VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

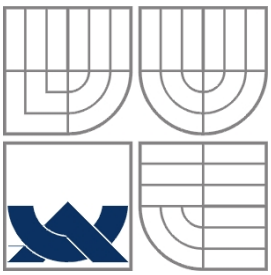
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

APLIKACE PRO ZOBRAZENÍ STRUKTURY KONEČNÉHO AUTOMATU

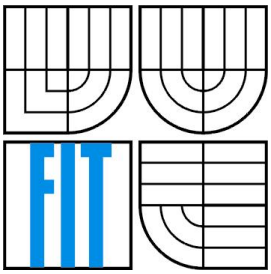
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE Ondřej Polcer
AUTHOR

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

APLIKACE PRO ZOBRAZENÍ STRUKTURY KONEČNÉHO AUTOMATU

APPLICATION FOR VISUALIZATION OF FINITE AUTOMATON STRUCTURE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — Ondřej Polcer
AUTHOR

VEDOUCÍ PRÁCE — ING. Eva Zámečníková
SUPERVISOR

BRNO 2010

Abstrakt

Cílem práce je aplikace, která převádí textovou podobu konečného automatu na tzv. „stavový diagram“, který je obecně pro vyjádření vzájemných vazeb názornější, než textová forma. Nejprve se strojová podoba konečného automatu určité syntaxe převede do objektové podoby a doplní se o další udaje. Následuje algoritmizace hledání vhodných umístění pro jednotlivé stavy, aby struktura jejich přechodu byla přehledná a srozumitelná s ohledem na komplexnost zadaného konečného automatu. Aplikace vytvoří sadu ekvivalentních zobrazení v různé formě a dodá vhodné informace pro snadnou interpretaci výsledku.

Abstract

The goal of this thesis is to create an application that converts text form of a finite state machine to a state diagram. First, the code form of finite state machine syntax is transformed into the object form and complemented by additional data. The algorithm of finding appropriate placement for individual state of structure follows to make their transition clear and understandable with considering the complexity of a given finite state machine. The application creates a set of equivalent views in various form and delivers the appropriate information for easy interpretation of results.

Klíčová slova

Grafická reprezentace, konečný automat, stavový diagram, c++, strojový zápis

Keywords

Graphic representation, finite state machine, state diagram, c++, computer-readable form

Citace

Ondřej Polcer: Aplikace pro zobrazení struktury konečného automatu, bakalářská práce, Brno, FIT VUT v Brně, 2010

Aplikace pro zobrazení struktury konečného automatu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Evy Zámečníkové
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Ondřej Polcer
18. 05. 2010

Poděkování

Všem diskutujícím myslitelům.

Zadání bakalářské práce/10070/2009/xpolce01

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2009/2010

Zadání bakalářské práce

Řešitel: **Polcer Ondřej**
Obor: Informační technologie
Téma: **Aplikace pro zobrazení struktury konečného automatu**
Application for Visualization of Finite Automaton Structure
Kategorie: Teorie informatiky

Pokyny:

1. Prostudujte problematiku konečných automatů, zejména způsob zápisu konečného automatu pro strojové zpracování.
2. Nastudujte potřebné kapitoly z tvorby uživatelského rozhraní.
3. Vytvořte návrh aplikace, která na základě zápisu konečného automatu pro strojové zpracování graficky zobrazí strukturu daného konečného automatu.
4. Dle pokynů vedoucího implementujte návrh aplikace.
5. Zhodnoťte dosažené výsledky a diskutujte další možný vývoj projektu.

Literatura:

- Meduna A.: Automata and Languages: Theory and Applications, Springer-Verlag, London, 2000.
- Žára, J., Beneš, B., Sochor, J., Felkel, P.: Moderní počítačová grafika, Computer Press, 2005.

Při obhajobě semestrální části projektu je požadováno:

- Body 1. až 3.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Zámečníková Eva, Ing.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2009
Datum odevzdání: 19. května 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Obsah

1 Úvod

Konečný automat (dále KA či FSM z anglického *finite state machine*) je abstraktní, výpočetní model používaný v informatice například pro zpracování regulárních výrazů, jako v lexikálním analyzátoru, či obecně pro studium formálních jazyků a v mnoha dalších vědních oborech jako jsou například matematika, logika, lingvistika, logistika či v překladech z programovacího či mluveného jazyka do jiného jazyka.

Jeho počátky se datují do druhého čtvrtletí dvacátého století. Do doby, kdy se teprve vytvářel koncept sálových počítačů. Samotný název KA je odvozen od konečné množiny vstupních symbolů, které automat čte ze svého vstupu a na základě kterých mění svůj aktuální stav.

Stavy reprezentují všechny možné konfigurace systému. Konečná, neprázdná množina vstupních symbolů (či abeceda) může představovat od zpráv objektů přes znaky regulárního jazyka po až příchozí data v síťovém protokolu. A konečná, neprázdná množina pravidel vytváří přechodovou funkci ze stavu do stavu na základě příchozího aktuálního symbolu na vstup KA. Díky své univerzálnosti umí KA zobrazovat rozličné situace a tím se stává šikovnou pomůckou nejen v informačních technologiích.

Cílem této aplikace je především grafické, ale i matematické zobrazení jednoduchých i rozsáhlejších definic KA na základě buď uživatelského vstupu či načteného, strojového zápisu konfigurace. Dále zprostředkování potřebných informací pro snadnější pochopení a interpretaci zobrazené problematiky a poskytnutí vhodných údajů pro samostatné vytváření KA na základě libovolné tématiky, kterou aplikace následně zobrazí.

2 Teoretický základ

Matematicky je konečný automat uspořádaná pětice:

$$KA = (Q, T, \delta, s, F)$$

(i) Q je abeceda stavů
(ii) T je abeceda vstupních symbolů
(iii) $\delta : Q \times T \rightarrow 2Q$ je přechodová funkce (2Q označuje množinu všech podmnožin množiny Q)
(iv) $s \in Q$ je počáteční stav
(v) $F \subseteq Q$ je množina koncových stavů [1].

Abeceda je libovolná, neprázdná, konečná množina. Prvky abecedy jsou symboly [1].

Konfigurace KA má tvar dvojice: (p,q); $p \in Q$, $q \in T^*$. Tedy, jeden stav a konečná sekvence vstupních symbolů na pásce [1].

Počáteční konfigurace KA má tvar dvojice: (s,q); $q \in T^*$. Tedy, jeden počáteční stav a celá vstupní páska [1].

Koncová konfigurace KA má tvar dvojice: (p,ε); $p \in F$. Tedy, jeden z konečných stavů a prázdná vstupní páska [1].

Rozšířená přechodová funkce umožňuje do konečné vstupní abecedy krom znaků zadávat i řetězce, tedy formálně mění definici na tento tvar:

(iii) $\delta : Q \times T^* \rightarrow 2Q$ je rozšířená přechodová funkce
(Q je abeceda stavů; $2Q$ označuje množinu všech podmnožin množiny Q;
T je abeceda vstupních symbolů a T^* je i-tá mocnina nad T, kde $i \geq 0$; $i \in N$) [4]

Výstup KA není povinný pro definici KA, není-li zadaný, jde o akceptory. Tedy o nástroje rozhodování, zda řetězec patří do určité gramatiky. A protože KA pracuje nad regulárními výrazy, musí nad nimi být vytvořena i posuzovaná gramatika. Cílová aplikace bude pracovat bez výstupů.

Činnost KA lze obecně popsat tak, že přijímá slovo, tvořené z abecedy nad regulární gramatikou. A to tím způsobem, že začíná v počátečním stavu, načte první znak ze vstupu a podle pravidla se posune do dalšího stavu (jeden takt), odkud opět čte další znak, hledá odpovídající pravidlo a tak dále, dokud nezpracuje celou vstupní sekvenci a pokud se po jejím dočtení nachází KA v jednom z konečných stavů, znamená to, že zadané slovo vyhovuje gramatice, podle které pracuje KA. V opačném případě či v případě zaseknutí nikoli.

Pokud se neomezíme na regulární tématiku, může KA rozhodovat v mnoha dalších situacích jako je například vyhodnocení správného pořadí příkazů síťového protokolu, činnost výdejového automatu či obsluhu stavů periferních zařízení v počítači.

Pro účel této aplikace jsou klíčové množiny stavů, vstupních symbolů a pravidel. Z této trojice lze sestavit grafickou reprezentaci KA, další dvojice počáteční stav a množina koncových stavů slouží k plnění funkce KA, tedy k ověřování zda dané vstupní slovo náleží do určitého regulárního jazyka, který reprezentuje KA. Grafická podoba však musí být schopná zachovat úplnost vstupní informace pro její budoucí využití, proto se musí počáteční stav a množina koncových stavů vizuálně odlišit.

2.1 Druhy konečných automatů

Rozlišujeme Mealyho a **Mooreův** konečný automat, rozdíl je mezi nimi ten, že Moore používá pro výpočet dalšího stavu pouze svůj vnitřní stav, Mealy ještě aktuální, čtený symbol na vstupu. Oba druhy jsou mezi sebou převoditelné těmito algoritmy:

Převod stavového automatu Moorova typu na automat Mealyho typu lze provést následujícím způsobem: hodnoty výstupních signálů uvedené v uzlech představujících stavy automatu přepíšeme ke hranám (šipkám), které do příslušných stavů směřují a typ těchto signálů změníme z kombinačního na registrový typ. V automatu takto vzniklém jsou ještě hodnoty výstupních signálů v každém stavu plně určeny tímto stavem. Tento automat lze nyní zjednodušit sloučením hran (přechodů) a stavů, které to dovolují. Je možno sloučit hrany, které vycházejí i končí ve stejných stavech a mají-li přiřazeny stejné hodnoty výstupních signálů. Přitom bude podmínka přechodu u nové hrany vytvořené sloučením rovna logickému součtu (disjunkce = nebo) podmínek slučovaných hran. Sloučit lze stavy, z nichž vycházejí hrany směřující do týchž nových stavů, přičemž jsou těmto přechodům přiřazeny stejné hodnoty výstupních signálů. Touto úpravou konečně vzniká automat Mealyho typu [8].

