v práci zmíněny:

- lepší vykreslování šipek na konci přechodu (viz str. 46)
- použití rozšíření Style pro Bobril (viz str. 44)

Další pak vyplynuly z porovnání s ostatními produkty:

- podpora dávkové simulace běhu automatů
- podpora interaktivní simulace běhu automatů, tedy okamžitá reakce na zadávaný symbol
- převod gramatiky do automatu v grafické podobě. Zde bude potřeba prozkoumat postupy či algoritmy, jak nově vytvořený automat vykreslit tak, aby dobře vypadal, přechody se příliš nekřížily atd.
- převod nedeterministického konečného automatu na deterministický v grafické podobě
- podpora operací zpět/znovu

Celá aplikace je také v současné době testovaná pouze manuálně, bylo by tedy vhodné testy co nejvíce automatizovat – tedy napsat jednotkové testy pro testování jádra aplikace a případně i testy grafického rozhraní.

Aplikace bude od příštího školního roku používána při výuce a i z toho důvodu budou zveřejněny její zdrojové kódy a zprovozněno diskusní fórum, kam budou moci její uživatelé přispívat své zkušenosti či požadavky, nebo se i díky dostupným zdrojovým kódům podílet na jejím dalším vývoji.

Cílem aplikace sice nebylo vytvořit prostředí použitelné i na čistě dotykových zařízeních, tj. zejména tabletech, ale v budoucnu by to mohlo být vhodné. Už teď je možné většinu činností na tabletech provádět, ale plnou podporou to rozhodně není možné nazývat.

Jako zajímavá se také jeví možnost „obalit" aplikaci pomocí NW.js (dříve zvané node-webkit) [57] a tím vytvořit nativní aplikaci pro všechny tři hlavní platformy (Windows, Linux, Mac). Aplikace by tak měla společné jádro použitelné jak na webu, tak na desktopu bez přímé potřeby webového prohlížeče.

13 SEZNAM POUŽITÉ LITERATURY

1. ŠITINA, J. *Grafové algoritmy a jejich vizualizace*. Diplomová práce. Univerzita Hradec Králové, 2010.

2. ČEŠKA, M. T. VOJNAR a A. SMRČKA. FIT VUT. In: *Teoretická informatika (TIN) - informace pro studenty* [online]. [cit. 2015-07-25]. Dostupné z: http://www.fit.vutbr.cz/study/courses/TIN/public/Texty/oporaTIN.pdf

3. JANČAR, P. In: *Teoretická informatika* [online]. 15. 8. 2015. Dostupné také z: http://www.cs.vsb.cz/jancar/TEORET-INF/ti-text.2010-01-20.pdf

4. AngularJS. *Superheroic JavaScript MVW Framework* [online]. [cit. 2015-06-15]. Dostupné z: https://angularjs.org/

5. A JavaScript library for building user interfaces. *React* [online]. [cit. 2015-06-15]. Dostupné z: http://facebook.github.io/react/

6. Github. *Virtual DOM Benchmark* [online]. [cit. 2015-06-17]. Dostupné z: http://vdom-benchmark.github.io/vdom-benchmark/

7. TypeScript [online]. [cit. 2015-06-18]. Dostupné z: http://www.typescriptlang.org/

8. *TypeScript Handbook* [online]. [cit. 2015-06-18]. Dostupné z: http://www.typescriptlang.org/Handbook

9. MICROSOFT. TypeScript Language Specification. [online]. [cit. 2015-06-18]. Dostupné z: http://www.typescriptlang.org/Content/TypeScript%20Language%20Specification.pdf

10. ORACLE. The Java Tutorials. In: *Interface* [online]. [cit. 2015-06-18]. Dostupné z: https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html

11. MOZILLA. Mozilla Developer Network. In: *Introduction to Object-Oriented JavaScript* [online]. [cit. 2015-06-18]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript

12. MOZILLA. Mozilla Developer Network. In: *this* [online]. [cit. 2015-06-18]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this

13. Header files. *LearnCpp* [online]. [cit. 2015-06-18]. Dostupné z: http://www.learncpp.com/cpp-tutorial/19-header-files/

14. DefinitelyTyped [online]. [cit. 2015-06-18]. Dostupné z: http://definitelytyped.org/

15. Github. *Bobril* [online]. [cit. 2015-06-18]. Dostupné z: https://github.com/bobris/bobril

16. Mithril. *A Javascript Framework for Building Brilliant Applications* [online]. [cit. 2015-06-18]. Dostupné z: https://lhorie.github.io/mithril/

