Česká zemědělská univerzita v Praze
Provozně ekonomická fakulta
Katedra informačního inženýrství

Diplomová práce

Návrh simulátoru deterministických konečných automatů

Bc. Leoš Novák

Vedoucí práce: Ing. David Buchtela

Studijní program: Systémové inženýrství a informatika

Obor: Informatika

8. dubna 2011

Poděkování

Na tomto místě bych rád poděkoval svému vedoucímu diplomové práce, Ing. Davidu Buchtelovi za cenné rady při návrhu aplikace, dále pak mé přítelkyni Evě Lobovské, za velkou podporu a trpělivost.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 8. 4. 2011 .....

Abstract

Leoš Novák, Design of Deterministic Finite Automata Simulator.

The target of this thesis is to design and implement a program, which will allow to construct and simulate finite automata for given input patterns. The application is especially intended for students of informatics as instrumental tool for teaching automata.

Abstrakt

Leoš Novák, Návrh simulátoru deterministických konečných automatů.

Cílem práce je návrh a implementace programu, který bude umožňovat konstrukci a simulaci konečných automatů pro zadané vstupní vzorky. Výsledná aplikace je především určena pro studenty informatických oborů jako pomocný nástroj při výuce automatů.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Automaty jsou využívány v mnoha oborech, ve výpočetní technice, zejména při návrhu počítačů a jejich programového vybavení. Problematika automatů patří mezi nejdéle studovanou a nejpodrobněji zdokumentovanou část oboru matematická informatika.

Výukou automatů se zabývají předměty vyučované na mnoha školách, které nabízejí informatické obory. Právě pro studenty těchto předmětů, ale i pro ostatní zájemce o tuto disciplínu, je určen tento projekt, který jsem nazval *ASim*, což představuje zkratku z anglického sousloví *Automata Simulator* (simulátor automatů).

Text ve druhé kapitole seznamuje s cílem této práce a následně definuje důležité pojmy týkající se problematiky automatů. V třetí kapitole jsou představeny exitující aplikace, umožňující návrh a simulaci automatů. Kapitola čtvrtá se zaměřuje na návrh simulátoru, což představuje katalog požadavků, volba programovacího jazyka, volba knihovny pro tvorbu grafického uživatelského rozhraní, implementace a testování. V závěru se hodnotí splnění cílů, shrnují se získané poznatky a představují se možnosti rozšíření.

Přílohy obsahují seznam použitých zkratek, uživatelskou příručku a XML schéma použité struktury, využívané k uchování dat. Uživatelská příručka seznamuje uživatele s instalací, návrhem automatů a simulací.

Kapitola 2

Cíl práce a metodika

2.1 Cíl práce

Cílem práce je vytvořit aplikaci, která bude umožňovat snadný návrh automatu s následnou možností simulace průchodu automatem pro zadaný vzorek. Preferuje se využití takových technologií, aby bylo možné program spustit na více operačních systémech.

Návrh automatu bude sestávat z definování stavů a přechodů. U stavů se bude moci nadefinovat jejich název a příznak, určující počáteční nebo koncový stav. Přechod bude určen počátečním a cílovým stavem přechodu, přechodovým symbolem a případně i zásobníkovou operací. Takto zadaný automat bude možno uložit jako rastrový nebo vektorový obrázek a do pevně definované XML struktury, pro případ využití navržených automatů externí aplikací.

Simulaci by mělo jít spouštět automaticky, krokovat a resetovat. Částečná podpora nedeterministických konečných automatů bude alespoň na úrovni definování více počátečních stavů, ε-přechodů a manuálního výběru jednoho z nedeterministických přechodů při běhu simulace.

2.2 Teoretický základ

2.2.1 Abeceda

Je-li Σ libovolná konečná množina (abeceda), pak Σ^* označuje množinu všech konečných neprázdných posloupností utvořených z prvků množiny Σ, ε označuje prázdnou

posloupnost a definuje $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$. Posloupnost $a_1 \ldots a_n \in \Sigma^*$ budeme zapisovat $a_1 \ldots a_n$. Každou takovou posloupnost nazýváme řetězem nebo slovem v abecedě Σ, ε nazýváme prázdným slovem. O Σ^* mluvíme jako o množině všech řetězců nad abecedou Σ, *o* Σ^+ jako o množině neprázdných řetězců nad Σ. Jestliže $u = a_1 \ldots a_n$ a $v = b_1 \ldots b_m \in \Sigma^*$, pak řetěz $uv = a_1 \ldots a_n b_1 \ldots b_m$ nazveme zřetězením slov u a v. Speciálně $eu = u\varepsilon = u$. Symbol u^n bude označovat n-násobné zřetězení slova u, tzn. $u^1 = u, u^2 = uu, u^{i+1} = u^i u$. Délku řetězu u budeme značit $|u|$, tj. $|a_1 \ldots a_n| = n$ $(a_i \in \Sigma)$ a $|e| = 0$. Je-li Σ konečná abeceda a $L \subseteq \Sigma^*$, pak L nazýváme jazykem nad abecedou Σ. [4]

Příklady abeced:

- $\{A, B, C, \ldots, Z\}, \{\alpha, \beta, \gamma, \ldots, \omega\}, \{0, 1\}$,
- $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /\}$,
- $\{begin, end, if, then, else, while, repeat, until, for, read, write, \ldots$ a další klíčová slova nějakého programovacího jazyka$\}$.

2.2.2 Jazyk

„Soubor slov a metod jak slova kombinovat, užitý pro dorozumívání v dané komunitě."

Je-li dána abeceda Σ, potom libovolná podmnožina množiny Σ^* všech slov nad touto abecedou se nazývá formální jazyk, zkráceně pouze jazyk, nad abecedou Σ. Neboli formální jazyk nad danou abecedou Σ je jakákoliv podmnožina množiny všech slov nad touto abecedou, formálně $L \subseteq \Sigma^*$. [3]

Existují dva navzájem odlišné typy konečné reprezentace jazyků:

- **reprezentace pomocí automatů** - umožňuje rozhodnout o libovolném slovu zda patří či nepatří do daného jazyka,
- **regulární výrazy** - specifikuje vlastnosti slov, která do jazyka patří.

Přechodovou funkci $\delta : Q \times \Sigma \to Q$ konečného automatu $A = (Q, \Sigma, \delta, q_0, F)$ rozšíříme na zobecněnou přechodovou funkci $\delta^* : Q \times \Sigma^* \to Q$ takto:

- $\delta^*(q, \varepsilon) = q$ pro každé $q \in Q$,

- $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$ pro každé $q \in Q, w \in \Sigma^*, a \in \Sigma^*$.)

Jazykem rozpoznávaným konečným automatem A pak nazveme jazyk

$L(A) = \{w; w \in \Sigma^* \ \& \ \delta^*(q_0, w) \in F\}$

Říkáme, že řetěz $w \in \Sigma^*$ je přijímán automatem A, právě když $w \in L(A)$.

Existuje-li konečný automat A takový, že jakyk $L = L(A)$, říkáme, že jazyk L je rozpoznatelný konečným automatem. Symbolem F budeme označovat množinu všech jazyků rozpoznatelných konečným automatem. [4]

Příklad vymezení jazyka:

- Množina všech slov zadané délky. Třeba byte (abeceda {0, 1}, délka 8 - jazyk má 256 slov),
- Množina slov nad abecedou {0,1}, kde počet jedniček je prvočíslo.
- Množina všech slov nad libovolnou abecedou, která jsou shodná se slovy vytvořenými opačným pořadím znaků ve slově.

2.2.3 Nerodova věta

Jazyk můžeme prohlásit za regulární, pokud sestrojíme regulární gramatiku nebo konečný (nedeterministický) automat, který jej rozpoznává. Toto tvrzení lze ovšem jen ztěží využít v případě, že chceme dokázat, že daný jazyk není regulární. Nerodova věta představuje jednu z nutných podmínek pro to, aby byl daný jazyk regulární.

Nerodova věta pracuje s pojmem pravá kongruence slov nad danou abecedou. Pokud máme nad množinou A^* všech slov nad danou abecedou A definovanou relaci ekvivalence $\sim$($\sim$ je tedy relace reflexivní, symetrická a tranzitivní), řekneme, že tato relace je pravá kongruence tehdy, pokud pro každá dvě ekvivalentní slova $u, v \in A^*$ platí, že pokud za ně zprava připíšeme stejný řetězec (též prvek A^*), budou obě slova opět ekvivalentní. Pokud například nad množinou $\{0, 1\}^*$ definujeme ekvivalenci vztahem $u \sim v$ právě když parita počtu nul a jedniček ve slovech u a v je stejná, potom ekvivalence $\sim$ je zřejmě i pravou kongruencí.

Každá pravá kongruence, protože je relací ekvivalence, dělí množinu A^* na třídy rozkladu. Pokud je těchto tříd konečný počet budeme říkat, že kongruence má konečný index. Výše uvedená kongruence dělí množinu A^* na dvě třídy rozkladu, v jedné budou

slova se sudým počtem jedniček, v druhé slova s lichým počtem jedniček. Kongruence je tedy konečného indexu.

Nerodova věta zní takto: Jazyk L nad abecedou A je regulární (generovatelný gramatikou typu 3, rozpoznatelný konečným automatem) právě tehdy, když na množině A^* existuje pravá kongruence konečného indexu $\sim$ taková, že L je sjednocením některých tříd rozkladu A^* podle relace $\sim$. [3]

2.2.4 Regulární jazyky

Důležité vlastnosti množiny R všech regulárních jazyků nad danou konečnou abecedou se nazývají *uzávěrové vlastnosti regulárních jazyků*. [3]

- Je-li L regulární jazyk nad abecedou Σ, je zřejmě i jeho dolněk $L' = \Sigma \div L$, do kterého patří ta a jen ta slova, která do L nepatří, regulární jazyk.
- Jsou-li L_1 a L_2 dva regulární jazyky, je i jejich sjednocení $L_1 \cup L_2$ tedy množina slov nad danou společnou abecedou, z nichž každé patří buď do L_1 nebo do L_2 regulativní jazyk.
- Jsou-li L_1 a L_2 dva regulativní jazyky nad společnou abecedou, je i jejich průnik $L_1 \cap L_2$ regulární jazyk.
- Zřetězení jazyků L_1 a L_2 nad společnou abecedou je jazyk, který obsahuje právě ta slova, která vzniknou zřetězením slov obou jazyků v daném pořadí.
  $L_1.L_2 = \{u.v : u \in L_1 \wedge v \in L_2\}$. Platí, že zřetězením dvou regulárních jazyků získáme opět regulární jazyk.
- Opakováním zřetězení se definuje mocnina slova a jazyka vzhledem ke zřetězení. Přesně se taková definice formuluje matematickou indukcí takto:
  $L^1 = L; L^{n+1} = L.L^n$ pro všechna $n = 1, 2, ...$ Místo $L^1 = L$ se někdy začíná pouze s požadavkem $L^0 = \{\varepsilon\}$ kde ε je prázdné slovo.

Množina všech regulárních jazyků je uzavřená vůči operacím doplňku, sjednocení, průniku, zřetězení, řetězové mocniny i řetězového uzávěru - iterace. Množina všech regulárních jazyků nad konečnou abecedou je průnikem všech množin jazyků, které mají tyto vlastnosti [3]:

- obsahující všechny jazyky, které mají jen konečný počet slov,
- jsou uzavřené vůči doplňku sjednocení, průniku, zřetězení a iretaci.

2.2.5 Konečný automat

Koněčný automat je nejjednodušší z formálních modelů počítače. Popisuje stroj, který má konečný počet stavů, v nichž se může nacházet. Konečný automat zpracovává posloupnost vstupních symbolů, vybíraných z dané konečné množiny, které říkáme abeceda automatu.

Při své práci automat začne pracovat ve svém počátečním stavu, čte znaky vstupního slova postupně zleva doprava a mění své stavy tak, že při čtení znaku x ve stavu q přejde do stavu $f(q, x)$[3].

Formálním popisem konečného automatu je uspořádaná pětice $\alpha = (Q, \Sigma, \delta, q_o, F)$, kde

- Q je konečná neprázdná množina stavů,
- Σ je konečná neprázdná množina vstupních symbolů (vstupní abeceda),
- δ je zobrazení $Q \times \Sigma \to Q$ (přechodová funkce),
- $q_o \in Q$ je počáteční stav,
- $F \subseteq Q$ je množina konečných stavů.

Pokud se po přečtení posledního vstupního symbolu bude automat nacházet ve stavu $q \in F$ říkáme, že automat vstupní slovo (posloupnost vstupních symbolů) přijímá, v případě, že po přečtení vstupního slova se nacházíme ve stavu $q \notin F$ je vstupní slovo automatem odmítnuto. Množina všech slov příjímaných konečným automatem je označována jako **automatem přijímaný jazyk**. [4]

2.2.6 Mealyho automat

K popisu konečného automatu lze přidat ještě jeden prvek, a to sice výstupní abecedu. V každém taktu konečný automat může (ale nemusí) umístit na výstup jeden ze symbolů výstupní abecedy. Výstupní abeceda může být libovolná neprázdná konečná množina O. Součástí výstupní abecedy je vždy i prázdný symbol ε, reprezentující situaci, kdy automat žádný výstup nedává. Přechodová funkce δ pak bude určovat nejen nový výstupní stav, ale i symbol výstupní abecedy. Jde tedy o zobrazení z množiny $Q \times I$ do množiny $Q \times O$. Konečný automat s takto upravenou definicí nazýváme konečný automat s výstupem nebo také Mealyho automat. Výstup Mealyho automatu záleží tedy na zpracovávaném vstupním symbolu i na stavu, ve kterém automat je. [3]

2.2.7 Moorův automat

U Moorova automatu závisí výstupní symbol pouze na aktuálním vnitřním stavu stroje. Nezáleží na právě čteném prvku vstupního slova, Pro formální popis Moorova automatu je tedy potřeba žádat zvlášť zadání přechodové funkce, což bude zobrazení $Q \times I \to Q$, a zvlášť výstupní funkce, což je zobrazení $Q \to O$. [3]

2.2.8 Deterministický automat

Deterministický konečný automat, též označován jako *DFA*, se vyznačuje tím, že v každém vnitřním stavu má automat při čtení vstupního znaku pouze jednu možnost, jak ve své činnosti pokračovat, tzn. jeho chování je jednoznačné. Může mít pouze jeden počáteční stav a nesmí obsahovat ε-přechody. [4]

2.2.9 Nedeterministický automat

Pojem nedeterministického konečného automatu je zobecněním pojmu konečného automatu. Také tyto automaty lze analogicky reprezentovat tabulkou či stavovým diagramem. Na rozdíl od deterministického případu nemusí být u nedeterministických automatů stavem a vstupním symbolem jednoznačně určen stav, do kterého automat přejde. Je pouze vymezena množina stavů, do kterých může přejít. Druhým rozdílným rysem je obecně víceprvková množina počátečních stavů. Pro pohodlnější provádění některých operací se u nedeterministických automatů připouštějí i tzv. ε-přechody. Při nich automat může změnit svůj stav, aniž by přitom přečetl symbol vstupního slova. Toto rozšíření není nezbytné, stejného chování automatu lze dosáhnout i tehdy, když ε-přehody neumožníme. [4]

Nedeterministickým konečným automatem budeme nazývat pětici $\alpha = (Q, \Sigma, \delta, I, F)$, kde

- Q je neprázdná konečná množina stavů,
- Σ je neprázdná konečná množiny vstupních symbolů,
- $\delta : Q \times \Sigma \to P(Q)$ je přechodová funkce ($P(Q)$ označuje množinu všech podmnožin množiny Q),
- $I \subseteq Q$ je množina počátečních stavů,
- $F \subseteq Q$ je množina koncových stavů.

2.2.10 Kritéria pro návrh konečného automatu

Dva konečné automaty α, β se nazývají ekvivalentní, jestliže rozpoznají tentýž jazyk, tzn. $L(\alpha) = L(\beta)$.

Ke každému konečnému automatu existuje nekonečně mnoho dalších s ním ekvivalentních automatů. To je vidět např. z toho, že ke každému automatu lze přidat libovolný počet zbytečných, z počátečního stavu nedosažitelných, stavů.

Problém najít k danému jazyku konečný automat, který by jej rozpoznával, tedy buď vůbec nemá řešení, anebo má nekonečně mnoho různých řešení. Při výběru jednoho z těchto řešení se zpravidla explicitně řídíme určitým kritériem. Většinou převládá jedno ze dvou základních kritérií.

Prvním z nich je požadavek, aby navržený automat byl co možná nejmenší, tj. aby měl co možná nejmenší počet stavů. Čím menší je totiž počet stavů automatu, tím menší paměť stačí pro jeho technickou realizaci.

Druhým kritériem je přehlednost a jasnost návrhu umožňující snadno ověřit jeho správnost.

Obě kritéria jsou často v rozporu. Zmenšení automatu obvykle dosáhneme za cenu, že se jeho struktura stane nepřehlednější. [4]

2.2.11 Redukce konečného automatu

Redukce konečného automatu připouští existenci stavů, do nichž se nelze dostat z počátečního stavu na základě žádné vstupní posloupnosti.

Stav $q \in Q$ konečného automatu $\alpha = (Q, \Sigma, \delta, q_0, F)$ se nazývá dosažitelný, jestliže existuje slovo $w \in \Sigma^*$ takové, že $\delta(q_0, w) = q$. Stavy, které nejsou dosažitelné, se nazývají nedosažitelné.