Převod stavového automatu Mealyho typu na automat Moorova typu: je možný, neobsahuje-li původní automat kombinační (asynchronní) výstupy, které reagují na vstupní signály bez čekání na aktivaci hodinového signálu. Při převodu rozštěpíme stavy původního automatu, do nichž směřují hrany s různými hodnotami výstupních signálů, na dílčí stavy tak, aby do každého z nich směřovaly pouze hrany se stejnými hodnotami výstupních signálů (jde o proces obrácený k popsanému slučování stavů). Pak můžeme přenést hodnoty výstupních signálů z hran do uzlů, do nichž hrany směřují, a jejich typ změníme na kombinační [8].

Pro Mealyho a Mooreuv KA platí, že automat Mealyho typu může obsahovat méně stavů než ekvivalentní automat Moorova typu. Mealyho automat je odolnější na logický hazard, díky registrovaným výstupům a také s nim potřebuje kratší čas na ustálení výstupních signálů po aktivní hraně hodinového signálu, oproti Mooreova automatu, protože tyto signály nemusí procházet kombinační logikou výstupu. Z těchto důvodů se obvykle automatům Mealyho typu dává přednost, i když jejich návrh je obtížnější. Popsaný převod automatu Mealyho typu na automat Moorova typu se tedy v praxi obvykle neprovádí a lze jej chápat spíše jako teoretickou možnost. Z uvedených důvodů bude aplikace znázorňovat Mealyho automat. To znamená, že KA bude brát v úvahu symbol na vstupu [8].

Dalším dělením je na **deterministický a nedeterministický** KA, kde deterministický má vždy pouze jedinou možnost přechodu, tedy pro aktuální konfiguraci existuje pouze jedno pravidlo a

opakovaným čtením stejné vstupní pásky dojdeme vždy k stejnému výsledku a zcela identickým taktům (krokům) řešení.

Jakmile obsahuje automat možnost volby (tj. existují alespoň jedna shoda pravidel typu: $Ab|\text{-}B$, $Ab|\text{-}C$,tedy s konfigurací *(A, b)* se podle náhody, či přesněji pseudonáhody, volí, zda se přesune do stavu *B* či *C*, protože obojí umožňují pravidla a definice), pak se jedná o nedeterministický KA.

Totální konečný automat, známý též jako úplný KA, je speciálním druhem deterministického KA. Používá se v těch situacích, kdy se vyžaduje, aby model reagoval (úplně) v každém stavu na všechny možné vstupy. A v případě chybného vstupu dokázal přejít do chybového stavu. Tato varianta KA se nemůže zastavit v průběhu zpracování. Při návrhu totálního KA se nejprve vytvoří deterministický KA, přidá se chybový stav a do něj povedou od každého stavu všechny hrany, které zatím nebyly u stavů použity. V chybovém stavu může zpracování setrvat až do dočtení vstupní pásky nebo z něj může vést hrana do počátečního stavu a plnit tak funkci resetu zařízení.

2.2 Další abstraktní vyjadřovací modely

Klasický KA pracuje s regulárními výrazy, ty však mají svá omezení a nezahrnují náročnější gramatiky, proto se využívají další abstraktní modely, s kterými se dále stručně seznámíme.

2.2.1 Zásobníkový konečný automat

Též ZKA, má větší vyjadřovací sílu než KA a umí pracovat s bezkontextovými gramatikami (viz.3.1Chomského hierarchie), které mohou reprezentovat třeba programovací jazyk. Proto tvoří základní součástí syntaktického analyzátoru v kompilátorech a sám o sobě je matematicky definován jako uspořádaná sedmice:

$$ZKA = (Q, T, Z, \delta, s, Z_0, F)$$

(i) Q je abeceda stavů
(ii) T je vstupní abeceda
(iii) Z je zásobníková abeceda
(iv) δ je konečná množina pravidel tvaru:

$$paA \rightarrow qw;\ kde\ p, q \in Q;\ a \in T \cup \{\varepsilon\},\ A \in Z,\ w \in Z^*$$

(v) s je počáteční stav, $s \in Q$
(vi) Z_0 je počáteční symbol na zásobníku, kde $Z_0 \in Z$
(vii) Z je množina koncových stavů, $F \subseteq Q$ [1]

Jak je vidět, do definice oproti KA přibyla zásobníková abeceda a počáteční znak na zásobníku. Jaké to poskytuje rozšíření??Klasický KA se rozhoduje podle aktuálního stavu a vstupní pásky, tedy podle přítomného stavu a vstupu. ZKA má navíc možnost pracovat s daty, uloženými v minulosti na zásobník typu LIFO (First in Last out, tj. zásobník s jediným přístupem: k vrcholu). Z uvedeného vyplývá, že pro konfiguraci bude potřeba o údaj víc pro stav zásobníku.

Konfigurace ZKA: je uspořádaná trojice: (q , w , z) , *kde* $q \in Q$; $w \in T^*$; $z \in Z^*$ *[1]*

Přechodová funkce ZKA: s dodržením předchozího významu symbolů, platí:

$$(q , aw , Yz) \vdash (p , w , yz)$$

znamená, že:

(i) automat přejde ze stavu q do stavu p,
(ii) symbol Y (tzn. vrchol zásobníku) bude nahrazen slovem y,
(iii) pokud $a \neq \varepsilon$, pak automat tento symbol přečte a posune čtecí hlavu na vstupní pásce a jedno pole doprava. Nebo pokud $a = \varepsilon$, zůstává čtecí hlava vstupní pásky na místě [1].

Pro úplnost je potřeba doplnit, že oproti situaci s KA, má nedeterministický ZKA větší vyjadřovací sílu, než deterministický ZKA, proto se právě tetko druha automatu využívá pro důkazy, kde se ukáže, že tento ZKA přijímá jazyk ekvivalentní nějakému jazyku generovanému bezkontextovou gramatikou (tj. Jazyku typu 2).

2.2.2 Turingův stroj

Má ještě vyšší vyjadřovací sílu, než ZKA, po specifikaci dokáže pracovat s kontextovými jazyky a umožňuje dokázat ekvivalenci mezi jazyky typu 0 a jazyky přijímanými Turingovým strojem (viz.3.1Chomského hierarchie). Byl vytvořen anglickým geniálním matematikem a průkopníkem informatiky Alanem Turingem, existuje k němu několik variací, všechny mají však stejnou vyjadřovací sílu jako originál [2].

Churchova (Church-Turingova) teze: Turingovy stroje (a jim ekvivalentní systémy) definuji svou výpočetní silou to, co intuitivně považujeme za efektivně vyčíslitelné [2].

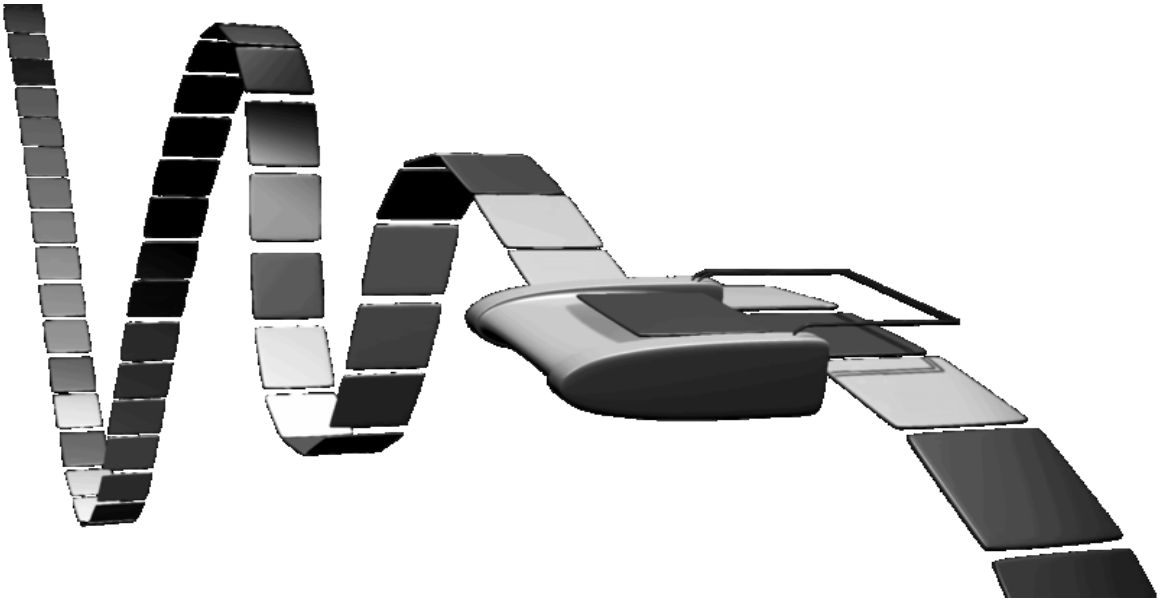
Matematicky lze Turingův stroj vyjádřit jako uspořádanou šestici:

$$\boldsymbol{TS = (Q, \Sigma, \Gamma, \delta, q_0, q_F)}$$

(i) Q je abeceda stavů
(ii) Σ je vstupní abeceda, platí, že *blank* (tj. prázdný znak) nenáleží Σ
(iii) Γ je konečná množina symbolů, $\Sigma \subset \Gamma$, blank $\in \Gamma$, nazývaná páková abeceda
(iv) parciální funkce $\delta : (Q \setminus \{q_F\}) \times \Gamma \rightarrow Q \times (\Gamma \cup \{L,R\})$, kde L,R nenáleží Γ, je přechodová funkce
(v) q_0 je počáteční stav, $q_0 \in Q$
(vi) q_F je počáteční stav, $q_F \in Q$ [2]

Popis činnosti Turingova stroje: uvažujme variantu s jednostranně nekonečnou páskou. Stroj postupuje podobně jako konečný automat, ale s tím rozdílem, že má nekonečně velkou pásku, na kterou může zapisovat a libovolně se po ní pohybovat. Na začátku definujeme stavy a přechodové funkce, přijímající a zamítající stav a abecedu, nad kterou stroj pracuje. Poté na pásku stroje vložíme

slovo, které se skládá ze symbolů abecedy a stroj se pokusí toto slovo přijmout. Při tom můžeme chtít, aby nám nějakým způsobem pozměnil symboly na pásce, čímž můžeme realizovat nějaký výpočet, například sčítání. Turingův stroj může skončit typicky ve třech situacích: pokud vstupní slovo *w* vyhovuje podmínkám, slovo přijme (akceptuje). Pokud slovo *w* nevyhovuje, slovo odmítne. Poslední možností je, že se stroj zacyklí, což znamená, že stroj bude donekonečna přecházet mezi nějakými stavy a nikdy se nezastaví. Turingův stroj tedy neumí vyřešit všechny problémy [12].