17. W3C. *Document Object Model (DOM)* [online]. [cit. 2015-06-18]. Dostupné z: http://www.w3.org/DOM/

18. W3C. *What is the Document Object Model?* [online]. [cit. 2015-06-18]. Dostupné z: http://www.w3.org/TR/DOM-Level-2-Core/introduction.html

19. Mozilla Developer Netword. *Element.innerHTML* [online]. [cit. 2015-06-18]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML

20. The Open Web Application Security Project (OWASP). *Cross-site Scripting (XSS)* [online]. [cit. 2015-06-18]. Dostupné z: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

21. Bobril. *Bobril.md* [online]. [cit. 2015-06-18]. Dostupné z: https://github.com/Bobris/Bobril/blob/master/src/bobril.md

22. recurial.com. *Understanding callback functions in Javascript* [online]. [cit. 2015-06-18]. Dostupné z: http://recurial.com/programming/understanding-callback-functions-in-javascript/

23. Bubbling and capturing. *JavaScript Tutorial* [online]. [cit. 2015-06-18]. Dostupné z: http://javascript.info/tutorial/bubbling-and-capturing

24. stackoverflow. *Capture key press (or keydown) event on DIV element* [online]. [cit. 2015-06-18]. Dostupné z: http://stackoverflow.com/questions/3149362/capture-key-press-or-keydown-event-on-div-element

25. w3schools. *Location hash Property* [online]. [cit. 2015-06-18]. Dostupné z: http://www.w3schools.com/jsref/prop_loc_hash.asp

26. Bobril. *Drag and drop sample* [online]. [cit. 2015-06-18]. Dostupné z: http://bobris.github.io/Bobril/dnd/index.html

27. W3C. *Localization vs. Internationalization* [online]. [cit. 2015-06-18]. Dostupné z: http://www.w3.org/International/questions/qa-i18n

28. Bobril. *Media detection sample* [online]. [cit. 2015-06-18]. Dostupné z: http://bobris.github.io/Bobril/media/index.html

29. Mozilla Developer Netword. *Promise* [online]. [cit. 2015-06-18]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

30. Bobril. *Sticky Header Bobril sample* [online]. [cit. 2015-06-18]. Dostupné z: http://bobris.github.io/Bobril/stickyheader/index.html

31. Bobril. *Vector Graphic Bobril sample* [online]. [cit. 2015-06-18]. Dostupné z: http://bobris.github.io/Bobril/vg/index.html

32. W3C. *SCALABLE VECTOR GRAPHICS (SVG)* [online]. [cit. 2015-06-21]. Dostupné z: http://www.w3.org/Graphics/SVG/

33. Can I use. *SVG (basic support)* [online]. Dostupné také z: http://caniuse.com/#feat=svg

34. W3C. *Document Structure* [online]. [cit. 2015-06-21]. Dostupné z: http://www.w3.org/TR/SVG/struct.html

35. Microsoft Connect. *Dynamically updated SVG path with a marker-end does not update* [online]. [cit. 2015-06-21]. Dostupné z: https://connect.microsoft.com/IE/feedback/details/801938/dynamically-updated-svg-path-with-a-marker-end-does-not-update

36. Microsoft Connect. *SVG marker is not updated when the SVG element is moved using the DOM* [online]. [cit. 2015-06-21]. Dostupné z: https://connect.microsoft.com/IE/feedback/details/781964/

37. Bootstrap. *The world's most popular mobile-first and responsive front-end framework.* [online]. [cit. 2015-06-25]. Dostupné z: http://getbootstrap.com/

38. FileSaver.js. *FileSaver.js* [online]. [cit. 2015-06-26]. Dostupné z: https://github.com/eligrey/FileSaver.js/

39. Mozilla Developer Network. *Blob* [online]. [cit. 2015-06-27]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/Blob

40. JSON. *Introducing JSON* [online]. [cit. 2015-06-28]. Dostupné z: http://json.org/

41. W3C. *The FileReader API* [online]. [cit. 2015-06-30]. Dostupné z: http://www.w3.org/TR/FileAPI/#dfn-filereader

42. Can I use. *FileReader API* [online]. [cit. 2015-07-02]. Dostupné z: http://caniuse.com/#search=filereader

43. Dive Into HTML5. *Local Storage* [online]. [cit. 2015-07-05]. Dostupné z: http://diveintohtml5.info/storage.html

44. Browser Cookie Limits [online]. [cit. 2015-07-05]. Dostupné z: http://browsercookielimits.squawky.net/

45. HTML5 Rocks. *Working with quota on mobile browsers* [online]. [cit. 2015-07-05]. Dostupné z: http://www.html5rocks.com/en/tutorials/offline/quota-research/