Nechť $\alpha = (Q, \Sigma, \delta, q_0, F)$ je konečný automat a nechť $P \subseteq Q$ je množina dosažitelných stavů automatu α. Potom automat $\beta = (P, \Sigma, \delta_p, q_0, F \cap P)$, kde δ_p je parcializaci přechodové funkce na $P \times \Sigma$, je automat ekvivalentní s α a neobsahuje nedosažitelné stavy. [4]

2.2.12 Gramatika

Generativní gramatika, zkráceně gramatika, je neformálně řečeno systém, jak pomocí daných přepisovacích pravidel vytvořit všechna slova daného jazyka z nějakého počá-

tečného symbolu. Gramatiky spolu s automaty jsou nejčastější způsoby reprezentace jazyků. [3]

Generativní gramatika je uspořádaná čtveřice $G = (\Sigma_N, \Sigma_T, P, S)$, kde

- Σ_N je neprázdná konečná množina neterminálních symbolů zvaná též abeceda proměnných,
- Σ_T je neprázdná konečná množina terminálních symbolů. Abeceda generovaného jazyka.
- Množiny Σ_N a Σ_T musí být disjunktní ($\Sigma_N \cap \Sigma_T = \oslash$). Označme dále $\Sigma_N \cup \Sigma_N = \Sigma$.
- P je neprázdná konečná množina tak zvaných produkčních, neboli přepisovaných pravidel. Tato množina se také nazývá *přepisovací systém*. Každé produkční pravidlo přiřazuje nějakému řetězci $\alpha \in \Sigma^*$ terminálních a neterminálních symbolů.
- $S \in \Sigma_N$ je vybraný počáteční symbol, který je vždy proměnnou (neterminálním symbolem).

Je-li $\alpha \to \beta$ produkční pravidlo, říkáme, že řetězec $y\alpha\delta$ lze přímo prepsat na řetězec $y\beta\delta$ (nebo, že $y\beta\delta$ lze přímo odvodit z $y\alpha\delta$), a zapisujeme to $y\alpha\delta \to y\beta\delta$. Přímé přepsání znamená tedy náhradu nějakého podřetězce, který je shodný s pravou stranou nějakého přepisovacího pravidla z množiny P pravou stranou tohoto přepisovacího pravidla.

Nechť $\alpha_1, \alpha_2, ..., \alpha_n$ jsou řetězce a nechť $\alpha_1 \to \alpha_2 \to \alpha_3, ..., \alpha_{n-1} \to \alpha_n$. V tomto případě říkáme, že α_1, lze přepsat na α_n (nebo, že α_n lze odvodit z α_1) a značíme to $\alpha_1 \to \alpha_n$. Přepsání tedy znamená postupné získání nového řetězce následným užitím konečného počtu (obecně různých) přepisovacích pravidel.

Stojí za povšimnutí, že množina všech slov, které lze odvodit z počátečního symbolu, je uzavřená vůči operaci přímého přepsání a je „nejmenší“ množinou řetězců nad danou abecedou, která obsahuje počáteční symbol a která je vůči operaci přímého přepsání uzavřená. Slovo „nejmenší“ je třeba chápat tak, že každá množina, která tyto vlastnosti má, ji již obsahuje jako podmnožinu. Jde tedy o tak zvaný tranzitní uzávěr vůči operaci přímého přepisování.

Jazyk generovaný gramatikou G je množina všech řetězců, na které lze počáteční symbol přepsat, a které se skládají pouze z terminálních symbolů (abecedy generovaného jazyka, již bez užití proměnných).

Pořadí výběru přepisovaných pravidel není nijak stanoveno. Je zde plná volnost. Užitím různých přepisovacích pravidel v různém pořadí můžeme získat různá slova jazyka. I stejné slovo lze postupným užíváním přepisovacích pravidel získat obvykle více způsoby. [3]

- $\Sigma_N = \{S\},\ \Sigma_T = \{0,1\},\ P = \{S \rightarrow 0S1,\ S \rightarrow 01\}$.
  Generovaný jazyk je $L(G) = \{0^n1^n\}$, kde $n = 1,2,\ ...,$ (0^n značí n nul bezprostředně za sebou, 1^n jedniček bezprostředně za sebou).
- $\Sigma_N = \{S, J, D, T\},\ \Sigma_T = \{0,1\},\ P = \{S \rightarrow 0S, S \rightarrow 1J, J \rightarrow 0J, J \rightarrow 1D, D \rightarrow 0D, D \rightarrow 1, D \rightarrow 0D, D \rightarrow 1, D \rightarrow 1T, T \rightarrow 1T, T \rightarrow 0, T \rightarrow 0T\}$, počáteční symbol S. Generovaný jazyk obsahuje ta a pouze ta slova, která obsahují tři nebo více symbolů 1.
- $V_N = \{S\}, V_r = \{a, b, c, ..., z\}, P = \{S \rightarrow aSa, S \rightarrow bSb, ..., S \rightarrow zSz, S \rightarrow \varepsilon\}$ generuje jazyk všech slov v anglické abecedě, která mají sudý počet znaků a jsou „souměrná podle svého středu“, tedy je lze „číst pozpátku“.

2.2.13 Chomského hierarchie gramatik

Obecná definice gramatiky poskytuje značnou volnost. Důležitou roli hrají gramatiky a jimi generované jazyky, kde jsou na přepisovací pravidla kladena doplňující omezení. Tato omezení vedou na tak zvanou Chomského hierarchii gramatik a formálních jazyků.[3]

2.2.13.1 Gramatika typu 0

Do tohoto typu Noam Chomsky řadí všechny gramatiky - jimi generované jazyky označuje jako jazyky typu 0. Množinu všech takových jazyků označme L_0. Protože ne každý jazyk lze generovat nějakou gramatikou, existují jazyky, které nejsou typu nula, natož pak některého z „vyšších“ typů. [3]

2.2.13.2 Gramatika typu 1

Kontextová gramatika je gramatika, u které každé produkční pravidlo musí být typu $\alpha X \beta \rightarrow \alpha\gamma\beta$, kde α a β jsou řetězce z Σ^*, X je nějaký neterminální symbol a $\gamma \in \Sigma^+$ je neprázdný řetězec. Jedinou vyjímkou může být pouze existence jediného pravidla

$S \to \varepsilon$, kde S je počáteční symbol gramatiky. V tom případě však nemůže S být na pravé straně žádného pravidla. Přepisováním nelze tedy řetězce „zkracovat“. - Gramatiky typu 1 generují jazyky typu 1 - kontextové jazyky. Množinu takových jazyků označme L_1. [3]

2.2.13.3 Gramatika typu 2

Bezkontextová gramatika je gramatika, u které každé produkční pravidlo je tvaru $X \to \gamma$, kde X je neterminální symbol a γ je řetězec Σ^* (složený z terminálních a neterminálních symbolů). Gramatiky typu 2 generují jazyky typu 2 - bezkontextové jazyky. Množinu všech takových jazyků označme L_2. Užití pravidla $\alpha X \beta \to \gamma$ a pravidla $X \to \gamma$ má samozřejmě týž efekt. Rozdíl je pouze v tom, že v druhém případě lze pravidlo užít vždy, v prvém pouze když X předchází α a následuje po něm β, tedy pouze když je X obklopeno „daným“ kontextem. [3]

2.2.13.4 Gramatika typu 3

Regulární gramatika je gramatika, u které každé produkční pravidlo je buď tvaru $A \to aB$, nebo tvaru $A \to a$, kde A a B jsou proměnné a a je terminální symbol. Gramatiky typu 3 generují jazyky typu 3 - regulární jazyky. Množinu všech takových jazyků označme L_3. Lze ukázat, že regulární gramatiku lze vymezit i tak, že se omezíme pouze na přepisovací pravidla typů $A \to aB, A \to B$ a $A \to \varepsilon$, kde velká písmena značí neterminální a malá terminální symboly. Tato pravidla mohou být v některých případech pohodlnější. Právě tak je možné místo pravidla $A \to aB$ uvažovat pravidlo $A \to Ba$. Vede opět na množinu regulárních gramatik, pouze slova generuje „pozpádku“. Nelze však nabídnout obě pravidla $A \to aB$ i $A \to Ba$ současně. V tom případě bychom dostali již širší třídu gramatik. Gramatika s přepisovacím systémem $\{A \to aB, A \to Ba, A \to a\}$ nazýváme lineární gramatika a příslušným jazykům lineární jazyk. Jde o obecnější množiny gramatik, než jsou regulární gramatiky, ale méně obecné než bezkontextové gramatiky. Někdy se jim proto říká i gramatiky a jazyky typu 2,5. [3]

2.2.14 Bezkontextové jazyky

Bezkontextový jazyk je formální jazyk, který je akceptovaný nějakým zásobníkovým automatem. Bezkontextové jazyky mohou být vygenerovány bezkontextovými gramatikami. [9]

2.2.15 Zásobníkový automat

Zásobníkový automat je rozšířením pojmu konečný automat. Zásobníkový automat obsahuje jednu potenciálně neomezenou paměť - zásobník, do kterého lze ukládat poznámky či mezivýsledky. Ze zásobníku je ke čtení či k zápisu přístupný vždy jen vrchní symbol, ostatní jsou nepřístupné. Zásobník pracuje podle disciplíny LIFO.

Zásobníkový automat pracuje tak, že v každém okamžiku zkoumá současně symbol vstupního slova a vrchol zásobníku. Na základě tří údajů, v jakém stavu je, čteného symbolu vstupního slova a vrcholu zásobníku může automat měnit svůj stav a uložit znak či několik znaků na vrchol zásobníku. Pokud by byl zásobník prázdný automat svoji práci ukončí. Po přečtení posledního znaku slova rozhodne, zda slovo přijímá. To může provést dvěma způsoby:

- Podobně jako konečný automat podle toho, zda byl stav, ve kterém se po přečtení celého slova zastavil, označen jako koncový stav.
- Přijmout slovo v případě, že po přečtení jeho posledního znaku zůstal zásobník prázdný, zůstal-li neprázdný, slovo odmítnout.

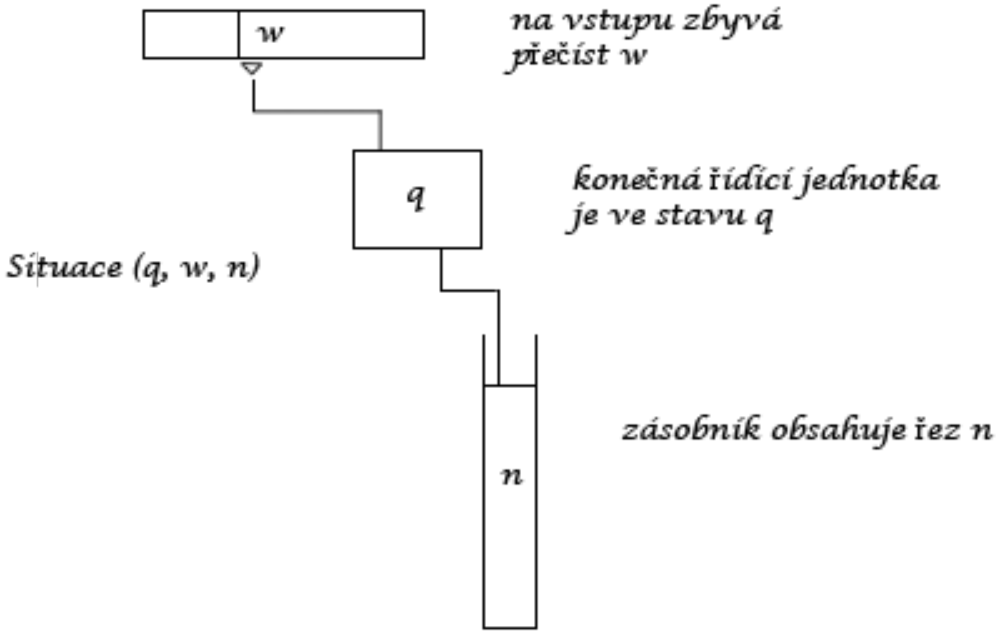
I takováto funkčně velmi omezená paměť podstatně zvýší možnosti stroje. Představme si úlohu, kdy máme na vstupu posloupnost znaků 0 a 1. Po libovolně dlouhé posloupnosti následuje znak c a po něm opět další nuly a jedničky až do konce vstupu. Úkolem je poznat, zda je druhá část posloupnosti zrcadlovým obrazem části první, či nikoliv. Pomocí konečného automatu je tato úloha zjevně neřešitelná. Konečný automat totiž nemá prostor, ve kterém by si mohl zapamatovat, „jak první část posloupnosti vypadala“. Formálně lze neexistenci konečného automatu, který by rozpoznával výše popsaný jazyk, odvodit z *Nerodovy věty*.

Zásobníkový automat je na tom lépe, může si první část posloupnosti ukládat do zásobníku. V okamžiku, kdy přečte znak c, přejde do jiného vnitřního stavu a změní své chování. Začne číst znaky ze vstupu a zároveň odebírat znaky ze zásobníku. Kontroluje jejich shodu. Pokud se znaky shodují pokračuje v kontrole, až do současného vyprázdnění zásobníku a konce vstupu. V takovém případě může konstatovat symetričnost posloupnosti. Pokud během porovnání dojde k neshodě znaku na vrchu zásobníku a znaku na vstupu, nebo pokud se zásobník vyprázdní dříve, než skončí vstup, nebo pokud naopak vstup skončí dříve, než se vyprázdní zásobník, lze konstatovat nesymetričnost posloupnosti. [3]

Zásobníkovým automatem nazýváme sedmici $\alpha = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$, kde Q je neprázdná konečná množina stavů, Σ je neprázdná konečná vstupní abeceda, Γ je neprázdná konečná zásobníková abeceda, $q_o \in Q$ je počáteční stav, $Z_o \in \Gamma$ je počáteční zásobníkový symbol (na začátku výpočtu bude zásobník obsahovat pouze symbol z_o), $F \subseteq Q$ je množina koncových stavů, δ je zobrazení $Q \times (\Sigma \cup \{\varepsilon\}) \times \delta$ do množiny všech konečných podmnožin $Q \times \Sigma^*$. Z definice je patrné, že zásobníkový automat je obecně nedeterministický. [4]

Situace zásobníkového automatu:

$\alpha = (Q, \Sigma, \Gamma, \delta, q_o, Z_o, F)$ nazveme trojici (q, w, n), kde $q \in Q, w \in \Sigma^*, n \in \Gamma^*$ (viz. obr. 2.1)



Obrázek 2.1: Situace zásobníkového automatu

2.2.16 Deterministický zásobníkový automat

Formálně lze deterministický zásobkový automat definovat jako uspořádanou sedmici $(Q, \Sigma, \Gamma, \delta, q_o, z_o, F)$. Významy symbolů Q, I, q_o a F jsou stejné jako u konečného automatu (množina vnitřních stavů, vstupní abeceda, počáteční stav a množina koncových stavů). Γ je konečná množina symbolů, které lze zapisovat do zásobníku. Nazýváme ji zásobníková abeceda. z_o je prvkem Γ^+, jde tedy o konečnou a neprázdnou posloupnost prvků zásobníkové abecedy, která je umístěna v zásobníku na počátku práce.

Přechodová funkce δ je zobrazení množiny $Q \times \Sigma \times \Gamma$ do množiny $Q \times \Gamma^*$. V tom případě není potřeba definovat množinu koncových stavů P a budeme říkat, že automat přijal slovo, pokud je po přečtení jeho posledního znaku zároveň prázdný zásobník. [3]

2.2.17 Nedeterministický zásobníkový automat

I u zásobníkového automatu je možné podobné zobecnění na nedeterministický zásobníkový automat jako u konečného automatu. Přechodová funkce δ nebude pouhým zobrazením do množiny $Qx\Gamma^*$, ale zobrazením do množiny všech neprázdných podmnožin této množiny. Pro danou situaci tedy nebude mít automat jednoznačně dáno, jak se zachovat, ale bude moci vybrat z několika variant další práce.

Nedeterministický zásobníkový automat přijímá dané slovo koncovým stavem, když existuje alespoň jeden přístupný postup jeho práce, při kterém po přečtení celého slova skončí v některém z koncových stavů. Přijímá slovo prázdným zásobníkem, když existuje alespoň jeden přístupný postup jeho práce, při kterém po přečtení celého slova bude zasobník prázdný.

U nedeterministických zásobníkových automatů jsou již oba způsoby rozpoznávání jazyků ekvivalentní. Pro libovolný jazyk L platí následující tvrzení: existuje nějaký nedeterministický zásobníkový automat, který přijímá slova jazyk L a pouze slova jazyka L tehdy a pouze tehdy, jestliže existuje nějaký nedeterministický zásobníkový automat, který přijímá jeho a jen jeho slova pomocí prázdného zásobníku. V případě rozpoznání prázdným zásobníkem lze dokonce vystačit se zásobníkovými automaty, které mají pouze jediný stav. [3]

2.2.18 Turingův stroj

Ani zásobníkový automat nepopisuje po formální strance dobře všechny možnosti, které by počítače mohly teoreticky využívat. Paměť zásobníkového počítače je konečná a zásobníkový přístup možnosti výpočtu značně limituje.