Obr. 2.2.2.1 Umělecké znázornění Turingova stroje [12]

2.3 Chomského klasifikace gramatik

Jde o hierarchii formálních gramatik, byla vytvořena v roce 1956 Noamem Chomskym, lingvistou, politikem a politickým anarchosyndikalistou z Philadelphie, který se významně zasloužil mimo jiné o rozvoj teoretické informatiky [3].

Obr. 2.3.1 Vztahy mezi třídami Chomského klasifikace

Pro snadné pochopení ještě dodejme, že termy (či terminály) jsou v programu všechny proměnné, konstanty a funkce, obsahující za parametry pouze termy. Tedy to, co je jakkoli zástupné pro vše, definované programátorem. A konečně neterminály (či non-term) jsou vše ostatní, tedy

jazykové konstrukce cyklu, klíčová slova (př.: if, then, else, do, break...), operátory, příkazy (př: lf, crlf, ;) a další.)

Gramatiky typu 0

Zahrnují v sobě všechny formální gramatiky, generují rekurzivně spočetné jazyky, které mohou být rozpoznané určitým Turingovým strojem. Tyto jazyky se někdy též nazývají gramatiky bez omezení, nebo fázové gramatiky [3].

Gramatiky typu 1 (kontextové gramatiky)

Generují kontextové jazyky. Tyto gramatiky se skládají z pravidel $\alpha A\beta \rightarrow \alpha\gamma\beta$, kde A je neterminál a α, β a γ řetězce terminálů a neterminálů. Řetězce α a β mohou být prázdné, ale γ musí být neprázdná. Pravidlo S-> ε je povoleno, pokud se S nevyskytuje na pravé straně žádného pravidla. Tyto jazyky jsou rozpoznatelné lineárně ohraničeným automatem [3].

Gramatiky typu 2 (bezkontextové gramatiky)

Generují bezkontextové jazyky. Skládají se z pravidel $A \rightarrow \gamma$, kde je A neterminálem a γ řetězcem terminálů a neterminálů. Pravidlo $S \rightarrow \gamma$ je povoleno, pokud se S nevyskytuje na pravé straně žádného pravidla. Tyto jazyky jsou právě jazyky rozpoznatelné určitým nedeterministickým zásobníkovým automatem [3].

Gramatiky typu 3 (regulární gramatiky)

Známé též jako lineární gramatiky, generují regulární jazyky. Pravidla těchto gramatik jsou omezena na jeden neterminál na levé straně. Pravá strana se skládá z řetězce terminálů, který může být následován jedním neterminálem [3].

A rozlišuje se levá a pravá lineární gramatika, podle následujícího systému: X a Y jsou neterminály a w je terminál, pak

$X \rightarrow Yw$ *a* $X \rightarrow w$, je levá lineární gramatika,

$X \rightarrow wY$ *a* $X \rightarrow w$, je pravá lineární gramatika.

Pravé lineární gramatiky a levé lineární gramatiky jsou ekvivalentní [3].

Další podskupinou jsou regulární gramatiky ve standartní formě. Ta zahrnuje podmínku, aby na pravé straně každého pravidla stál buď jeden term s jedním neterminálem nebo prázdný znak. Všechny tyto jazyky jsou rozpoznatelné konečným automatem [3].

2.4 Základní způsoby reprezentace konečného automatu

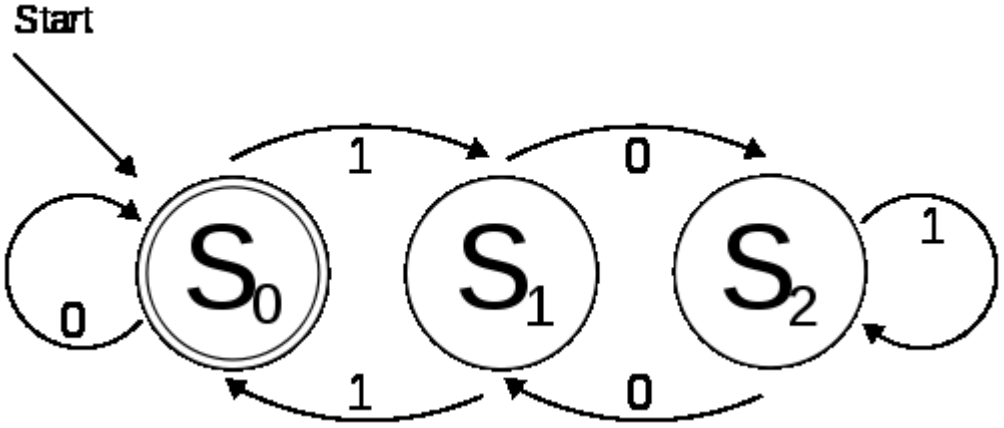
Nejrozšířenější jsou tyto čtyři způsoby reprezentace:

- stavový diagram

- výčet prvků
- tabulka přechodů
- stavový strom

Postupně si je představíme a popíšeme na této definici KA:
KA={{S0, S1, S2}, {0,1}, δ, S0, {S0}}

Stavový diagram je z nich neklasičtější a bude ho také používat aplikace. Zásady jeho zobrazení jsou tyto: stavy představují kružnice, které obsahují název stavu. Vstupní abeceda je zobrazena u šipek, počáteční stav je označen samostatnou šipkou a koncové stavy jsou tvořeny dvěma soustřednými kružnicemi pro odlišení od nekoncových.



Obr. 2.4.1 Ukázka stavového diagramu [3]

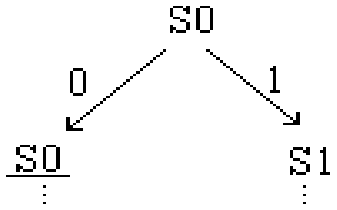
Výčet prvků je další, leč negrafickou reprezentací. Pravidla KA by se v něm vyjádřila takto:

```
δ(S0, 0)=S0 ;      δ(S0, 1)=S1;      δ(S1, 0)=S2;
δ(S2, 1)=S2;       δ(S2, 0)=S1;      δ(S1, 1)=S0;
```

Tabulka přechodů je velmi přehledná a používaná, leč opět negrafická. Sloupce obsahují vstupní abecedu, řádky stavy a jejich propojením nalezneme stav, do kterého automat přejde na základě této konfigurace. Dodejme, že počáteční stav má obvykle před sebou šipku a koncový bývá označen nejčastěji podtržením. Pravidla našeho KA by v ní byla vyjádřena takto:

Stavy / Vstupní řetězce	0	1
->S0	S0	S1
S1	S2	S0
S2	S1	S2

Stavový strom je minoritně používanou grafickou reprezentací a zobrazí jen ty KA, které neobsahují nedosažitelné stavy. Automat bez těchto stavů je však ekvivalentní.



Obr. 2.4.2 Ukázka struktury stavového stromu

Počáteční stav je kořenem stromu. Z každého vrcholu, který není listem, vychází právě tolik hran, kolik má příslušný stav následníků. Tyto hrany k následníkům jsou označeny popisky podle očekávaného řetězce v množině pravidel. Jestliže nějaký stav odpovídá více uzlům, pak hrany vycházejí jen z jednoho z těchto uzlů. Výsledný graf může nabývat více podob podle toho, zda ho procházíme do šířky či do hloubky [5].

3 Zápis konečného automatu pro strojové zpracování

Aby mohl být KA zobrazen, musí existovat jednoznačný způsob zápisu, s kterým bude aplikace pracovat, tedy ukládat vytvořené KA do této formy i v ní očekávat vstupy.

Nároky na volbu způsobu zápisu jsou:

- rychlá orientace pro snadnou práci
- využití již známé syntaxe, proto, aby nevznikala nutnost učit se speciální zápis pro použití jedné aplikace,
- snadná rozšiřitelnost, například, kdyby měla aplikace zobrazovat místo uspořádané pětice složitější matematické *n*-tice,
- přesně definované nepovolené či zástupné znaky, aby nevznikala nejednoznačnost zápisu a tím pravděpodobná chyba zobrazení
- možnost vkládat vlastní komentáře či komentovat části definice a tím změnit popis KA
- odolnost na chybové vstupy

3.1 Minimální textová forma

První uvažovanou formou zápisu byla **minimální textová podoba**, která se skládala pouze ze tří částí: pravidla, počáteční stav a koncové stavy. Jejich rozborem by bylo možné extrahovat všechny použité znaky a stavy a tím doplnit definici. Jednotlivé množiny by byly od sebe odděleny pouze zalomením řádku a díky fixnímu pořadí jim nemusí předcházet žádná kódová značka. Tento systém by tak automaticky odřadil nedosažitelné stavy a nevyužité vstupní symboly a tím prováděl přirozeně minimalizaci KA, při ekvivalentním zobrazení.

Příklad zápisu v minimální textové podobě:

```
#priklad konfigurace konecneho automatu v minimalni, textove podobe
#fixni poradi zapisu mnozin umoznuje absenci kodovych znacek
#zacina pravidly
Ab->B
Ba->A
Aa->A
Bb->B
Bc->C
```

```
#hned nasledovane jednim pocatecnim stavem
A
#na poslednim miste je mnozina koncovych stavu
B, C
#konec konfigurace
```

Tato forma má nespornou výhodu ve velké efektivnosti zápisu, protože postrádá logické oddělovače mezi množinami či jiné značky, které by usnadňovaly orientaci a strojové zpracování. Vyhraněný oddělovač v pravidlu je v tomto případě šipka, tu nesmí obsahovat žádný stav či hrana, protože by došlo k záměně a tím k chybě zpracování.

