FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Webový editor konečných automatů
Student:	Petr Svoboda
Vedoucí:	Ing. Jan Trávníček
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

Nastudujte definici různých druhů konečných automatů.
Navrhněte webový nástroj pro návrh a kreslení konečných automatů.
- Podporujte načtení a uložení souboru s definicí konečného automatu.
- Umožněte automatický výpočet pozicování stavů a přechodů editovaného konečného automatu.
- V návrhu se zaměřte na možnost rozšíření o další pozicovací algoritmy a formáty reprezentace uloženého konečného automatu v souboru.

Implementaci proveďte v jazyce vhodném pro použití ve webovém prohlížeči.
Proveďte uživatelské testování výsledné implementace.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 20. listopadu 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Webový editor konečných automatů

Petr Svoboda

Katedra softwarového inženýrství
Vedoucí práce: Ing. Jan Trávníček

15. května 2019

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 15. května 2019 .....

Odkaz na tuto práci

Svoboda, Petr. *Webový editor konečných automatů*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Bakalářská práce popisuje proces návrhu a realizace webové aplikace v podobě dynamického editoru konečných automatů. Analyzují se stávající řešení a jejich přínos. Následně se zdůvodňuje výběr zvolených webových technologií a jak byly využity. Práce dále rozebírá problematiku importu a exportu dat pomocí souborů různých formátů a automatické pozicování vytvořených automatů pozicovacími algoritmy. Tato funkčnost je navržena tak, aby byla modulární a nezávislá na zbytku aplikace. V práci se také řeší tvorba přívětivého uživatelského prostředí.

Klíčová slova webová aplikace, editor, konečný automat, vykreslování grafů, pozicovací algoritmy, funckionální programování, SVG, TypeScript, React

Abstract

Bachelor thesis describes process of designing and implementing web application in the form of dynamic editor of finite automata. Existing solutions and their benefits are analyzed. It describes the selection and utilization of web technologies used in its creation. It also analyzes solutions for importing and exporting data of various file formats and automatic positioning of created

state machines with the help of positioning algorithms. Lastly it also describes development of user-friendly enviroment.

Keywords web application, editor, finite automata, graph drawing, positioning algorithms, funcional programming, SVG, TypeScript, React

Obsah

Seznam obrázků

Seznam tabulek

Seznam algoritmů

Seznam ukázek kódu

Úvod

Stavové automaty jsou užitečným konstruktem pro vizualizaci reálných i teoretických problémů. Jednou z jejich výhod je možnost situaci jednoduše a rychle zakreslit na papír. Problém vzniká pokud chceme se stejnou rychlostí automat převést do digitální podoby.

Určitě není třeba diskutovat nad tím, že existuje nepřeberné množství různých nástrojů, ať už založených na webových nebo nativních platformách, řešících různými způsoby kreslení grafů na digitální ploše. Některé z nich se dokonce přímo soustředí na automaty. Žádný z nich ale neobsahuje kompletní funkcionalitu, kterou si požadoval autor práce. Jedná se hlavně o velmi jednoduché uživatelské prostředí, které umožní rychlé tvoření automatů, které by byly následně napozicovány do přehledné podoby a exportovány v uživatelsky definovaných formátech a to bez nutnosti cokoli instalovat.

Jak už bylo řečeno, stavové automaty lze využít k mnoha účelům. Existuje JavaScript knihovna `xstate` [1], umožňující, v rámci definovaného automatu, manipulaci s aktuálním stavem automatu na základě prováděných akcí. Zde se využívá výhod automatů a může se například definovat, jak se bude chovat stav uživatelského prostředí aplikace nebo jaké manipulace se budou provádět na položce databáze v závislosti na jejím aktuálním stavu.

Automaty je nejdříve potřeba nadefinovat, což knihovna `xstate` neumožňuje a pouze čte definici automatu ve svém formátu ze souboru. Pro menší automaty není problém soubor upravit, čím je ale automat rozsáhlejší, tím se úprava ztěžuje. Proto vznikl nápad na aplikaci přímo určenou k rychlé a jednoduché definici automatů. Aplikace by pak umožňovala export do různých formátů, jedním z nich by byl formát automatů knihovny `xstate`.

Webové technologie jsou neustále na vzestupu [2] a zmíněná aplikace je ideálním kandidátem pro jejich využití. Použití webu zpřístupní aplikaci všem uživatelům, kteří mají moderní prohlížeč. Web také umožňuje jednoduchou tvorbu dynamického uživatelského prostředí, což bude pro vytvoření kvalitního editoru nezbytné. Velkou výhodou bude podpora vektorové grafiky prohlížeči,

což je pro vykreslování grafů, v tomto případě automatů, nejlepší přístup z hlediska různých transformací elementů automatu.

Pokud se jedná o vizualizaci stavových automatů, určitě kromě kreslení automatu přímo uživatelem je možnost automatického pozicování podle algoritmu také výhodou, která výsledné automaty zpřehlední. Aplikace tak bude mít možnost využít různých pozicovacích algoritmů, které uspořádají automat pro nejlepší čitelnost podle vybraných parametrů.

V první kapitole se představí přesné cíle pro navrhované řešení. Druhá kapitola sepisuje teoretický základ potřebný k návrhu řešení. Dále ve třetí kapitole následuje podrobná analýza ostatních řešení a stručnější popis cílů. Ve čtvrté kapitole je popsán proces návrhu a odůvodnění zvolených řešení. Na to úzce navazuje pátá kapitole popisující konkrétní implementaci a ukazuje některou složitější funkcionalitu. V poslední kapitole jsou uvedeny výsledky testování výsledné aplikace.

Kapitola 1

Cíl práce

Cílem práce je navrhnout a vytvořit webový editor konečných automatů. Editor bude zpracován jako klientská aplikace běžící v prohlížeči uživatele. S využitím moderních webových technologií, jako jsou TypeScript a React, aplikace poskytne jednoduché uživatelské prostředí, ve kterém bude uživatel moci definovat automaty.

Uživatel bude editor ovládat pomocí dynamické funkcionality poskytované editorem. Editor musí zvládat dostatečně rychle a plynule vykreslovat elementy stavového automatu.

Po vytvoření nebo úpravě automatů editor poskytne možnost automaty exportovat do souborů různých formátů a následný import z nich zpět do aplikace. Součástí je také automatické nebo ruční pozicování automatu pomocí pozicovacích algoritmů, které převede stavy a přechody automatu do maximálně čitelné podoby.

Tyto možnosti jsou navrženy tak, aby jejich rozšíření o další formáty či algoritmy bylo co nejjednodušší a nebylo nutno znát fungování celé aplikace, pouze těchto modulů.

Kapitola 2

Teorie

Pro správný návrh aplikace si je potřeba přesně ujasnit a definovat, co jsou to konečné automaty, které se budou s pomocí aplikace tvořit. Dále se rozeberou řešení a algoritmy vhodné pro pozicování vytvořených algoritmů. Nakonec se popíše programovací model, který je využit při programování aplikace.

2.1 Základní pojmy

Následující pojmy jsou převzaty z knihy Automaty a gramatiky [3].

Abeceda Abeceda je konečná množina symbolů, značíme ji σ nebo T (pro příklad množiny: {0, 1}, {a, b, c, d}).

2.1.1 Konečný automat

Konečný automat je výpočetní model, který se na základě vstupu (symbolu) ze vstupní abecedy přesune do dalšího stavu definovaného v přechodové funkci. Na počátku se automat nachází v počátečním stavu. Výstupem výpočtu je signalizace, jestli se automat nachází v konečném stavu.

Konečný automat M je pětice M = (Q, Σ, δ, q_0, F), kde

- Q je konečná neprázdná množina stavů,
- Σ je konečná vstupní abeceda,
- δ je přechodová funkce,
- $q_0 \in Q$ je počáteční stav,
- $F \subset Q$ je množina koncových stavů.

2.1.2 Deterministický konečný automat

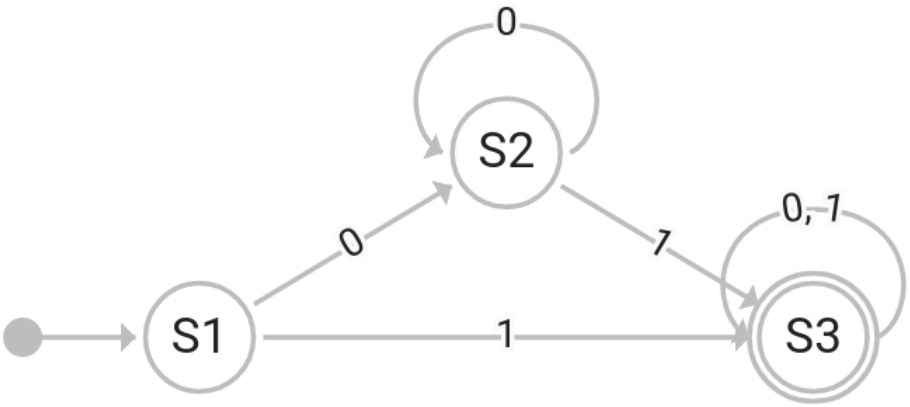
Deterministický konečný automat má vždy jasně dáno v přechodové funkci, jak výpočet pokračuje.

Přechodová funkce δ je zobrazení z množiny $Q \times \Sigma$ do množiny stavů Q, neboli δ: $Q \times \Sigma \to Q$.

Obrázek 2.1: Deterministický konečný automat

Přechodová funkce nemusí být jasně definována pro všechny dvojice z množin Q a Σ. Pokud automat dostane na vstup symbol, pro který nemá definovaný přechod, končí výpočet s chybou.

Automat, který má definovaný přechod pro všechny dvojice Q a Σ, se nazývá **úplně určený deterministický konečný automat.**

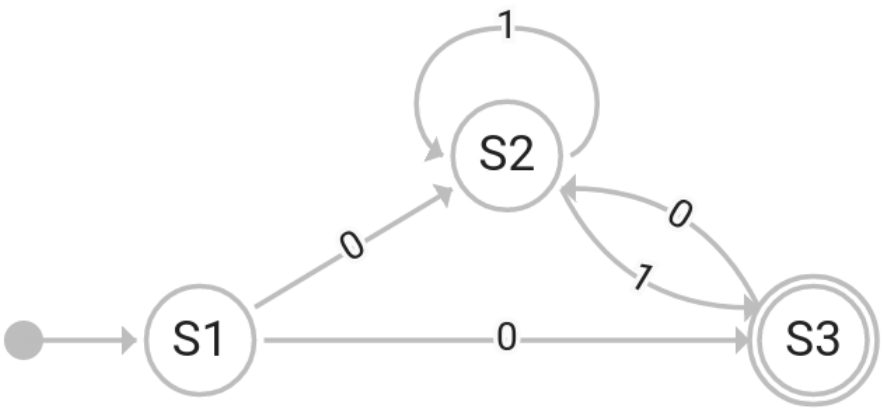


Obrázek 2.2: Úplně určený deterministický konečný automat

2.1.3 Nedeterministický konečný automat

Nedeterministický konečný automat má přechodovou funkci δ takovou, že při výpočtu si lze vybrat, do jakého stavu se přejde. Například pokud je automat ve stavu S_1, tak při vstupním symbolu 1 může přejít jak do stavu S_2, tak do stavu S_3.

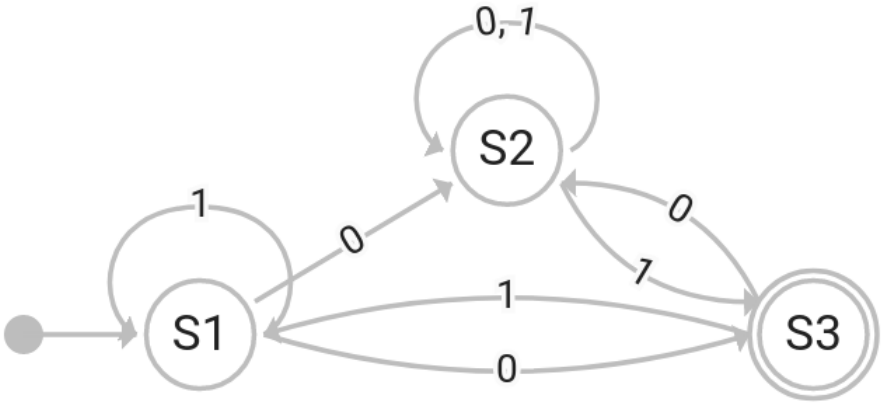
Přechodová funkce δ je zobrazení z množiny $Q \times \Sigma$ do množiny všech podmnožin množiny stavů Q, neboli δ: $Q \times \Sigma \to 2^Q$.



Obrázek 2.3: Nedeterministický konečný automat

Opět jako u deterministického konečného automatu platí, že přechodová funkce δ nemusí být definována pro všechny dvojice z Q a Σ. Výsledek pro tuto dvojici je pak prázdná množina stavů, do kterých se lze přesunout.

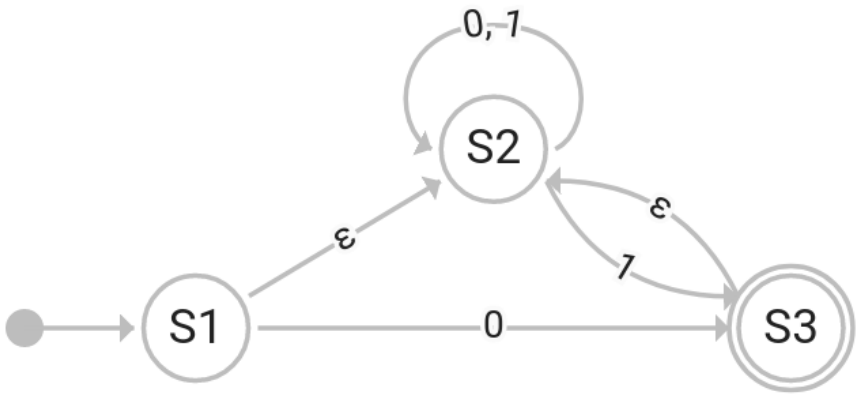
Automat, který má definovaný přechod pro všechny dvojice Q a Σ, se nazývá **úplně určený nedeterministický konečný automat.**



Obrázek 2.4: Úplně určený nedeterministický konečný automat

Nedeterministický konečný automat s ε-přechody

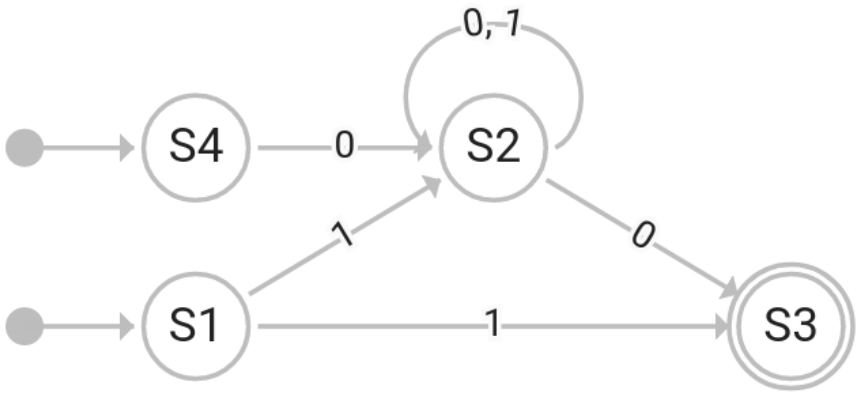
Pokud se přechodová funkce definuje jako: δ: $Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$, povolí se v automatu přechody, kde automat nečte na vstupu žádné symboly a automaticky přejde do jiného stavu. Takový automat se nazývá nedeterministický konečný automat s ε-přechody.