Turingův stroj je konečný automat doplněný o paměť v podobě pásky. Na pásku se zapisují symboly tzv. páskové abecedy. Páska je na obě strany potencionálně nekonečná, i když samozřejmě popsaná a používaná část pásky je po konečném počtu kroků vždy jen konečná. Turingův stroj pásku čte a zapisuje na ni pomocí čtecí a zapisovací hlavy, která se může po pásce libovolně pohybovat v každém taktu vždy o jedno políčko buď v pravo, nebo vlevo, případně zůstane na místě. Pro zjednodušení situace lze předpokládat, že páska s nějakým obsahem existuje i před započtením práce Turingova stroje. Obsah pásky lze pokládat za vstupní údaje, které má stroj k dispozici před svou prací, tedy za slovo, o jehož přijetí či nepřijetí stroj rozhoduje. Podobně páska existuje i po skončení práce stroje a obsahuje jeho výstup. V tomto smyslu není při definici Turingova stroje narozdíl od zásobníkového automatu třeba rozlišovat vstupní a výstupní abecedu. Obě jsou shodné a nazývají se pásková abeceda. Formálně tedy Turingův stroj uspořádaná pětice (Q, P, δ, q_o, F), kde:

- Q je libovolná konečná množina vnitřních stavů,
- P je konečná množina symbolů, které lze zapisovat na pásku, takzvaná pásková abeceda. Pásková abeceda obsahuje prázdný symbol - mezeru, kterou označíme $\sqcup$. Vždy je jen konečná část pásky popsaná jinými symboly než $\sqcup$,
- $q_o \in Q$ je počáteční stav stroje,
- F je neprázdná podmnožina Q, množina koncových stavů, tato množina je sjednocením dvou disjunktních podmnožin, přijímající (odpověď ANO) a odmítající (odpověď NE),
- δ je zobrazení z množiny $(Q \div F) \times P)$ do množiny $Q \times P \times \{l, n, p\}$. Každé situaci, kdy se stroj nachází v nekoncovém vnitřním stavu a na pásce pod čtecí hlavou je čten symbol páskové abecedy, je tedy přiřazen:
  - Nový vnitřní stav stroje.

- Nový symbol páskové abecedy zapsaný hlavou stroje. Ten může být pochopitelně totožný se čteným symbolem, obsah pásky se tedy může, ale nemusí změnit.
- Pokyn pro posun hlavy stroje: l znamená posun o jedno políčko vlevo, p posun o jedno políčko vpravo, n znamená ponechat hlavu stroje na místě.

Stav práce stroje je určen jeho konfigurací (α, q, β), kde α je část pásky před hlavou $(a1, a_2, ..., a_{i-1}), q$ je stav řídicí jednotky stroje a β je část pásky od hlavy vpravo $a_1, a_{i+1}, a_{i+2}, ..., a_n)$. [3]

Kapitola 3

Přehled současných metod a technologií

3.1 Existující řešení

3.1.1 Automaton Simulator

Program pro návrh a simulaci turingových strojů, deterministických a nedeterministických konečných automatů. Autor na stránkách uvádí i možnost simulace deterministických zásobníkových automatů, ale na možnost definovat zásobníkové symboly či operace jsem nenarazil. Celkově je program založen na jednoduchosti a účelnosti, při návrhu máme k dispozici pouze tři tlačítka: *vytvořit stav, vytvořit přechod a napsat popisek*. Vkládání stavů je intuitivní, nastavení stavu jako počátečního nebo konečného a smazání stavu je prováděno přes kontextovou nabídku, která se zobrazí po kliknutí pravým tlačítkem myši na prvek. Přes kontextovou nabídku lze také přiřazovat přechodové symboly nebo mazat přechody. Navrhnuté automaty lze ukládat do souborů s vlastní strukturou. Program dále umožňuje tisk a simulaci.

První verze tohoto simulátoru vyšla v roce 2001 a od té doby doznala jen několika drobných změn, jako je oprava chyb a řešení problémů s kompatibilitou, jedná se tedy s největší pravděpodobností o finální verzi. Naprogramováno v programovacím jazyce Java, pro spuštění je potřeba mít nainstalovaný Java Virtual Machine, ve verzi alespoň 1.3. Program je uvolněn pod GNU General Public License v.2 a lze ho stáhnout z oficiálních stránek projektu (velikost 357 kB). K dispozici je velmi stručná dokumentace, která je součástí programu.

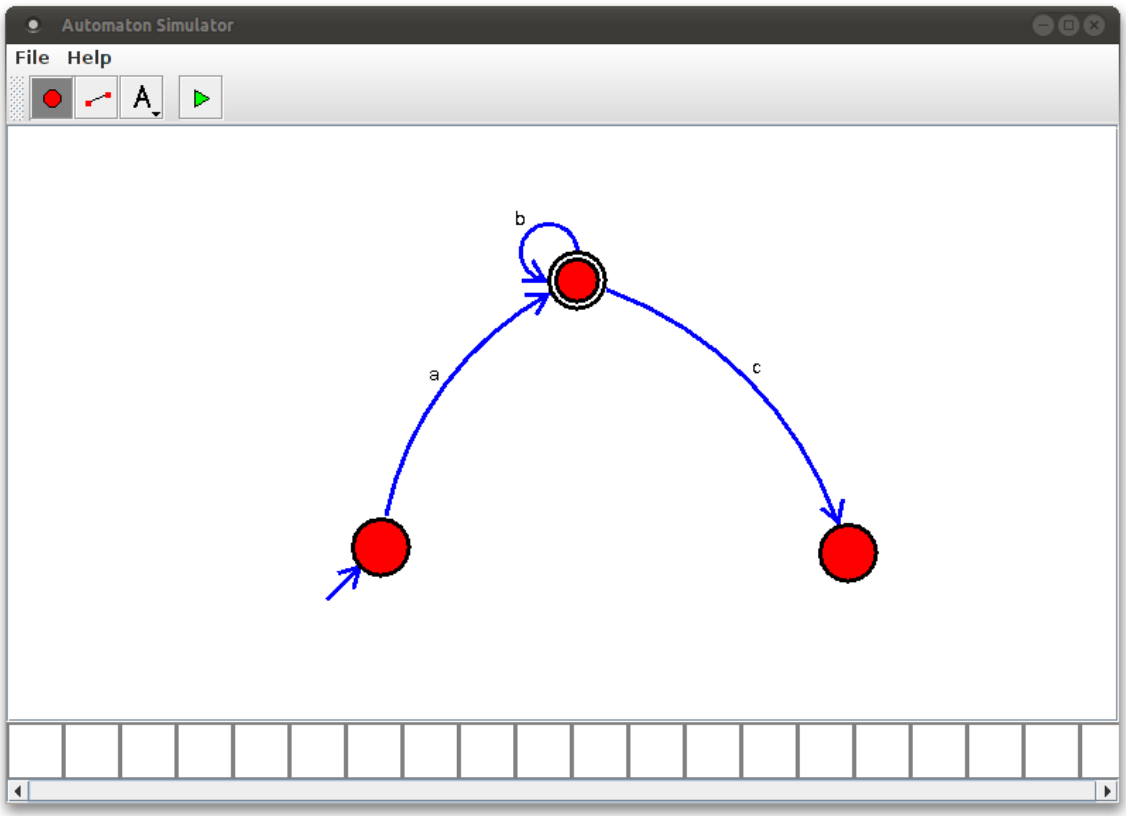
WWW: http://ozark.hendrix.edu/~burch/proj/autosim

3.1.1.1 Výhody

- Přenositelnost.
- Možnost ukládání a tisku.

3.1.1.2 Nevýhody

- Špatná kvalita zpracování.
- Nemožnost zadat k jednomu přechodu více přechodových symbolů.



Obrázek 3.1: AutomataSimulator

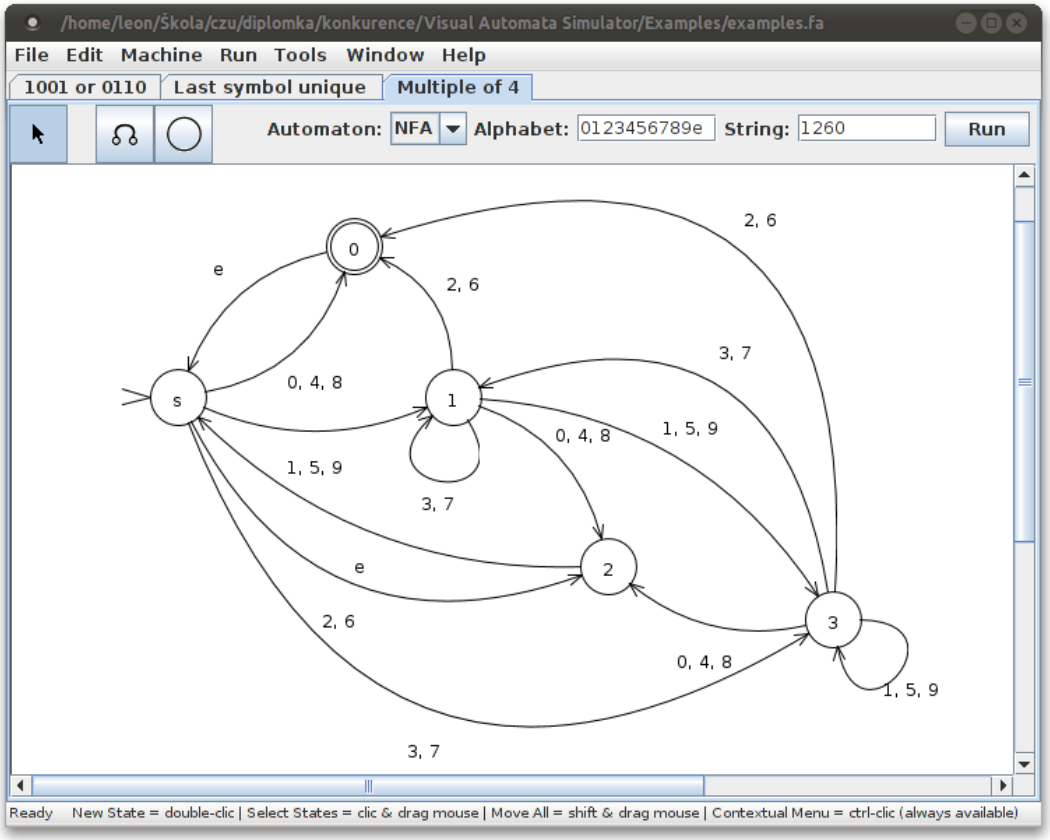
3.1.2 Visual Automata Simulator

Aplikace umožňuje návrh, simulaci a transformaci deterministických a nedeterministických konečných automatů, dále pak návrh a simulaci turingových strojů. Výsledkem

návrhu je graficky dobře zpracovaný automat, nad kterým je možno provádět automatickou a postupnou simulaci průchodu pro zadaný vzorek. Mezi další funkce patří převod nedeterministických automatů na deterministické a export do rastrového a vektorového formátu. Data jsou uchovávána v XML struktuře.

Naprogramováno v Javě v letech 2004-2006 Jeanem Bovetem, k dispozici včetně zdrojových kódů, uvolněných pod BSD licencí. Součástí projektu jsou tři ukázkové příklady a stručná dokumentace. Na webu jsou k dispozici video návody.

WWW: http://www.cs.usfca.edu/~jbovet/vas.html



Obrázek 3.2: Visual Automata Simulator

3.1.2.1 Výhody

- Přenositelnost.
- Záložky - projekt může obsahovat více automatů.

- Kvalitní zpracování a přehlednost.
- Převod nedeterministických automatů na deterministické.

3.1.2.2 Nevýhody

- Pomalost aplikace.
- Stručná dokumentace.
- Chybějící podpora zásobníkových automatů.

3.1.3 jFAST

jFAST je jednoduchý, snadno použitelný nástroj pro návrh, úpravu a simulaci konečných automatů. Mezi podporované automaty patří DFA, NFA, PDA a TM. Pro zadaný vstup simuluje průchod libovolným konečným automatem. Program umožňuje také tisk a uložení automatu jako rastrového obrázku. Data uchovává v XML struktuře.

Nejnovější verze 1.3 byla vydána v roce 2006. Naprogramováno v Javě Thimoty Whitem, uvolněno pod GNU/GPL licencí. K dispozici není dokumentace, pouze krátký tutorial doplněný dvěma příklady na webové prezentaci.

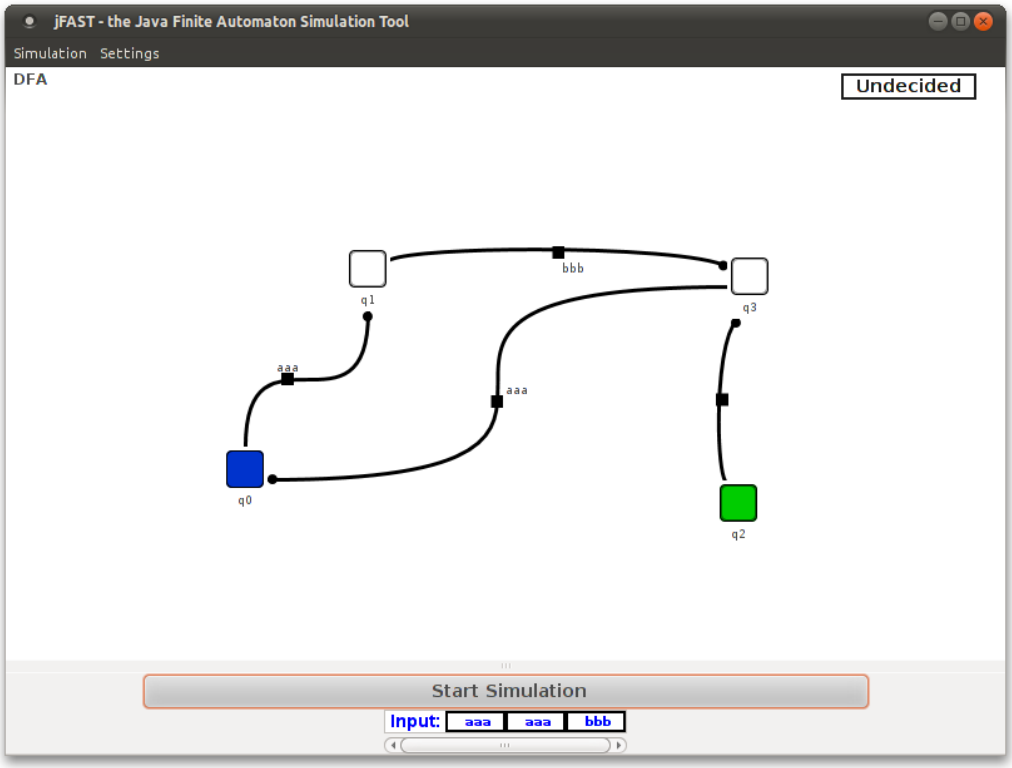
WWW: http://www46.homepage.villanova.edu/timothy.m.white/

3.1.3.1 Výhody

- Přenositelnost.
- Podpora zásobníkových automatů.

3.1.3.2 Nevýhody

- Pomalost aplikace.



Obrázek 3.3: jFAST

3.1.4 JFLAP

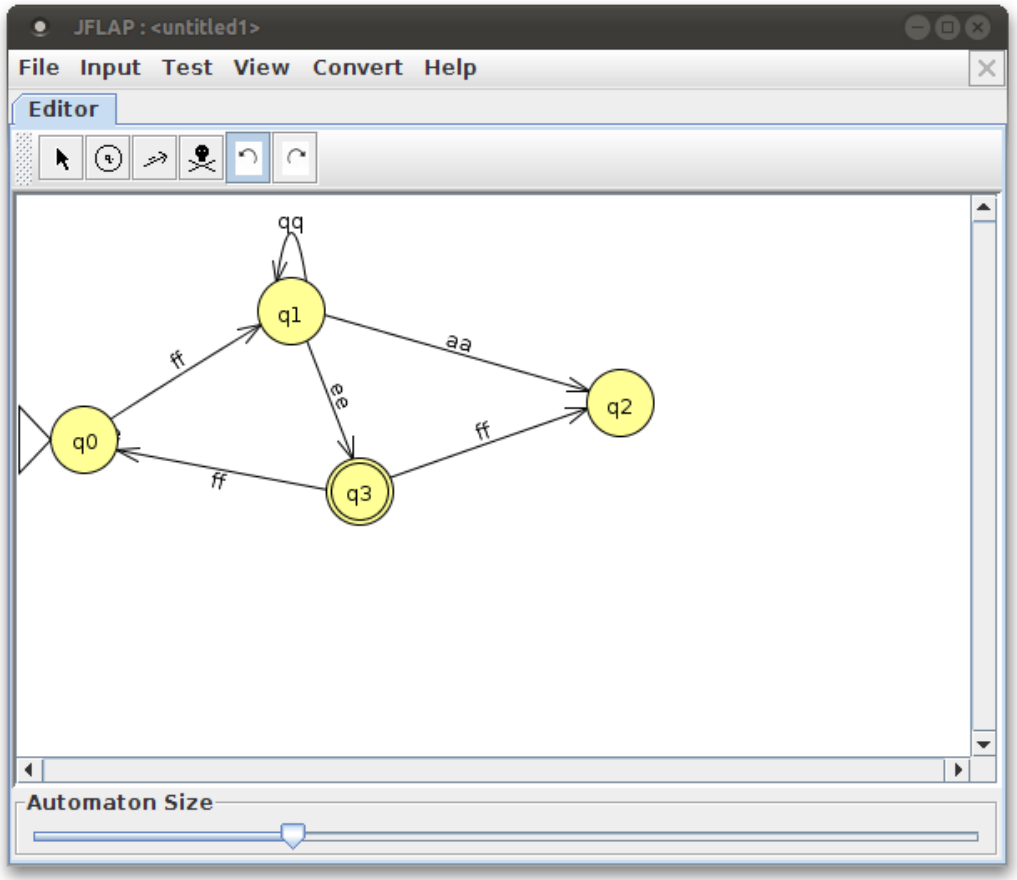
JFLAP je rozsáhlý grafický nástroj pro práci nebo výuku základních znalostí formálních jazyků a teorie automatů. Mezi podporované funkce patří:

- regulární jazyky
  - návrh DFA, NFA, regulárních gramatik a regulárních výrazů
  - převody
    - NFA → DFA → DFA minimální
    - NFA ↔ regulární výraz
    - NFA ↔ regulární gramatika
- bezkontextové jazyky
  - návrh PDA a bezkontextových gramatik
  - transformace

  - PDA → CFG
  - CFG → PDA (LL analyzátor)
  - CFG → PDA (SLR analyzátor)
  - CFG → CNF
- jedno i více páskové turingovy stroje
- L-systémy

K dispozici je velké množství tutoriálů i několik ukázkových příkladů na webu. Vyvíjeno v Javě, aktuální verze 7.0 zdarma ke stažení po vyplnění dotazníku. K dispozici jsou i zdrojové kódy opatřené vlastní licencí.