Nevýhodou minimální, textové formy zápisu je obtížná orientace, protože není srozumitelná na první pohled - intuitivně, bez doplňujícího vysvětlení. Dále odstranění nedosažitelných stavů je sice jeden z druhů optimalizace a výsledné KA jsou si ekvivalentní, nikoli však identické a v případě, že takové stavy zavedeme úmyslně a záleží na jejich korektním zobrazení, tato forma zápisu to nedovolí.

3.2 Úplná XML forma

Další variantou je zápis pomocí syntaxe **XML**. Tento příklad vytvořila aplikace na základě uživatelského vstupu a bude na něm vysvětlena práce s touto formou zápisu.

Příklad zápisu ve formátu xml:

```
<!--  Ondrej Polcer 4BIT FIT-VUTBR 2010  -->
    <KA>
<!--  Definice konecneho automatu  -->
<!--  Konecna mnozina stavu  -->
    <STAV>F</STAV>
    <STAV>E</STAV>
    <STAV>D</STAV>
    <STAV>C</STAV>
    <STAV>B</STAV>
    <STAV>A</STAV>

<!--  Vstupni abeceda  -->
    <ABECEDA>a</ABECEDA>
    <ABECEDA>b</ABECEDA>
    <ABECEDA>c</ABECEDA>
```

```
    <ABECEDA>d</ABECEDA>
    <ABECEDA>e</ABECEDA>
    <ABECEDA>f</ABECEDA>

<!--  Mnozina pravidel  -->
    <PRAVIDLO>Bc C</PRAVIDLO>
    <PRAVIDLO>Cd-D</PRAVIDLO>
    <PRAVIDLO>Ee->D</PRAVIDLO>
    <PRAVIDLO>Ae E</PRAVIDLO>
    <PRAVIDLO>Af F</PRAVIDLO>

<!--  Pocatecni a konecne stavy     -->
    <POCATEK>A</POCATEK>
    <KONEC>D</KONEC>
    <KONEC>E</KONEC>
    <KONEC>F</KONEC>

    </KA>
<!--  Konec dokumentu   ->
```

Pravidla syntaxe XML v rámci potřeb aplikace se dají stručně shrnout takto:

- zápis řetězce x do uvozovací a koncové značek `< a >` se nazývá tag x
- existují párové a nepárové tagy, budeme používat párové, mají tvar *`<x> tělo tagu </x>`*
- xml dokument musí obsahovat právě jeden kořenový element (párový tag), který ohraničuje celý obsah
- zanořování tagů je povolené, překrývání nikoli. To znamená, že ohraničující tag končí v těle vnořeného tagu např.: `<ohranicujici><vnoreny></ohranicujici></vnoreny>`
- komentáře se vnořuji pomocí uvozovací „`<!--`“ a koncové „`-->`“ značky

Tato varianta je výhodnější, než minimální textová forma, protože na rozdíl od ní umožňuje kompletní popis KA. Dále za cenu většího objemu dat umožňuje mnohem přehlednější zápis s rychlejší orientací v něm, což shledávám jako přednější. Navíc poskytuje ochranu typu nepříjetí pravidla, které nemá definované prvky v abecedě stavů a vstupních symbolů. Podotýkám, že minimální textová forma by takové pravidlo přijala s tím, že by obě množiny doplnila o chybějící údaje. Z těchto důvodů jsem se rozhodl pro upřednostnění xml formy zápisu a přizpůsobil jí aplikaci, detaily tohoto řešení obsahuje kapitola 4.2.Specifikace strojového zápisu.

4 Návrh

Při návrhu aplikace jsem věnoval velkou důležitost těmto požadavkům:

- multiplatformní přístup je nejen spravedlivý, ale i výhodný, protože v sobě nenese omezení pro uživatele, zvyklého na určitý operační systém, a nenutí ho/ji pořizovat si či zavádět konkrétní operační systém pokud si chce spustit tuto aplikaci
- pokud možno nabídnout co nejvíce intuitivní ovládání, aby se minimalizoval čas nutný pro seznámení s aplikací
- způsob uložení konfigurace do strojového kódu musí vyhovovat těmto požadavkům sestupně podle důležitosti: využití již existující syntaxe, jednoznačnost, rozšiřitelnost, malý objem přidaných-redundantních dat
- v případě neočekávaných komplikací, které znemožní grafické zobrazení KA, musí aplikace umět minimálně poskytnout výstup ve formě čistě textového vyjádření.

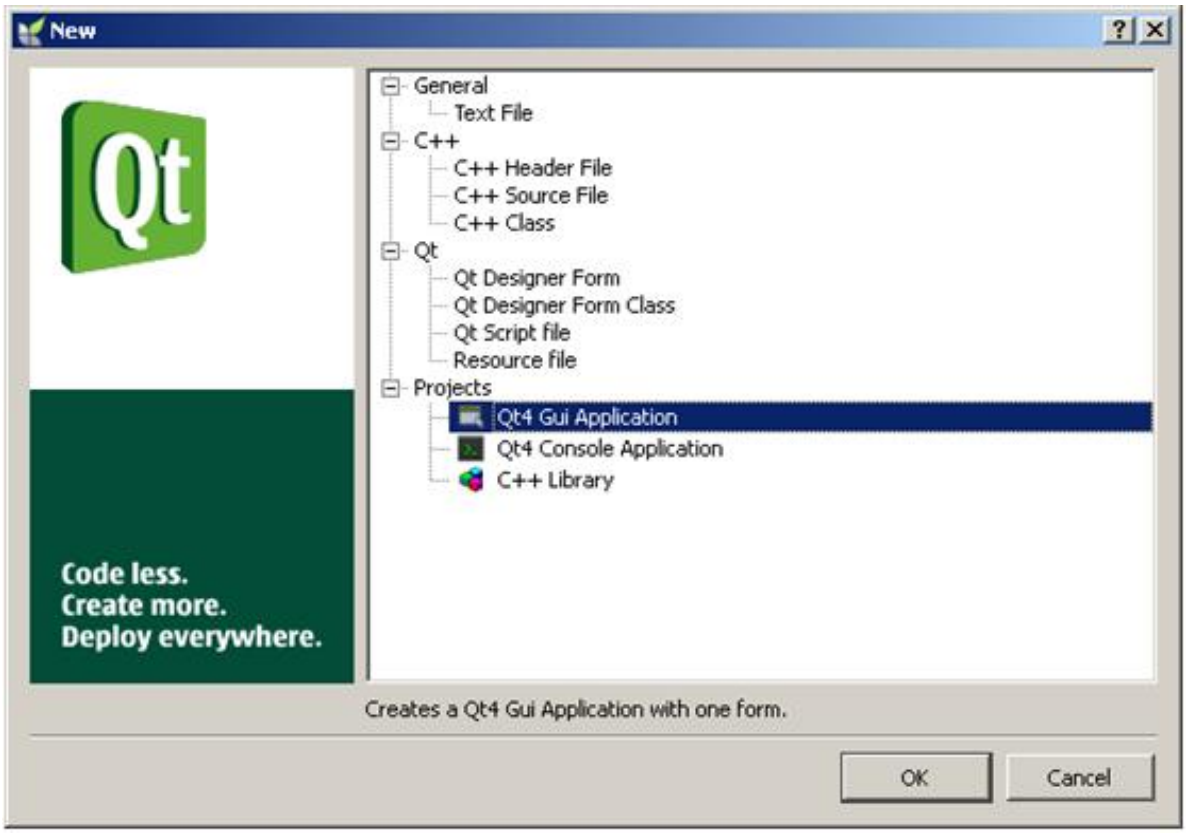
4.1 Výběr vývojového prostředí

Základní myšlenkou je, že větší smysl, než specializovaný software pro minoritní platformy, má pragmatický a kdekoli spustitelný program. Hlavním kritériem pro výběr prostředí je tedy míra přenositelnosti. Po pečlivém hledání a důkladném diskutování jsem jako odpovídající vybral dvě knihovny: Qt a Swing, které v následujících kapitolách představím.

Po zvážení všech výhod a nevýhod jsem upřednostnil knihovnu Qt, protože je rychlejší, nabízí velké množství nástrojů, pracuje nad c++, a nikoli nad javou jako Swing, a v neposlední řadě pro její oblíbenost a rozšířenost mezi programátory, což dokazuje i množství softwaru, které je v něm napsané.

4.1.1 Qt

Knihovnu vyvíjí norská společnost Trolltech (dříve Quasar Technologies, nyní vlastněná společností Nokia) pro platformy X Window System (Qt/X11), Mac OS X (Qt/Mac), MS Windows (Qt/Windows), Windows CE (Qt/WinCE), Java a Embedded. QT patří mezi nejpopulárnější mutliplatformní knihovnu pro vytváření uživatelských grafických rozhraní. Mezi nejznámější software využívající Qt je: prostředí KDE, webový prohlížeč Opera, Google Earth, Skype, VirtualBox, Qtopia a OPIE. QT má rozsáhlou strukturu, což je výhod z pohledu množství nástrojů a nevýhoda z pohledu práce s ní [6].



Obr. 4.1.1.1 Ukázka z vytváření aplikace v QT [7]

4.1.2 Swing

Jde o knihovnu tříd a komponent pro tvorbu grafického uživatelského rozhraní v Javě. První verze se objevila v r. 1997. Swing má vlastní knihovnu komponent a tudíž nevyužívá komponenty operačního systému, proto všude vypadá a chová se stejně. Další výhodou je, že jde o přiměřeně rozsáhlou knihovnu, oproti QT a tudíž u ní předpokládám rychlejší orientaci a práci díky přehlednější objektové struktuře. Obsahuje metody pro práci s datovou strukturou stromů (JTree), snadné změny chování a vzhledu (Look and feel), podpora interpretace pro nevidomé a celkově je přizpůsobeno snadnému grafickemu návrhu. Naopak nevýhodou je, že se aplikace odlišuje od ostatních aplikací v OS a že má vyšší nároky na strojový čas, takže zejména na starších počítačích se aplikace ve swingu mohou zdát pomalé [9].