Obrázek 2.5: Nedeterministický konečný automat s ε-přechody

Nedeterministický konečný automat s více počátečními stavy

Místo počátečního stavu q_0 můžeme u automatu definovat množinu počátečních stavů I. Výpočet pak může začít ve více stavech. Takový automat se nazývá nedeterministický konečný automat s více počátečními stavy.



Obrázek 2.6: Nedeterministický konečný automat s více počátečními stavy

2.2 Pozicovací algoritmy

Konečné automaty zobrazené v aplikaci bude možno pozicovat pomocí algoritmů. Pozicovacích algoritmů existuje velký výběr a je potřeba si ujasnit jakých parametrů z výsledného napozicování algoritmem je vyžadováno pro pozicování stavových automatů. Takové algoritmy se soustředí na kompaktní vykreslení automatu, zobrazení jasné struktury a zamezení nadbytečnému křížení přechodů.

Pro účely této práce se vybraly dva algoritmy, každý z nich se soustředí na odlišnou prezentaci grafu.

2.2.1 Hierarchical

Algoritmus hierarchical se využívá pro kreslení hierarchické struktury orientovaných grafů. Vrcholy jsou uspořádány do vrstev. Což znamená, že hrany grafu směřují převážně jedním směrem po směru vrstev hierarchie.

Nejpopulárnější metodou je metoda Sugiyamova [4]. Většina prozkoumaných nástrojů využívá právě tuto metodu. Metoda dle [5] definuje pravidla pro kreslení estetických grafů:

- Hrany by měly směřovat stejným směrem.
- Krátké hrany jsou čitelnější.
- Rovnoměrně rozprostřené vrcholy zabraňují nepřehlednosti.
- Křížící se hrany znepřehledňují graf.
- Rovné hrany jsou čitelnější.

Algoritmus provádí několik obecných kroků:

1. Odstranění cyklů.
2. Přiřazení vrstev.
3. Minimalizace křížení hran.
4. Přiřazení pozic.

Každý z těchto oddělených kroků je možné řešit různými algoritmy, které se liší ve složitosti. V další kapitole bude popsána jejich volba pro tuto práci.

2.2.2 Force-directed

Algoritmus force-directed je určený pro vykreslování neorientovaných grafů. Byl vybrán z důvodu, že dobře znázorňuje jednotlivé komponenty grafu, takže sice nepracuje s orientovanými hranami, ale pokud má stavový automat více logických komponentů, jsou viditelně oddělené. Pro účely práce se používá

algoritmus Fruchtermana a Reingolda [2.1]. Vrcholy na sebe působí silami podobnými pružinám. Vrcholy se všechny navzájem odpuzují a vrcholy které mají mezi sebou hranu se přitahují. Rozšířený algoritmus ještě přichází s myšlenkou ochlazování sil, kdy tyto síly pomalu ztrácí na intenzitě každou iteraci. [6]

Tento algoritmus je vhodný pro menší grafy (v rámci stovek vrcholů), což pro rámec práce odpovídá. Standardní algoritmy pracují v euklidovském prostoru, existují i rozšířené algoritmy, které umí pracovat s velkými grafy. [6]

Algoritmus 2.1 Fruchterman and Reingold

$area \leftarrow W * H$ ▷ W a H je šířka a výška okna
$G \leftarrow (V, E)$ ▷ vrcholům je přiřazena náhodná počáteční pozice
$k \leftarrow \sqrt{\frac{area}{|V|}}$
function $f_a(x)$
    return $\frac{x^2}{k}$
function $f_r(x)$
    return $\frac{k^2}{x}$
for $i = 1$ **to** $iterations$ **do**
    for u **in** V **do** ▷ vypočte odpuzující síly
        $v.disp \leftarrow 0$ ▷ každý vrchol má dva vektory: `.pos` a `.disp`
        for v **in** V **do**
            if $u \neq v$ **then**
                $\delta \leftarrow v.pos - u.pos$ ▷ rozdíl mezi pozicemi vektorů
                $v.disp \leftarrow v.disp + (\frac{\delta}{|\delta|}) * f_r(|\delta|)$
    for e **in** E **do** ▷ vypočte přitažlivé síly
        $\delta \leftarrow e.v.pos - e.u.pos$
        $e.v.disp \leftarrow e.v.disp - (\frac{\delta}{|\delta|}) * f_a(|\delta|)$
        $e.u.disp \leftarrow e.u.disp + (\frac{\delta}{|\delta|}) * f_a(|\delta|)$
    for v **in** V **do** ▷ limituje maximální přemístění na teplotu t a zabraňuje napozicování mimo okno
        $v.pos \leftarrow v.pos + (v.disp/|v.disp|) * min(v.disp, t)$
        $v.pos.x \leftarrow min(\frac{W}{2}, max(-\frac{W}{2}, v.pos.x))$
        $v.pos.y \leftarrow min(\frac{H}{2}, max(-\frac{H}{2}, v.pos.y))$
    $t \leftarrow cool(t)$ ▷ snižuje teplotu, rozmístění se přibližuje lepšímu sestavení

2.3 Funkcionální programování

Funkcionální programování je styl programování využívající pro tvorbu programu „čisté“ funkce, které nepracují se vnějším stavem a nemění přímo data. Kód programu je deklarativní. Funkcemi definujeme chování, kde funkce na základě parametrů provede výpočet, který není vázán na žádný vnější stav mimo funkci a vždy vrátí stejný výsledek. Pokud funkce manipuluje s objekty, tak místo změny daného objektu vytvoří nový objekt, který je složen ze změn a původního objektu. Program se tvoří skládáním jednotlivých funkcí. [7]

Tento styl programování má několik výhod. Výstup kódu je vždy jasně daný a nemůže se stát, že by ho ovlivnily nějaké vnější okolnosti. Také lze kód jednoduše rozdělit na malé moduly, které zprostředkovávají jednoduchou funkcionalitu. To pak umožňuje častější znovuvyužití některých funkcí a také jednoduché testování.

JavaScript není přímo funkcionální jazyk, ale poskytuje mnoho nástrojů, které dovolují kód psát ve funkcionálním stylu. Převážně v posledních letech se podpora s příchodem novějších verzí JavaScriptu rozšiřovala. Funkce se dají definovat do proměnných a v rozšíření ES6 [8] s doplněním blokového kontextu a anonymních funkcí se přístup zlepšil. TypeScript pak umožnuje přesně definovat vstupy a výstupy funkcí.

Práce většinou využívá funkcionálního programování, pokud není potřeba manipulovat se vnějším stavem, což je většinou dáno architekturou webu nebo použitých knihoven, například u zpracování uživatelských událostí.

Kapitola 3

Analýza

Pro správný návrh aplikace je potřeba podrobněji ujasnit cíle, které musí výsledná aplikace splňovat. Před tím je vhodné prozkoumat řešení podobných problémů a seznámit se s problematikou implementace vybrané funkcionality v již existujících nástrojích. Následně se objasní konkrétní požadavky na jednotlivé části editoru a popíše se způsob, jak by měl uživatel s aplikací zacházet.

3.1 Podobné práce

V rámci rešerše bylo prozkoumáno několik bakalářských a diplomových prací, které se zabývaly podobným problémem vykreslování grafů nebo manipulací se stavovými automaty.

3.1.1 Grafický editor konečných automatů a kaskády

Diplomová práce popisující webový grafický editor kaskád a stavových automatů. Zaměřuje se na pozicování entit na pracovní ploše pro maximální přehlednost. K tomu využívá moderních webových technologií a zároveň se snaží být nezávislá na externích knihovnách. [9]

Práce je velmi podobná v cílech, které si klade, této práci. Je použit jazyk JavaScript a s ním knihovna jQuery [10]. Aplikace používá DND přístup tvorby elementů a podporuje historii akcí. Nicméně někdy volí velmi rozdílné přístupy k jejich dosažení. Vizualizaci elementů provádí v HTML elementu canvas na rozdíl od SVG.

Přístup práce k automatickému pozicování není příliš estetický a optimální. Soustředí se převážně na pozicování přechodů, kdy si přechody hledají na pracovní ploše cestu k cílovým stavům s co nejmenším počtem křivek. Bohužel přístup nebude nejspíše ideální řešení pro kreslení automatů, protože ve výsledku vznikají velmi vlnité cesty, které jsou často delší než by musely být. Tento

výsledek také upevnil rozhodnutí autora této práce zvolit pozicovací algoritmy, které pozicují pouze stavy a přechody se přímo nepozicují.

3.1.2 Vizualizace konečných automatů

Bakalářská práce popisující tvorbu grafického programu pro vizualizaci konečných automatů. Program podporuje základní tvorbu automatů – přidání, přejmenování a odebrání stavů nebo přechodů. Zaměřuje se především na vizualizaci algoritmů pracujících s konečnými automaty. Umožňuje například odstranění epsilon přechodů, determinaci, sjednocení, průnik nebo zřetězení. [11]

Výsledkem je jednoduchá aplikace fungující jako Java applet [12], která umožnuje jak samotnou tvorbu automatů, tak její import a export mimo aplikaci v podobě gramatiky.

Většina práce se soustředí na implementaci algoritmů pro konečné automaty a samotnou vizualizaci automatu, pozicování nebo export přenechává externím knihovnám. Proto hlavním užitkem, který přinesla pro tuto práci, byl především rozbor ostatních řešení editorů stavových automatů nebo grafů. Autor odkazované práce využil Java knihovny JUNG [13], která zprostředkovává vykreslování a pozicování.

3.1.3 Vektorový grafický editor v prostředí webového klienta

Bakalářská práce popisující tvorbu webového vektorového editoru, umožňujícího tvořit jednoduchou vektorovou grafiku v prohlížeči. Dále rozebírá technologii SVG a její přínosy. Porovnává různá řešení již existujících vektorových editorů. Také se snaží, aby byla práce snadno rozšiřitelná. [14]

Výsledkem je editor, který užívá technologie SVG a JavaScript. Editor je snadno použitelný jako modul do aplikací.

Tato práce řeší převážně obecnou tvorbu vektorové grafiky, a ne přímo automaty či grafy. Nicméně dobře popisuje postup tvorby uživatelského prostředí a také práci s SVG. Také používá Bézierovy křivky, které jsou využity v této práci.

Aplikace pro práci s SVG používá knihovnu Raphaël [15], která obstarává deklaraci SVG elementů a nabízí pomocné funkce pro práci s nimi. Pro uživatelské prostředí je zvolena knihovna jQueryUI [16].

3.1.4 Webový nástroj pro kreslení UML diagramů tříd

Bakalářská práce zabývající se tvorbou webového nástroje pro tvorbu UML diagramů. Využívá webové technologie SVG a HTML5, je napsána v JavaScriptu, kde používá moderní prvky tohoto jazyka. [17]

Práce má jen slabá podobnost s touto prací, ale využívá knihovny React, manipuluje s elementy v SVG a přináší jiný pohled na téma.

3.1.5 Nástroj pro vizualizaci a kreslení konečných automatů

Bakalářská práce, která popisuje tvorbu nástroje pro vytváření konečných automatů. Tvorba automatů probíhá v grafickém a také v textovém režimu. Grafický režim pracuje s podrežimy, které umožňují přidávat, upravovat nebo mazat elementy. Textový režim podporuje psát automat v nástrojové liště, kdy se změny automaticky zobrazují v grafické podobě. Automat je pozicován pomocí jednoduchého algoritmu. Aplikace podporuje ukládání a nahrávání mimo nástroj. [18]

Nástroj je navrhnut jako Java applet, který používá k vizualizaci HTML a JavaScript.

Rozdíl v řešení je především v použití více režimů kreslení grafu. Tato práce volí jednodušší způsob uživatelské interakce, protože využívat více režimů k úpravám může být matoucí a nejspíše také pomalejší způsob. Dále se neklade důraz na rozšíření pozicování a import/export.

3.1.6 Grafická reprezentace grafů

Diplomová práce zabývající se především problémem rozmisťování uzlů grafu na ploše. Probírá několik řešení od jednoduchého uspořádání do kruhu, až po složitější pružinový algoritmus. Nakonec je vyvinuto rozšířené řešení pružinového algoritmu. Jako vstup a výstup nástroje je použito XML. [19]

Tato práce blíže rozvádí nebo hodnotí některé jiné algoritmy použité k pozicování.

3.1.7 Shrnutí

Většina jmenovaných prací řeší pouze část požadavků této práce nebo používá externí knihovny k jejich řešení. Navíc pokud se jedná o pozicování, tak se nezabývají adekvátními algoritmy vhodnými k tomuto účelu nebo implementují pouze triviální řešení.

V rámci této práce byla snaha minimalizovat závislost na externích knihovnách k dosažení snadné rozšiřitelnosti, zmenšení komplexnosti pokrytého kódu a také k minimalizaci velikosti aplikace.

U většiny autorů je shoda, že je třeba klást důraz na jednoduchost uživatelského prostředí. Je vidět i trend velké pracovní plochy a malé lišty nástrojů, která nezabírá příliš místa. Také je často použit DND přístup. Těmito preferencemi se shodují s přístupem zvoleným v této práci.

3.2 Existující nástroje

Pro inspiraci a lepší zanalyzování možných řešení se zkoumaly přístupy ostatních editorů grafů, a to webových i nativních. Většina popsaných řešení má své výhody a nevýhody, které jsou popsané u jednotlivých aplikací. Snahou bylo

odnést si jejich nejlepší vlastnosti, a pokud jsou užitečné, použít je ve výsledné práci.

Graphviz

Komplexní grafový vizualizační nástroj. Nabízí velkou škálu formátů grafů, kde konečné automaty jsou jen malou částí grafů, kterou umožňuje tvořit. Podporuje mnoho výstupních formátů, mimo jiné i SVG, obrázky nebo PDF. Nabízí bohaté možnosti úpravy vzhledu grafů. [20]

Důležitou součástí je i pozicování pomocí propracovaných algoritmů, které jsou vyvíjeny opensource komunitou. Příklad je nástroj DOT [21], který je často zmiňován v odkazech v pracích pojednávajících o pozicování stavů.

Dagre

Klientský JavaScript nástroj pro pozicování grafu. Podporuje pouze pozicování, ale existují nadstavby, které slouží k vizualizaci. [22]

Rappid

Webový framework pro tvorbu diagramů, používající HTML5 a SVG. Poskytuje mnoho různých formátů diagramů. Vyžaduje zakoupení. [23]

nomnoml

Webový nástroj pro kreslení UML diagramů. Vykresluje a automaticky pozicuje diagramy podle textové definice. [24]

Canviz

JavaScript knihovna pro vykreslování graphviz generovaných grafů v prohlížeči v canvasu. Slouží pouze k vizualizaci. [25]

JFLAP

Java balíček sloužící k tvorbě konečných automatů. Nepodporuje automatické pozicování. Nemá webovou podobu. [26]

Automata editor

Vektorový editor konečných automatů, který podporuje řadu výstupních formátů (SVG, obrázky, GraphML, ...). Program je napsán v jazyce C a používá OpenGL pro renderování. [27]

yEd Graph Editor

Další desktopová aplikace pro práci s diagramy, podporuje automatické pozicování a export. Nezaměřuje se přímo na automaty. [28]

DRAW.IO, Lucidchart, gliffy

Webové editory diagramů. [29, 30, 31]

3.3 Požadavky

Požadavky aplikace se odvíjí od zadání práce. Je potřeba definovat veškerou funkcionalitu, od které se bude odvíjet další analýza, návrh a realizace.