WWW: http://www.jflap.org



Obrázek 3.4: JFLAP

3.1.4.1 Výhody

- Přenositelnost.
- Velký rozsah funkcí.
- Profesionální zpracování.

3.1.4.2 Nevýhody

- Nefunkční tlačítka zpět a dopředu.

3.1.5 Shrnutí

S vyjímkou programu *JFLAP* jsou nástroje vyvinuty studenty výpočetní techniky na vysokých školách jako bakalářské nebo diplomové práce a po několik let nejsou upravovány nebo rozšiřovány, jedná se pravděpodobně o jejich finální verze. Všechny projekty jsou naprogramovány v programovací jazyku Java, k uchování dat využívají XML strukturu (*s vyjímkou projektu AutomataSimulator*) a jsou uvolňovány včetně zdrojových kódů. Společnou negativní vlastností všech projektů je pomalý start a občasná dlouhá odezva při návrhu automatu.

Kapitola 4

Vlastní návrh simulátoru

4.1 Katalog požadavků

4.1.1 Funkční požadavky

- Program bude umožňovat zadání automatů pomocí návrháře.
- Program bude simulovat průchod automatem pro libovolný vzorek.
- Program bude ukládát automaty do souboru.
- Program bude načítat automaty ze souboru.
- Program půjde spouštět s parametrem.
- Bude použito vektorové vykreslování.

4.1.2 Nefunkční požadavky

- Grafické rozhraní programu bude jednoduché a intuitivní.
- Program bude stabilní.
- Program bude přenositelný na co největší počet operačních systémů.
- Program nebude sloužit k distribuci nelegálního software.

4.2 Volba programovacího jazyka

Při výběru programovacího jazyka se nabízí mnoho možností, proto jsem si hned na počátku zvolil kritéria:

- typ (z důvodu rychlosti preferuji jazyky kompilované);
- rozšířenost programovacího jazyka;
- úroveň mých znalostí,

díky kterým jsem výběr zůžil na C, C++ [1], C# [2] a Javu.

4.2.1 C

Nízkoúrovňový přístup jazyka C mi příjde spíše vhodný pro vývoj jádra operačních systémů, vestavěných systémů, ovladačů nebo aplikací zaměřených na vysokou výpočetní rychlost, než pro potřeby grafické aplikace, a tak i s ohledem na absenci objektově orientovaného programování tento jazyk zamítám.

4.2.2 C#

Vysokoúrovňový objektově orientovaný jazyk vyvinutý firmou Microsoft, který je velmi často programátory využíván především pro aplikace běžící pod MS Windows. Běh aplikací na MAC OS a OS Linux je sice z velké části podporován díky projektu Mono, ale jako základ multiplatformní aplikace se příliš nehodí a tak jej i přes nesporné kvality .NET frameworku také nevyužiji.

4.2.3 Java

Zřejmě nejpopulárnější programovací jazyk současnosti, zavrhuji už jen z toho důvodu, že bych chtěl aplikaci odlišit od všech testovaných aplikací, napsaných právě v Javě, u kterých jsem se potýkal s pomalejšími starty a odezvou zejména při návrhu automatů.

4.2.4 C++

Programy napsané v objektově orientovaném C++ se vyznačují vysokou rychlostí startu aplikací i provádění algoritmů. Při samotném vývoji aplikací poskytuje C++ mnoho možností a volnosti, které však při nedostatečných zkušenostech programátora mohou snadno vést k neoptimálnímu a těžko udržovanému kódu. I přes tyto hrozby se mi však jeví C++ jako nejlepší volba.

Pro programování přenositelných aplikací s grafickým uživatelským rozhraní v C++ nejčastěji využívají knihovny wxWidgets a Qt.

4.3 Knihovny pro tvorbu grafického uživatelského rozhraní

4.3.1 wxWidgets

Projekt byl původně distribuován pod názvem wxWindows, na nátlak Microsoftu byl v roce 2004 přejmenován na wxWidgets, což představuje zkratku „Windows and X widgets“. Jedná se o knihovnu základních elementů pro tvorbu grafického uživatelského rozhraní (GUI). wxWidgets umožňuje zkompilovat a spustit program na několika počítačových platformách s minimálními nebo žádnými změnami kódu. Mezi podporované platformy patří Windows, Macintosh, Linux/Unix (X11, Motif, a GTK+), OpenVMS a OS/2. [11]

4.3.1.1 Knihovny

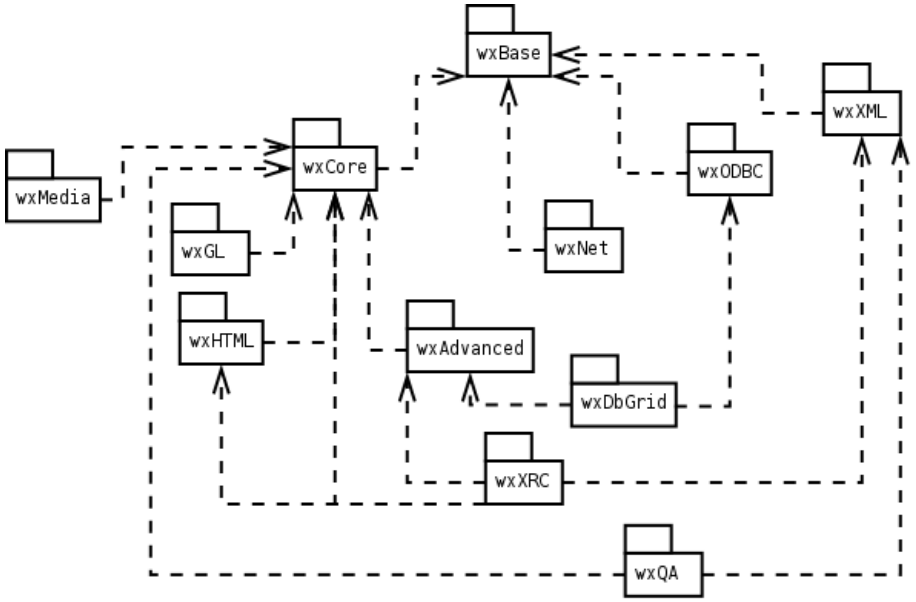
Toolkit wxWidgets je k dispozici buď jako jedna velká knihovna - „monolitická knihovna“, nebo jako několik menších knihoven. Závislosti jednotlivými knihovnami znázorňuje obrázek 4.1.

wxAui

Obsahuje nástroje pro pokročilé uživatelské rozhraní.

wxBase

Každá wxWidgets aplikace musí obsahovat odkaz na knihovnu `wxBase`, je základem každého objektu *wxWidgets*. `wxBase` může být použit k vytvoření konzolové nebo grafické aplikace, nevyžaduje žádné grafické knihovny nebo spuštění *X Window System* na Unixu.



Obrázek 4.1: Struktura knihoven wxWidgets.

wxNet

Obsahuje třídy pro přístup k síti:

- wxSocket, wxSocketClient, wxSocketServer,
- wxSocketOutputStream, wxSocketInputStream,
- wxTCPServer, wxTCPClient, wxTCPConnection,
- wxURL,
- wxInternetFSHandler.

wxRichText

Představuje základní funkce pro práci s formátovanými texty.

wxXML

Tato knihovna obsahuje jednoduché třídy pro zpracování XML dokumentů. V budoucnu se plánuje nahradit novější knihovou, zpětná kompatibilita nebude zajištěna, a tak se její použití nedoporučuje.

wxCore

Základní třídy grafického uživatelského rozhraní. Všechny aplikace s GUI musí obsahovat odkaz na tuto knihovnu.

wxAdvanced

Obsahuje pokročilé nebo málo používané GUI třídy, jako je kalendář, joystick, spořič obrazovky, průvodce akcí nebo algoritmy pro dynamické rozmístění prvků v aplikaci.

wxMedia

V současné době obsahuje pouze `wxMediaCtrl`, v budoucnu bude rozšířena.

wxGL

Tato knihovna obsahuje třídu `wxGLCanvas` pro integraci knihovny *OpenGL* s *wxWidgets*. Na rozdíl od všech ostatních knihoven není tato knihovna součástí monolitické knihovny.

wxHTML

Obsahuje jednoduchý HTML vykreslovač a další HTML vykreslovací třídy.

wxODBC

Knihovna zajišťující propojení aplikace s databázovým systémem.

wxQA

Obsahující extra třídy pro zajištění kvality aplikací. V současné době obsahuje pouze ladící `wxDebugReport` a jeho související třídy. V budoucnu se plánuje další rozšíření.

wxDbGrid

Obsahuje třídu `wxDbGridTableBase`, která spojuje prvek `wxGrid` s `wxDbTable`. Využívá se pro jednoduchý výpis obsahu databáze do grafické aplikace.

wxXRC

Tato knihovna obsahuje třídu `wxXmlResource`, která poskytuje aplikaci přístup ke zdrojovým souborům, načítaným z XRC souboru.

4.3.1.2 Příklad zdrojového kódu

```
#include <wx/app.h>
#include <wx/frame.h>

class MyApp : public wxApp
{
public:
  MyApp() {}
  ~MyApp() {}

  virtual bool OnInit()
  {
    bool wasSuccess = false;
    wxFrame *frame = new wxFrame(0, wxID_ANY, _T("NOVE OKNO"));

    if(frame) // kontrola alokace pameti
    {
      SetTopWindow(frame); // nejvyssi okno aplikace
      frame->Show(true); // zobrazeni okna
      wasSuccess = true;
    }

    return wasSuccess;
  }
};

IMPLEMENT_APP(MyApp)
```

Třída wxApp má tyto úkoly [11]:

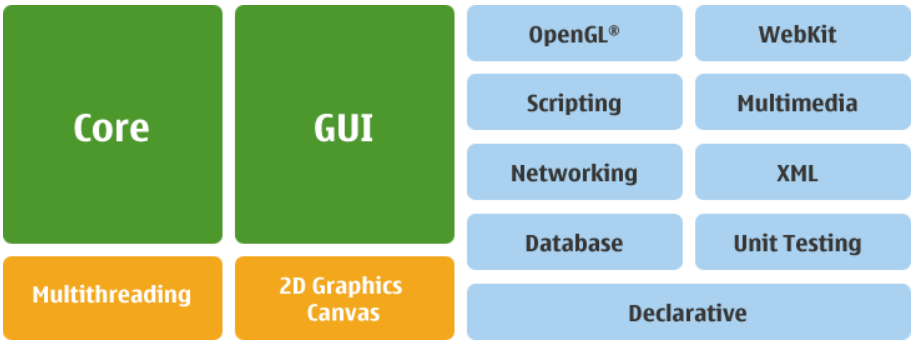
- nastavit a získat globální vlastnosti aplikace,
- zajistit komunikaci s operačním systémem např. vytvoření tzv. „Event Loop“, což je nekonečná smyčka čekající na zprávy od operačního systému, které následně převádí na události pro tento toolkit,
- provést kód v metodě OnInit, tu definuje programátor, pokud vrátí false, dáme wxWidgets najevo, že se nezdařila inicializace a aplikace se ukončí,
- zpracovávat události, které nejsou zachyceny v jiných objektech v aplikaci.

4.3.2 Qt

Qt toolkit byl vytvořen v roce 1999 společností Trolltech, která jej v roce 2008 prodala firmě Nokia. Qt toolkit je multiplatformní nástroj, ve kterém lze vyvíjet konzolové nebo GUI aplikace v odlišných programovacích jazycích pro různé platformy. Aplikace napsané s pomocí toolkitu je možno distribuovat pod licencí GPL, LGPL, nebo po splnění určitých podmínek i komerčně.

Qt je knihovna programovacího jazyka C++, ale existuje i pro jazyky Python (PyQt), Ruby (QtRuby), C, Perl, Pascal, C#, Java (Jambi) a Haskell. Podporuje lokalizaci aplikací a také SQL, zpracování XML, správu vláken, přístup k souborům, práci s grafikou a multimédii. Velkou výhodou Qt je velmi přehledně zpracovaná dokumentace a také vývojové programy Qt Creator nebo Qt Designer. Aplikace vytvořené pro grafické uživatelské prostředí používají nativní vzhled operačního systému, takže vyvinuté aplikace se vždy přizpůsobí do používaného prostředí. [10]

4.3.2.1 Moduly



Obrázek 4.2: Moduly frameworku Qt 4.7

QtCore

Obsahuje negrafické třídy využívané ostatnímy moduly frameworku, jako jsou `QList`, `QString`, `QVector`, apod.

QtGui

Součástí QtGui jsou komponenty grafického uživatelského rozhraní – např. `QFileDialog`, `QWidget`.

QtNetwork

Nástroje pro programování síťových protokolů přes protokoly TCP, UDP, FTP, nebo šifrované SSL.

QtOpenGL

QtOpenGL obsahuje funkce pro snadné využití OpenGL v Qt aplikacích.

QtOpenVG

Modul zajišťující podporu pro kreslení vektorové grafiky.

QtSQL

Třídy pro připojení k některým databázím využivajícím dotazovací jazyk SQL.

QtSvg

Modul vhodný pro zobrazování a kreslení vektorových SVG souborů.

QtWebKit

QtWebKit obsahuje třídy pro zobrazení a editaci webového obsahu.

QtXml

Třídy z modulu QtXml jsou určené pro tvoření a zpracování XML souborů metodou DOM nebo SAX.

QtXmlPatterns

Dotazovací jazyky XQuery, XPath, tvorba XML schémat a validace XML dokumentů.

QtDeclarative

Modul QtDeclarative poskytuje funkce pro vytváření dynamických uživatelských rozhraní.

Phonon

Multimediální framework pro přehrávání audio a video souborů.

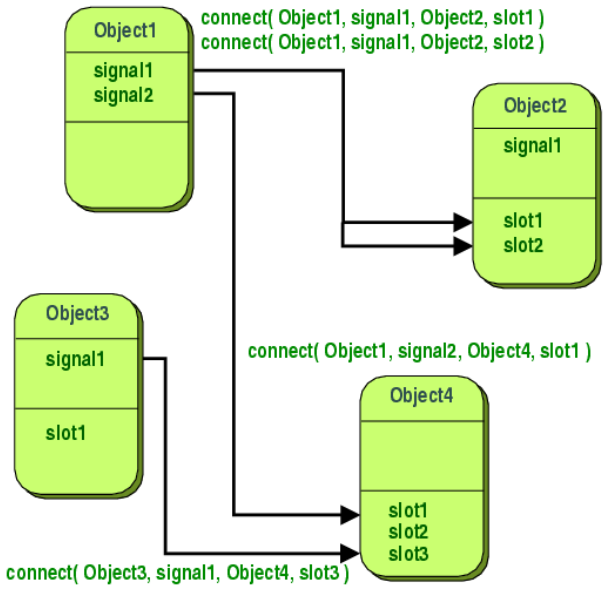
Qt3Support

Modul zajišťující kompatibilitu s předchozí verzí Qt frameworku.

4.3.2.2 Signály a sloty

Signály a sloty, pravděpodobně hlavní vlastnost odlišující Qt od ostatních frameworků, jsou využívány pro komunikaci mezi objekty v projektu. Nahradily původně využívaný *callback*, což je ukazatel na metodu objektu, kterou chceme vyvolat po nějaké události jiného objektu.

Sloty a signály mohou být využity ve všech objektech, které jsou přímo nebo nepřímo zděděny ze třídy `QObject`. Při propojování signálů a slotů může být s jedním slotem spojeno několik různých signálů a stejně tak na jeden signál napojeno několik slotů. Sloty mohou být použity pro přijímání signálů a zároveň mohou být použity jako standardní metoda objektu [8].



Obrázek 4.3: Komunikace mezi objekty v Qt pomocí signálů a slotů

Příklad komunikace

Intuitivní návrh třídy čítače:

```
class Counter
{
public:
  Counter() { m_value = 0; }
  int value() const { return m_value; }
  void setValue(int value);

private:
  int m_value;
};
```

Třídu `Counter` obohatíme o signály a sloty, přičemž funkčnost třídy zůstane zachována, nadále bude možné využívat veřejné metody třídy:

```
#include <QObject>

class Counter : public QObject
{
  Q_OBJECT

public:
  Counter() { m_value = 0; }
  int value() const { return m_value; }

public slots:
  void setValue(int value);

signals:
  void valueChanged(int newValue);

private:
  int m_value;
};
```

Takto upravená třída může okolnímu světu vysláním signálu `valueChanged(int)` oznámit změnu svého stavu. Slot `setValue(int)` může naopak zachytávat signály vyslané ostatními objekty. Všechny třídy obsahující signály a sloty musí být potomky `QObject` a také musí obsahovat v deklaraci třídy makro `Q_OBJECT`.