4.2 Specifikace strojového zápisu

Opět si ukážeme příklad úplného XML zápisu a na něm si vysvětlíme možnosti a omezení této formy.

```
<!--  Ondrej Polcer 4BIT FIT-VUTBR 2010  -->
    <KA>
```

```
<!--  Definice konecneho automatu  -->
<!--  Konecna mnozina stavu  -->
    <STAV>C</STAV>
    <STAV>B</STAV>
    <STAV>A</STAV>
<!--  Vstupni abeceda  -->
    <ABECEDA>a</ABECEDA>
    <ABECEDA>b</ABECEDA>
    <ABECEDA>c</ABECEDA>
<!--  Mnozina pravidel  -->
    <PRAVIDLO>Bc C</PRAVIDLO>
    <PRAVIDLO>Ca-A</PRAVIDLO>
    <PRAVIDLO>Ab->B</PRAVIDLO>
    <PRAVIDLO>Ac C</PRAVIDLO>
<!--  Pocatecni a konecne stavy  -->
    <POCATEK>A</POCATEK>
    <KONEC>C</KONEC>
    </KA>
<!--  Konec dokumentu  -->
```

Všechna jména párových tagů (např: KA, STAV, POCATEK apod.) jsou definované konstanty aplikace a jejich změna ovlivní jak čtení, tak ukládání konfigurace tak, aby nedošlo k nekonzistenci. To je vhodné třeba při převodu aplikace do dalších jazyků, Najednou však může přijímat pouze jednu variantu každého výrazu.

Důležité je, aby množiny stavů a vstupních symbolů (dále též hrany, ohraničené párovým tagem ABECEDA) byla uvedena na začátku. Program totiž provádí automaticky kontrolu a zadá-li se neplatný údaj, ať již v pravidle či v počátečním stavu, aplikace jej neakceptuje. Kdyby abeceda pravidel byla uvedena jako první, nebylo by ještě s čím porovnávat a aplikace by v krajním případě musela upustit od sekvenčního zpracování, které je pro xml typické, a složitě dohledat přítomnost množin stavů a vstupních symbolů na jiných, než očekávaných místech xml dokumentu.

Za povšimnutí stojí kořenový tag KA je zde pro korektnost xml zápisu, který vyžaduje přítomnost kořenového, párového tagu, který ohraničuje celý dokument.

Další zajímavostí je benevolence v zápisu pravidel. Aplikace, či přesněji její xml parser, očekává tento tvar: „*stav1*“„*oddelovac1*“„*hrana*“„*oddelovac2*“„*stav2*“ přitom *stav1* a *stav2* se mohou rovnat, všechny musejí být již definované, *oddelovac1* je prázdný znak a *oddelovac2* může nabývat tří hodnot: mezera, pomlčka a šipka (tj „->“). Všechny tři možnosti demonstruje předchozí příklad.

Přítomnost oddělovačů je nezbytná, protože jinak by nebylo možné přesně určit začátek a konec jednotlivých položek. Tuto chybu demonstruji na následujícím příkladu:

```
    …
<STAV>DD</STAV>                <!--  stav č. 1   -->
<STAV>DDD</STAV>               <!--  stav č. 2   -->
<ABECEDA>Da</ABECEDA>          <!--  hrana č. 1  -->
<ABECEDA>a</ABECEDA>           <!--  hrana č. 2  -->
<PRAVIDLO>DDDa DD</PRAVIDLO>
<!-- !! stav č 1 + hrana č.1 nebo stav č.2 a hrana č.2 !! -->
    …
```

Jak je vidět, nepřítomností oddělovačů by aplikace mohla nesprávně zpracovat definici KA. Jejich přítomnost tomuto zamezuje za cenu menší množiny znaků, nad kterými lze vytvářet řetězce pro jednotlivé množiny. Těmto znakům budu říkat **vyhraněné** a jejich možné, alternativní vyjádření budu nazývat **zástupné**.

Výčet vyhraněných a zástupných znaků:

- prázdný znak („“) doporučuji nahrazovat řetězcem „epsilon“
- čárka („ , “) je automaticky vyjmuta a je vhodné ji nahrazovat středníkem („ ; “)
- pomlčka („-“) lze zadat jejím zdvojením („--“) či sémanticky podobnou dvojtečkou „:“
- šipka („->“) může být nahrazena podobně vypadajícím řetězcem („=>”)
- mezera není vyhrazená, přesto může způsobit nejednoznačnosti, proto ji doporučuji nahrazovat podtržítkem („_“) či alespoň ohlídat, aby zbytek slova za mezerou nebyl samo o sobě také stavem či hranou.

Za povšimnutí stojí také pravidlo, že jeden tag obsahuje pouze jednu hodnotu. Způsob zpracování neukládá tuto podmínku, rozhodl jsem se však tuto variantu upřednostnit z těchto důvodů: odpovídá principu xml a je názorný a pochopitelný na první pohled. V opačném případě (např: povolením této anotace <STAV>A, AA, AAA<STAV>) by uživatel (např při ruční editaci) byl veden do situace, kdy by nevěděl, zda položka obsahuje jeden, složitě zapsaný údaj a nebo tři, oddělené hodnoty. Nutnost studovat manuál v těchto případech považuji za nevyhovující, obzvlášť pokud se jí dá vyhnout lepším návrhem, který jsem upřednostnil.

Jak by aplikace měla reagovat na chybné vstupy??V žádném případě nesmí způsobit pád aplikace, pokud jde o nekorektně pojmenované tagy v xml dokumentu, dovolil jsem si upřednostnit variantu přeskočení takového údaje a pokračování nejbližším korektním. Tedy, z dokumentu se extrahuje maximum korektních údajů a nevyhovující ignoruji.

4.3 Hlavní myšlenky algoritmů zobrazení

Způsobů, kterými se dají pro KA vypočítávat pozice a orientace všech elementů, je celá řada. Během studia problematiky jsem vytvořil několik plánů a datových struktur, které umožní efektivně zobrazit stavový diagram. Přepínačem u diagramu si uživatel může vybrat, který z naimplementovaných algoritmů uspořádá scénu. Ty nejdůležitější postupy představím v následující kapitole.

Efektivitou se zde rozumí podobnost se zobrazením, které by dokázal vytvořit člověk, byť za cenu podstatně delšího času či několika oprav grafu.

Všechny algoritmy mají společné tyto základní úvahy:

- Nejprovázanější stavy (to jsou ty stavy, které se nejčastěji vyskytují v pravidlech) je obtížnější umístit než méně provázané a nejlehčí je pozicování nedosažitelných stavů. Tedy, s počtem relací roste složitost nalezení efektivního umístění, protože krom podmínky nekolizní pozice bere v úvahu i souřadnice dalších stavů, s kterými se má spojit hranou.
- Z předchozí úvahy vyplývá, že pokud začneme umisťovat nejprovázanější stavy dokud nemáme zaplněnou scénu, ušetříme mnoho strojového času, který by jinak padl na detekci kolizí a propočítávání vhodných pozic s ohledem na stavy, s kterými se má spojit. A symetricky: není výrazný rozdíl, pokud umisťujeme málo provázané stavy do téměř prázdné či skoro kompletní scény. Proto pokud se tyto stavy nechají na poslední iterace algoritmů uspořádání a upřednostní se složité (nejprovázanější) stavy, dojde k urychlení činnosti.
- Kolize je problém, mnohdy však snaha o její vyřešení vede k zbytečnému zesložitění výpočtu z těchto důvodů: a) lepší pozice nemusí existovat a jejím hledáním ztrácíme čas či může dojít k zacyklení, b) pokud docílíme optického odlišení kolidujících předmětů (obzvláště hran a popisků), není tato situace závažná. Důsledkem tohoto přístupu je však to, že se stavový diagram stává méně efektivní oproti jejímu ručnímu uspořádání.
- Pokud se dá scéna přibližovat a oddalovat, je irelevantní velikost písma a elementů. Protože zvětšením písma i např. poloměru stavových elementů (typicky kruhů) je rovnocenné k přiblížení diagramu. Z tohoto důvodu je nadbytečné nabízet možnost volby velikostí prvků, důležitější je nalézt vhodný poměr obou hodnot a implementovat transformace. Špatný výběr poměru obou velikostí se projeví tehdy, pokud jsou jména příliš dlouhá a přesahují svá ohraničení v grafu či obráceně jsou-li tvořeny krátkými řetězci a nevyplňují vhodně svá ohraničení.
- V definici KA mohou být pravidla, která tvoří relaci mezi stejnými stavy pomocí víc než jedné hrany (př: $\delta(S0, 1)=S1$;$\delta(S0, A)=S1$). V tom případě nepřidávám fyzicky další šipku do grafu, ale pouze rozšířím popisek u již existující a vstupní řetězce oddělím čárkou.

4.4 Osové zobrazení

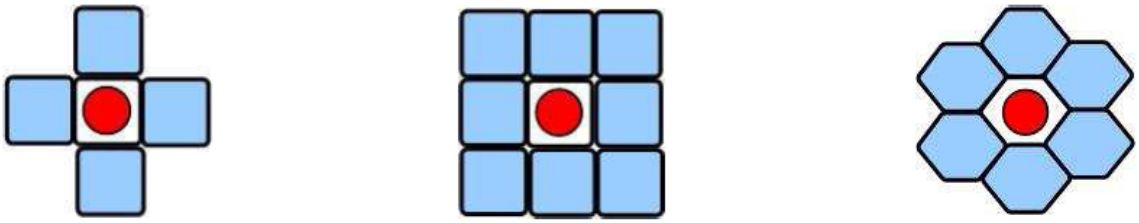
Osa je vždy přímka, která má v každém svém bodě stejnou vzdálenost k objektům, ke kterým tvoří osu. Nejlépe se hledá osa ke dvěma prvkům, ale obdobný princip lze použít i pro větší množství entit.

Jak tedy osové zobrazení funguje?Stavy, které zatím na scéně nemají ty stavy, s kterými se mají spojit relací (též partnerské stavy), se umisťují na nejbližší volné pozice u středu scény. Ty, které již v okamžiku hledání souřadnic mají na scéně své partnery, berou v potaz jejich polohu a vytvoří seznam kandidátních bodů na osách mezi partnery. Osa se nejsnadněji vytváří mezi dvěma objekty, v případě, že je pouze jeden, tvoří kandidátní body kružnici kolem něj. Horší situace nastává, pokud je scéna plná závislostí, potom se naplno projevují nedostatky tohoto způsobu. Představme si nejhorší situaci, kdy se spojujeme se stavy na okraji vygenerované scény. Potom drtivá většina bodů bude zamítnuta z důvodu množství kolizí a efektivní řešení se naskýtá buď v přerovnání scény, inkrementu povoleného množství kolizí či v jiném principu.