Při vývoji editoru se využije moderních webových funkcionalit a knihoven, které zjednoduší implementaci požadavků. Bude kladen důraz na uživatelskou přívětivost a jednoduché používaní. Grafické prostředí bude co nejjednodušší, kdy hlavním elementem je tvořený automat. Ovládací prvky nemusí být výrazné, avšak nesmí se obětovat srozumitelnost, což by mohlo mást nové uživatele. Jako pomoc k ovládání automatu se bude zobrazovat dynamická nápověda na pracovní ploše podle právě prováděné akce. Pro zkušené uživatele se zpřístupní ovládání, které nezpomaluje ruční generování automatů.

3.3.1 Funkční požadavky

Popisují všechny parametry editoru, které musí být pokryty a editorem zpřístupněny uživateli.

1. F1 – Umístění stavů na pracovní ploše a jejich smazání.
2. F2 – Spojení dvou stavů (i sebe samého) přechodem a jeho smazání.
3. F3 – Přejmenování přechodů a stavů.
4. F4 – Označení stavu za počáteční a koncový.
5. F5 – Automatické pozicování.
6. F6 – Uložení a nahrání automatu.
7. F7 – Import a export automatu v různých formátech.

3.3.2 Nefunkční požadavky

Popisují technickou stránku editoru.

1. Využití moderních webových technologií.
2. Podpora moderních webových prohlížečů (Chrome, Firefox).
3. Plynulá práce v editoru.

4. Snadné doplnění nových formátů pro import a export.
5. Snadné doplnění pozicovacích algoritmů.

3.4 Případy užití

Definování případů užití pomáhá s konkrétním návrhem aplikace. Popis případů detailně rozebírá, jak bude uživatel s aplikací pracovat, a podle tohoto popisu bude navrženo uživatelské prostředí a chování aplikace.

Následuje výčet důležitých případů užití a jejich popis.

- UC1 – Přidání nového stavu.
- UC2 – Přidání nového přechodu.
- UC3 – Označení stavu nebo přechodu.
- UC4 – Úprava stavu nebo přechodu.
- UC5 – Smazání stavu nebo přechodu.
- UC6 – Posunutí stavu.
- UC7 – Výběr pozicovacího algoritmu.
- UC8 – Napozicování aktuálního automatu.
- UC9 – Uložení aktuálního automatu.
- UC10 – Nahrání aktuálního automatu.
- UC11 – Export aktuálního automatu.
- UC12 – Import automatu.

3.4.1 Uživatelské role

Editor je volně přístupný pro každého uživatele, který s ním pracuje. Všechna funkcionalita je odemčená a proto v rámci analýzy existuje jen jedna role – **uživatel**. Uživatel v editoru tvoří automat podle svých potřeb.

3.4.2 UC1 – Přidání nového stavu

Umožní uživateli v editoru přidat nový stav do automatu.

1. Uživatel klikne na tlačítko Add state v panelu nástrojů.
2. Uživatel přejede kurzorem na místo na pracovní ploše, kam chce stav umístit.
3. Uživatel pustí tlačítko myši.
4. Editor umístí nový stav s automaticky generovaným jménem na dané místo a zobrazí panel pro úpravu stavu.
5. Pokud chce uživatel upravit stav, postupuje jako v UC5 – úprava stavu nebo přechodu.

Obrázek 3.1: Případy užití

3.4.3 UC2 – Přidání nového přechodu

Umožní uživateli v editoru přidat nový přechod mezi stavy do automatu.

1. Uživatel najede kurzorem na stav na pracovní ploše, odkud přechod vychází.
2. Editor zobrazí u stavu tlačítko na přidání přechodu.
3. Uživatel klikne na tlačítko na přidání přechodu.
4. Uživatel přejede kurzorem na stav na pracovní ploše, ve kterém přechod končí (může končit i začínat ve stejném stavu).
5. Uživatel pustí tlačítko myši.
6. Editor umístí nový přechod s automaticky generovaným jménem mezi vybrané stavy a zobrazí panel pro úpravu přechodu.
7. Pokud chce uživatel upravit přechod, postupuje jako v UC5 – úprava stavu nebo přechodu.

3.4.4 UC3 – Označení stavu nebo přechodu

Umožní uživateli vybrat existující stav nebo přechod.

1. Uživatel najede kurzorem na stav nebo přechod na pracovní ploše.
2. Uživatel stiskne levé tlačítko myši.
3. Editor zobrazí panel pro úpravu stavu nebo přechodu.

3.4.5 UC4 – Úprava stavu nebo přechodu

Umožní uživateli upravit parametry existujícího stavu nebo přechodu.

1. Uživatel označí stav nebo přechod jako v UC3 – označení stavu nebo přechodu.
2. Pokud chce uživatel přejmenovat prvek, v okénku se jménem vyplní nové jméno. Pokud chce označit stav jako počáteční nebo koncový, tak klikne na přepínací tlačítko s odpovídajícím jménem.
3. Editor automaticky uloží úpravy.

3.4.6 UC5 – Smazání stavu nebo přechodu

Umožní uživateli smazat existující stav nebo přechod.

1. Uživatel označí stav nebo přechod jako v UC3 – označení stavu nebo přechodu.
2. Uživatel zmáčkne klávesu Del nebo klikne na tlačítko Smazat.
3. Editor prvek odznačí a smaže.

3.4.7 UC6 – Posunutí stavu

Umožní uživateli přesunout existující stav na pracovní ploše.

1. Uživatel najede kurzorem na vybraný stav.
2. Uživatel stiskne levé tlačítko myši.
3. Uživatel pohne kurzorem na vybrané místo.
4. Editor zobrazuje označený stav na místě kurzoru, pokud do stavu vedou přechody, tak je také automaticky přesouvá.
5. Uživatel pustí levé tlačítko myši.

3.4.8 UC7 – Výběr pozicovacího algoritmu

Umožní uživateli vybrat algoritmus, kterým se automaticky pozicuje automat.

1. Uživatel klikne na tlačítko výběr pozicovacího algoritmu.
2. Uživatel zvolí algoritmus ze seznamu a klikne na něj.
3. Editor uloží vybraný algoritmus.

3.4.9 UC8 – Napozicování aktuálního automatu

Umožní uživateli automaticky napozicovat automat na pracovní ploše.

1. Uživatel vybere pozicovací algoritmus pomocí UC8 – výběr pozicovacího algoritmu nebo ponechá stávající algoritmus.
2. Uživatel klikne na tlačítko Automaticky napozicovat.
3. Editor automaticky napozicuje stavový automat podle algoritmu a rozmístí stavy na pracovní ploše.

3.4.10 UC9 – Uložení aktuálního automatu

Umožní uživateli uložit rozpracovaný automat v rámci aplikace.

1. Uživatel klikne na tlačítko Save.
2. Editor uloží v rámci prohlížeče stavový automat z pracovní plochy.
3. Editor zobrazí hlášku o úspěšném uložení.

3.4.11 UC10 – Nahrání aktuálního automatu

Umožní uživateli nahrát dříve rozpracovaný automat v rámci aplikace.

1. Uživatel klikne na tlačítko Load.
2. Editor nahraje stavový automat z prohlížeče a rozmístí ho podle uložených dat na pracovní plochu.

3.4.12 UC11 – Export aktuálního automatu

Umožní uživateli exportovat stavový automat do souboru a stáhnout ho.

1. Uživatel klikne na tlačítko Export.
2. Editor zobrazí nabídku možných formátů uložení automatu.
3. Uživatel klikne na zvolený formát.
4. Editor vygeneruje soubor ve zvoleném formátu pošle soubor uživateli.

3.4.13 UC12 – Import automatu

Umožní uživateli importovat stavový automat ze souboru.

1. Uživatel klikne na tlačítko Import.
2. Editor zobrazí nabídku možných formátů automatu.
3. Uživatel klikne na zvolený formát.
4. Editor zvaliduje data podle zvoleného formátu. Pokud není soubor validní, zobrazí chyby ve formátu uživateli. Jinak pokračuje dále.
5. Editor napozicuje automat podle zvoleného algoritmu z UC8 – výběr pozicovacího algoritmu.
6. Editor zobrazí importovaný automat na pracovní ploše.

3.5 Pokrytí požadavků

Následující tabulka slouží k prozkoumání, jestli definované případy užití pokrývají všechny funkční požadavky.

	F1 – Umístění a smazání stavů	F2 – Spojení stavů přechodem a smazání	F3 – Přejmenování přechodů a stavů	F4 – Označení stavu za počáteční a koncový	F5 – Automatické pozicování	F6 – Uložení a nahrání automatu	F7 – Import a export automatu
UC1 – Přidání nového stavu.	x						
UC2 – Přidání nového přechodu.		x					
UC3 – Označení stavu nebo přechodu.			x	x			
UC4 – Úprava stavu nebo přechodu.			x	x			
UC5 – Smazání stavu nebo přechodu.	x	x					
UC6 – Posunutí stavu.	x						
UC7 – Výběr pozicovacího algoritmu.					x		
UC8 – Napozicování aktuálního automatu.					x		
UC9 – Uložení aktuálního automatu.						x	
UC10 – Nahrání aktuálního automatu.						x	
UC11 – Export aktuálního automatu.							x
UC12 – Import automatu.							x

Tabulka 3.1: Pokrytí funkčních požadavků

Kapitola 4

Návrh

Před implementací editoru je důležité popsat jednotlivé části aplikace, jejich podobu a jak spolu budou části vzájemně komunikovat. Editor je možné funkčně rozdělit na několik samostatných částí – správu vnitřních dat, interakce s uživatelem, komunikace uživatelského rozhraní s vnitřními daty, export dat mimo systém a zpět a nakonec pozicování.

4.1 Použité technologie

Pro návrh aplikace byla zvážena řada technologií, které by byly vhodné pro implementaci požadavků. Nakonec byly zvoleny moderní technologie, které se v době vypracování bakalářské práce stále ještě aktivně vyvíjely. Editor slouží i jako ukázka, jak dané technologie fungují ve složitější aplikaci, a ukazuje, jak implementovat jejich integraci.

Jako programovací jazyk byl zvolen TypeScript, který nabízí oproti tradičnímu JavaScriptu několik rozšíření, oba jazyky jsou vhodné pro vypracování interaktivního uživatelského prostředí. Jednotlivé výhody jsou blíže popsány v dalších částech.

Aplikace je vyvíjena ve frameworku React, který už sice není několik let přímo novou technologií a je běžně užíván ve webových aplikacích [2], ale v rámci práce byla využita jeho nová funkcionalita, resp. React Hooks, která procházela vývojem ještě při tvorbě editoru.

Pro reprezentaci stavového automatu na pracovní ploše aplikace je zvolena technologie SVG, která umožňuje deklarativně zobrazovat jednotlivé prvky.

Technologie jsou blíže popsány dále v této kapitole. Vysvětlí se důvod jejich volby, jak se v práci používají a co umožňují.

4.1.1 Typescript

Pro vytvoření webové aplikace existuje několik vhodných jazyků. Dvěma hlavními kandidáty jsou JavaScript a PHP. Jelikož editor automatů by měl být

dynamická aplikace, která zpracovává události vzniklé uživatelskou interakcí, je vhodné použít JavaScript. [32]

Samotný JavaScript by určitě byl dostatečným nástrojem pro vytvoření editoru, ale existují různé nadstavby tohoto jazyka, které obohacují vývojářskou funkcionalitu. Jedna z takových nadstaveb je jazyk TypeScript.

TypeScript je nový jazyk vyvíjený společností Microsoft a zejména v posledních letech se mu dopřává vyšší a vyšší popularity. [2] Zásluhu na tom má určitě rostoucí zájem o vývoj aplikací pro webovou platformu. Webové aplikace nabývají na komplexnosti a občas JavaScript v určitých oblastech pokulhává ve funkcionalitě. Jednou z takových oblastí je určitě statické typování proměnných. JavaScript je dynamicky typovaný jazyk, což je někdy výhoda, hlavně třeba pokud se vývoj soustředí na velmi rychlou implementaci, ale ve větších a komplexních aplikacích, kdy na sebe navazují a komunikují spolu různé systémy vzniká prostor pro chyby. TypeScript je automaticky řeší. [33]

Další z obohacujících funkcionalit TypeScriptu jsou rozšíření tříd (celkově objektově orientovaného programování), dekorátory a generiky. [34]

Přímo TypeScript není možné v současné době používat nativně v prohlížeči, ale jeho velkou výhodou je, že veškerý kód lze jednoduše zkompilovat do JavaSriptu. TypeScript kompilátor automaticky vygeneruje JavaScript soubory se stejnou funkcionalitou, které akorát ztrácí informace o typech. Výsledný zkompilovaný kód už prohlížeč spustí.

Pro účely práce se využije hlavně statického typování, umožňující napsání dobře čitelného funckionálního kódu. JavaScript v posledních letech prochází rychlým vývojem oproti minulým letům [35]. Aktualizace jazyka přidávají funkcionalitu, která je pro funkcionální typ programování vhodná (například uložení definice funkce do proměnné). TypeScript zabezpečuje funkcionální kód v rámci typů funkcí a definuje přesně, co musí mít funkce za parametry a jakých návratových hodnot nabývají.

4.1.2 React

React je jedna z moderních knihoven a frameworků (dále ještě Angular [36] a Vue [37]), které umožnily novodobý rozkvět webových aplikací. React, vyvíjený Facebookem, je pouze knihovnou, dokonce relativně malou oproti ostatním možnostem. [38]

Velkým problémem při vykreslování HTML elementů je rychlost jejich překreslování [39]. React přichází s řešením, které nazývá Virtual DOM. Jak název napovídá, jedná se o virtuální DOM [40] uloženém v paměti prohlížeče, ve kterém probíhají veškeré manipulace s elementy a výpočty aplikace. React pak sám výsledky z Virtual DOM vykresluje do DOM a provádí vlastní optimalizaci vykreslování. Výsledkem je možnost vytvářet rozsáhlé dynamické aplikace v měřítku dříve jen stěží možném při použití čistého JavaScriptu.

React aplikace je postavená z jednotlivých komponentů [41], které představují chytré virtuální HTML elementy. Komponentům lze specifikovat argumenty

„props“, které může využívat při jeho vykreslování. Komponent samozřejmě může vykreslovat další klasické HTML elementy nebo React komponenty. Jednotlivé komponenty by měly být kompaktní a neřešit příliš funkcionality najednou. Zvyklostí je rozdělit aplikaci na jednoduché komponenty, každý obstarává úzkou a specifickou část funkcionality.

Populární metodou tvorby komponentů je rozdělení na „chytré“ a „hloupé“ komponenty [42]. Hloupé komponenty vykreslují pouze obsah na základě props. Chytré komponenty odebírají vnitřní stav aplikace, provádějí výpočty a řeší komplikovanější logiku aplikace.