46. Can I use. *Web Storage - name/value pairs* [online]. [cit. 2015-07-05]. Dostupné z: http://caniuse.com/#search=webstorage

47. Using HTML5 Web Storage For Interprocess Communication. *Wintellect* [online]. [cit. 2015-07-05]. Dostupné z: http://www.wintellect.com/devcenter/jprosise/using-html5-web-storage-for-interprocess-communication

48. gulp.js. *the streaming build system* [online]. [cit. 2015-08-17]. Dostupné z: http://gulpjs.com/

49. Node.js [online]. [cit. 2015-08-17]. Dostupné z: https://nodejs.org/

50. *Automaton Simulator* [online]. [cit. 2015-07-07]. Dostupné z: http://www.cburch.com/proj/autosim/index.html

51. *JFLAP* [online]. [cit. 2015-07-09]. Dostupné z: http://www.jflap.org/

52. Mealyho automat. *Wikipedia* [online]. [cit. 2015-07-09]. Dostupné z: https://cs.wikipedia.org/wiki/Mealyho_automat

53. Wikipedia. *Mooreův stroj* [online]. [cit. 2015-07-09]. Dostupné z: https://cs.wikipedia.org/wiki/Moore%C5%AFv_stroj

54. *Automaton Simulator* [online]. [cit. 2015-07-11]. Dostupné z: http://automatonsimulator.com/

55. Java Finite Automata Simulator. *jFAST* [online]. [cit. 2015-07-11]. Dostupné z: http://jfast-fsm-sim.sourceforge.net/

56. *FSM simulator* [online]. [cit. 2015-07-13]. Dostupné z: http://ivanzuzak.info/noam/webapps/fsm_simulator/

57. NW.js [online]. [cit. 2015-07-29]. Dostupné z: http://nwjs.io/

14 SEZNAM OBRÁZKŮ

Obrázek 1 Grafická reprezentace automatu, zdroj: autor ... 5
Obrázek 2 Nedeterministický konečný automat, zdroj: autor ... 7
Obrázek 3 Přechod v zásobníkovém automatu, zdroj: autor ... 8
Obrázek 4 Přechod v Turingově stroji, zdroj: autor ... 10
Obrázek 5 Diagram typových úloh, zdroj: autor ... 16
Obrázek 6 Návrh rozhraní, zdroj: autor ... 17
Obrázek 7 Návrh tříd, zdroj: autor ... 19
Obrázek 8 Návrh uživatelského rozhraní, zdroj: autor ... 20
Obrázek 9 Logo TypeScriptu [7] ... 21
Obrázek 10 Logo Bobrilu [15] ... 26
Obrázek 11 Příklad DOM stromu [18] ... 26
Obrázek 12 Počet vlastností DIV elementu, zdroj: autor ... 27
Obrázek 13 Přidání uzlu za už existující uzly, zdroj: autor ... 27
Obrázek 14 Přidání nového uzlu před existující uzly, zdroj: autor ... 28
Obrázek 15 Příklady převodu virtuálních uzlů do skutečných [21] ... 30
Obrázek 16 Životní cyklus komponenty [21] ... 34
Obrázek 17 Skládání komponent do větších celků, zdroj: autor ... 35
Obrázek 18 Logo SVG [32] ... 45
Obrázek 19 Příklad použití SVG v aplikaci, zdroj: autor ... 46
Obrázek 20 Příklad vykreslení stavu, zdroj: autor ... 49
Obrázek 21 Příklad zobrazení přechodu, zdroj: autor ... 50
Obrázek 22 Příklad zobrazení přechodu pomocí Bézierových křivek, zdroj: autor ... 50
Obrázek 23 Příklad zobrazení přechodu jako smyčky, zdroj: autor ... 51
Obrázek 24 Použití komponenty dropdown, zdroj: autor ... 52
Obrázek 25 Použití komponenty modal, zdroj: autor ... 52
Obrázek 26 Podpora FileSaver.js v prohlížečích [38] ... 53
Obrázek 27 Logo JSONu [40] ... 54
Obrázek 28 Okno programu Automaton Simulator, zdroj: autor ... 59
Obrázek 29 Okno programu JFLAP, zdroj: autor ... 60
Obrázek 30 Okno programu Automaton Simulator, zdroj: autor ... 62
Obrázek 31 Okno programu jFAST, zdroj: autor ... 63