Pravděpodobná implementace slotu `setValue(int)` programátorem:

```
void Counter::setValue(int value)
{
  if(value != m_value)
```

```
  {
    m_value = value;
    emit valueChanged(value); // vyslani signalu
  }
}
```

Funkčnost pak lze ověřit následujícím příkladem – synchronizací dvou instancí třídy `Counter`:

```
Counter a, b;
QObject::connect(&a, SIGNAL(valueChanged(int)), &b, SLOT(setValue(int)));

a.setValue(12); // a.value() == 12, b.value() == 12
b.setValue(48); // a.value() == 12, b.value() == 48
```

4.3.2.3 Příklad zdrojového kódu

Ve funkci `main()` se nejprve inicializuje `QApplication`, které se předají ke zpracování parametry spouštěného programu. Následuje vytvoření instance třídy `QWidget` nebo některého z jejich potomků – `QMainWindow`, `QDialog` apod.

```
#include <QtGui>

int main(int argc, char *argv[])
{
  QApplication app(argc, argv);

  QMainWindow window;
  window.resize(600, 480);
  window.show();

  return app.exec();
}
```

4.3.3 Volba knihovny GUI

Po detailním prozkoumání obou výše popsaných knihoven jsem se rozhodl využít obsáhlejší Qt framework, který je dle mého názoru lépe navržený a obsahuje perfektně zpracovaný modul pro kreslení 2D grafiky. Dalšími výhodami Qt je detailní popis všech modulů a tříd doplněný množstvím ukázkových příkladů včetně zdrojových kódů.

4.4 Implementace

Projekt byl vyvíjen na platformě *Linux*, konkrétně distribuci *Ubuntu 10.10*. Vývojovým prostředím byl zvolen *Netbeans 6.9.1* pro C++ s integrovanou podporou Qt. K překladu byl použit kompilátor *g++*, z rodiny překladačů *GNU GCC*, ve verzi 4.5.2. Překlad na platformu *Microsoft Windows* zajistil překladač vývojového prostředí *Microsoft Visual Studio 2008*. V projektu z důvodu zajištění plné kompatibility mezi překladači nebyly využity nové vlastnosti a funkce jazyka C++, uvedené ve standardu nazývaném *C++0x*. Pro vývoj byla zvolena, v době programování aplikace, poslední stabilní verze Qt frameworku - 4.7, z níž byly pro potřeby aplikace využity knihovny *QtCore, QtGui, QtSvg a QtXml*.

Projekt ASim sestává z 25 hlavičkových souborů a 25 zdrojových souborů, které obsahují přibližně 5500 řádků vlastního kódu. Při vývoji byl kladen důraz na snadnou čitelnost programového kódu, proto byly striktně dodržovány následující pravidla:

- Každý hlavičkový soubor obsahuje deklaraci právě jedné třídy, jejíž název je shodný s názvem hlavičkového souboru.
- Hlavičkový soubor obsahuje pouze deklarace proměnných a metod, s vyjímkou funkcí typu *get* a procedur typu *set*, jejichž definice obsahuje pouze návratovou hodnotu u funkce, respektive přiřazení hodnoty do členské proměnné u procedur.
- Zdrojový soubor obsahuje definice proměnných a metod třídy, jejíž název je shodný s názvem zdrojového souboru.
- Odkazy na hlavičkové soubory jsou přednostně uváděny v hlavičkových souborech, pouze v případech, kdy tato konstrukce vedla ke vzniku křížových závislostí mezi hlavičkovými soubory, byla v hlavičkovém souboru uvedena dočasná deklarace příslušné třídy, která byla ve zdrojovém kódu předdefinována odkazem na příslušný hlavičkový soubor.
- Celý projekt je uzavřen v jednom jménném prostoru, čímž se předchází případným problémům s kolizemi názvů tříd, definovaných uživatelem, s třídami Qt frameworku.
- Názvy tříd jsou uvedeny s počátečním velkým písmenem, názvy proměnných a metod s malým počátečním písmenem. V případě, že je třída hlavním oknem aplikace (dědí z *QMainWindow*) nebo dialogem (dědí z *QDialog*), předchází

její název prefix *W*, představující počáteční písmeno anglického slova Window (okno).

- Makra jsou využita pouze v nejnutnější případech.

U každé metody nechybí komentář s popisem činnosti, doplněný typem návratových hodnot u funkcí. Komentáři jsou také opatřeny složitější konstrukce kódu. Celá aplikace je psána se snahou maximálně využít vlastnosti objektově orientovaného programování, jako je abstrakce, zapouzdření, skládání, dědičnost a polymorfysmus. V projektu se tak lze setkat například s děděním, virtuálními i čistě virtuálními funkcemi, šablonami, rekurzí nebo přetěžováním funkcí a operátorů.

Z důvodu připojování zdrojů (obrázků, ikon, atd.) a zajištění funkčnosti signálů a slotů se musí ještě před startem C++ překladače spustit Qt preprocesor, který pro třídy obsahující v deklaraci makro `Q_OBJECT` generuje zdrojové soubory s prefixem `moc_`. Z XML souboru, obsahujícího odkazy na připojené zdroje, vytvoří zdrojový soubor s prefixem `qrc_`, který obsahuje pole s hexadecimálním zápisem připojených souborů. K automatickému provedení všech potřebných operací během sestavování aplikace definuje Qt vlastní formát platformově nezávislého Makefilu.

Nyní uvedeme nejdůležitější třídy z aplikace ASim.

4.4.1 Třída WMain

Třída WMain představuje hlavní okno aplikace, sestává ze tří částí grafického rozhraní. V hlavní části je zobrazena grafická scéna (automat), v pravém sloupci pak hierarchicky seřazený seznam objektů umístěných na scéně a pod ním tabulkové zobrazení parametrů aktivního objektu (dále jen „tabulka“).

4.4.1.1 Komunikace mezi grafickými prvky

Grafický prvek tabulka není jednotný, sestává ze tří prvků, které se dynamicky zobrazují dle typu označeného objektu na scéně:

- parametry stavu,
- parametry přechodu,
- parametry automatu (zobrazuje se v případě, že není označen stav ani přechod).

Záměnu prvků provádějí signály, které se vysílají z třídy `Automaton` při označení nebo odznačení objektu. V případě označení vysílají jako parametr ukazatel na daný objekt, při odznačení hodnotu 0. Demonstrujeme na příkladu 4.1, kdy na počátku není označen žádný objekt, následuje označení stavu a na závěr bude označen přechod.

pořadí	označeno	akce	signál	tabulka
1.	nic	žádná	žádný	automat
2.	stav	označení stavu	stateSelectedSignal(State *)	stav
3.	stav	označení přechodu	stateSelectedSignal(0)	automat
4.	přechod	žádná	transitionSelectedSignal(Transition *)	přechod

Tabulka 4.1: Komunikace mezi scénou a tabulkou pomocí signálů

Přičemž doba zobrazení kroku 3. je natolik krátká, že není okem uživatele postřehnutelná.

Činnost slotu pro zobrazení tabulky stavu a přechodu je až na zobrazení jiné instance totožná, proto uvádíme pouze slot pro stavy.

```
void WMain::showStateTblWidget(State *state)
{
    // oznaceni polozky v hiearchickem stromu
    explorer->selectItem(state);

    if(state)
    {
        // skryti tabulky automatu a prechodu
        automatonTbl->setVisible(false);
        transTbl->setVisible(false);

        // predani ukazatele na oznaceny stav
        stateTbl->setState(state);
        // zobrazeni tabulky stavu
        stateTbl->setVisible(true);

        // povoleni polozek v menu upravy
        disabledEditMenuAct(false);
    }
    else
    {
        // skryti tabulky stavu
        stateTbl->setVisible(false);
        // zobrazeni tabulky automatu
        automatonTbl->setVisible(true);
        // zneplatneni polozek v menu upravy
        disabledEditMenuAct(true);
    }
}
```

Slot odchytávající signály je rozdělen na dvě větve, dle hodnoty parametru - zda ukazuje na instanci stavu, nebo nabývá hodnoty 0. V případě, že je parametr nenulový, provede se skrytí tabulek automatu a přechodů, přiřazení ukazatele a zobrazení tabulky stavů. Poslední řádek povoluje akce z menu úpravy (smazat, přenést dopředu a přenést dozadu), ty jsou dostupné pouze při označení stavu nebo přechodu. Aby byla synchronizace co nejvíce uživatelsky přívětivá, je ve slotu prováděno voláním procedury `explorer->selectItem(QGraphicsItem *)` i zvýraznění položky v hierarchickém stromě. V případě nulového parametru se skryje tabulka stavu, zakryjí se položky v menu úpravy a zobrazí se tabulka automatu.

Pro úplnost ještě uvedeme propojení signálů se sloty, které se provádí vždy při inicializaci nového automatu.

```
connect(automaton, SIGNAL(stateSelectedSignal(State *)),
    this, SLOT(showStateTblWidget(State *)));
connect(automaton, SIGNAL(transitionSelectedSignal(Transition *)),
    this, SLOT(showTransTblWidget(Transition *)));
```

Při zavření scény s automatem se provede volání funkce `disconnect` s identickými parametry jako u `connect`.

4.4.1.2 Příznak modifikace automatu

Třída `WMain` dále obsahuje dialogy pro uložení a otevření automatu ze souboru. Chování funkcí *otevřít* a *zavřít* automat je závislé na hodnotě příznaku modifikace. V případě, že je automat od posledního uložení změněn, či je nově založen, nabízí se před provedením těchto akcí nejprve uložení scény. Neuloženou scénu signalizuje hvězdička v záhlaví okna aplikace a aktivace ikony *uložit* v panelu nástrojů, respektive položky *uložit* v menu soubor.

Při zavírání aplikace je volána funkce `close()`, která plní zároveň funkci slotu.

```
bool WMain::close()
{
    // je otevrena scena s automatem?
    if(automaton)
    {
        // je automat modifikovan
        if(automaton->isModified())
        {
```

```
            // dotazovaci dialog s tlacitky Ano, Ne a Storno
            // ...
        }
        else
            terminAutomaton(); // uzavreni automatu
    }

    return true;
}
```

Funkce vrací hodnotu `true` v případě, že scéna není otevřena, nebo je v průběhu funkce `close()` zavřena, což nastane tehdy, pokud není scéna modifikována, nebo pokud jsou její změny v dialogu uloženy nebo zahozeny). Stornování *dialogu uložení* nastavuje návratovou hodnotu funkce na `false`.

Volání funkce `close()` se mimo zavírání okna aplikace a scény provádí před otevřením jiného souboru s automatem.

```
void WMain::open()
{
    if(close())
    {
       // dialog pro otevreni souboru, nacteni souboru
       // ...
    }
}
```

4.4.1.3 Uložení scény jako PNG a SVG

K uložení scény automatu jako rastrového obrázku ve formátu PNG a vektorového obrázku SVG jsou využity třídy `QPixmap` a `QSvgGenerator` z modulu `QtSvg`. Rastrový i vektorový obrázek jsou ukládány včetně bílého pozadí a jejich rozměry jsou totožné s rozměry scény. Export do PNG se provádí se zapnutým vyhlazováním.

4.4.2 Třída Automaton

Třída `Automaton` je základním prvkem GUI aplikace, která umožňuje návrh a simulaci automatu. Z hlediska funkčnosti lze třídu rozdělit na čtyři základní funkce:

1. návrh a simulace automatu,
2. komunikace s ostatními prvky GUI,
3. ukládání a načítání souboru s automatem,
4. metody vztahující se k funkčnosti automatu.

4.4.2.1 Návrh a simulace automatu

Základem třídy `Automaton` je grafická scéna (`QGraphicsScene`), což je kreslíci plátno, které navíc umožňuje ochytávat různé externí události, jako je například pohyb myši. Na scénu se mohou umisťovat grafické prvky (`QGraphicsItem`), které v aplikaci představují stavy a přechody. Aby byla instance třídy `Automaton` použitelná zároveň pro návrh i pro simulaci, musí pracovat ve dvou zobrazovacích módech. Mód návrhu povoluje prvkům scény reagovat na události vyvolané pohybem myši (přesun, označení prvku), v módu simulace je prováděno pouze zvýrazňování prvků.

4.4.2.2 Komunikace s ostatními prvky GUI

Objekt udržuje příznak modifikace, který mění svou hodnotu na **true** v případě změny objektu samotného, nebo změny objektů s automatem souvisejících. Uživatel aplikace ocení tento příznak především při zavírání okna aplikace nebo otevírání nové scény, kdy se v případě, že je evidována modifikace, aplikace dotáže na uložení změn, čímž se předejde nechtěné ztrátě dat. Třída `Automaton` tvoří most pro komunikaci mezi hlavním oknem aplikace a instancemi stavů a přechodů, jenž vysílají signály o změně svého vnitřního stavu. Jedná se o procedury:

- `void stateSelected(State *state)` - označení stavu,
- `void stateChangePosition(State *state)` - změna pozice stavu na scéně,
- `void stateModified(State *state)` - změna názvu nebo příznaku stavu,
- `void transitionSelected(Transition *transition)` - označení přechodu,
- `void transitionModified(Transition *transition)` - změna přechodových operací, typu, počátečního nebo koncového stavu přechodu.

4.4.2.3 Ukládání a načítání souboru s automatem

Automat je ukládán do textových souborů do XML struktury, která je popsaná pomocí schéma XSD. K načítání XML souborů se využívá způsob DOM, při kterém se načítá celý XML soubor do stromové struktury objektů, které reprezentují uzly jazyka XML. Zvolený způsob je snadněji požitelný, ale pomalejší a paměťově náročnější než způsob API, který čte všechny prvky sekvenčně, což ale vzhledem k relativně malé velikosti vstupních XML souborů nebude vadit.

4.4.2.4 Metody vztahující se k funkčnosti automatu

Pro potřeby identifikace typu automatu, simulace a dalších funkcí obsahuje třída **Automaton** metody:

- `bool reduction()` - Hledání nedosažitelných stavů.
- `bool haveTransitionsDefineSymbol()` - Zjištění, zda mají všechny přechody definovaný alespoň jeden přechodový symbol - nutná podmínka před spuštěním simulace.
- `bool isDeterministic()` - Zjišťuje, zda jsou splněny všechny podmínky, aby byl navrhnutý automat deterministický.
- `bool isPushdown()` - Test, zda některý z přechodů automatu obsahuje zásobníkovou operaci.
- `bool hasStartingState()` - Obsahuje automat alespoň jeden počáteční stav?
- `QList<State *> startingStates()` - Vrací seznam počatéčních stavů.

Hledání nedosažitelných stavů

Vyhledání nedosažitelných stavů v automatu obstarává funkce `reduction()`, která má návratovou hodnotu buď `true` (pokud existují nedosažitelné stavy) nebo `false`, v případě, že lze dosáhnout všech stavů. Vyhledávací algoritmus využívá rekurzivního volání procedury `reductionStep(State *state, QList<State *> &list)`.

```
bool Automaton::reduction()
{
    // mnozina pocatecnich stavu
    QList<State *> startList = startingStates();
    // mnozina dosazitelnych stavu
    QList<State *> achievableList;

    // do mnoziny dosazitelnych stavu pridame vsechny pocatecni stavy
    // rekurze zajisti, ze krom pocatecnich stavu i vsechny stavy z
    // pocatecniho stavu dosazitelne
    foreach(State *s, startList)
    {
        reductionStep(s, achievableList);
    }

    // zde jiz je mnozina dosazitelnych stavu kompletni, zbyva tyto stavy zvyraznit
```

```
    // odznaceni prvku automatu
    clearSelection();

    // zvyrazneni vsech stavu, ktere nejsou obsazeny v mnozine dosazitelnych stavu
    foreach(State *s, stateList)
    {
        if(!achievableList.contains(s))
            selectItem(s, false);
    }

    return (stateList.count() != achievableList.count());
}

void Automaton::reductionStep(State *state, QList<State *> &list)
{
    // telo procedury se provadi pouze pokud stav jiz neni
    // v mnozine dosazitelnych stavu obsazen
    if(!list.contains(state))
    {
        // pridani stavu do mnoziny dosazitelnych stavu
        list << state;

        // do mnoziny dosazitelnych stavu pridame vsechny stavy, do kterych smeruje
        // prechod ze stavu state
        for(int i = 0; i < state->transitions().count(); i++)
        {
            if(state->transitions().at(i)->fromState() == state)
            {
                // rekurze - pro kazdy dosazitelny stav aktualniho stavu state
                // budeme hledat dosazitelne stavy
                reductionStep(state->transitions().at(i)->toState(), list);
            }
        }
    }
}
```

4.4.3 Třída WVisualization

Třída zajišťující simulaci průchodu automatem pro vstupní vzorek. Nejdůležitějšími metodami jsou procedury:

- `void step()` - Představuje jeden krok v simulaci. V každém kroku volá právě jednou proceduru `process()`, která zajišťuje hlavní logiku simulace. Metoda `step()` spouští dialogy při nedeterministických operacích, eviduje počet dosažení koncových stavů a zvýrazňuje aktuální stavy a přechody automatu.
- `void run()` - Spustí / pozastaví časovač, který při uplynutí daného intervalu volá proceduru `step()`.

- `void reset()` - Zastaví časovač, nastaví všem proměnným jejich počáteční hodnoty.
- `void process(Operation *operation, const QString &symb)` - Představuje rozhodovací logiku. Určuje, do jakého stavu se má automat přenést, zajišťuje provádění zásobníkových operací a zaznamenává uskutečněné kroky do boxu.

4.4.4 Seznam hlavičkových souborů

Kompletní seznam hlavičkových souborů je uveden v tabulce 4.2.

4.4.5 Překlad

Překlad čistého zdrojového kódu, který není udržován v projektu některého z vývojových rozhraní, se na operačním systému Linux provádí v následujících třech krocích.