Důležitou součástí Reactu je správa vnitřního stavu aplikace. React využívá architektury Flux [43], která funguje na základě jednosměrného odebírání změn stavu a jejich zpracování a zobrazení výsledku v komponentech. React nabízí více možností, jak využívat stav aplikace. Jednou z nich je lokální stav komponentů, kde je v rámci komponentu možno stav měnit, například uživatelskou interakcí, a také je možné ho předávat „child“ komponentům (komponenty definované v komponentu, dále nazývané děti). Ve větších aplikacích s touto možností vznikají problémy předávání spousty stavových argumentů mnoha dětem, proto se využívá převážně jen pro lokální stav v aplikaci. Pro stav, který je používaný více komponenty aplikace, je možné využít Context API [44]. Context je v podstatě objekt, který lze nadefinovat v některém z rodičovským komponentů. Místo předávání argumentů postupně po dětech se komponenty přihlásí k potřebné části Contextu s odebírají změny samostatně.

Hooks

Při úvodní vlně popularity Reactu se hojně využívaly třídy k definici komponentů. Nejnovější vývoj se přesouvá k využití funkcí. Už delší dobu existují bezestavové funkcionální komponenty, což jsou pouze jednoduché funkce přijímající `props` a které se využívají převážně pro již dříve zmíněné „hloupé“ komponenty. Jejich výhodou je jednoduchá možnost optimalizace. [45]

V době vzniku práce přišel React vývojářský tým s dalším způsobem využití funkcionálních komponentů. Nový model pojmenovávají Hooks [46]. Jedná se o funkce poskytující stejnou funkcionalitu, jako klasické React komponenty definované třídami. Odpadá tak nutnost třídy používat pro komponenty, které pracují se stavem. Jelikož se jedná o velice novou technologii, výhody přístupu nejsou ještě přesně otestovány. Nicméně existují jednotlivé zátěžové testy, ukazující, že využití Hooks namísto tříd zlepšuje rychlost testované aplikace [47].

Pro využití práce se zvolily Hooks, které přešly v časovém rozmezí tvorby práce z alfa verze do regulární funkcionality React. Podle dřívější zkušenosti autora s knihovnou React hodnotí Hooks pozitivně, jak z pohledu většího komfortu a jednoduchosti kódu, tak z předběžného dojmu, že výsledné aplikace jsou rychlejší nebo alespoň Hooks podporuje takovou architekturu aplikace, která automaticky vede k tvorbě lépe zoptimalizovaného kódu.

Redux

Pro jednoduché aplikace je standardní způsob správy stavu v React dostačující, má však své limity. Problémy se začínají projevovat, pokud má aplikace mnoho komponentů připojených ke Contextu. Například pokud aplikace aktualizuje Context příliš často nebo pokud je vnitřní stav příliš hluboký objekt. Část těchto problému je popsaná dále v práci, kde se popisuje architektura vnitřního stavu.

S řešením pro zmíněné problémy přichází knihovna Redux [48], která zprostředkovává vnitřní stav a s ní spojená knihovna React-redux [49], která propojuje stav s React komponenty.

Knihovna funguje na principu podobném architektuře Flux, ale nabízí další optimalizace a nástroje pro manipulaci se stavem. Také ve své dokumentaci specifikuje přesná praktika, jak se stavem zacházet a pomáhá tak k dosažení lepšího výkonu.

Redux využívá globálního stavu, což je jednoduchý objekt, jehož podobu si nadefinuje vývojář. Ke stavu, či spíše ke specifickým částem, se přihlásí komponenty k odběru a zpracovávají jeho hodnoty. Pro změnu stavu se používají akce, které vyvolávají komponenty. Akce jsou zpracovány reducery, podle kterých se vytvoří nová podoba stavu na základě akce. Stav, ke kterému jsou komponenty přihlášeny, se automaticky aktualizuje. [50]

Redux je velmi využívaná knihovna [51] se spojením s React, a proto má mnoho rozšiřujících knihoven vytvořených komunitou. [52] Jednou z nich je rozšíření prohlížeče Dev tools [53], které zobrazuje současnou podobu stavu se všemi předchozími akcemi a s možností si prohlédnout zpětnou podobu stavu. Knihovna byla využita při vývoji aplikace a to převážně pro debuggování. Další knihovna je redux-undo [54], která podporuje vracení se zpět v rámci vnitřního stavu přímo v aplikaci, je to umožněno přesnou definicí akcí a jejich vlivem na stav.

Těchto knihoven se využívá pro veškerou správu vnitřního stavu výsledné aplikace, s výjimkou lokálních výjimek.

4.1.3 SVG

Před tvorbou uživatelského prostředí bylo potřeba rozhodnout způsob vykreslování pracovní plochy a prvků na ní. HTML5 nabízí dvě možnosti – canvas a SVG.

Canvas je HTML element zapracovávající vykreslování bitmapové grafiky. Vykresluje prvky a změny v nich v programovém cyklu, kdy se prvky renderují. [55]

Zatímco u SVG jsou elementy deklarovány pomocí XML, jejich vykreslování obstarává prohlížeč. Navíc se používá vektorová grafika. Další výhodou SVG je podpora událostí elementy, kdy se může využít naslouchání na kliknutí SVG elementu v rámci aplikace. SVG také podporuje CSS stylování, což

umožní indikaci uživatelských akcí. Pomocí vektorů lze dosáhnout snadného přibližování plochy, aniž by se snížila kvalita jednotlivých elementů. Nevýhodou je snížení výkonu, když SVG začne obsahovat přes 1000 elementů [56], což by v rámci editoru nebyl problém. [57]

SVG umožní editoru provádět veškeré výpočty mimo konkrétní elementy, které budou jen odebírat jejich aktuální parametry. Nakonec se autor podle všech vyjmenovaných výhod rozhodl využít SVG.

Původně se očekávalo, že se v rámci aplikace nebude využívat přímo SVG, ale knihovna, která by SVG elementy vytvářela podle poskytnutých parametrů programově. Nakonec se ukázalo, že SVG je velmi přívětivá technologie a v rámci aplikace nebylo potřeba takových knihoven.

4.1.4 Ostatní knihovny

Material UI React komponenty založené na Material design od Google [58]

File saver Automatické stahování souborů [59]

4.2 Uživatelské prostředí

Uživatelské prostředí klade důraz na tři hlavní body.

1. Moderní a minimalistický vzhled.
2. Jednoduché a jasné ovládání.
3. Rychlá tvorba automatu.

Cílem aplikace je umožnit uživatelům tvořit stavové automaty. Je důležité, aby uživatelské prostředí nepřekáželo uživateli. Klade se důraz na minimalismus. Většinu okna zabírá pracovní plocha určená pro tvorbu automatů. Při návrhu nástrojové lišty se musel udělat kompromis mezi jasností a velikostí. Pro zvýšení jasnosti se na pozadí aplikace nachází nápověda, která slouží hlavně novým uživatelům.

S pomocí knihovny material-ui editor poskytuje podobu, která si bere příklad z Material Design od Google. Prostředí je dynamické a každou uživatelskou akci jasně indikuje.

Pro tvoření automatu se zvolil přístup Drag and drop [60]. Uživatelé přetahují stavy z nástrojové lišty a přechody z počátečních stavů do koncových. Elementy automatu je možné označit. Označeným elementům mohou být upraveny jejich parametry pomocí panelu, který se objeví u nástrojové lišty.

Mimo ovládání Drag and drop aplikace také poskytuje ovládání pomocí klávesových zkratek. Klávesové zkratky umožní pokročilému uživateli pohodlnější ovládání a také daleko rychlejší manipulaci s elementy. Zkratky pokrývají veškerou funkcionalitu ovládání.

4.2.1 Kreslení automatu

Jak už bylo zmíněno, ke kreslení se používá Drag and drop. Uživatel přetahuje stavy nebo přechody a tvoří automat. Také je uživateli poskytnuta možnost označit stav za koncový nebo počáteční.

Editor vykresluje automaty s využitím technologie SVG. Jednotlivé elementy automatu jsou jen primitivní objekty, které mají definované parametry, které sbírají z datového stavu aplikace.

Každý element je definován typem a id. React component se podle id s pomocí reduxu napojí na datový stav konkrétního elementu. Data pak může mírně poupravit k signalizaci stavu elementu a předá je SVG elementům reprezentující elementy automatu.

Stav

Je reprezentován kruhem. Jeho pozice je definována pomocí souřadnic `(x, y)`. Stav je možné posouvat pomocí DND na pracovní ploše. Při pozicování přejde stav do režimu `draggable`, poslouchá pozici kurzoru a automaticky se na ni přesouvá. Velikost stavu se odvíjí od jména. Po každé změně jména stavu nebo při iniciálním vykreslením se přečte velikost ohraničujícího kruhu textu a to se použije jako poloměr stavu.

Pro identifikaci počátečního a koncového stavu se používají speciální indikátory. Do počátečního stavu vede šipka a koncový stav je dvakrát ohraničen.

Stav může nabývat různých módů. Stav je indikuje podle zvýraznění ohraničení různými barvami. Zvýrazňuje se najetí myší na stav, výběr stavu a pokud je vybrán přechod, který vede do nebo z tohoto stavu.

Obrázek 4.1: Stav S1, počáteční stav S2 a koncový stav S3

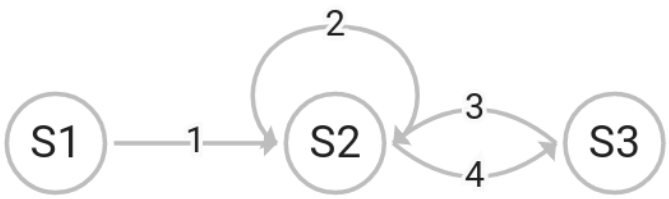
Speciálním typem stavu je pak `ghost` stav, který slouží jako pomocný indikátor, když uživatel vytváří nový stav. Tento stav je částečně průhledný a pokaždé je napozicován na souřadnice kurzoru. Pokud uživatel nezruší tvorbu stavu a stav umístí, pak se na místě vytvoří regulární stav a ghost stav zmizí.

Přechod

Přechody reprezentují orientované přímky, případně Bézierovy křivky [61]. Přechod vždy vychází ze stavu a míří do stavu. Jeho pozice je definována těmito dvěma stavy. Přechod uchovává id těchto stavů a při kreslení si je najde v jejich mapě ve vnitřním stavu aplikace. Počáteční a koncový bod přímky je spočítán z pozic stavů. Přechod míří přímo ze středu počátečního stavu do středu koncového stavu a začíná/končí na okraji stavů.

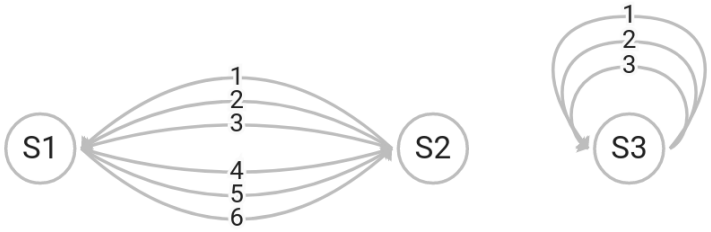
Název přechodu se zobrazuje uprostřed přímky a je pro větší zřetelnost ohraničen. Také jako u stavu se zvýrazňuje, pokud je na přechodu kurzor nebo pokud je zvolen. Koncový stav přechodu je indikován šipkou na přechodu.

Název přechodu je automaticky otáčen vzhledem k úhlu přechodové přímky, aby byl maximálně v rotaci 90 stupňů vůči vodorovné čáře. Dosahuje se tak lepší přehlednosti.



Obrázek 4.2: Přechod 1, přechod na sebe sama 2 a zakulacené přechody 3 a 4

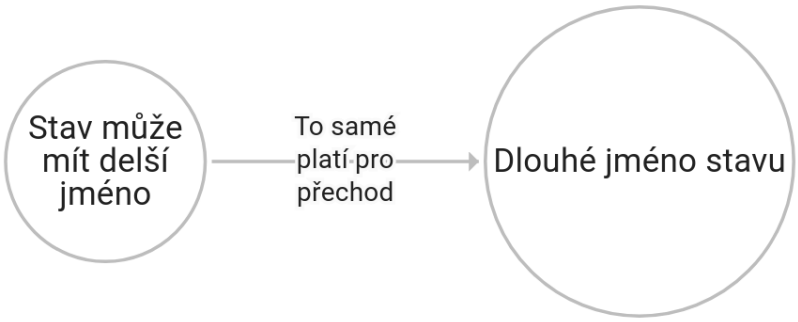
Přechod v několika případech místo přímky používá křivky. Když přechod vychází ze stejného stavu jako do kterého vede, pak přechod obchází stav z jedné strany na druhou a je zakulacen; dále pokud přechod vede do stavu, ze kterého vede zpětný přechod. V tomto případě by stavy splývaly, takže se přechod a zpětný přechod částečně zakulatí. Nakonec rozšíření předešlého případu je, když ze stavu vede do jiného stavu více různých přechodů, každý přechod se výrazněji zakulatí, aby byly přechody snadno rozlišitelné.



Obrázek 4.3: Mnoho přechodů mezi stavy S1 a S2, mnoho přechodů stavu S3 na sebe sama

Názvy

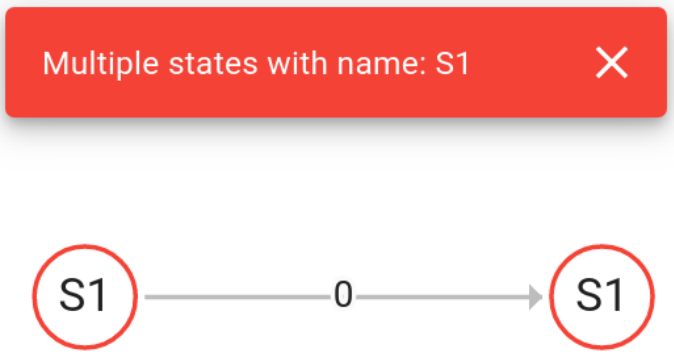
Editor podporuje možnost víceřádkových jmen elementů. Při editaci jména stačí jednoduše použít `Shift + Enter` pro odřádkování.



Obrázek 4.4: Vzhled víceřádkových jmen elementů aplikace

Chyby

V rámci tvorby automatu uživatelem editor při jednotlivých krocích částečně kontroluje jestli jsou elementy automatu validní. Pokud uživatel například definuje stav beze jména, editor zobrazí chybovou hlášku. Původně bylo předpokládáno, že funkcionalita bude kontrolovat kompletní validitu automatu a zobrazovat větší různorodost chyb. Nakonec se tato možnost projevila jako nadbytečná, předpokládá se, že uživatelé budou sami zodpovědní za validitu svých automatů. A pokud jim chyby nevadí, tak místo rušivých chybových hlášek ze nezobrazuje nic. V budoucnu je možnost, že by se zavedl samostatný modul na validaci automatu, který by automat jednorázově kontroloval.



Obrázek 4.5: Chyba, když mají dva stavy stejné jméno

4.3 Architektura

Sekce se zabývá především správou vnitřcích dat aplikace. Pro uložení stavu aplikace se používá dříve zmíněný redux.