Obrázek 32 Příklad automatu vytvořeného programem FSM simulator, zdroj: autor 64

15 SEZNAM TABULEK

Tabulka 1 Tabulka přechodů automatu 5

Tabulka 2 Převod NKA na DKA... 7

Tabulka 3 Srovnání funkcí jednotlivých programů.. 65

16 SEZNAM PŘÍLOH

Příloha 1 Uživatelská příručka

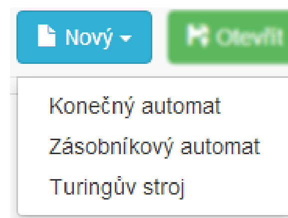
Příloha 2 CD se zdrojovými kódy aplikace

UŽIVATELSKÁ PŘÍRUČKA

Aplikace je dostupná na adrese http://automaty.wikan.com. Offline verzi je možné získat na přiloženém CD v adresáři „publish“. Na následujícím obrázku je zobrazeno okno aplikace s vytvořeným konečným automatem:

V horní části okna je navigační lišta umožňující vytvořit nový automat, otevřít už dříve vytvořený či současný automat uložit.

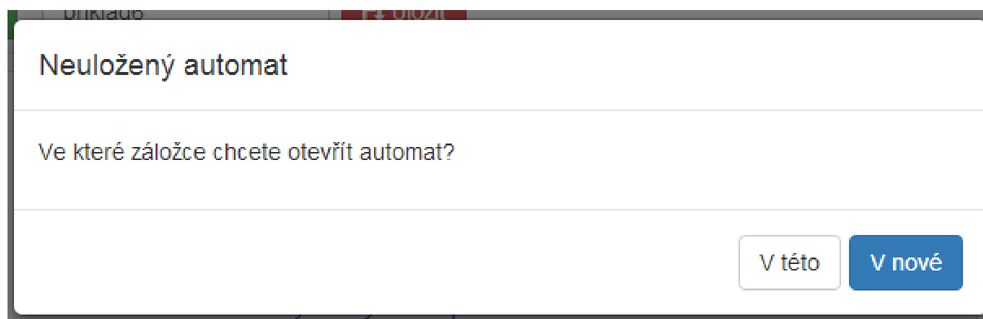
Vytvoření nového automatu



Po kliknutí na tlačítko „Nový“ se objeví nabídka s podporovanými typy automatů. Poté, co uživatel vybere požadovaný typ, se otevře nová záložka prohlížeče, ve které bude možno tento automat editovat.

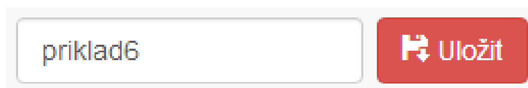
Otevření automatu

Dříve uložený automat je možné otevřít po kliknutí na tlačítko „Otevřít“. Aplikace po vybrání příslušného souboru zkontroluje, zda už nemá uživatel nějaký automat otevřený, a podle toho buď automat ihned vytvoří, nebo zobrazí dialogové okno s dotazem, ve které záložce se má tento automat otevřít.