Shrnutí: bohužel pro složité scény tento princip neposkytuje lepší časové výsledky než náhodný výběr bodů z těchto důvodů: čím více závislostí, tím větší se generuje seznam bodů a pro každý z nich se vytváří seznam kolizí, což je výpočetně náročná operace. Dále neposkytuje záruku nalezení nekolizních souřadnic při sebedelším běhu algoritmu.

4.5 Celulární zobrazení

Celulární čili buňkové zobrazení si lze představit jako včelí plástve (či celulární automat), kde každá buňka představuje ohraničení území pro reprezentaci právě jednoho stavu a každá hrana této buňky představuje potenciální relaci přechodu.



Obr. 4.5.1 Neumannovské, Moorovo a šestiúhelníkové okolí [10]

Obvykle se hodnota okolí - počet všech buněk, které bezprostředně sousedí s jednotlivým prvkem sítě - definuje na čísle šest podle včelího vzoru, ale může samozřejmě nabývat i jiných hodnot, což jak si ukážeme může být nejen vhodné, ale i potřebné.

Jak tedy celulární zobrazení funguje?Nejprve se projde seznam pravidel a zjistí se maximální počet spojení mezi stavy s tím, že mají-li dva stavy mezi sebou více možných přechodů, počítá se tato relace pouze za jeden. Globální maximum definuje počet okolí v celulární síti. Tento krok předchází umisťování, protože ve chvíli, kdy by aplikace zjistila, že její definované okolí nestačí (počet potřebných přechodů ze stavu by byl vyšší než kolik umožňuje síť), musela by inkrementovat okolí a přepočítat celou scénu od začátku.

Opět začínáme umisťovat nejprovázanější stavy na nejbližší volné pozice od středu a ve chvíli, kdy již máme usazené oba stavy z pravidla, zavedeme mezi mini přechodovou šipku. Součástí tohoto kroku je kontrola překrývání objektů scény, která po zhodnocení stavu a podmínek může zavolat funkci, která přeskládá scénu těmito způsoby:

- rošáda: prohodí dva stavy mezi sebou, pokud by tato změna neměla horší dopady na efektivnost zobrazení oproti původní situaci. To se může stát pokud pohybující se stavy mají již usazené přechodové hrany a nové spojení by vedlo přes další buňky (či jejich hrany) a celkový počet křížení by byl vyšší než před touto operací. Proto je vhodná pokud alespoň jeden stav nemá zatím relaci s okolím.
- Posun či rotace části sítě je jednoduchá, nikoli však častá operace. Dochází k ní pokud již umístěný stav (*A*) má být spojen se vzdáleným a nepropojitelným stavem (*B*), má již na sebe napojené další stavy a ty mají relace pouze mezi sebou a *A*. Současně, *B* může navrhnout výhodnější pozici a autonomní část scény, ta obsahující stav *A*, může být na ní přesunuta , aniž by porušila závislosti v celé scéně.
- Štěp scény je vhodný, pokud do již usazené části grafu (*A*) má být přidán jeden či více stavů a v celém okolí *A* není taková vhodná pozice, která by efektivně propojila stavající a nové stavy (tedy bez kolizí s prvky scény, například: střed grafu má zcela zaplněné své bezprostřední okolí a nový stav se má propojit právě s ním). Za těchto okolností může oblast *A* posunout všechny stavy od hraniční přímky štěpu o vzdálenost jedna k okraji scény a tím vytvořit volný prostor v bezprostředním okolí partnerského stavu, na který se napojí nově přidaný stav. Funkce si sama zjistí další stavy s identickou situací (musela by být volána opětovně i pro jejich umístění), propočítá jejich nároky na prostor a právě o tolik posune část scény *A* od hraniční přímky.

Shrnutí: celulární zobrazení je líbivé pro oko, protože se vyskytuje běžně v přírodě i moderním designu. Poskytuje příjemně přehlednou práci, co se týká pozicování (místo souřadnicového systému pracuje s celulárním okolím), mnohé její uspořádávací funkce pracují rekurzivně, když zjišťují zda by určitá operace přinesla zefektivnění zobrazení či nikoli. A nabízí pro mě osobně nejlákavější možné rozšíření, kdy si, podle definovaného způsobu zápisu, poznamenává charakteristiky poloh a vazeb elementů (v části grafu) na počátku uspořádávání a jejich výsledné, efektivnější pozice po dokončení operace. Tento soubor „řešených postupů“ by byl čten a porovnáván při každé potřebě zpřerovnání scény tím způsobem, že by izoloval klíčové elementy, našel v souboru identickou situaci a podle ní upravil požadovanou část grafu. A samozřejmě při každém úspěšném přerovnání by byl aktualizován o nový údaj. Jedinými, a obávám se že také nejzávažnější nevýhodami proti jakýmkoli dalším zobrazením, je jeho složitost, těžká laditelnost (hlavně kvůli rekurzivnímu přístupu) a programová „robustnost“. Algoritmus jako i další musí mít ustanovený práh (časový či iterační), za kterým nehledá lepší uspořádání, ale spokojí se se stávající, nejméně kolizní variantou řešení.

4.6 Stochastické zobrazení

Je inspirováno metodou Monte Carlo a její schopností díky pravděpodobnosti a množství experimentů docílit kvalitních výsledků, které jsou matematicky velmi obtížně dosažitelné. Počítačové zpracování snadno umožňuje velký počet opakování pokusu (v našem případě nalezení vhodné pozice pro nový stav s ohledem na jeho relace k okolním stavům), důraz musí být kladen hlavně na minimalizaci příkazů a jejich časové náročnosti v každém cyklu, protože s každou interací se celková cena (časová, výpočetní) prudce zvedá.

Jak tedy stochastické zobrazení funguje?Opět, jako u předchozích metod, uspořádáme abecedu stavů podle množství pravidel, v kterých figurují, sestupně a postupně začneme hledat jejich pozice. Vhodně nastavíme několik prahových hodnot, konkrétně jde o celkový počet experimentů pro jeden stav, dále procento z celkových naplánovaných experimentů bez efektu, po kterém se rozšíří hledaná scéna o velikost dosazovaného objektu na každou stranu. A maximální počet experimentů bez zlepšení efektivnosti uspořádání, po kterém se iterace ukončí s dosavadním nejlepším výsledkem. Jediná nepřípustná situace je v tom případě, že by došlo ke kolizím stavů. Jejich překrytí je závažnější než křížení relací či překrytí popisků. Pokud nalezený výsledek i po doběhnutí algoritmu není odpovídající našim představám, potom začnu od pozice nejlepšího dosavadního výsledku systematicky prověřovat jeho nejbližší okolí, nenalézá-li se v něm ještě lepší umístění, které zatím nebylo prověřeno. A samozřejmě nové výsledky konfrontuji s dosavadními.

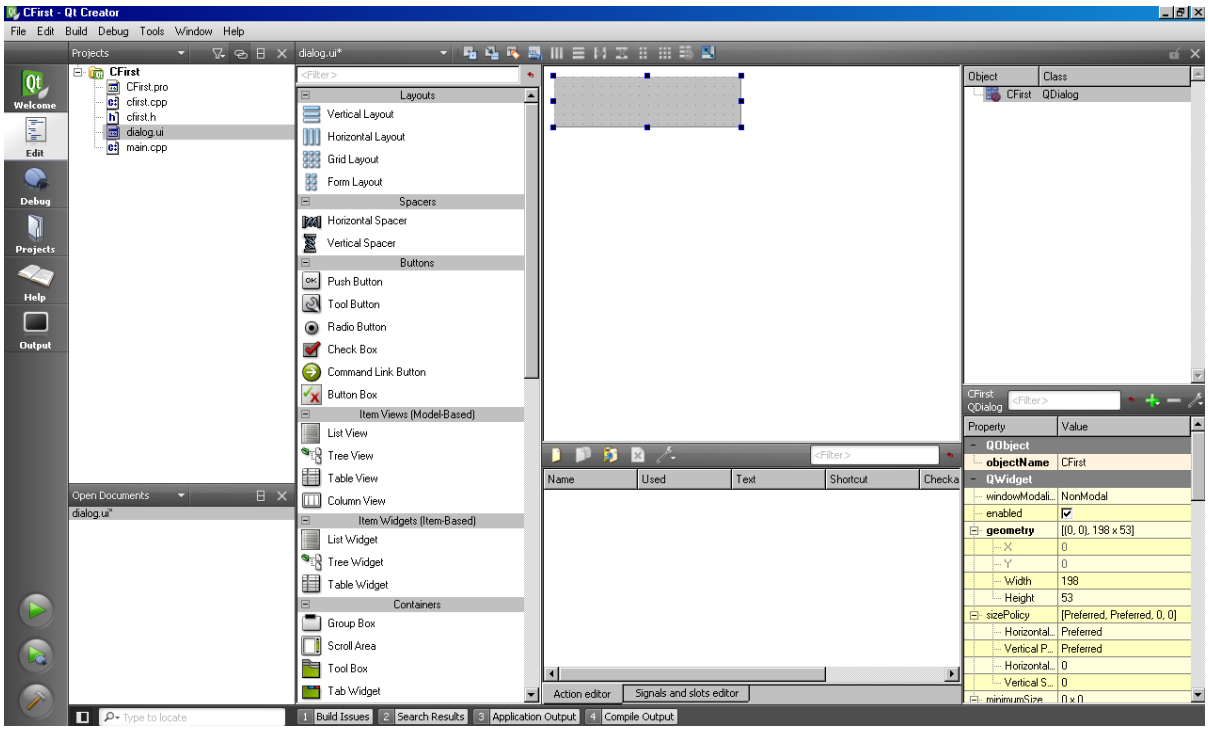
Shrnutí: princip je jednoduchý, ze stochastických základů algoritmu vyplývá, že pro tutéž definici KA bude existovat nespočet různě efektivních zobrazení a dále, že kolizní situace nejsou pro toto zobrazení kritické a spouštějí pouze funkci řešící grafické odlišení objektů. Algoritmus v nejlepším případě nalezne efektivní/ideální rozmístění v nejhorším případě skončí se značným počtem překrývajících se objektů na scéně, které alespoň opticky odliší od sebe.