4.3.1 Vnitřní stav

Vnitřní stav udržuje veškeré informace o aktuální podobě editoru. Uchovává strukturu stavového automatu, podobu prvků nástrojové lišty a údaje o aktuálně zvolených elementech automatu s dalšími pomocnými údaji potřebnými ke kreslení automatu.

Pro změnu stavu se používají akce, které se vysílají a zpracovávají pomocí knihovny `redux`. Akce se spouštějí uživatelsky iniciovanými událostmi nebo automaticky při splnění některých nastavených podmínek.

Při zpracování akce `redux` provede programem definované změny a přepíše celý objekt stavu novým stavem. Vzniká tak jasná historie podoby stavu, které se využilo při implementaci funkcionality vracení se o krok zpět v editoru.

Správa vnitřního stavu proběhla několika iteracemi. Původně bylo zamýšleno, že nebude potřeba využít externí knihovnu. Místo toho chtěl autor použít funkcionalitu Reactu a vnitřní stav ukládat pomocí Context API.

Než se editor dostal do finálního stavu, tak prošel třemi iteracemi návrhu.

První byl s využitím React Context API, kdy všechen stav byl uložen v globálním kontextu. Tento návrh se po krátké době neosvědčil, protože jednotlivé akce při modifikaci stavu upravovaly parametry v příliš mnoho větvích globálního stavu najednou a nebylo zřejmé, co jednotlivé akce provádí.

Jako odpověď na problém autor zvolil rozdělit globální stav na více oddělených stavů, kde se každý zabýval svojí specifickou oblastí. Místo akcí, které provádí mnoho změn, v vnitřním stavu aplikace, vyvolává více akcí, které mají jednoduchý a jasný účinek. Jednotlivé hlavní stavy spravovaly: vnitřní reprezentaci automatu, pracovní plochu, s čímž se také pojila stavba automatu, a panel nástrojů. Existovaly pak další pomocné stavy například pro uložení pozice kurzoru. Tento model pracoval dostatečně, bohužel se po nějakém čase objevil problém s optimalizací editoru. React vždy překresluje celou větev stromu komponentů, která byla aktualizovaná a jelikož každý stav musí naslouchat pozici kurzoru, tak se velká část automatu neustále překreslovala při pohybu myší. Tomuto je možno zabránit, kdyby se před překreslováním ověřil stav atributů jednotlivých komponentů a pokud se nezměnily, tak by se element přestal překreslovat. Nicméně každý element odebíral stav a samotné ověřování se časově prodražilo. Není optimální ověřovat všechny komponenty. Navíc vznikaly problémy, kdy se zastavilo překreslení komponentu, který ale obsahoval jiný komponent, také odebírající stav, jako dítě a ten se již nemohl překreslit při změně.

S problémy by se dalo vypořádat i bez použití externích knihoven, ale vyžadovalo by to velké úsilí. Redux se s těmito problémy umí vypořádat

automaticky, takže se pro finální verzi použil. Může se využít, že redux každý prvek jednotlivě připojuje ke globálnímu stavu a poslouchá jen změny, které se ho týkají. Také používá „batchování“ překreslování, kde při více různých změnách globálního stavu za sebou se elementy překreslí jen jednou a naráz.

Při migraci stavu z Context API na redux byl zvolen lepší návrh, takže se lépe promysleli jednotlivé akce, které se jasněji navrhly. Stále se využívá module vysílání více akcí, ale všechny ovlivňují pouze jeden globální stav, který je pak dále podrozdělen. S využitím reduxu se také nabídla zajímavá funkcionalita kroků zpět a dopředu při vytváření automatu.

4.4 Import a export

Důležitým prvkem editoru je možnost importu a exportu mimo aplikaci pomocí souborů různých formátů. V rámci bakalářské práce byly zvoleny čtyři typy formátů:

json Standardní formát určení pro uchování dat. Data jsou podobně strukturovaná jako ve vnitřním stavu aplikace.

json-xstate Formát knihovny xstate.

text Formát zvolený pro jeho jednoduchost a snadnou možnost úpravy.

text-fit Formát, který je využíván v předmětech na fakultě, kde vznikla práce.

4.4.1 Import

Editor mimo tvorby automatu poskytuje funkci importu a zobrazení automatu ze souborů nahraných uživatelem. Soubory jsou do aplikace načteny a z nich se vygeneruje automat přímo v aplikaci. V souborech se udává pouze definice automatu a ostatní údaje, jako jsou pozice a velikost stavu, se automaticky vypočítávají až po načtení.

Funkce je řešena maximálně obecně, aby umožňovala přidání dalších formátů. Data jsou načtena ze souboru a podle jeho formátu jsou poslána do transformačního modulu pro něj definovaného. Pro funkci se používá jasně definované rozhraní, které jednotlivé moduly musí implementovat, ale veškerou ostatní manipulaci provádí libovolně. Důležité je pouze, aby byl vstup zformátován do datové podoby vnitřního stavu.

4.4.2 Validace

Před nahráním importovaných dat do vnitřního stavu aplikace a jejich zobrazení je potřeba zvalidovat formát automatu. Tato validace se už neváže na jednotlivé formáty souborů. Jelikož uživatelsky importovaná data můžou mít v podstatě

jakoukoli podobu, mohlo by se stát, že po nahrání dat celá aplikace spadne kvůli špatnému formátování. Proto se provádí validace.

Po úvodním kroku importu všechna data projdou ještě vrstvou validace, která prověří důležité vlastnosti automatu. Zkontroluje se, že všechny přechody vedou mezi skutečnými stavy a že popsané počáteční a koncové stavy existují v datech.

Pokud uživatelský soubor neprojde validací, je potřeba uživatele upozornit na chybu a co nejjasněji mu sdělit, jaká chyba formátování nastala, aby ji mohl opravit.

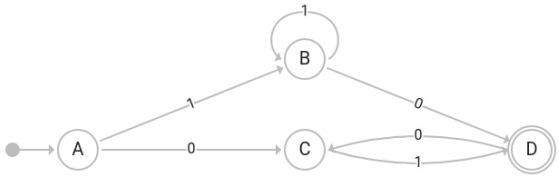
Po úspěšné validaci se mohou data nahrát do vnitřního stavu aplikace.

4.4.3 Export

Export automatu z aplikace do souboru je jednodušší. Strukturu dat definuje aplikace a proto není nutné provádět validaci, ale stačí převést vnitřní stav do požadovaného formátu. Data prochází jednoduchými iteracemi, kdy se mění jejich podoba. Rozdělení do jednotlivých modulů exportu podle formátu je stejný jako u importu.

4.4.4 Ukázka syntaxe různých formátů souborů

Následuje ukázka, jak je automat z obrázku [4.6] definovaný v různých formátech.



Obrázek 4.6: Ukázkový automat

```
{
    "initial": ["A"],
    "final": ["D"],
    "states": {
        "A": {
            "0": "C",
            "1": "B"
        },
        "C": {
            "1": "D"
        },
        "B": {
            "0": "D",
            "1": "B"
        },
        "D": {
            "0": "C"
        }
    }
}
```

Ukázka kódu 4.1: Definice automatu ve formátu json

```
{
    "id": "statemaker",
    "initial": "A",
    "states": {
        "A": {
            "on": {
                "0": "C",
                "1": "B"
            }
        },
        "C": {
            "on": {
                "1": "D"
            }
        },
        ...
    }
}
```

Ukázka kódu 4.2: Definice automatu ve formátu json-xstate

```
A   C   B   D
A
D
A   0   C
A   1   B
C   1   D
B   0   D
D   0   C
B   1   B
```

Ukázka kódu 4.3: Definice automatu ve formátu text

```
FA  0   1
>A  C   B
C   -   D
B   D   B
<D  C   -
```

Ukázka kódu 4.4: Definice automatu ve formátu text-fit

4.5 Automatické pozicování

Uživatel má možnost vyvolat akci automatického pozicování automatu, který se automaticky napozicuje do maximálně čitelné podoby.

Jako u importu se kladl důraz na jednoduchou možnost rozšiřitelnosti o další algoritmy. Data automatu jsou vstupem do jednotlivých modulů algoritmů, kde jsou zpracována a vrácena zpět opět v podobě napozicovaných dat. Aplikace přichází do kontaktu s logikou algoritmu pouze v těchto dvou bodech. Vše, co se děje mezitím, už je otázkou implementace algoritmu.

Každý použitý algoritmus pracuje s daty, která si naformátuje do podoby grafu, kde stavy jsou vrcholy a přechody orientované hrany. Tyto elementy pak mají dodatečné pomocné parametry, které se užijí v rámci výpočtu pozice v algoritmu.

4.5.1 Hierarchical

Pro potřeby editoru se zvolily jedny z jednodušších, ne však naivních možností implementace jednotlivých kroků tohoto algoritmu. Zvolené algoritmy se dále rozeberou a uvede se jejich obecný pseudokód.

4.5.1.1 Odstranění cyklů

K odstranění nebo v tomto případě pouze označení hran, které v grafu tvoří cykly, se může využít algoritmus DFS. [62]

Nejprve se zvolí nějaký vrchol grafu s preferencí vrcholů, které jsou v automatu počáteční stavy. Následně se prochází sousední vrcholy, které vedou po existujících hranách. Každý vrchol se přitom označí za navštívený. Pokud se při průchodu algoritmus setká s již navštíveným vrcholem, tak označí hranu k němu vedoucí příznakem.

Pokud graf obsahuje více komponent, tak se DFS pustí vícekrát, pokaždé na ještě neoznačené vrcholy.

Po označení všech vrcholů výsledné hrany nosí příznaky o tom, jestli tvoří cyklus později, při další manipulaci s grafem je můžeme zvlášť vyfiltrovat. Hrany se přímo nemažou, jelikož je potřeba je zachovat pro závěrečnou transformaci grafu zpět na data aplikace.

4.5.1.2 Přiřazení vrstev

Pro rozřazení vrcholů do vrstev se použije podobně jako v [63] topologické řazení. Využije se při tom pouze necyklových hran.

V grafu se vyhledají vrcholy, do kterých nevede žádná hrana a přidají se do fronty. Pak následuje cyklus, kdy se postupně odebírá vrchol z fronty a přiřadí se do vrstvy. Dále se z množiny všech vrcholů odeberou vrcholy z fronty a z množiny hran se odeberou hrany vedoucí z vrcholů fronty. Následně

Algoritmus 4.1 Hierarchical – označení cyklů

```
1: G ← (V, E)  ▷ graf s vrcholy V a hranami E, vrcholy jsou seřazeny podle
   toho, jestli jsou počáteční, kde počáteční jsou na začátku pole

2: function DFSMarkVertices(u)
3:     if u.isVisited = true then          ▷ vynechá již navštívené vrcholy
4:         return
5:     u.isVisited ← true
6:     u.isMarked ← true
7:     outEdges ← GetOutEdges(u)           ▷ hrany vycházející z vrcholu
8:     for e in outEdges do                                  ▷ hrana (u, v)
9:         if e.v.isVisited ≠ true then
10:            DFSMarkVertices(e.v)
11:        else                                          ▷ hrana tvoří cyklus
12:            SetEdgeMakesCycle(e)
13:    u.isMarked ← false

14: for u in V do
15:    if u.isVisited ≠ true then          ▷ vynechá již navštívené vrcholy
16:        continue
17:    DFSMarkVertices(u)
```

se opět vyhledají vrcholy, do kterých nevede žádná hrana, které tvoří novou frontu. Cyklus se opakuje, dokud není množina vrcholů prázdná. [4]

Algoritmus 4.2 Hierarchical – přiřazení vrstev

```
1: G ← (V, E) ▷ graf s vrcholy V a hranami E, hrany, které tvoří cyklus jsou
   ignorovány

   ▷ Najde vrcholy bez příchozích hran, tyto vrcholy a odchozí hrany z těchto
   vrcholů odstraní z původních množin V a E
2: queue ← GetVerticesWithoutIncomingEdges(V, E)
3: layer ← 0
4: while queue.empty ≠ true do
5:     for u in queue do
6:         u.layer ← layer
7:     queue ← GetVerticesWithoutIncomingEdges(V, E)
```

4.5.1.3 Minimalizace křížení hran

Nejprve se z grafu odstraní hrany, které vedou mezi vrstvami, které spolu přímo nesousedí. Tyto hrany se nahradí řetězem provizorních vrcholů a hran,

kde provizorní vrchol vede vždy na každé mezi-vrstvě.

Následně se zkoušejí různé transpozice jednotlivých vrstev a hledá se ta, která má nejmenší počet křížení hran mezi vrstvami. Počet iterací toho cyklu se použil 24, dle poznatku [4].

Nejprve se pro každý vrchol ve vrstvách spočítá medián pozice podle pozic z předchozí vrstvy. Následně se ve vrstvách prohazují jednotlivé vrcholy, pokud to snižuje lokální křížení hran. Po těchto krocích se spočítá počet křížení celého grafu a porovná se s počtem z předešlé iterace. Pro následující iteraci se použije graf s menším počtem křížení. [4]

Algoritmus 4.3 Hierarchical – minimalizace křížení hran

```
1: G ← (V, E)                                   ▷ graf s vrcholy V a hranami E

   ▷ Hrany s vrcholy, které nejsou v sousedících vrstvách nahradí řetězem
   hran a vrcholů
2: RemoveSpanningEdges(G)
3: order ← InitOrder(G)                         ▷ Původní pozice vrcholů
4: best ← order
5: for i = 0 to 24 do
6:     AssignMedians(order)
7:     Transpose(order)
8:     if CountCrossings(best) > CountCrossings(order) then
9:         best ← order
```

4.5.1.4 Přiřazení pozic

Tento krok se dělí na dva mezikroky. Nejprve se napozicují vrcholy ve vrstvě a pak následně jednotlivé vrstvy.

Při hledání správné pozice na pozici `x` se nejprve vrcholům přiřadí priorita podle toho, jestli jsou to vrcholy dočasné a jestli z nich vedou dočasné hrany. Tyto vrcholy budou mít nejvyšší prioritu a budou jim přiřazované pozice jako první. Dočasné vrcholy se ve výsledném grafu nahradí za původní hrany, které vedou přes více vrstev a algoritmus bude preferovat jejich vykreslování, čímž se docílí kratších hran, což vyústí ve větší přehlednost. [4]

Vrcholy se podle priorit vybírají a je jim přiřazena pozice ve středu dříve napozicovaných vrcholů předchozí vrstvy, ze kterých vedou do aktuálního vrcholu hrany. Pokud by se vrchol na tuto pozici nevešel a vznikalo by křížení stavů, tak buď se posune vrchol na jednu ze stran, pokud je místo a nebo se musí posunout všechny napozicované vrcholy vrstvy směrem od aktuálního vrcholu. [4]

Při hledání správné pozice na pozici `y` se nejprve napozicuje úvodní vrstva na nultou pozici, najde se v ní největší vrchol. Výsledná velikost spolu s auto-

Algoritmus 4.4 Hierarchical – přiřazení pozic

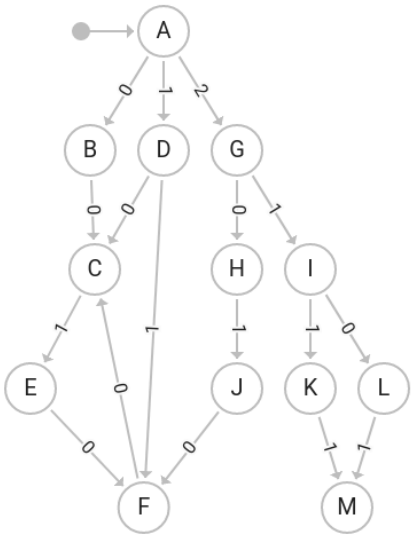
```
1: G ← (V, E)    ▷ graf s vrcholy V a hranami E, vrcholy jsou rozděleny do pole vrstev L

2: for l in L do
3:     sorted ← SortByPriority(l)
4:     for v in sorted do
5:         v.position.x ← GetBestXPosition(v)
6:         if !CanPlace(v.position) then
7:             MoveLowerPriorityVertices(l, v)
```

matickou mezerou mezi vrstvami se použijí k pozici další vrstvy, kde se celá vrstva vykresluje do jedné linie. Tento krok se opakuje pro všechny vrstvy. [4]

4.5.1.5 Ukázka napozicovaného automatu

V obrázku [4.7] je vidět, jak může vypadat stavový automat napozicovaný algoritmem hierarchical.