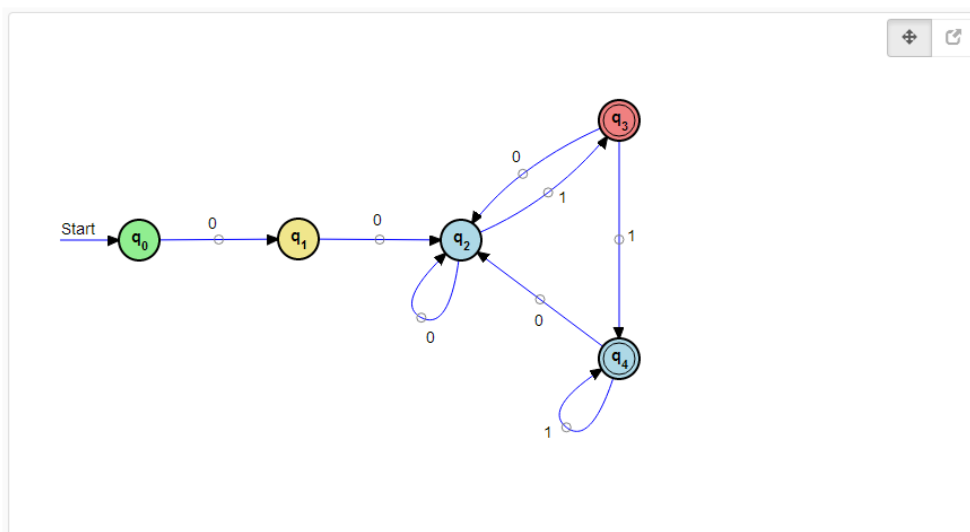
Uložení automatu

Aby bylo možné automat uložit, je nutné nejprve vyplnit jeho jméno a poté kliknout na tlačítko „Uložit“. Podle nastavení webového prohlížeče se pak buď zobrazí dialog pro uložení souboru, či se soubor automaticky stáhne do nastavené složky (obvykle složka „Stažené soubory“ v profilu uživatele).



Editace automatu

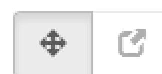
Automat je možné editovat na kreslicí ploše pod navigační lištou.



Na této ploše je možné provádět následující operace:

- **Vytvoření stavu**: Nový stav se na plochu přidá po dvojkliku na místo, kde má být stav umístěn.
- **Vytvoření přechodu**: Nový přechod je možné vytvořit stisknutím klávesy „Ctrl“ následovaným stiskem levého tlačítka myši na počátečním stavu a tažením myši (při stále stisknutém levém tlačítku) na koncový stav. Nový stav se vytvoří po uvolnění tlačítka myši. Alternativně je možné kliknout na přepínací tlačítko v pravém horním rohu kreslicí plochy, viz dále.

- **Odstranění stavu či přechodu**: Požadovaný stav resp. přechod je možné po vybrání levým tlačítkem myši odstranit stiskem klávesy „Delete“. Alternativně je možné je smazat v editačním panelu stavu resp. přechodu, viz dále.
- **Vytvoření smyčky**: Princip je stejný jako u běžného přechodu mezi dvěma stavy. Je pouze nutné potáhnout myší mimo prostor stavu a pak zase zpět.
- **Zakřivení přechodu**: Uprostřed každého přechodu se nachází malé kolečko, které je možné chytit myší a tažením přechod upravit do požadovaného stavu.
- **Napřímení přechodu**: Po dvojkliku na výše zmíněné kolečko se přechod opět napřímí.
- **Změna umístění stavu**: Stav je možné přesunout myší na požadované místo.
- **Změna umístění popisku přechodu**: Stejný princip jako v předchozím případě.



V pravém rohu kreslicí plochy se nachází dvojice tlačítek, která slouží jako přepínač toho, jak aplikace reaguje na potažení stavu myší. Při standardním chování se tímto potažením mění pozice příslušného stavu. Po přepnutí tlačítka na druhou pozici se pozice stavů uzamkne a tažením myši se vytvářejí nové přechody. Není tak nutné stále držet tlačítko „Ctrl“ při vytváření více přechodů.

Panel „Automat“

Vpravo od kreslicí plochy se nachází panel s informacemi o současném automatu. Nachází se zde zejména zobrazení typu automatu a podle tohoto typu až dvě tlačítka, která slouží pro zobrazení dalších informací. U konečných a zásobníkových automatů je možné zobrazit ekvivalentní gramatiku, u nedeterministických konečných automatů ještě tabulku s převodem tohotou automatu na deterministický.

Automat

Typ: Konečný automat

Zobraz gramatiku

Zobraz deterministický

Gramatika ×

G = ({S,A,B,C,D,E},{0,1},P,S)
S → A
A → 0B
B → 0C
C → 1D | 0C
D → 0C | 1E
E → 1E | 0C

Převod na deterministický KA ×

	a	b
{q0}	{q1,q2}	
{q1,q2}		{q1}
{q1}		

Tyto dodatečné informace je možné skrýt opětovným stiskem daného tlačítka či klikem na křížek v pravém horním rohu příslušného panelu.

Panel „Stav“

Po vybrání stavu myší je možné tento stav editovat v panelu, který se nachází pod panelem „Automat“. Je zde možné editovat název stavu, jeho barvu a nastavit, zda je počáteční či přijímající. U Turingových strojů je navíc možné nastavit stav i jako zamítající. Změněné informace je nutné uložit kliknutím na tlačítko „Uložit“. Tlačítko „Odstranit“ slouží k odstranění vybraného stavu z automatu.