5 Implementace

Programová část bakalářské práce spočívá ve vytvoření aplikace pro grafickou reprezentací konečného automatu. V této kapitole detailně popíši jednotlivé úvahy, včetně tvorby uživatelského rozhraní, které ovlivnily směr výsledné implementace, zkušenosti a komplikace, které se v průběhu implementace naskytly a řešení, která jsem aplikoval.

5.1 Grafické nástroje Qt a uživatelské rozhraní

Knihovna Qt, jak již bylo řečeno, je určena pro vytváření grafických uživatelských rozhraní. V tomto směru poskytuje celou řadu tříd a možností implementace pro tytéž požadavky. Jako první zmíním integrovaný designer:



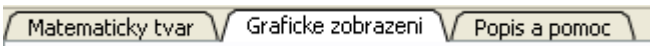
Obr. 5.1.1 Integrovaný designer, [11]

Na první pohled jde o sympatický nástroj, umožňující celou řadu běžných situací řešit nikoli programově, ale „poklikáním“ a úpravou parametrů objektů. Designer umožňuje také definici signálů a slotů (reakcí na uživatelské vstupy jako např.: kliknutí, dvojklik, rolování, stisk klávesy apod.). Rozhodl jsem se ho však využít jen minoritně na zobrazení statutu-zápatí, protože má svoje limity jako jsou neznalost všech nástrojů (jako třeba mnou upřednostněné záložky), neumožnění vkládání vlastních grafických tříd, dále fakt, že úspora kódu je minimální z toho důvodu, že „graficky přetažené“ nástroje lze obvykle naprogramovat na několika málo řádcích. A v neposlední řadě, že možnosti výběru nastavení jednotlivých objektů jsou zaměřeny na nejčastěji používané funkce (př: otevření/zavření/minimalizace okna) a složitější akce (př: spuštění sekvenční zpracování xml souboru) je s ním obtížněji naprogramovatelné, než když se dělá pouze kódově.

Druhým nástrojem je QPainter. Stručně řečeno jde o třídu s množstvím metod a pokud je použita v kódu, vyžaduje přepsání virtuální funkce pro generování pozadí objektu, s kterým je svázána. To v praxi znamená, že námi vytvořená třída si přidá do svých privátních proměnných objekt instance Qpainter a virtuální metodu void paintEvent(QPaintEvent *), která se obvykle volá při vytváření prvku (či zřídkakdy na požádání) a ve svém těle obsahuje celé vykreslení grafického pozadí objektu. Uvádím tento nástroj, protože je velmi praktický, rychlý, příjemně snadno se s ním pracuje a ze začátku jsem v něm umisťoval dokonce i samotný KA. Jeho omezením je špatná práce s velkým množstvím prvků a výslovné nedoporučení od autorů spouštět vykreslování jindy než v konstruktoru.

Pokročilejšími, či spíše nejrozsáhlejšími, nástroji pro umisťování grafických objektů a jejich reprezentaci v Qt jsou QgraphicsScene a QgraphicsView. Dělí si mezi sebe funkce tímto způsobem: třída QgraphicsScene (též scéna) je jako plátno, které uchovává údaje o své velikosti, aktuální transformaci, všech objektech, které se v ní nacházejí, jejich pozicích a nastaveních atd. A QgraphicsView (též náhled) obstarává vše, co se týče zobrazení scény, kterou získá přidělenou ve svém konstruktoru (či později změněnou voláním určité metody), tedy například ořezání, přiblížení, skrývání, průhlednost a umístění v třetím rozměru (tj. osa *z* = umístění v hloubce scény). Prakticky si tedy lze náhled představit jako *widget* (z angličtiny udělátko), který obstarává správný grafický výstup a vstupem je mu určitá scéna. A instance scény zpracovává všechny zobrazované objekty, podle jejich typů a parametrů, a její výstup slouží jako základ pro náhled.

Když jsem řešil téma, co vše za informace ocení uživatel aplikace pro snadné rozhodování a porozumění, dospěl jsem k názoru, že rozhodně nesmím vynechat samotné grafické zobrazení, potom může přijít vhod, když bude mít možnost si přepnout a vidět kompletní výpis všech množin, a hodí se také samostatné zobrazení nápovědy s manuálem k programu a informacemi o aplikaci. Přepínání nesmí být výpočetně náročné a také není vhodné, aby všechny tři vrstvy generovala scéna. To z důvodu, že se do ní nedají umisťovat objekty provázané s určitou akcí (signály a sloty), takže zobrazení by nebylo problém, ale nedala by se ovládat. Řešením je třída QtabWidget (též záložka), s kterou se pracuje tímto způsobem: instance této třídy stanovím jako centrální *widget* a naplním ji třemi dalšími *widgety* s požadovaným obsahem a patřičným popiskem. Záložka se pak zobrazí jako klasická internetová lišta.



Obr. 5.1.2 Ukázka QtabWidgetu přímo v aplikaci

Vstupy a nastavení jsem se rozhodl koncentrovat tak, jak bývají uživatelé zvyklí. Tedy do několika přehledných menu. Hlavní z nich je instancí QmenuBar, je umístěno pod hlavičkou, je dostupné vždy a obsahuje dvě podmenu: první pro načtení a ukládání definice KA do xml plus ukončení aplikace a druhé slouží pro ruční vkládání (či přepis v případě počátečních a koncových stavů) dat do některé z množin KA. Vedlejší menu je umístěno v záložce grafického zobrazení a obsahuje nástroje pro transformaci scény, výběr druhů popisku (textově či barevně odlišené), volbu umístění legendy, výběr použitého algoritmu pro vytvoření scény a tlačítko refresh, které vyčistí paměť a spustí výpočet grafu. Podrobněji funkčnost vysvětluji v manuálu či v záložce Popis a pomoc.

Pro úplnost ještě doplním mnou velmi často používanou třídu QVBoxLayout a QHBoxLayout, která obstarává postupně vertikální a horizontální uspořádání prvků a po svém naplnění se aplikuje na určitý objekt, který řeší samotné umístění v aplikaci. Jak lze vidět například v záložce grafické zobrazení.

5.2 Použité datové struktury

Konečný automat je tvořen 5-ticí, nároky jednotlivých množin na uložení a uchování doplňujících údajů se různí. Dále každý algoritmus zobrazení má odlišné nároky na další datové položky které potřebuje pro svůj výpočet. Z tohoto důvodu jsem se rozhodl vytvořit jeden, společný způsob uložení definice KA (též stálý), který zůstává neměnný do doby přepsání jiným KA. A jeden pro každý způsob zobrazení a z důvodu nekompletnosti zpracování dalších navrhovaných uspořádání (celulární apod.), tu uvedu pouze ten, který jsem dovedl do stabilní verze až po odladění a vytvoření série testových definic KA.

Stálé datové struktury pro uložení definice KA:

- Počáteční stav je tvořen obyčejným řetězcem, přitom akceptuje pouze data již obsažená v množině stavů.
- Koncové stavy jsou uchovány v seznamu řetězců, čemuž v Qt odpovídá datový typ QstringList a opět akceptují pouze prvky již obsažené v množině stavů. V případě přímého vložení skrz menu oddělené čárkami či ve více xml značkách v případě strojového čtení.
- Zbývající množiny pravidel, vstupní abecedy a stavů mají podobné charakteristiky, proto pro ně používám stejný datový typ: ABECEDA. Jde o obousměrně provázaný seznam, který krom ukazatelů uchovává dále řetězec value (textová hodnota položky), ukazatel na grafický objekt gitem (grafická reprezentace údaje), booleovskou hodnotu set (je-li již obsažena ve scéně) a celočíselnou hodnotu vaha (množství relací s okolím).

Přímo při generování scény pomocí výsledného zobrazení jsem narazil na potřebu uchovávat informace typu ukazatel na popisek přechodové hrany (v případě potřeby o jeho rozšíření) či ukazatel přímo na přechodovou hranu s údajem mezi kterými stavy se nachází. Qt nabízí implicitní nástroje k vyřešení jako je a) možnost vytvářet grafické instance jako potomky (nejde o dědění, ale o příslušnost k nadřazenému objektu na scéně) jiných instancí či b) detekcí kolizních objektů zjistit, mezi které byl zasazen. Bohužel (add a) mezi potomky nerozlišuje a pracovat s nimi lze pouze jako se seznamem a tím nevyhovuje všem potřebám. A dále (add b) v případě kolizí jde o náročnou operaci, která nezaručuje jednoznačné a vždy požadované odpovědi. Z těchto důvodů vytvářím v průběhu zpracování pomocnou tabulku prvků scény. Ta má podobu seznamu typu Qlist, který obsahuje *n*-krát tyto čtyři prvky:

- řetězec value, pro indexaci a vyhledávání, obsahuje jméno aktuálního stavu
- seznam ukazatelů stavy, typu ABECEDA, který ukazuje přímo do stálého obousměrného seznamu na konkrétní stavy, s kterými je spojen aktuální.
- seznam ukazatelů hrany, který obsahuje textový popisek jednotlivých přechodových hran
- seznam ukazatelů primka, který je spojen přímo s konkrétní přechodovou hranou.

Díky tomu po naindexování položky seznamu (dle hodnoty value) najdu seznamy všech objektů, s kterými mám vazby včetně jejich typu, orientace, hodnot. Obecně lze říci, že každý ze seznamů vytváří jednorozměrné pole, což s ohledem na samotnou strukturu pomocné tabulky prvků vytváří tří rozměrnou tabulku prvků s indexací podle jména stavu. Toto řešení poskytuje velmi rychlou práci, oproti detekci kolizí, a nabízí přesně ty informace, které pro práci s grafem potřebuji.