4.4.5.1 Vytvoření projektového souboru

Nejprve se přejde do složky, která obsahuje zdrojové kódy. Následným zadáním příkazu

```
qmake -project QT="core gui svg xml" DESTDIR="out" OBJECTS_DIR="obj"
```

se vygeneruje projektový soubor **ASim.pro** (za předpokladu, že se nadřazená složka jmenuje ASim). Význam parametrů příkazu `qmake` je uveden v tabulce 4.3

4.4.5.2 Vytvoření souboru Makefile

Pomocí příkazu

```
qmake ASim.pro
```

vytvoříme soubor **Makefile**, který obsahuje popis pravidel pro překlad a popis závislostí. Při volání programu `qmake` tentokrát nemusíme zadávat hodotu přepínače, `-makefile` je výchozí hodnotou. Jediným argumentem je tak cesta k projektovému souboru vygenerovaném v předchozím kroku.

Hlavičkový soubor	Popis
`Arrow.h`	Šipka značící směr přechodu.
`Automaton.h`	Grafická scéna, metody vztahující se k funkčnosti automatu.
`AutomatonItem.h`	Základní třída pro všechny objekty umisťované na scénu - stavy a přechody.
`AutomatonTable.h`	Tabulkové zobrazení parametrů automatu.
`Clipboard.h`	Jednoprvková paměť, využívaná při kopírování operací u přechodu.
`Explorer.h`	Hiearchický seznam prvků umistěných na scéně.
`Functions.h`	Doplňující funkce využívané v celém projektu.
`Mathematics.h`	Obsahuje matematické konstanty a funkce.
`Operation.h`	Přechodové operace, složené ze vstupních symbolů a zásobníkových operací.
`OperationList.h`	Seznam sdružující operace, které jsou přiděleny jednomu přechodu.
`Pattern.h`	Grafický prvek simulující čtení vstupního slova.
`Stack.h`	Grafický prvek simulující činnost zásobníku.
`State.h`	Prvek scény automatu - stav.
`StateTable.h`	Tabulkové zobrazení parametrů stavu.
`Symbols.h`	Obsahuje funkce pro práci se symboly - převod z řetězce na seznam a opačně.
`Transition.h`	Obecný přechod bez definovaného tvaru, základ pro tvarově definované přechody.
`TransitionArc.h`	Obloukové přechody.
`TransitionLoop.h`	Smyčkové přechody.
`TransitionStraight.h`	Rovné přechody.
`TransitionTable.h`	Tabulkové zobrazení parametrů přechodů.
`WInitVisualization.h`	Dialogové okno, sloužící k nastavení počátečních hodnot simulace - vzorku a počatečních symbolů na zásobníku.
`WMain.h`	Hlavní okno aplikace.
`WOperations.h`	Dialogové okno určené pro editaci přechodových operací.
`WQuestion.h`	Dialogové okno určené pro výběr přechodu při simulaci nedeterministických automatů.
`WVisualization.h`	Okno zobrazující simulaci průchodu automatem pro zadané slovo.

Tabulka 4.2: Seznam hlavičkových souborů

4.4.5.3 Vytvoření spustitelného programu

V posledním kroku spustíme program

```
make
```

jehož funkce spočívá v

- načtení souboru Makefile,

Argument	Hodnota	Popis
`-projekt`		Přepínač, kterým určíme, že bude qmake generovat projektový soubor.
`QT`	`core gui svg xml`	Knihovny Qt frameworku, které jsou v projektu využívány, jedná se o `QtCore, QtGui, QtSvg, QtXml`.
`DESTDIR`	`output`	Spustitelný soubor bude po kompilaci umístěn ve složce **out**.
`OBJECTS_DIR`	`objects`	Objektové soubory budou umisťovány do složky **obj**.

Tabulka 4.3: Parametry příkazu qmake při vytváření projektového souboru

- kontroly změn v souborech,
- vyhodnocení závislostí,
- kompilování a „slinkování“ změněných částí.

Nejprve se spustí preprocesor `moc`, který vytvoří pro soubory obsahující makro `Q_OBJECT` soubory s prefixem **moc_**. Poté se do složky **obj** vytvoří objektové soubory, obsahující tzv. „kód relativních adres“. Při kompilaci je spouštěn překladač `g++` s parametry uvedenými v tabulce 4.4

Argument	Popis
`-pipe`	Zrychluje kompilaci.
`-O2`	Nastavuje úroveň optimalizace kódu.
`-Wall`	Překladač vypisuje všechna varování.
`-DQT_NO_DEBUG`	Kompilace se provádí bez podpory debuggování (ladění), výsledkem je menší výstupní soubor.
`-DQT_SVG_LIB`	Připojení modulu QtSvg.
`-DQT_XML_LIB`	Připojení modulu QtXml.
`-DQT_GUI_LIB`	Připojení modulu QtGui.
`-DQT_CORE_LIB`	Připojení modulu QtCore.
`-DQT_SHARED`	Qt knihovny nejsou připojeny staticky (Pozn. aby šla aplikace spustit na Windows, musí být k ní výše uvedené knihovny přiloženy, případně se musí nacházet ve standardním systémovém uložišti).

Tabulka 4.4: Parametry kompilátoru g++

Nakonec se provede „slinkování“ objektových souborů, jehož výsledkem je spustitelný program **ASim**, umístěný v adresáři **out**.

4.4.6 Platformová přenositelnost

Přenos aplikace na ostatní platformy je závislý na podpoře dané platformy Qt frameworkem, který je však k dispozici pro většinu dnes používaných operačních systémů (kromě OS Linux a Windows, také například Mac). Mezi další požadavky patří existence standardní knihovny jazyka C++ a kompilátor jazyka C++ kompatibilní s Qt frameworkem (g++ na OS Linux, VS 2008 nebo MinGW na Windows).

4.5 Testování

Aplikace **ASim** byla úspěšně otestována na počítačích s následujícími konfiguracemi:

- **Notebook ASUS F3Jr** - Intel Core Duo T2450 2.0 GHz, 2048 MB RAM, 160 GB, ATI Radeon X2300 256 MB, OS Linux, distribuce Ubuntu 10.10 (32 bit)
- **Notebook Lenovo G560** - Intel Core i3 2.53 GHz, 4096 MB RAM, 500 GB, NVIDIA GeForce 310M 512 MB, OS Microsoft Windows 7 Home Premium (64 bit)
- **Stolní počítač** - Intel Pentium IV 2.8 GHz, 512 MB RAM, 80 GB, Intel 82845G 64 MB, OS Microsoft Windows XP Professional SP3 (32 bit)

4.5.1 Příklad 1

Navrhněte konečný automat nad abecedou {a, b}, který zjišťuje, zda slovo obsahuje počet symbolů „a“ dělitelný třemi.

Návrh

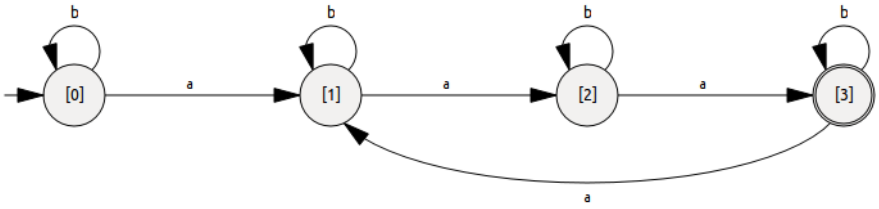
Pro tento příklad nám bude postačovat navrhnout konečný deterministický automat, který může vypadat například takto:

$\alpha = (\{[0], [1], [2], [3]\}, \{a, b\}, \delta, [0], [3]),$

kde je přechodová funkce δ dána následující tabulkou:

		a	b
→	[0]	[1]	[0]
	[1]	[2]	[1]
	[2]	[3]	[2]
←	[3]	[1]	[3]

Tabulka 4.5: Příklad 1 - Přechodová tabulka automatu přijímajícího vstupy s lichým počtem výskytu znaku „a“



Obrázek 4.4: Příklad 1 - Automat příjímající lichý počet znaků „a“.

Simulace

Na následujících vstupních vzorcích jsme otestovali správnou funkčnost automatu:

- **aaa** - automat přijal
- **bbababaabbbab** - automat nepřijal
- **aababbbbaaaabaababaa** - automat přijal

4.5.2 Příklad 2

Navrhněte automat, který bude kontrolovat syntaktickou správnost aritmetických výrazů včetně správného uzavření závorek. Ve výrazu se mohou vyskytovat proměnné i čísla, z operatorů pak +, -, . (násobení), / (dělení).

Návrh

Tento příklad již představuje náročnější řešení, je zřejmé, že si již nevystačí s konečným automatem, ale z důvodu kontroly uzavření závorek budeme muset využít zásobníkový automat.

Příklad jedné z možných gramatik:

(1) $Rovnice \rightarrow Element$

(2) $Element \rightarrow - \ Element$

(3) $Element \rightarrow (\ Element\ )$

(4) $Element \rightarrow Hodnota$

(5) $Element \rightarrow Element\ Operator\ Element$

(6) $Hodnota \rightarrow Cislo$

(7) $Hodnota \rightarrow Promenna$

(8) $Operator \rightarrow +$

(9) $Operator \rightarrow -$

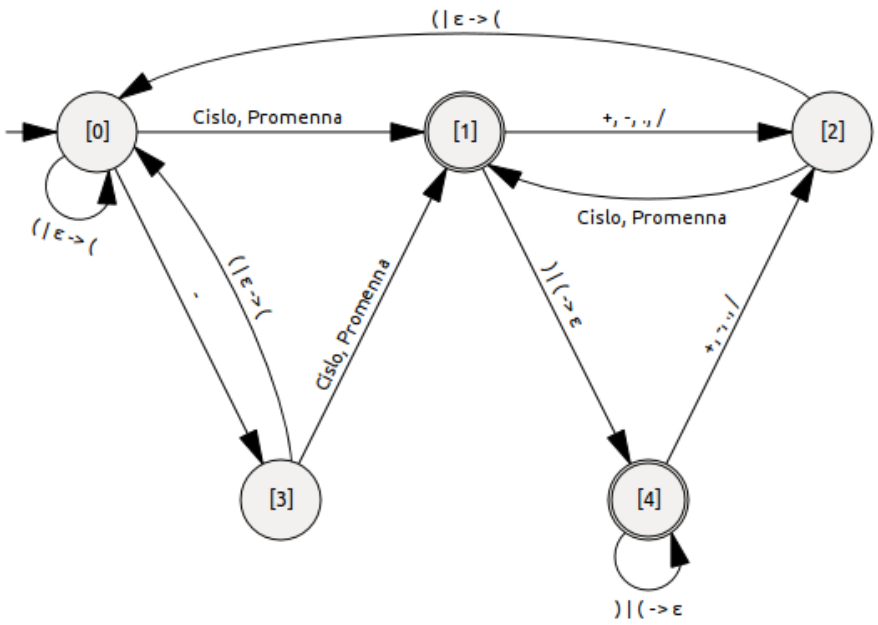
(10) $Operator \rightarrow .$

(11) $Operator \rightarrow /$

Jazyk, generovaný výše definovanou gramatikou, přijímá například automat:
$\alpha = (\{[0],[1],[2],[3],[4]\}, \{(,),+,-,.,/,Cislo,Promenna\}, \{(,)\}, \delta, [0], \varepsilon, [1,4])$,
kde vstupní symbol **Cislo** představuje realné číslo a **Promenna** je libovolné symbolické označení zastupující v rovnici číslo. Rovnice je správně zadaná v případě, že po přečtení všech vstupních symbolů se automat bude nacházet v jednom z koncových stavů, tj. **[1]** nebo **[4]**. Závorky jsou v rovnici správně uzavřeny, pokud nenastane chyba srovnání na zásobníku a po přečtení všech vstupních symbolů bude zásobník prázdný. Přechodová funkce δ je dána tabulkou 4.6.

	operátor +, ., /	operátor -	otev. z. (	uzav. z.)	číslo, proměnná
→ [0]		[3]	[0] | $\varepsilon \mapsto$ (		[1]
← [1]		[2]		[4] | ($\mapsto \varepsilon$	
[2]			[0] | $\varepsilon \mapsto$ (		[1]
[3]			[0] | $\varepsilon \mapsto$ (		[1]
← [4]	[2]	[2]		[4] | ($\mapsto \varepsilon$	

Tabulka 4.6: Příklad 2 - Přechodová tabulka automatu kontrolujícího správnost aritmetických výrazů

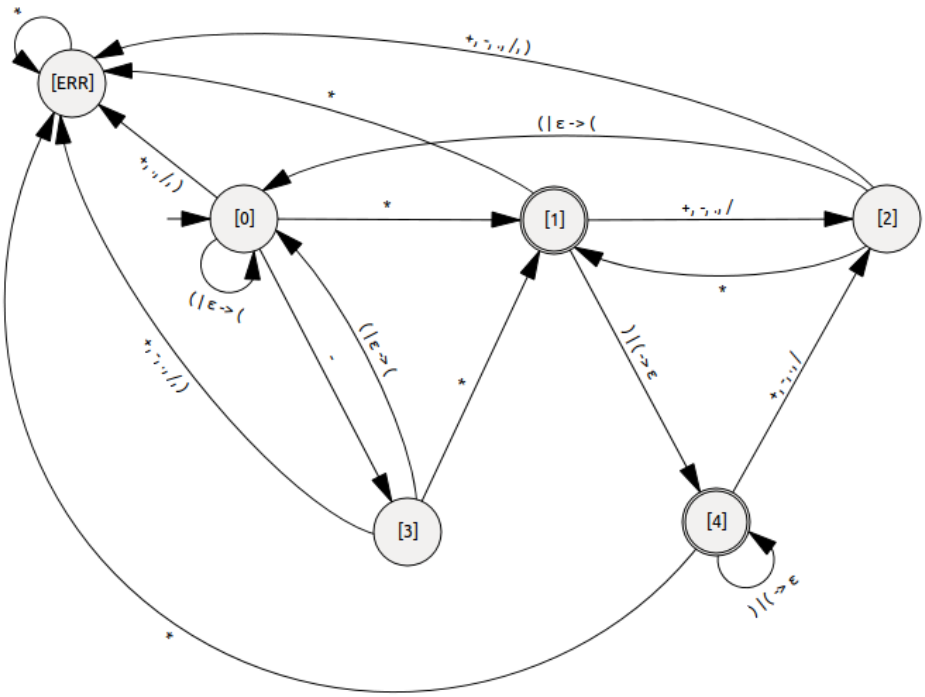


Obrázek 4.5: Příklad 2 - Automat kontrolující správnost aritmetických výrazů.

Rozeznávání čísel a libovolně pojmenovaných proměnných vyžaduje složitou konstrukci automatu, proto si situaci ulehčíme využitím přechodů s nedefinovaným přechodovým symbolem, označovaným v aplikaci ASim znakem *****. Automat využije tohoto přechodu v případě, že čtený symbol není uveden u žádného z ostatních přechodů vedoucích ze stavu, ve kterém se automat aktuálně nachází. Musíme tedy do automatu zavést chybový stav **[ERR]**, do něhož přecházíme, pokud je na vstupu nežádoucí symbol. Činnost upraveného automatu je znázorněna přechodovou tabulkou 4.7.

	operátor +, ., /	operator -	otev. z. (	uzav. z.)	číslo, proměnná *
→ [0]	[ERR]	[3]	[0] | $\varepsilon \mapsto$ (	[ERR]	[1]
← [1]	[ERR]	[2]	[ERR]	[4] | ($\mapsto \varepsilon$	[ERR]
[2]	[ERR]	[ERR]	[0] | $\varepsilon \mapsto$ (	[ERR]	[1]
[3]	[ERR]	[ERR]	[0] | $\varepsilon \mapsto$ (	[ERR]	[1]
← [4]	[2]	[2]	[ERR]	[4] | ($\mapsto \varepsilon$	[ERR]
[ERR]	[ERR]	[ERR]	[ERR]	[ERR]	[ERR]

Tabulka 4.7: Příklad 2 - Upravená přechodová tabulka automatu kontrolujícího správnost aritmetických výrazů



Obrázek 4.6: Příklad 2 - Automat kontrolující správnost aritmetických výrazů s chybovým stavem.

Simulace

Funkčnost byla vyzkoušena na následujících vzorcích:

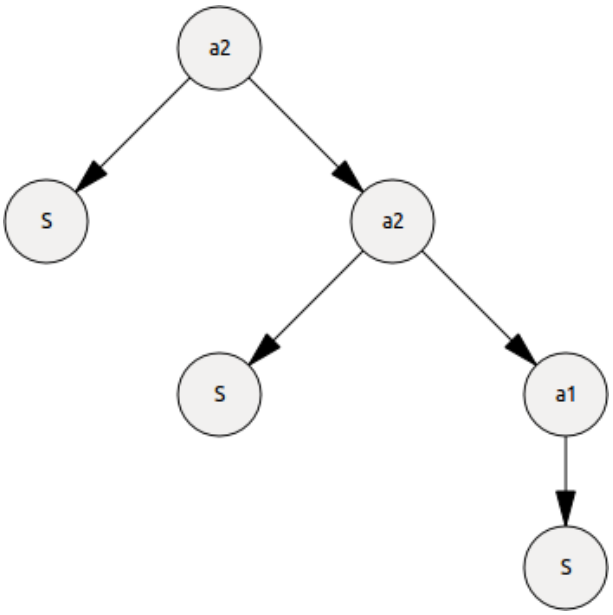
- **(8 + b) . (b + 15)** - automat přijal, zásobník prázdný
- **(-(a + 13) . (b + 11)** - automat přijal, zásobník není prázdný
- **a / * 5** - automat nepřijal, zásobník prázdný
- **-(a (-) (56 / c) . (44 + b / d)** - automat nepřijal, zásobník není prázdný
- **((-(a + 5) . (b /, (-c . a - d)) + d) . (a - c . d) - f) / a** - automat přijal, zásobník prázdný

4.5.3 Příklad 3

V dalším příkladě si otestujeme funkčnost aplikace na hledání vzorku (podstromu) ve stromě. Řešením této problematiky za zabývá algoritmická disciplína, která se nazývá

Arbologie. [6]

Příkladem je vytvoření automatu pro hledání vzorku, zobrazené na obr. 4.7.