Obrázek 4.7: Automat napozicovaný algoritmem hierarchical

4.5.2 Force-directed

Algoritmus force-directed je už v teorii popsán dostatečně podrobně. Ale musí se více specifikovat pomocné proměnné.

Pro přiřazení úvodních pozic se počítá náhodná pozice v rozmezí středu obrazovky, kde jednotlivé hrany tvoří polovinu skutečné hrany prohlížeče. Výsledný automat je tak umístěn doprostřed obrazu. Dále s pomocí kamery implementované aplikací je možné posouvat a oddalovat obraz a tak není potřeba limitovat výsledný posun vrcholů v algoritmu mezi limity okna prohlížeče a jsou povoleny pozice i mimo okno.

Teplotu ochlazování sil je potřeba definovat tak, aby měl algoritmus možnost najít správnou pozici vrcholů. To se pozná tak, že po vícenásobném puštění algoritmu na stejný graf se dostane stejného nebo skoro stejného výsledku. Dle [x] je doporučený počet iterací působení sil v rámci několika stovek. Pro snížení časové náročnosti se zvolil počet, který je dostatečný pro aplikaci.

$$iterations = 200$$

Od toho se pak odvíjela teplota, která musí navazovat na velikost okna prohlížeče a která by se měla pohybovat v prostoru jednoho kvadrantu v rozmezí náhodných pozic.

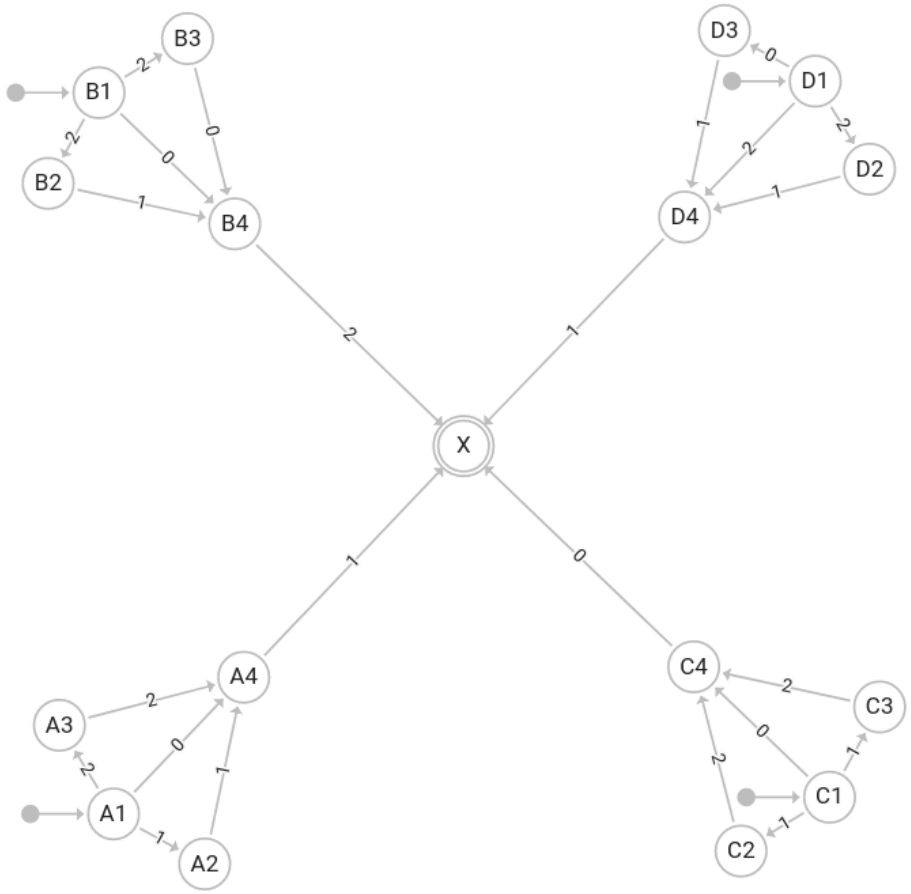
$$temperature = min(width, height) * \frac{1}{4}$$

Míra ochlazování pak musí být taková, aby zůstala teplota v posledních iteracích na takové úrovni, aby byly pohyby s vrcholy stále zřetelné, ale také aby se s nimi pohybovalo maximálně v rozmezí jednotek.

$$cooling\_rate = 0.97$$

4.5.2.1 Ukázka napozicovaného automatu

V obrázku [4.8] je vidět, jak může vypadat stavový automat napozicovaný algoritmem force-directed.



Obrázek 4.8: Automat napozicovaný algoritmem force-directed

Kapitola 5

Realizace

V této kapitole se rozebírá detailní implementace konkrétních částí editoru. Je zde popsaná struktura kódu a jsou uvedeny některé ukázky kódu složitější funkcionality, kterou aplikace řeší.

5.1 Struktura

Zdrojový kód aplikace je strukturován do složek podle funkcionality kódu. Zde je uvedena podoba rozdělení a krátký popis jednotlivých složek.

```
src
├── components
│   ├── Elements
│   └── Toolbar
├── hooks
├── icons
├── interfaces
├── reducers
├── utils
│   ├── export
│   ├── import
│   ├── positioning
│   └── validation
├── index.tsx
└── store.ts
```

components Deklarace React komponentů sestavujících aplikaci. Komponenty zobrazují uživatelské prostředí a zpracovávají uživatelskou interakci. Každý důležitý komponent má vlastní složku. Elements jsou komponenty pro zobrazení elementů automatu. Toolbar obsahuje komponenty spojené s nástrojovou lištou.

hooks Pomocné React hooks, které jsou používány komponenty. Zpracovávají uživatelskou interakci a napojení na redux.

icons Ikony použité pro stavy.

interfaces Definice datové podoby objektů, které jsou použity v aplikaci. Soubory mohou obsahovat dodatečné pomocné funkce pro manipulaci s objekty.

reducers Definice jednotlivých modulů vnitřního stavu aplikace, akcí reduxu a reducerů, které akce zpracovávají.

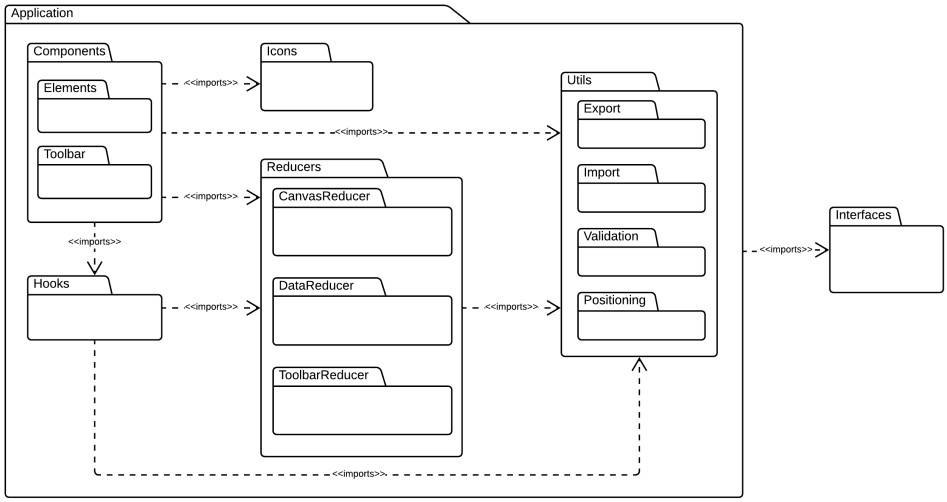
utils Ostatní funkcionalita aplikace, která je oddělená od uživatelského prostředí.

export Moduly pro export do souborů.

import Moduly pro import ze souborů.

positioning Moduly pro automatické pozicování algoritmy.

validation Moduly pro validaci souborů.



Obrázek 5.1: Diagram komponentů

5.2 Vnitřní stav

Vnitřní stav aplikace je rozdělen do tří hlavních částí – canvas, data a toolbar.

5.2.1 Canvas

Udržuje informace o pracovní ploše. Zpřístupňuje informace o stavu kamery a kurzoru, které se používají v interakci s automatem. Dále spravuje „ghost“ elementy, které se zobrazují na ploše při vytváření elementů nových. A také jsou zde uloženy právě vybrané elementy.

```
interface CanvasState {
    cursorMode: 'select' | 'move'
    cursor: Position
    offset: Position
    scale: number
    ghostState: IState
    ghostTransition: ITransition
    selectedState: string
    selectedTransition: string
    positioningState: string
    stateNameId: number
    transitionNameId: number
}
```

Ukázka kódu 5.1: Definice vnitřního stavu – modul Canvas

5.2.2 Data

Uchovává vykreslený automat. Stavy a přechody jsou uloženy v mapách podle jejich id. To umožňuje snadné odkazování na elementy a hledání v čase $\mathcal{O}(1)$.

```
interface StateMap {
    [key: string]: State
}

interface TransitionMap {
    [key: string]: Transition
}

interface DataState {
    initialStates: string[]
    finalStates: string[]
    states: StateMap
    transitions: TransitionMap
}
```

Ukázka kódu 5.2: Definice vnitřního stavu – modul Data

5.2.2.1 Stav

Stav automatu je jednoduchý objekt, který uchovává údaje jen o svém jméně, pozici a velikosti. Dále má každý stav id, které funguje i jako identifikátor v mapě stavů.

```
interface Position {
    x: number
    y: number
}

type StateType = 'default' | 'initial' | 'final'

interface IState {
    id: string
    name: string
    position: Position
    size: number
}
```

Ukázka kódu 5.3: Definice elementu reprezentujícího stav automatu

5.2.2.2 Přechod

Přechod je také jednoduchý objekt se stejným principem identifikátoru jako u stavu. Informace o stavech, mezi kterými přechod vede, přechod uchovává s pomocí dvou id, první je id stavu, ze kterého přechod vychází, a druhé je, ve kterém stav končí. Tento přístup nám umožňuje lehce a rychle najít stav v mapě stavů.

```
interface ITransition {
    id: string
    name: string
    startState: string
    endState: string
}
```

Ukázka kódu 5.4: Definice elementu reprezentujícího přechod automatu

5.2.3 Toolbar

Informace o manipulaci s lištou nástrojů. Ukládá vybraný nástroj, obsah zadaného textu a zobrazené chyby.

```
interface ToolbarState {
    input: string
    errors: string[]
    buildState: boolean
    buildStateType?: StateType
    buildTransition: boolean
    showHint: boolean
    showExplore: boolean
    algorithm: Algorithm
    confirmText: string
    grid: boolean
    mouseMode: MouseMode
}
```

Ukázka kódu 5.5: Definice vnitřního stavu – modul Toolbar

5.3 Lišta nástrojů

Většina uživatelské interakce s aplikací se provádí skrz nástrojovou lištu. Tento prvek se zvolil, jelikož se předpokládá, že se s ním většina uživatelů už setkala v jiných aplikacích. Lišta zabírá minimální prostor v editoru. Všechna tlačítka jsou označena pouze ikonou. Pro jasnost se zobrazí jejich popis, když se na ikonu najede. Dělí se na 6 hlavních částí:

1. Nástroje pro manipulaci s pracovní plochou
2. Práce s elementy automatu
3. Pozicování
4. Import a export
5. Nástroje pro práci s vnitřním stavem dat aplikaci - uložení, kroky zpět a dopředu
6. Nápověda

5.4 Manipulace s elementy

Pozicování elementů a jejich posouvání byl jedním ze složitějších problémů, které se v této práci řešily. Důvodem bylo sloučení funkcionality automatického pozicování, vytváření elementů a jejich následné posouvání po pracovní ploše.

Bylo zváženo několik postupů. Prvním bylo mít všechny elementy interaktivní a dovolit manipulaci s pozicí jak stavů, tak i přechodů. Přechody by mohly například mít „klouby“ a tak obcházet stavy pro zlepšení přehlednosti

nebo by také bylo povolené uživateli zahýbat přechody a dělat z přímek křivky. Tento postup by poskytl veškerou funkcionalitu k manipulaci uživateli. Hlavním nedostatkem by pak ale bylo, kdyby se na uživatelsky nakreslený automat pustilo automatické pozicování. Použité algoritmy by musely projít velkými úpravami, jen aby braly v potaz uživatelské úpravy elementů, a vznikalo by řádově daleko více krajních případů, kdy by se pozicování mohlo rozbít. Také by jen v rámci pohybování elementů, kdy uživatel pohne přechodem, který má kloub nebo který je zahnutý, se tyto informace špatně překládaly do režimu pohybování.

Druhým postupem by mohlo být, že by se uživateli odebrala veškerá možnost manipulace a při přidávání elementů by se celý automat rovnou sám napozicoval do nejoptimálnější polohy. Výsledkem by byly vzhledné automaty. Někdy má uživatel svoje představy, jak má automat vypadat, ale toto by mu odebralo možnost veškeré manipulace.

Nakonec se zvolil kompromis mezi těmito metodami. Manipulace se stavy bude umožněna i uživateli, ale přechody budou definované natvrdo v editoru a bude se jim automaticky vypočítávat pozice a zahnutí. Přístupem se vyvarujeme většině krajních případů, které nastávají při manipulaci s přechody, které se tak budou vykreslovat esteticky. Jediné krajní případy, které se musí řešit, je vykreslování více přechodů vedle sebe, když vedou mezi stejnými stavy.