5.3 Řadící algoritmus stavů

Jak bylo řečeno a vysvětleno v kap. 4.3., je výhodné generovat scénu systematicky, od objektů s nejvyšším počtem relací s okolím a končit u nedosažitelných stavů. Abychom do téměř kompletně zaplněného grafu přidávali snadno přidatelné položky a nikoli obráceně. Tato operace řazení (obsažená ve funkci void prepare_gitem()) tedy proběhne pouze jednou pro každou definici KA, je však velmi obsáhlá a stojí za detailnější pohled. Prvním pod-úkolem je potřeba ohodnotit jednotlivé položky množiny stavů podle počtu pravidel, ve kterých figurují. To zajistí stejný algoritmus, který prověřuje korektnost nově přidaného pravidla. Během jeho činnosti je pravidlo rozparsováno na dva stavy a jednu přechodovou hranu, ony stavy jsou dohledány ve stálém seznamu stavů a jejich proměnná vaha je inkrementována.

Po dokončení tohoto kroku máme ve stálém seznamu prvků správně vyplněné váhy a podle jejich hodnot setřídíme samotný obousměrný seznam stavů. Při výběru algoritmu řazení jsem bral v potaz především zaručenou časovou náročnost, nároky na paměť a v poslední řadě datový typ, nad kterými pracuje. Mým požadavkům nejlépe vyhovoval heap sort (řazení haldou), probíraný na škole, který má zaručenou časovou náročnost O(N log N) a řadí data na původním místě. Jedinou komplikací byla provázanost heap sortu a datové struktury haldy, která má velmi efektivní operace výběru nejvyššího prvku. Ten jsem nahradil vlastním postupem. V něm procházím obousměrný seznam od počátku po zarážku (místo, za kterým je již seřazeno) a ukládám si pozice nikoli jedno nejvyššího, ale všech položek se stejným, nejvyšším ohodnocením. Tato úprava vychází z pozorování, že množina hodnot (v testovaných definicích KA), určených k seřazení, má často nikoli jednu nejvyšší hodnotu, ale více stejně ohodnocených položek. Jejich současné seřazení ušetří jak cykly, tak zmenší prohledávaný prostor.

6 Závěr

Zadání mé bakalářské práce bylo splněno, laťky, které jsem si osobně stanovil, však nikoli. Konkrétně mluvím o implementaci pokročilejších zobrazovacích algoritmů, které jsem popsal detailně v návrhu. Některé z nich prošly značnou cestu vývoje, ale z důvodu nestability nebyly vloženy do výsledné aplikace. I přesto jejich teoretické zpracování má svoji hodnotu a může sloužit jako námět pro rozšíření této či podobné aplikace.

Způsob zápisu abstraktních výpočetních modelů bere v potaz hned několik hledisek jako jsou přehlednost zápisu, rozšiřitelnost, výběr známé syntaxe a definici pravidel pro zápis apod., v tomto směru si troufám tvrdit, že volba xml a konkrétních pravidel je dostatečně univerzální, aby sloužila jako podklad pro další aplikace a modely s vyšší vyjadřovací silou.

Přínosem pro uživatele, pravděpodobně majoritně z řad studentů FIT, je aplikace, která umožňuje vytvořit stavový diagram konkrétního konečného automatu. Mám osobní zkušenost s projekty, kdy mi právě tento krok významně pomohl porozumět a přehledněji vyvíjet projekty. S možností grafického zobrazení pomocí aplikace odpadá nutnost řešit tento úkol na papíře a výsledek aplikace může sloužit současně jako podklad pro samotný návrh, část dokumentace projektu a svým užitím zkvalitnit výsledky programátorské práce. S významem přidané hodnoty grafické reprezentace pro studenty se však moje úvaha střetává s náplní cvičení a písemných zkoušek v některých předmětech na bakalářském studiu FITu, kdy se právě vyžaduje nejen znalost, ale i praktické zkušenosti s ruční tvorbou konečných automatů. Proto může tato aplikace sloužit maximálně jako doplněk stávajících možností.

V závěrečné etapě své práce jsem objevil bakalářskou práci bc. Ondřeje Syrového s názvem: Vizualizace práce konečných automatů, zásobníkových automatů a Turingova stroje, z roku 2009. Po seznámení se stavem jeho práce jsem ocenil myšlenku generovat graf po krocích a tím umožnit uživateli názorně vidět všechny mezikroky, které vedou k výsledku. A fakt, že jsme nezávisle na sobě došli k podobným výsledkům v oblastech použitých datových struktur a výběru údajů vhodných pro vytvoření stavového diagramu. S ohledem na obě zadání zle soudit, že tato tématika je jednak velmi žádaná a vhodná jako studijní pomůcka pro abstraktní výpočetní modely a dále, že systematickým vývojem či spojením výsledků našeho snažení, by uživatelé měli velmi praktickou pomůcku, která by jim pomáhala porozumět zajímavé tématice teoretické informatiky, jejíž největším úskalím, si troufám tvrdit, je právě absence vhodných, vizuálních, výukových pomůcek pro snadnější porozumění.

Literatura

[1] Meduna ,A., Lukáš, R. Formální jazyky a překladače. Opora předmětu IFJ. FIT-VUT Brno. Verze 1 2006

[2] M. Češka, T. Vojnar, A. Smrčka Teoretická informatika TIN, Opora předmětu TIN. FIT-VUTBR Brno 1.12.2008

[3] VolkovBot a Oashi : Chomského hierarchie - Wikipedie, otevřená encyklopedie [online] [cit. 1.5.2010]. Dostupné na URL: <http://cs.wikipedia.org/wiki/Chomského_hierarchie>

[4] Hubert Dostál: Teorie konečných automatů, regulárních gramatik, jazyků a výrazů [online] [cit. 1.5.2010]. Dostupné na URL: <http://iris.uhk.cz/tein/teorie/rozsPrechodova.html>

[5] Hubert Dostál: Teorie konečných automatů, regulárních gramatik, jazyků a výrazů [online] [cit. 1.5.2010]. Dostupné na URL: <http://iris.uhk.cz/tein/teorie/stavovyStrom.html>

[6] Nokia: Reference knihovny Qt [online] [cit. 1.5.2010]. Dostupné na URL: <http://doc.trolltech.com/>

[7] Martin Chudoba: Qt framework – základy práce (1) linux software [online] [cit. 1.3.2010]. Dostupné na URL: <http://www.linuxsoft.cz/article.php?id_article=1620>

[8] Josef Vochyán: Transformace stavového automatu Moorova typu na Mealyho typ a naopak [online] [cit. 1.3.2010]. Dostupné na URL: <http://www.urel.feec.vutbr.cz/~kolouch/pld/1_prednasky/kapitola06_01.html#kap06_01>

[9] Tým vývojářů knihovny Swing: Reference knihovny Swing [online] [cit 1.3.2010]. Dostupné na URL: <http://www.swingwiki.org>

[10] Martina Husáková: Celulární automaty [online] [cit 1.5.2010]. Dostupné na URL: <http://lide.uhk.cz/fim/ucitel/fshusam2/lekarnicky/zt3/zt3_dokumenty/CelularniAutomaty.pdf>

[11] Martin Chudoba: Qt framework – základy práce (1) linux software [online] [cit. 1.3.2010]. Dostupné na URL: <http://www.linuxsoft.cz/img/qtframework/12.png>

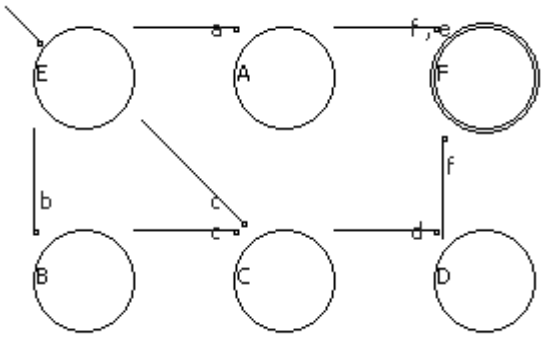
[12] Oashi, TobeBot, Timy, Shadel a Zvonicek : Turingův stroj - Wikipedie, otevřená encyklopedie [online] [cit. 1.5.2010]. Dostupné na URL: <http://cs.wikipedia.org/wiki/Turingův_stroj>

[13] Meduna A.: Automata and Languages: Theory and Applications. Springer London 2000

Seznam příloh

Příloha A

Vzorový příklad demonstruje stavový diagram (Obr. A.1) určitého konečného automatu, k němu odpovídající textovou podobu (Obr. A.2) a strojový zápis:



Obr: A.1 Ukázka stavového diagramu vytvořeného aplikací

Matematicka podoba konecneho automatu

Konecna mnozina stavu konecneho automatu: **F, E, A, B, C, D**

Vstupni abeceda/hrany konecneho automatu: **a, b, c, d, e, f**

Pravidlo/a konecneho automatu: **Eb B, Bc C, Cd D, Df F, Ec C, Ea A, Af F, Ae F**

Pocatecni stav konecneho automatu: **E**

Mnozina koncovych stavu konecneho automatu: **F**

Obr. A.2 Výpis množin-též matematický tvar-vytvořený aplikací

```
<!-- Ondrej Polcer 4BIT FIT-VUTBR 2010     -->
    <KA>
<!-- Definice konecneho automatu     -->
<!-- Konecna mnozina stavu -->
        <STAV>F</STAV>
        <STAV>E</STAV>
        <STAV>A</STAV>
        <STAV>B</STAV>
        <STAV>C</STAV>
        <STAV>D</STAV>
<!-- Vstupni abeceda -->
        <ABECEDA>a</ABECEDA>
        <ABECEDA>b</ABECEDA>
        <ABECEDA>c</ABECEDA>
        <ABECEDA>d</ABECEDA>
        <ABECEDA>e</ABECEDA>
```

```
        <ABECEDA>f</ABECEDA>

<!-- Mnozina pravidel-->
        <PRAVIDLO>Eb B</PRAVIDLO>
        <PRAVIDLO>Bc C</PRAVIDLO>
        <PRAVIDLO>Cd D</PRAVIDLO>
        <PRAVIDLO>Df F</PRAVIDLO>
        <PRAVIDLO>Ec C</PRAVIDLO>
        <PRAVIDLO>Ea A</PRAVIDLO>
        <PRAVIDLO>Af F</PRAVIDLO>
        <PRAVIDLO>Ae F</PRAVIDLO>

<!-- Pocatecni a konecne stavy  -->
        <POCATEK>E</POCATEK>
        <KONEC>F</KONEC>

    </KA>
<!-- Konec dokumentu -->
```