Obrázek 4.7: Příklad 3 - Hledaný vzorek (podstrom)

Vytvoříme si postfix vzorku (průchod stromem metodou postorder): $S\ S\ S\ a_1\ a_2\ a2$.

Teorie [7] říká, že pro postfixové vzorky stromů s ohodnocením uzlů 0, 1, 2 je gramatika následující:

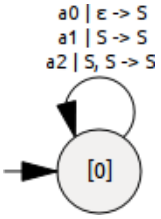
(1) $S \rightarrow a0$

(2) $S \rightarrow S\ a_1$

(3) $S \rightarrow S\ S\ a_2$

přechodové funkce δ:

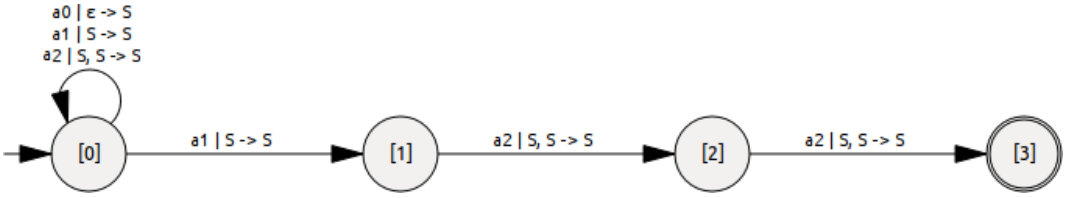
- $\delta(0,\ a_0,\ \varepsilon) = (0,\ \mathrm{S})$
- $\delta(0,\ a_1,\ \mathrm{S}) = (0,\ \mathrm{S})$
- $\delta(0,\ a_2,\ \mathrm{S\ S}) = (0,\ \mathrm{S})$
- $\delta(\mathrm{q},\ \mathrm{a},\ \alpha) = \{(q, S) : S \rightarrow \alpha a \in P\}$



Obrázek 4.8: Příklad 3 - Konstrukce automatu pro vyhledávání vzorků ve stromech

Konstrukce automatu

Postupným procházením postfixu vzorku se pro každý přechodový symbol vytvoří nový stav, pro vzorek $S\ S\ S\ a_1\ a_2\ a_2$ vypadá nedeterministický automat následovně:



Obrázek 4.9: Příklad 3 - Nedeterministický automat pro vzorek $S\ S\ S\ a_1\ a_2\ a2$

Formální definice automatu, znázorněném na obr. 4.9:

$$\alpha = (\{[0], [1], [2], [3]\}, \{a_0, a_1, a_2\}, \{S\}, \delta, [0], \varepsilon, [3]),$$

přechodová funkce δ je dána tabulkou:

		a_0	a_1	a_2
$\rightarrow$	[0]	$[0] \mid \varepsilon \mapsto S$	$[0] \mid S \mapsto S$ $[1] \mid S \mapsto S$	$[0] \mid S, S \mapsto S$
	[1]			$[2] \mid S, S \mapsto S$
	[2]			$[3] \mid S, S \mapsto S$
$\leftarrow$	[3]			

Tabulka 4.8: Příklad 3 - Přechodová tabulka nedeterministického automatu pro vyhledávání vzorků ve stromech I.

Převod nedeterministického automatu na deterministický

Vytvoříme nový stav **[01]** sloučením stavu [0] a [1] se shodnou zásobníkovou operací $S \mapsto S$.

		a_0	a_1	a_2
$\rightarrow$	[0]	[0] $\mid \varepsilon \mapsto S$	[01] $\mid S \mapsto S$	[0] $\mid S, S \mapsto S$
	[1]			[2] $\mid S, S \mapsto S$
	[2]			[3] $\mid S, S \mapsto S$
$\leftarrow$	[3]			
	[01]	[0] $\mid \varepsilon \mapsto S$	[01] $\mid S \mapsto S$	[0] $\mid S, S \mapsto S$ [2] $\mid S, S \mapsto S$

Tabulka 4.9: Příklad 3 - Přechodová tabulka nedeterministického automatu pro vyhledávání vzorků ve stromech II.

Vytvořením stavu [01] nám v přechodové tabulce vznikla nová nedeterministická operace - u přechodu se vstupním symbolem a_2, který vede ze stavu [01] souběžně do stavu [0] i [2] s identickou zásobníkovou operací $SS \mapsto S$.

		a_0	a_1	a_2
$\rightarrow$	[0]	[0] $\mid \varepsilon \mapsto S$	[01] $\mid S \mapsto S$	[0] $\mid S, S \mapsto S$
	[1]			[2] $\mid S, S \mapsto S$
	[2]			[3] $\mid S, S \mapsto S$
$\leftarrow$	[3]			
	[01]	[0] $\mid \varepsilon \mapsto S$	[01] $\mid S \mapsto S$	[02] $\mid S, S \mapsto S$
	[02]	[0] $\mid \varepsilon \mapsto S$	[01] $\mid S \mapsto S$	[0] $\mid S, S \mapsto S$ [3] $\mid S, S \mapsto S$

Tabulka 4.10: Příklad 3 - Přechodová tabulka nedeterministického automatu pro vyhledávání vzorků ve stromech III.

Stále se však v přechodové tabulce objevuje nedeterministická operace, a tak analogicky vytvoříme ještě další stav **[03]**.

		a_0	a_1	a_2
$\rightarrow$	[0]	[0] $\mid \varepsilon \mapsto S$	[01] $\mid S \mapsto S$	[0] $\mid S, S \mapsto S$
	[1]			[2] $\mid S, S \mapsto S$
	[2]			[3] $\mid S, S \mapsto S$
$\leftarrow$	[3]			
	[01]	[0] $\mid \varepsilon \mapsto S$	[01] $\mid S \mapsto S$	[02] $\mid S, S \mapsto S$
	[02]	[0] $\mid \varepsilon \mapsto S$	[01] $\mid S \mapsto S$	[03] $\mid S, S \mapsto S$
$\leftarrow$	[03]	[0] $\mid \varepsilon \mapsto S$	[01] $\mid S \mapsto S$	[0] $\mid S, S \mapsto S$

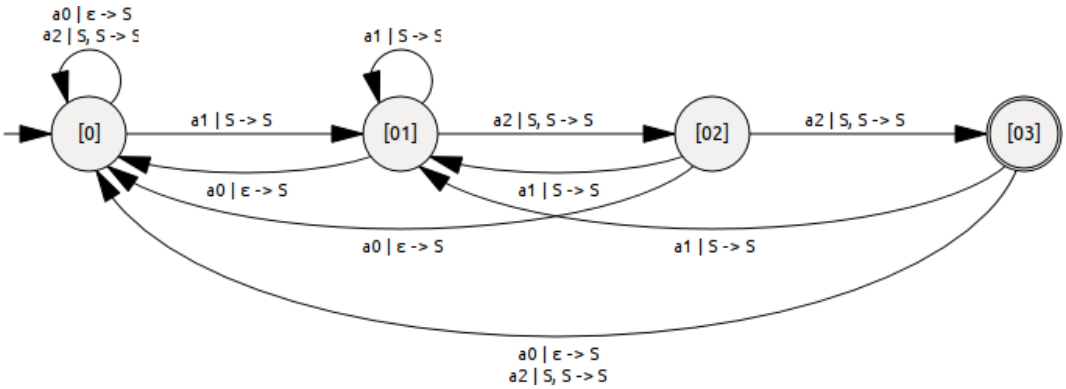
Tabulka 4.11: Příklad 3 - Přechodová tabulka deterministického automatu pro vyhledávání vzorků ve stromech I.

Nyní je již automat deterministický, v tabulce přechodů se však vyskytují nedosaži-

telné stavy, které odebereme.

		a_0	a_1	a_2
$\rightarrow$	[0]	[0] $\mid \varepsilon \mapsto S$	[01] $\mid S \mapsto S$	[0] $\mid S, S \mapsto S$
	[01]	[0] $\mid \varepsilon \mapsto S$	[01] $\mid S \mapsto S$	[02] $\mid S, S \mapsto S$
	[02]	[0] $\mid \varepsilon \mapsto S$	[01] $\mid S \mapsto S$	[03] $\mid S, S \mapsto S$
$\leftarrow$	[03]	[0] $\mid \varepsilon \mapsto S$	[01] $\mid S \mapsto S$	[0] $\mid S, S \mapsto S$

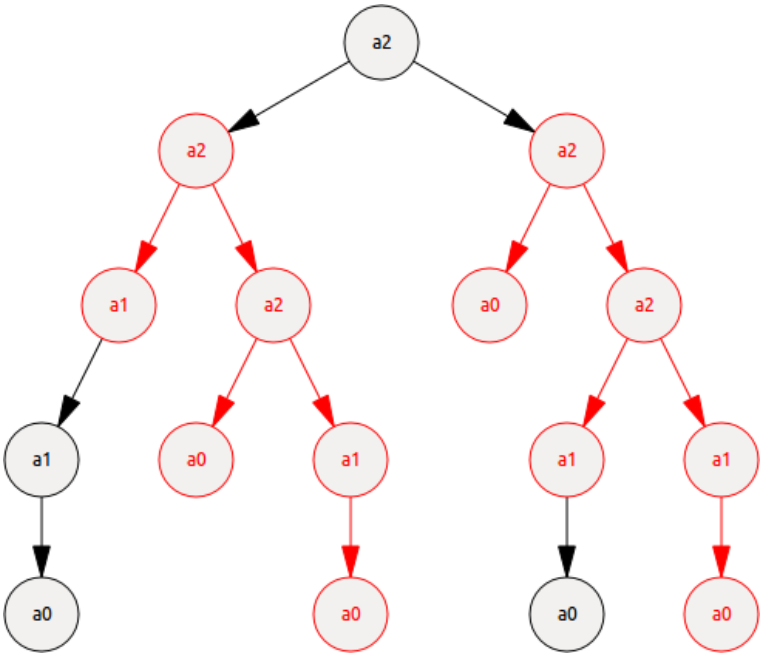
Tabulka 4.12: Příklad 3 - Přechodová tabulka deterministického automatu pro vyhledávání vzorků ve stromech II.



Obrázek 4.10: Příklad 3 - Deterministický automat pro vzorek $S\ S\ S\ a_1\ a_2\ a2$

Test

Funkčnost vizualizace vyzkoušíme na binárním stromu, který je zachycen na obr. 4.11. Automat dosáhne 2x koncového stavu [3], což představuje dvojí výskyt vzorku v zadaném stromu. Na obrázku jsou tyto výskyty zbarveně červeně.



Obrázek 4.11: Příklad 3 - Strom $a0\ a1\ a1\ a0\ a0\ a1\ a2\ a2\ a0\ a0\ a1\ a0\ a1\ a2\ a2\ a2$

4.5.4 Příklad 4

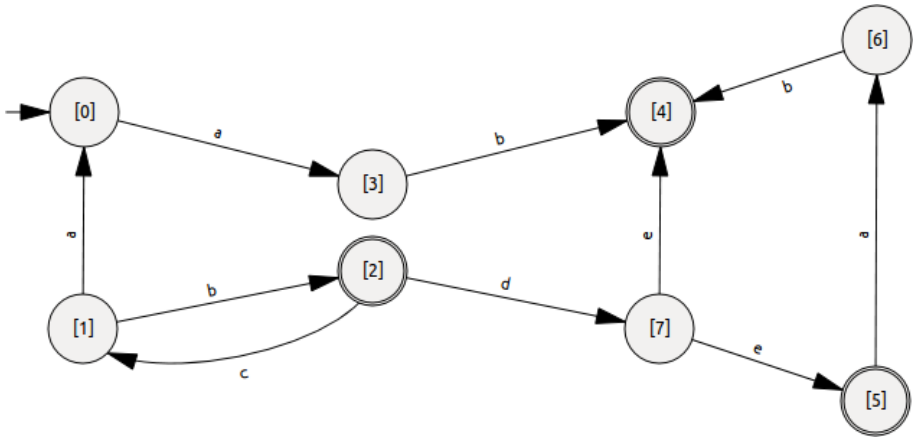
Posledním příkladem otestujeme funkci označování nedosažitelných stavů. Správnou funkčnost si ověříme na nedeterministickém automatu

$$\alpha = (\{[0], [1], [2], [3], [4], [5], [6], [7]\}, \{a, b, c, d, e, f, g, x, y\}, \delta, [0], [4]),$$

kde je funkce δ definována přechodovou tabulkou 4.13.

		a	b	c	d	e
→	[0]	[3]				
	[1]	[0]	[2]			
←	[2]			[1]	[7]	
	[3]		[4]			
←	[4]					
←	[5]	[6]				
	[6]		[4]			
	[7]					[4], [5]

Tabulka 4.13: Příklad 4 - Přechodová tabulka nedeterministického automatu obsahujícího nedosažitelné stavy



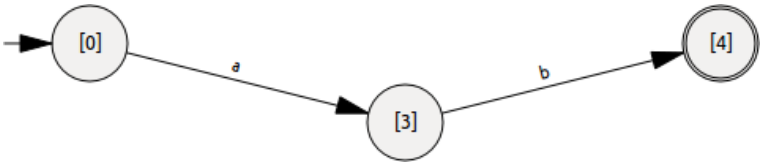
Obrázek 4.12: Příklad 4 - Nedeterministický automat obsahující nedosažitelné stavy

Provedli jsme označení nedosažitelných stavů, které jsme následně odstranili. Výsledný automat β je již deterministický a sestává pouze ze tří stavů a dvou přechodů.

$$\beta = (\{[0], [3], [4]\}, \{a, b\}, \delta, [0], \{[4]\}),$$

zobrazení δ:

- $\delta([0], a) = [3]$
- $\delta([3], b) = [4]$



Obrázek 4.13: Příklad 4 - Deterministický automat

Kapitola 5

Závěr

Cílem práce bylo vytvořit aplikaci, která bude umožňovat snadný návrh automatu s následnou možností simulace průchodu automatem pro zadaný vzorek. Dovolím si tvrdit, že zadaný cíl se podařilo splnit, výsledkem práce je plnohodnotná multiplatformní aplikace, která představuje jednoduchý nástroj pro návrh automatů. V požadovaném rozsahu jsou splněny i nároky na simulaci. Program *ASim* byl úspěšně otestován na operačních systémech Linux a Windows.

Při vývoji aplikace jsem si rozšířil znalosti programovacího jazyka C++ a nastudoval obsáhlý Qt framework, který poskytuje grafické uživatelské prostředí pro aplikace, u kterých je požadována snadná migrace na další operační systémy.

Jak jsem již předeslal v úvodní kapitole, aplikace *ASim* bude moci být využita především jako podpůrná pomůcka pro studenty při výuce předmětů, zabývajících se automaty.

V budoucnu by mohla být aplikace v oblasti návrhu automatu rozšířena o funkci převodu nedeterministických automatů na deterministické, zobrazení tabulky přechodů nebo export navrhnutého automatu pomocí příkazů sázecího systému LaTeX. Simulační část by mohla být rozšířena o možnost nastavení rychlosti automatické simulace, nebo o možnost editovat symboly na zásobníku při probíhající simulaci.

Kapitola 6

Seznam použitých zdrojů

Literatura

[1] B. Stroustrup. *Programovací jazyk C++*. BEN - technická literatura, Praha, 1997.

[2] Ch. Nagel, B. Evjen, J. Glynn, K. Watson, M. Skinner. *C# - Programujeme profesionálně*. Computer Press, 2009.

[3] J. Vaníček, M. Papík, R. Pergl, T. Vaníček. *Teoretické základy informatiky*. Kernberg Publishing, Praha, 2007.

[4] M. Chytil. *Automaty a gramatiky*. SNTL Praha, Praha, 1984.

[5] M. Demlová, V. Koubek. *Algebraická teorie automatů*. SNTL Praha, Praha, 1990.

[6] Arbologie.
`http://www.arbology.org`, stav z 5. 4. 2011.

[7] Arbologie - popis konstrukce automatů.
`http://www.dcs.kcl.ac.uk/events/LSD&LAW09/slides/Melichar1.pdf`, stav z 5. 4. 2011.

[8] Oficiální stránky projektu Qt.
`http://qt.nokia.com`, stav z 12. 3. 2011.

[9] Wikipedia: Bezkontextový jazyk.
`http://cs.wikipedia.org/wiki/Bezkontextový_jazyk`, stav z 5. 4. 2011.

[10] Wikipedia: Qt.
http://cs.wikipedia.org/wiki/Qt_(knihovna), stav z 12. 3. 2011.

[11] Wikipedia: wxWidgets.
http://cs.wikipedia.org/wiki/WxWidgets, stav z 12. 3. 2011.

Kapitola 7

Přílohy

7.1 Seznam použitých zkratek

CFG Bezkontextová gramatika, tj. formální gramatika, ve které mají všechna přepisovací pravidla tvar $A \to \beta$.

CNF Konjuktivní normální forma.

DFA Deterministický konečný automat.

DOM Objektově orientovaná reprezentace XML dokumentu.

GUI Grafické uživatelské rozhraní.

LIFO Charakteristický způsob manipulace s daty, data uložena jako poslední budou čtena jako první. „Last In – First Out“.

NFA Nedeterministický konečný automat.

OS Operační systém.