Velikost stavu

Velikost stavu je automaticky přepočítávána pomocí React hooku, který analyzuje referenci na textové pole názvu, zjišťuje maximum mezi výškou a šířkou a větší hodnotu ukládá jako poloměr stavu, podle kterého se vykresluje prvek stavového kruhu.

```
const getSizeFromRef = (el: SVGTextElement): number => {
    if (!el) return MIN_STATE_SIZE

    const bounds = el.getBBox()
    let size = distance(toPosition(bounds.width, bounds.height))
    size /= 2 // Get radius
    size = Math.max(MIN_STATE_SIZE, size)
    size += BORDER_OFFSET

    return size
}
```

Ukázka kódu 5.6: Výpočet velikosti stavového elementu

```
// Updates state's size to fit current text content
const useTextSize = (
    text: string,
    onUpdate: (state: Partial<IState>) => void,
    runOnInit = true
): React.RefObject<SVGTextElement> => {
    const [ref] = React.useState(React.createRef<SVGTextElement>
        ())
    const [prevText, setPrevText] = React.useState(text)

    const firstUpdate = React.useRef(true) // Checks for 1st
        render
    const refContent = ref ? ref.current : ''
    React.useLayoutEffect(() => {
        if (firstUpdate.current && !runOnInit) {
            firstUpdate.current = false
            return // Don't update on first render
        }
        if (!refContent || (prevText === text && firstUpdate.
            current !== true))
            return // Don't update if no text or text hasn't
                changed

        setPrevText(text)
        const size = getSizeFromRef(refContent)
        onUpdate({ size }) // Dispatch redux action to update
            state
    }, [runOnInit, refContent, text, prevText, onUpdate])

    return ref
}
```

Ukázka kódu 5.7: Výpočet aktuální velikosti stavu

Posouvání po ploše

Pro umožnění DND posunu stavů po ploše jsou jednotlivým elementům přiřazeny funkce pro naslouchání událostí iniciovanými myší. Pokud stav zaznamená podržení tlačítka, tak se zapne mód posunu, kdy se povolí automatické pozicování podle React hooku, který přepočítává aktuální pozici kurzoru a předává jí elementu reprezentující stav. Pozice stavu se uloží natrvalo do vnitřního stavu aplikace až po puštění myši, aby se redukovalo vyvolávání akcí reduxu. V opačném případě by neustálé přepisování vnitřního stavu snižovalo rychlost aplikace a také by nadbytečné akce znepříjemňovaly vracení se v čase při kroku zpět.

Naslouchání události pro puštění myši je na rozdíl od ostatních zmíněných připojen k celému dokumentu, jelikož uživatel může manipulovat s myší rychleji, než aplikace vykresluje stav a pak by se bez puštění už neregistrovala událost posunu myši a stav by se nepohyboval i beze spuštění.

Následuje ukázka, jak se vypočítává reálná pozice kurzoru na pracovní ploše, která bere v úvahu přiblížení a posunutí pracovní plochy. Tato pozice se pak používá při pohybu se stavem.

```
// Returns cursor position relative to application scale and offset
function usePosition(allowChange: boolean): Position {
    // Get current offset from state
    const { offset, scale } = useSelector(
        (state: ReduxState) => ({
            offset: state.canvas.offset,
            scale: state.canvas.scale
        }),
        []
    )
    // Get cursor position if the indicator is true
    const cursor: Position = useCursor(allowChange)
    const [position, setPosition] = React.useState<Position>(cursor)

    React.useLayoutEffect(() => {
        if (!allowChange) return

        // Normalize the cursor position
        setPosition(normalize(cursor, offset, scale))
    }, [allowChange, cursor, scale, offset])

    return position
}
```

Ukázka kódu 5.8: Výpočet aktuální pozice stavu s pomocí React hooku

Vykreslení přechodu

Pozice křivky přechodu se přímo odvíjí od jejích stavů. Samotná křivka ukládá odkaz na stavy pouze v podobě id, takže si musí najít stavy z mapy. Podle pozice stavů zjistí úhel přímky mezi stavy. S pomocí úhlu a velikosti nalezne hraniční body stavu, kde přímka končí. Tyto body použije jako počáteční a koncové body přímky.

```
// Gets distance between the center of circle to the edge on an
    angle
const edgeOffset = (size: number, angle: number): Position => {
    const actualSize = Math.min(size, MIN_STATE_SIZE)
    const degrees = toPosition(Math.cos(angle), Math.sin(angle))
    return multiplyByX(degrees, actualSize + BORDER_OFFSET)
}

// Gets points on start and end states that start on their edges
    and are on line with shortest distance possible
export const getEdgePoints = (
    startState: State,
    endState: State
): EdgePoints => {
    const startPoint = startState.position
    const endPoint = endState.position

    const angle = getAngle(startPoint, endPoint)

    // Change the points from center to the edges
    const from = subtract(startPoint, edgeOffset(startState.size,
        angle))
    const to = add(endPoint, edgeOffset(endState.size, angle))

    return { fromPoint: from, toPoint: to }
}
```

Ukázka kódu 5.9: Výpočet koncových bodů křivky přechodu

Vykreslení více přechodů

Před vykreslením přechodu se hledají ostatní přechody, které prochází mezi stejnými stavy. Pokud se takové přechody naleznou, tak je potřeba přímky přímky přechodu změnit na křivky, aby se nepřekrývaly. Z těchto podobných přechodů se vytvoří pole, kde první polovinu tvoří přechody ze stejného počátečního stavu a druhá polovina je tvořena těmi, které mají počáteční a koncový stav přehozený. Z pole se najde pozice aktuálního přechodu a znormalizuje se tak, aby přechod v polovině pole měl index 0, dolní část přechodů pak záporné indexy a horní kladné. Tento index se používá jako offset pro zakroužení křivky.

```
// Array of transitions going between the same states as this
    transitions
```

```
// [...from -> to, ...to -> from]
const sameTransitions = [
    ...transitions.filter(
        t => (
            t.startState === startState &&
            t.endState === endState
        )
    ),
    ...transitions.filter(
        t => (
            t.startState === endState &&
            t.endState === startState
        )
    )
]

// Index of current transition on the array
const offset =
    sameTransitions.length === 0
        ? 0
        : sameTransitions.map(t => t.id).indexOf(id) -
          Math.floor(sameTransitions.length / 2)


// Creates bezier curve between points start and end
// Curve can have offset from center, which makes it more round outwards
const curve = (
    startPoint: Position,
    endPoint: Position,
    offset: number
): string => {
    // Find points on the 1/3 and 2/3 of the line
    const p1Line = linePoint(startPoint, endPoint, 1 / 3)
    const p2Line = linePoint(startPoint, endPoint, 2 / 3)
    // Get points on the perpendicular of current line using offset to calculate distance
    const p1Curve = curvePoint(startPoint, endPoint, p1Line, offset)
    const p2Curve = curvePoint(startPoint, endPoint, p2Line, offset)

    const start = `${startPoint.x} ${startPoint.y}`
    const end = `${endPoint.x} ${endPoint.y}`
    const p1 = `${p1Curve.x} ${p1Curve.y}`
    const p2 = `${p2Curve.x} ${p2Curve.y}`

    // SVG path
    return `M ${start} C ${p1} ${p2} ${end}`
}
```

Ukázka kódu 5.10: Výpočet cesty křivky reprezentující přechod

Vykreslení několika-řádkových názvů

Tato zdánlivě jednoduchá funkcionalita přinesla několik potíží v rámci implementace.

SVG přímo nepodporuje řádkování v textovém elementu a většina pomocných elementů, které obstarávají řádkování a jsou popsány v SVG2 [64], nejsou zatím implementovány v prohlížečích. Musí se proto zvolit jiný způsob.

Místo využití funkcionality SVG se text rozdělí na pole řádků, které se pak jednotlivě namapují na textové elementy a napozicují se, aby se zobrazovaly v řádcích. V jménech přechodů se používá SVG funkce textPath, která tento přístup mírně komplikuje, ale s uvedenými použitými parametry funguje.

```
const lines = data.name.split('\n')
const textYPosition = (i: number) => i~* 20 - ((lines.length - 1)
    * 20) / 2
const mapLine = React.useCallback(
    (line: string, i: number) => (
        <React.Fragment key={i}>
            <text className={classes.text} dy={textYPosition(i)}>
                <textPath
                    xlinkHref={'#path_' + data.id}
                    startOffset={'50%'}
                    textAnchor="middle"
                    alignmentBaseline="middle"
                >
                    {line}
                </textPath>
            </text>
        </React.Fragment>
    ),
    [lines]
)
```

Ukázka kódu 5.11: Transformace popisku přechodu do více řádků

Grid

Pro přesnější manipulaci s polohami stavů uživatelem nabízí aplikace mód grid, který po zapnutí zobrazí mřížku na pracovní ploše a umožňuje posouvání po jednotlivých čarách mřížky.

Dosáhne se toho tak, že místo reálné pozice kurzoru se jeho hodnota zaokrouhluje k nejbližší čáře podle jednoduchého vzorce.

```
Snaps position to the closest point on the grid of size
const snap: NumOperator = (position, gridSize) =>
    multiplyByX(
        round(divideByX(position, gridSize)), gridSize
    )
```

Ukázka kódu 5.12: Výpočet pozice stavu na gridu

5.5 Pozicování

Pozicování bylo řešeno tak, aby mohlo být implementováno v jednotlivých modulech nezávislých na zbytku aplikace. Jediné, co tyto moduly musí splňovat, je implementace sdíleného a jasně definovaného rozhraní. Rozhraní je vedeno jako funkce, která má parametr datového stavu automatu a jež vrací datový stav automatu.

Toto umožňuje pro jednotlivé algoritmy, aby mohly svoje řešení implementovat jakýmkoli způsobem, nezávislým na programovacím modelu nebo i na jazyku.

V rámci implementace algoritmů se stále využívalo funkcionálního programování. K tomu bylo potřeba upravit obecné algoritmy popsané v předchozích částech. Oba algoritmy si prvně datový stav převedou na graf, který má obdobnou strukturu, ale navíc uchovává informace o pomocných proměnných používaných v rámci výpočtu. Prvním a posledním krokem obou algoritmů je tedy konverze z datového stavu na graf a zpět.

Manipulace s grafy probíhá nedestruktivně. Místo úpravy grafu se vždy funkcemi vypočítá nová podoba grafu, která nahradí starou. Jednoduchým příkladem může být označení vrcholů, které představují počáteční stavy.

```
1  // Returns new graph with vertices set to initial if their id is
       in the input array
2  const setInitialVertices = (initialIds: string[]) =>
3      (graph: Graph): Graph => ({
4      ...graph,
5      vertices: {
6          ...graph.vertices,
7          ...initialIds.reduce<VertexMap>(
8              (acc, curr) => ({
9                  ...acc,
10                 [curr]: {
11                     ...graph.vertices[curr],
12                     isInitial: true
13                 }
14             }),
15             {}
16         )
17     }
18 })
19
20 ...
21 graph = setInitialVertices(data.initialStates)(graph)
22 ...
```

Ukázka kódu 5.13: Transformace grafu – označení počátečních stavů

5.5.1 Objekty grafu

Oba algoritmy pracují se základními objekty `Vertex`, `Edge` a `Graph`, které používají k definici grafu a dále je rozšiřují podle své konkrétní potřeby.

```
interface Vertex {
    id: string
    name: string
    position: Position
}
interface Edge {
    id: string
    from: string
    to: string
    name: string
}
type VertexMap<V˜extends Vertex> = Map<V>
interface Graph<V˜extends Vertex, E extends Edge> {
    vertices: VertexMap<V>
    edges: E[]
}
```

Ukázka kódu 5.14: Základní definice grafu pro pozicovací algoritmy

Jako rozhraní pro každou pozicovací funkci, která se použije v aplikaci a kterou musí algoritmy implementovat, je jednoduchá funkce přijímající data automatu a vracející nově napozicovaná data.

```
type PositionFn = (data: Data) => Data
```

Ukázka kódu 5.15: Funkční rozhraní modulů pro pozicování

5.5.2 Hierarchical

Algoritmus rozšiřuje grafové objekty o několik parametrů. Rozšířený graf pak uchovává pole id vrcholů jednotlivých vrstev.

```
interface Vertex extends TVertex {
    layer: number
    size: number
    priority: number        \\ Used when assigning x position
    isInitial: boolean
    isVisited: boolean      \\ DFS helper attribute
    isMarked: boolean       \\ DFS helper attribute
    isTemp: boolean         \\ Temporary vertex
}
interface Edge extends TEdge {
    makesCycle: boolean     \\ Creates cycle in original graph
}
interface Graph extends TGraph<Vertex, Edge> {
    layers: string[][]
}
```

Ukázka kódu 5.16: Definice grafu pro algoritmus hierarchical

5.5.3 Force-directed

Algoritmus rozšiřuje vrchol pouze o polohu `displacement`, kam se ukládá posun vrcholu na základě vypočítaných sil v iteraci.

```
interface Vertex extends TVertex {
    displacement: Position
}
```

Ukázka kódu 5.17: Definice vrcholu pro algoritmus force-directed

Zde jsou uvedeny funkce použité k výpočtu hodnoty `displacement` při počítání odpudivých sil, přitažlivých sil a limitování hodnoty na základě teploty.

```
const repulse = (diff: Position, dist: number, k: number) =>
    multiply(divide(diff, dist), fr(dist, k))
const attract = (diff: Position, dist: number, k: number) =>
    multiply(divide(diff, dist), fa(dist, k))
const displace = (disp: Position, temperature: number) =>
    multiply(
        divide(disp, absolute(disp)),
        min(absolute(disp), temp)
    )
```

Ukázka kódu 5.18: Funkce pro výpočet sil force-directed algoritmu

5.6 Import a export

Jednotlivé moduly pro importování a exportování pomocí souborů musí implementovat jednoduché rozhraní, kterým transformují data ze souborů do dat

aplikace nebo obráceně. Podobné rozhraní implementují i validační funkce, kde navíc toto rozhraní implementuje i interní validační funkce, která kontroluje správnost definice automatu.

```
type ImportFn = (data: string) => Data
type ExportFn = (data: Data) => string

type ValidateArg<T> = T extends Data ? Data : any
export type ValidateFn<T> = (data: ValidateArg<T>) => string[]
```

Ukázka kódu 5.19: Funkční rozhraní modulů pro import, export a validaci

5.6.1 Import

Aplikace umožňuje uživateli nahrát soubor s automatem. Po nahrání souboru se aplikace pokusí o rozpoznání formátu souboru podle přípony souboru a pomocných identifikátorů v definovaných v souborech. Pokud se nepodaří soubor aplikaci přečíst, tak zobrazí uživateli chybovou hlášku.

Pokud dojde k úspěšnému přečtení souboru, tak následuje validace dat souboru. Každý formát musí implementovat tuto validaci, kde se dle možností formátu kontroluje, jestli jsou data validní. V případě textového formátu je tato možnost validace omezená pouze na kontrolu počtu sloupců a podobné kontroly. Pokud se jedná o JSON soubor, tak se může i navíc zkoumat datová podoba objektů.

Data následně putují do modulů importů podle rozpoznaného formátu. Data souboru se přeloží do podoby dat popisujících automat v aplikaci.

Tato data jsou ještě zpracována datovým validátorem. Ten ověří, jestli má importovaný automat korektní strukturu, převážně jestli identifikátory, které odkazují na specifické stavy opravdu existují.

Když vznikne chyba kdykoli v průběhu tohoto procesu, tak se zaznamená do pole chyb a na konci mezikroku se import zruší a nashromážděné chyby se zobrazí uživateli.

5.6.2 Export

Funkce exportu pouze nakonfiguruje formát výstupního souboru podle vybraného typu a pomocí daného modulu exportu naformátuje data do jedné `string` proměnné, která je přeměněná na `Blob` [65]. S pomocí knihovny `file-saver` se vyvolá v prohlížeči uložení souboru.