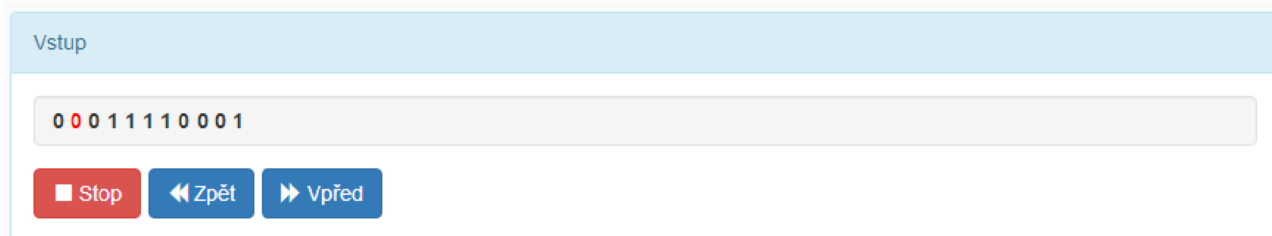
Panel „Přechod“

Vzhled panelu „Přechod“ je velmi závislý na vybraném typu automatu. Všechny typy automatů zde mají editační políčko pro zadání vstupního symbolu. U konečných automatů je možné zadat více vstupních symbolů. Tyto symboly není nutné nijak oddělovat, stačí je pouze zapsat za sebou. Zásobníkové automaty mají navíc políčka pro zadání symbolu na vrcholu zásobníku a symbolu, který se má do zásobníku vložit. Turingovy stroje zase mají vstupní pole pro symbol, který má být zapsán na pásku, a výběr kterým směrem se má po pásce posunout čtecí hlava. Opět je zde dvojice tlačítek „Uložit“ a „Odstranit“, která mají stejný význam jako u předchozího panelu.

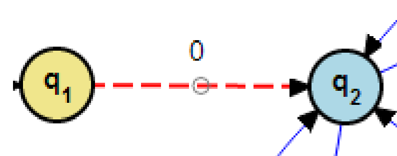
Panel „Vstup“

Pod kreslicí plochou se nachází panel pro simulaci činnosti vytvořeného automatu. Vzhled tohoto panelu je opět závislý na typu automatu. U konečných automatů se zde zadává posloupnost vstupních symbolů a u Turingových strojů počáteční stav symbolů na pásce. U zásobníkových automatů ještě navíc počáteční stav zásobníku a zda automat přijímá vyprázděním zásobníku. V opačném případě automat přijímá dosažením koncového stavu.

Dále se zde nachází tlačítko „Start“, kterým se spustí samotná simulace. Po stisku tohoto tlačítka už není možné automat editovat, stejně tak jako není možné editovat obsah panelu „Vstup“. Samotné tlačítko „Start“ zmizí a místo něj se objeví trojice tlačítek „Stop“, „Zpět“ a „Vpřed“.



Tlačítko „Stop“ slouží k zastavení simulace. Tlačítka „Zpět“ a „Vpřed“ slouží k posunu po jednotlivých krocích simulace. Při každém kroku je v automatu zvýrazněn přechod (resp. přechody u nedeterministických konečných automatů), který je v tomto kroku použit a zároveň je zvýrazněn současný vstupní symbol.



Po ukončení simulace (kromě zastavení tlačítkem „Stop“) je nad kreslicí plochou zobrazeno hlášení, zda byla simulace úspěšná či neúspěšná.

Automat přijímá vstup.

UNIVERZITA HRADEC KRÁLOVÉ
Fakulta informatiky a managementu
Rokitanského 62, 500 03 Hradec Králové, tel: 493 331 111, fax: 493 332 235

Zadání k závěrečné práci

Jméno a příjmení studenta: **Milan Kopsa**

Obor studia: Aplikovaná informatika (2)
Jméno a příjmení vedoucího práce: **Andrea Ševčíková**

Název práce:
Výukový nástroj pro vizualizaci a kreslení automatů.

Název práce v AJ:
Learning tool for visualizing and drawing machines.

Podtitul práce:

Podtitul práce v AJ:

Cíl práce: Vytvoření výukového nástroje pro vizualizaci a kreslení automatů pro podporu výuky automatů v předmětu Teoretická informatika a jeho srovnání s obdobnými produkty.

Osnova práce:

- Prohloubení si znalostí z oblasti teorie automatů a formálních jazyků
- Teoretické zpracování problematiky teorie automatů
- Analýza a návrh aplikace
- Volba nástrojů pro vývoj aplikace
- Implementace aplikace
- Testování aplikace
- Správa zdrojových kódů
- Zpracování získaných poznatků a srovnání s obdobnými produkty

Projednáno dne:

Podpis studenta

Podpis vedoucího práce