PDA Deterministický zásobníkový automat.

SAX Proudové zpracování XML dokumentu.

TM Turingův stroj.

XML Rozšířitelný značkovací jazyk.

7.2 Instalační a uživatelská příručka

7.2.1 Instalace

Aplikaci není třeba instalovat, postačí zkopírovaní adresáře se spustitelným programem na lokální disk. Druhou možností je překlad na lokálním stroji, ke kterému je třeba:

1. nainstalovat standardní knihovny jazyka C++ pro vývojáře,
2. nainstalovat Qt framework verze alespoň 4.7,
3. zkopírovat zdrojové soubory s obrázky z CD na lokální disk,
4. spustit terminál a přejít do složky ASim,
5. zadat příkaz `qmake -project QT="core gui svg xml" DESTDIR="out"OBJECTS_DIR="obj"`,
6. poté `qmake ASim.pro`,
7. nakonec příkaz `make`

7.2.2 Návrh

Hlavní funkce návrhu automatu představuje 5 přepínacích tlačítek:

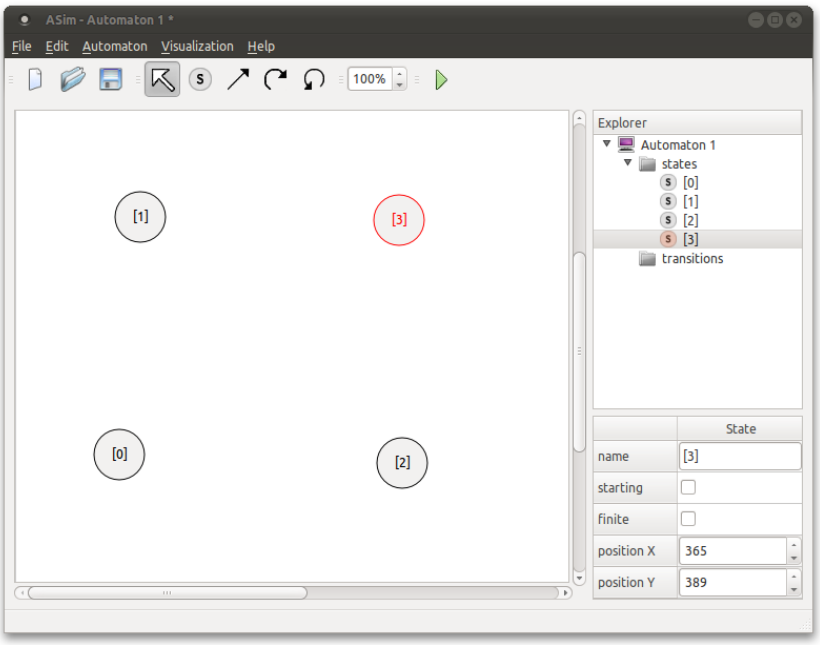
- posun,
- vytvořit stav,
- vytvořit rovný přechod,
- vytvořit obloukový přechod,
- vytvořit smyčkový přechod.

Posun

Mód posunu slouží pro označování prvků, přesunu stavů na jinou pozici, změnu velikosti oblouků u obloukových přechodů, změnu úhlu natočení smyčkových přechodů a úpravy parametrů stavů a přechodů.

Přidání stavu

K přidání nového stavu na scénu automatu postačí kliknout na libovolném místě plátna. Pod kurzorem se objeví nově vytvořený stav, automaticky nazvaný pořadovým číslem umístěným v hranatých závorkách. Název stavu lze poté změnit v pravém panelu vlastností objektu, stejně jako lze nastavit příznak „počáteční stav“ nebo „koncový stav“. V pravém panelu lze také určit přesnou polohu středu stavu na plátně, hodnoty pozice se obarví červeně v případě, že zadaná pozice stavu je mimo rozměry plátna. Pod kontextovou nabídkou vyvolanou pravým kliknutím na stav se skrývají akce umožňující přenést stav dopředu/dozadu před/za ostatní kolidující stavy a smazání stavu ze scény. Při smazání stavu se odstraní případně i stavy, které s daným stavem inklinují.



Obrázek 7.1: Ukázka aplikace - vytváření stavů

Přidání přechodu

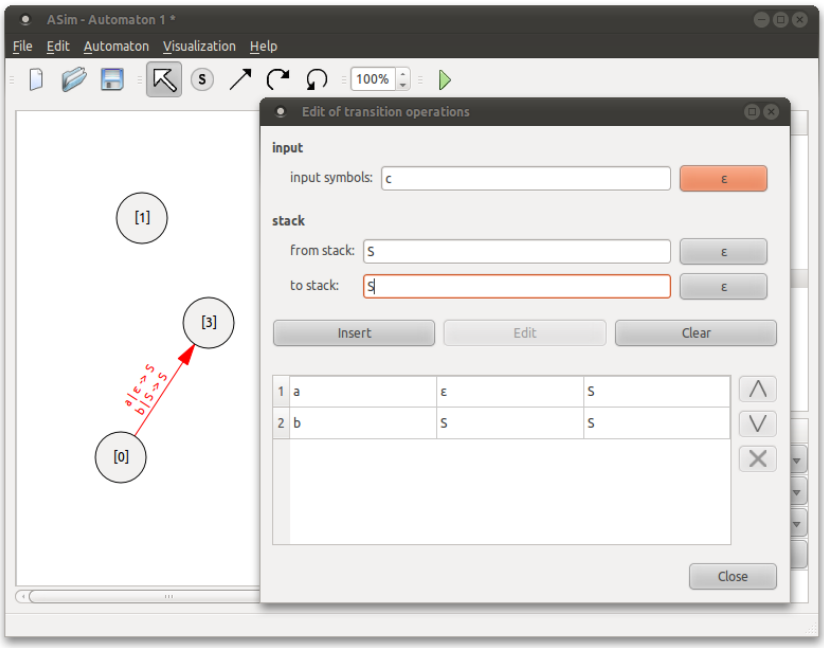
Rovný a přímý přechod přidáme kliknutím na výchozí stav přechodu, táhnutím myši a uvolněním tlačítka nad koncovým stavem přechodu. Smyčky se přidávají kliknutím na příslušný stav. Typ přechodu, vstupní a výstupní stavy lze dodatečně měnit v pravém panelu aplikace. Dvojklikem na přechod, nebo stiskem tlačítka **edit...** v

pravém panelu, otevřeme dialogové okno spravující operace přechodu, kde se definují vstupní symboly, případně zásobníkové operace. Více symbolů, vstupních i zásobníkových, se odděluje pomocí čárky, nebo se každá operace píše na nový řádek. Oba zápisy jsou ekvivalentní, víceřádkové zadávání je k dispozici z důvodu možného zadání více zásobníkových operací pro jeden přechod.

Symbol ε, který se zadává kliknutím na příslušné tlačítko, představuje prázdný znak. Zadáním symbolu ε jako přechodového symbolu vytvoříme tzv. ε-přechod. U zásobníkových operací znak ε značí, že se neprovádí srovnání na zásobníku, respektive se na vrchol zásobníku nevkládá žádný symbol. Znak * značí libovolný nedefinovaný přechodový symbol. Posledním speciální znakem je #, který je v zásobníkových operacích nahrazován použitým vstupním symbolem.

Stejně jako u stavů, lze i u přehodů vyvolat kontextovou nabídku s akcemi přenést dopředu, přenést dozadu a smazat.

Pro ulehčení práce při zapisování přechodových operací je možno využít možnosti jejich kopírování. Stiskem kláves **Ctrl+C** zkopírujeme operace z označeného přechodu do clipboard paměti, stiskem **Ctrl+V** je vložíme k označenému přechodu.



Obrázek 7.2: Ukázka aplikace - vytváření přechodů

7.2.2.1 Další funkce

Uložit

Navrhnutý automat se ukládá do pevně definované XML struktury, typicky s předdefinovaným názvem souboru ve formátu <**název_automatu**>**.xas**.

Otevřít

Otevřít je možné pouze uložené XAS soubory, načítání automatu z formátů SVG nebo PNG není možné.

Program umožňuje i spouštění aplikace s parametrem, kterým může být cesta k XAS souboru:

```
./ASim projekty/priklad.xas
```

Označit vše

Možnost **Označit vše** v nabídce **Úpravy** označí všecny stavy a přechody na plátně. Kliknutí na libovolný stav a následným tažením přemístíme všechny prvky na plátně na novou pozici.

Označit nedosažitelné stavy

Pod hlavní nabídkou **Automat** je první položkou možnost **Označit nedosažitelné stavy**, jejíž algoritmus prochází automatem od počátečních stavů do všech možných směrů. Stavy, které nebyly při průběhu algoritmu navštíveny, jsou nadbytečné, automat se do těchto stavů za žádných podmínek nedostane. Pokud je konstrukce automatu správná, je vhodné tyto stavy smazat pomocí klavesy **Delete**.

Typ automatu

V dialogovém okně se zobrazí zařazení navrhnutého automatu, které může nabývat hodnot:

- deterministický automat,
- nedeterministický automat,

- deterministický zásobníkový automat,
- nedeterministický zásobníkový automat.

Výstup do PNG a SVG

Poslední možností v návrhu je uložení plátna jako rastrový (*.PNG) nebo vektorový (*.SVG) obrázek.

7.2.2.2 Klávesové zkratky

- **Ctrl+B** Přenést označený prvek dozadu.
- **Ctrl+C** Zkopírovat operace označeného přechodu do clipboard paměti.
- **Ctrl+F** Přenést označený prvek dopředu.
- **Ctrl+N** Vytvořit nový automat.
- **Ctrl+O** Otevřít soubor s automatem.
- **Ctrl+Q** Zavřít aplikaci.
- **Ctrl+R** Start simulace.
- **Ctrl+S** Uložit.
- **Ctrl+Shift+S** Uložit jako.
- **Ctrl+V** Vložit zkopírováné operace z clipboard paměti.
- **Ctrl+W** Zavřít automat.
- **Del** Smazat prvek.
- **F1** Přepnout na mód Posun.
- **F2** Přepnout na mód vytvořit stav.
- **F3** Přepnout na mód vytvořit rovný přechod.
- **F4** Přepnout na mód vytvořit obloukový přechod.
- **F5** Přepnout na mód vytvořit smyčkový přechod.

7.2.3 Simulace

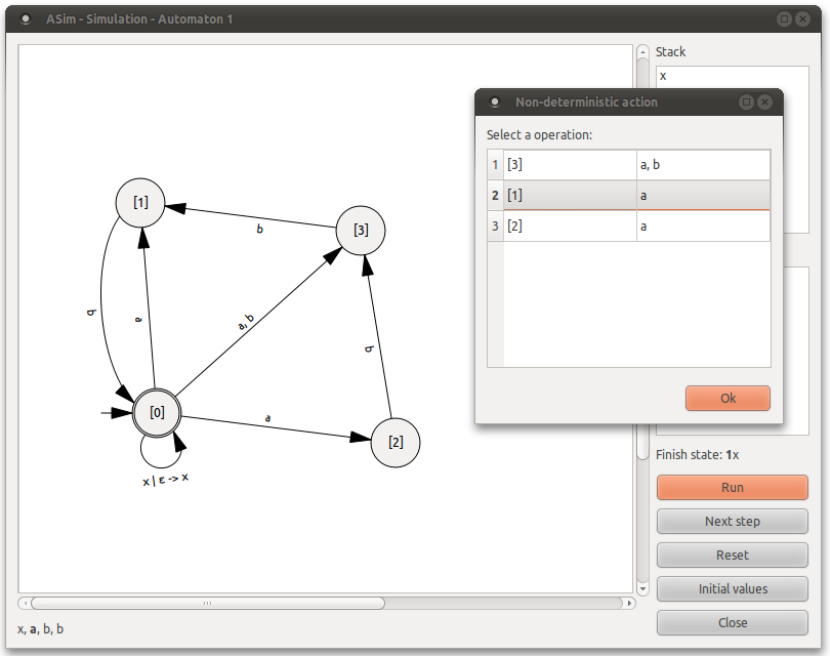
Před startem simulace se spustí dialog vyžadující zadání vstupního vzorku, případně počátečních zásobníkových symbolů, tyto hodnoty lze kdykoli změnit kliknutí na příslušné tlačítko.

Průchod automatem je možné buď automaticky spustit tlačítkem **Start**, nebo krokokovat pomocí tlačítka **Další krok**. Průběh vizualizace se dá vynulovat stisknutím tlačítka **Reset**.

Aktuálně zpracovávaný symbol je ve vzorku označen tučným textem. Jednotlivé kroky se zaznamenávají do boxu umístěném nad ovládacími tlačítky. Stav zásobníku je zobrazen v pravém horním boxu, posledně vložený prvek je zobrazen jako první.

Simulace ohlašuje chybu srovnání na zásobníku a neexistenci přechodu pro vstupní symbol z aktuálního stavu, po které je simulace s chybou ukončena.

Pokud existuje více počátečních stavů, nebo z aktuálního stavu vede pro čtený symbol více přechodů, zobrazí se dialog, kde si počáteční stav, nebo přechod zvolíme.



Obrázek 7.3: Ukázka aplikace - simulace

7.3 XML schéma použité struktury

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="asim">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="automaton">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="name" type="xs:string" />
       <xs:element name="states">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="state" minOccurs="0" maxOccurs="unbounded">
           <xs:complexType>
             <xs:simpleContent>
             <xs:extension base="xs:string">
              <xs:attribute name="id" type="xs:int" use="required"/>
              <xs:attribute name="posX" type="xs:int" use="required"/>
              <xs:attribute name="posY" type="xs:int" use="required"/>
              <xs:attribute name="zVal" type="xs:float" use="required"/>
              <xs:attribute name="starting" type="xs:boolean"
                default="false"/>
              <xs:attribute name="finite" type="xs:boolean"
                default="false"/>
             </xs:extension>
            </xs:simpleContent>
           </xs:complexType>
          </xs:element><!-- /state -->
         </xs:sequence>
        </xs:complexType>
       </xs:element><!-- /states -->
       <xs:element name="transitions">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="transition" minOccurs="0"
            maxOccurs="unbounded">
           <xs:complexType>
            <xs:sequence>
             <xs:element name="operations">
              <xs:complexType>
               <xs:sequence>
                <xs:element name="operation" minOccurs="0"
                  maxOccurs="unbounded">
                 <xs:complexType>
                  <xs:sequence>
                   <xs:element name="input">
                    <xs:complexType>
                     <xs:sequence>
                      <xs:element name="symbol" type="xs:string"
```

```
                    minOccurs="0" maxOccurs="unbounded" />
                  </xs:sequence>
                 </xs:complexType>
                </xs:element><!-- /input -->
                <xs:element name="fromStack">
                 <xs:complexType>
                  <xs:sequence>
                   <xs:element name="symbol" type="xs:string"
                    minOccurs="0" maxOccurs="unbounded" />
                  </xs:sequence>
                 </xs:complexType>
                </xs:element><!-- /fromStack -->
                <xs:element name="toStack">
                 <xs:complexType>
                  <xs:sequence>
                   <xs:element name="symbol" type="xs:string"
                    minOccurs="0" maxOccurs="unbounded" />
                  </xs:sequence>
                 </xs:complexType>
                </xs:element><!-- /toStack -->
               </xs:sequence>
              </xs:complexType>
             </xs:element><!-- /operation -->
            </xs:sequence>
           </xs:complexType>
          </xs:element><!-- /operations -->
         </xs:sequence>
         <xs:attribute name="type" type="xs:int" use="required"/>
         <xs:attribute name="refFrom" type="xs:int" use="required"/>
         <xs:attribute name="refTo" type="xs:int" default="-1"/>
         <xs:attribute name="zVal" type="xs:float" use="required"/>
         <xs:attribute name="angle" type="xs:float" default="0.0"/>
         <xs:attribute name="height" type="xs:float" default="0.0"/>
        </xs:complexType>
       </xs:element><!-- /transition -->
      </xs:sequence>
     </xs:complexType>
    </xs:element><!-- /transitions -->
   </xs:sequence>
   <xs:attribute name="width" type="xs:int" use="required"/>
   <xs:attribute name="height" type="xs:int" use="required"/>
   <xs:attribute name="stateCounter" type="xs:int" use="required"/>
  </xs:complexType>
 </xs:element><!-- /automaton -->
</xs:sequence>
<xs:attribute name="version" type="xs:float" use="required"/>
</xs:complexType>
</xs:element><!-- /asim -->
</xs:schema>
```

7.4 Použitý software

- **Gimp** Drobné úpravy a tvorba obrázků.
- **Microsoft Visual Studio 2008** Překlad na platformu Windows.
- **Netbeans 6.9.1** Vývojové prostředí.
- **Texmaker** Sazba dokumentace.

7.5 Obsah přiloženého CD

- **doc/** - adresář obsahuje dokumentaci ve formátu PDF
  - **src/** - adresář obsahuje zdrojové soubory sázecího systému LaTeX
- **project/**
  - **source/** - adresář obsahuje zdrojové kódy připravené k překladu přes terminál nebo příkazový řádek
  - **ubuntu - Netbeans 6.9.1/** - adresář obsahuje zdrojový projekt vytvořený programem Netbeans 6.9.1 na Ubuntu
  - **windows - Visual Studio 2008/** - adresář obsahuje zdrojový projekt vytvořený programem Microsoft Visual Studio 2008 na Windows
- **public/**
  - **ubuntu 32b/** - adresář obsahuje spustitelný program ASim, včetně testovacích příkladů, zkompilovaný na Ubuntu (32bit)
  - **windows 32b/** - adresář obsahuje spustitelný program ASim, včetně testovacích příkladů, zkompilovaný na Windows (32bit)