5.7 Ukládání automatu

Aktuální automat z pracovní plochy lze uložit a dále nahrát z paměti i po znovuotevření aplikace. Využívá se k tomu local storage prohlížeče, kdy po akci kliknutí se zkopíruje datová část vnitřního stavu aplikace, která se převede

na `string`. Ten se pak může zpětně nahrát ze storage, převést zpět do objektu a výsledný objekt se nahraje do vnitřního datového stavu.

V dřívějším návrhu aplikace bylo zapnuté ukládání po každé uživatelské akci. Tento přístup byl nakonec opuštěn kvůli situacím, kdy uživatel si sám uloží automat, pak provede několik změn, není s nimi spokojen a chce vrátit stav do uložené podoby, která se ale už sama změnila.

5.8 Kamera

Pro podporu manipulace s rozsáhlejšími automaty aplikace umožňuje posun kamerou a přibližování s oddalováním.

Uživatel může posouvat pracovní plochu prostředním tlačítkem myši nebo zapnutím módu posunu na liště nástrojů a pak levým tlačítkem myši. Mód posunu zablokuje interakci s vykreslenými elementy a zjednodušuje posun kamerou.

Také je poskytnuta možnost přibližování a oddalování s pomocí kolečka myši či tlačítky na liště.

Tato funkcionalita představuje několik obtíží při její implementaci. Nejdříve se musí rozhodnout mezi dvěma přístupy transformace elementů. Buď se bude transformovat celá pracovní plocha a na ní se vypočítávat aktuální pozice elementů. Nebo se budou transformovat samotné elementy.

První přístup nabízí menší náročnost na výpočty v rámci této práce, jelikož druhý přístup by vyžadoval neustálé přepisování vnitřního stavu. Také výsledné exportované automaty by bez transformace pozic zpět obsahovaly stavy s transformovanými pozicemi, které by mnohdy mohly být uloženy jako velmi malá nebo velmi velká čísla.

Stavy tedy budou mít pevně definovanou pozici ve vnitřním stavu aplikace, která se mění pouze jejich posunem po pracovní ploše, ale při vykreslování jednotlivých stavů se započítává do jejich pozice aktuální transformace pracovní plochy. Zde se od pozice odečte posun plochy a výsledek se vynásobí převrácenou hodnotou přiblížení.

```
1 // Normalizes position on canvas with scale and offset
2 const normalize = (
3     position: Position, offset: Position, scale: number
4 ): Position =>
5     multiplyByX(subtract(position, offset), 1 / scale)
```

Ukázka kódu 5.20: Výpočet normalizované pozice na základě polohy kamery

U transformace pracovní plochy se nesmí zapomenout aplikovat přiblížení na aktuální posun. Pro transformaci se používá SVG atribut `transform`.

```
1 const scaledOffset = divideByX(offset, scale)
2 const translate = `translate(
3     ${scaledOffset.x}, ${scaledOffset.y}
4 )`
```

```
5 const transform = `scale(${scale}), ${translate}`
```

Ukázka kódu 5.21: Transformace pracovní plochy na základě polohy kamery

Přibližování je nejsložitější část. Bylo cílem, aby při přibližování kamera centrovala pohled na pozici kurzoru, pokud se používá kolečko myši, a na střed plochy, pokud se používá nástroj z lišty. Pro přiblížení je potřeba aplikovat několik kroků:

1. Zvolit faktor přiblížení (může být i záporný).
2. Vypočítat nové přiblížení a rozdíl mezi ním a tím starým.
3. Bod přiblížení se vynásobí rozdílem, aby se vypočítala relativní pozice bodu přiblížení, která bere v úvahu aktuální přiblížení.
4. Přičíst k upravenému bodu aktuální posun.

Zde je ukázka těchto kroků v kódu.

```
1  // Calculates new offset depending on the factor of scaling
2  // Offset is calculated so the zooming is directed to specific
       position
3  const zoom = (
4      to: Position,
5      offset: Position,
6      scale: number,
7      factor: number
8  ): Position => {
9      const newScale = scale + factor
10     const change = newScale - scale
11     return add(multiplyByX(to, -change), offset)
12 }
```

Ukázka kódu 5.22: Výpočet přiblížení/oddálení kamery

5.9 Historie

Už bylo zmíněno, že s pomocí reduxu je implementace historie akcí uživatele jednoduchá, jelikož existuje jasná historie úprav vnitřního stavu. Implementaci historie navíc ještě zlehčuje knihovna `redux-undo`. Knihovna poskytuje transformující funkci, která se aplikuje na reducer. Výsledkem je proměna původního stavu na stav, který je pod-rozdělený na past, present a future bloky.

Aktuální podoba stavu je v present bloku, na který jsou napojena aktuální data aplikace. Části past a future slouží k uchování minulých stavů a budoucích stavů, generovaných při akcích vrácení se zpět nebo vpřed. Podobu aktuálního stavu jednoduše změníme vyvoláním akce `undo` nebo `redo`, která použije dřívější nebo budoucí verze stavu a posune je na aktuální pozici. Původně aktuální stav zároveň posune do past nebo future bloku.

Historie je aplikována na datovou část vnitřního stavu, která uchovává definici automatu. Pro správnou funkcionalitu je potřeba dbát na to, aby se při uživatelské interakci nebo programově nevyvolalo více akcí najednou, které by se aplikovaly na datový reducer. Toto chování by vyústilo ve zmatení uživatele, který by pro vrácení jedné jeho akce musel klikat na tlačítko zpět vícekrát kvůli skrytým akcím aplikace.

5.10 Dokumentace

Pro zlepšení čitelnosti kódu a ulehčení rozšíření aplikace dalšími vývojáři byla vytvořena dokumentace kódu. K dokumentaci je použita knihovna TypeDoc [66]. TypeDoc využívá komentářů v kódu a automaticky z nich generuje HTML dokumentaci.

Kapitola 6

Testování

Pro ověření funkčnosti výsledné aplikace se provedlo uživatelské testování a zároveň jsou naprogramované testy kódu pro většinu důležitých modulů.

6.1 Uživatelské testování

K ověření přívětivosti aplikace a zároveň k nalezení skrytých chyb se uskutečnilo uživatelské testování. Zjišťovalo se, jestli jsou všechny součásti editoru jasné a jestli uživatelé s poskytnutými nástroji umí zacházet bez problémů. Pro cíl maximální uživatelské přívětivosti se uživatelé pozorovali při práci s editorem a zkoumalo se, ve kterých částech by se mohlo stávat, že není editor k uživatelům přívětivý.

6.1.1 Použité instrukce

Instrukce pro testování jsou rozděleny na čtyři hlavní části. Každá se zabývá oddělenou problematikou.

6.1.1.1 Tvorba automatu

V této části se zkoumá tvorba automatů. Jestli je jasné, jak se pracuje v editoru se stavy a přechody, jestli se tyto prvky chovají, jak si uživatelé představují a zda uživatelé najdou všechny nástroje k tomu využívané a budou s nimi umět zacházet.

1. Vytvořte stav a umístěte ho přibližně do středu pracovní plochy.
2. Označte vytvořený stav a pojmenujte ho A.
3. Vytvořte počáteční stav a umístěte ho na pracovní ploše nad stav A.
4. Označte vytvořený stav a pojmenujte ho B.
5. Vytvořte přechod mezi stavem A a B. Přechod vychází ze stavu B.

6. Přechod označte a pojmenujte 1.
7. Vytvořte alespoň další tři stavy a libovolně je pojmenujte.
8. Vytvořte alespoň dalších pět přechodů a libovolně je pojmenujte.
9. Označte jeden ze stavů jako koncový.
10. Do tohoto stavu přidejte přechod ze sebe sama a přechod pojmenujte.
11. Automat uložte.

6.1.1.2 Import automatu

Tato část zkoumá, jestli uživatelé dokáží rozpoznat, jak nahrát data automatu ze souboru. Při práci s externími soubory se může stát, že se uživatel setká s automaty, které mají cizího autora, proto se také pozoruje ovládání pracovní plochy.

1. Resetujte pracovní plochu.
2. Nastavte pozicovací algoritmus na volbu force-directed.
3. Importujte automat ze souboru `aut1.json`.
4. Posuňte pracovní plochu tak, aby byl počáteční stav na středu obrazovky.
5. Oddalte obraz tak, aby byl celý automat vidět na obrazovce.

6.1.1.3 Pozicování automatu

V této části se testuje ergonomika manipulace s elementy.

1. Nahrajte původně vytvořený automat.
2. Zapněte pomocnou pozicovací mřížku.
3. Přepněte se do posouvacího modu kurzoru.
4. Přesuňte stavy na ploše do pozice, která je dle vás nejčitelnější.
5. Automat uložte.
6. Automat exportujte ve formátu json-xstate.

6.1.1.4 Úprava automatu

V poslední části se zkoumají ostatní pokročilé funkce, jako je úprava automatu, práce s historií akcí a vrácení kroků.

1. Zvolte pozicovací algoritmus-layered.
2. Importujte automat ze souboru `aut2.json`.
3. Odznačte stav A, aby nebyl počáteční.
4. Zvolte stav B jako počáteční.

5. Automaticky napozicujte automat.
6. Smažte stav G.
7. Smažte nějaké dva přechody.
8. Automaticky napozicujte automat.
9. Vraťte se o 4 kroky zpět.
10. Několikrát vyzkoušejte vytvořit nebo smazat stav nebo přechod a následně napozicovat automat.
11. Automaticky napozicujte automat.
12. Automat uložte.

6.1.2 Dotazník

Po uživatelském testování editoru následovalo několik otázek mířených na uživatele. Následuje seznam otázek:

1. Bylo vám jasné, jak přidávat nové stavy a přechody?
2. Bylo vám jasné, jak upravovat stavy a přechody?
3. Bylo vám jasné, jak tvořit nebo nastavovat počáteční a koncové stavy?
4. Vyhovuje vám spíše klikání na ploše nebo používaní klávesových zkratek?
5. Bylo hned jasné, jak použít editor pro splnění úkolů v testování?
6. Pokud nebylo, pomohly jasnosti tooltipy a nápověda v aplikaci?
7. Přijde vám aktuální verze nástrojové lišty srozumitelná?
8. Byla manipulace s přechody a stavy na pracovní ploše intuitivní?
9. Bylo vám jasné, jak se posouvat po pracovní ploše a přepínání mezi mody kurzoru?
10. Je vám jasný rozdíl mezi ukládáním a nahráváním v rámci aplikace a importem a exportem ze souborů?
11. Uvítaly byste další funkcionalitu automatu? Jakou?
12. Máte nějaké další připomínky k aplikaci?

6.1.3 Výsledky

Při testování nebyl objeven žádný významný problém s implementací ovládání editoru a uživatelům byla většina funkcionality jasná.

U počátečního testování bylo objeveno několik chyb, které rozbily celou aplikaci a znemožnily další práci. Většinou byly chyby vyvolány přímo uživatelskou interakcí a jednalo se o nedomyšlené krajní případy. Všechny nalezené chyby jsou ve finální verzi opraveny.

Mimo kritických chyb bylo objeveno pár dalších:

- Po resetování automatu by se měl vybraný element sám odznačit, už neexistuje.
- Některé klávesové zkratky nefungovaly.

Dále bylo objeveno několik případů, kdy uživatelé byly zmateni funkcionalitou editoru nebo očekávali jinou odpověď na jejich akce:

- Když je vybrán element automatu a uživatel klikne na pracovní plochu, měl by se výběr zrušit.
- Nefungovala zkratka `Ctrl + S` pro uložení.
- Move mode nefungoval korektně.
- DND z nástrojové lišty nefungoval korektně.
- Stisknutí levého tlačítka myši nad prázdnou plochou by mělo umožnit její posunutí.

Při sledování uživatelů a jejich reakcí při práci v editoru si autor povšiml možných vylepšení:

- Varování by mělo zmizet samo, uživatelé ho nezavírali a pak si nevšimli změny jeho obsahu.
- Úspěšné uložení by mělo být indikováno.
- V hierarchical algoritmu by se měly počáteční stavy pozicovat do vrchních vrstev, pokud je to možno.
- Popisky a ikony nástrojové lišty musí být jasnější.
- Dvojklik na tlačítko pro tvorbu přechodu by měl automaticky vytvořit přechod do stejného stavu.

Všechny zmíněné problémy nebo vylepšení byly opraveny a přidány do aplikace.

V dotazníku uživatelé většinou potvrdili odpozorované poznatky. Klávesové zkratky nepoužívali, ale vítali jejich přítomnost a někteří zmínili, že by je více využívali po lepším seznámení s editorem. Dále po implementaci úvodní obrazovky s nápovědou se snížil čas, kdy uživatel tápal a zkoumal funkcionalitu nástrojů.

Uživatelé měli několik připomínek k rozšíření funkcionality editoru. Nejvíce zazněl nápad přidat více možností pro kreslení dalších typů modelů. Objevily se požadavky na rozšíření uživatelské volby vzhledu elementů a jejich širší funkcionalitu. Zaznělo i několik dalších nápadů:

- Generování obrázku.
- Zobrazit menu elementu po kliknutím pravého tlačítka.

- Povolit označení více elementů najednou.

Nakonec je potřeba ještě zmínit několik problémů, které se vyskytly, ale byly moc komplikované pro řešení v rámci práce nebo je nelze s vybranou architekturou řešit.

- Klikání na přechody občas potřebuje velmi precizní práci s myší, obzvláště při oddálení. Kvůli případům, kdy se nachází mnoho přechodů v blízkosti, nelze oblast pro kliknutí zvětšit, protože by pak mohly existovat přechody, na které nelze kliknout.
- Indikátor počátečního stavu se může překrývat s dalším stavem. Jedná se o poměrně složitý algoritmický problém a pro správné fungování by se musela pozice indikátoru počítat v reálném čase při posunu stavu, aby indikátor obtékal jiné stavy.

6.2 Testování kódu

Pro automatické ověření funkčnosti aplikace se používá testování kódu. K tomu je využita knihovna Jest [67], s pomocí které byly vytvořeny unit testy [68] hlavních modulů aplikace.

Výhodou funckionálního programování je, že je snadné testovat jednotlivé funkce samostatně. V aplikaci se jen na několika místech nedodržují zásady funckionálního programování, ale v ostatních případech se pracuje s čistými funkcemi, které v rámci testů mají vždy jasně definovaný výstup.

S automatickými testy se může zaručit, že při změnách kódu nedojde k nevědomému rozbití části aplikace, která je pokryta testy. Pokud se tak stane, testy ohlásí, že byl změněn očekávaný výstup funkce a chyba se může opravit.

Nejdůležitějšími částmi aplikace jsou moduly importu a exportu, pozicování a vnitřní stav. Vše zmíněné je proto pokryto testy na 100%.

Soubory	Výrazy (%)	Větve (%)	Funkce (%)	Řádky (%)
src	x	x	x	x
src/components	0	0	0	0
src/hooks	0	0	0	0
src/interfaces	100	100	100	100
src/reducers	100	100	100	100
src/utils	x	x	x	x
src/utils/export	100	100	100	100
src/utils/import	100	100	100	100
src/utils/positioning	100	100	100	100
src/utils/validation	100	100	100	100

Tabulka 6.1: Pokrytí kódu testy

Závěr

Cílem práce bylo navrhnout dynamický webový editor automatů. Toho se dosáhlo s pomocí implementovaného přívětivého uživatelského prostředí, které poskytuje uživatelům snadné a responzivní ovládání s využitím DND a klávesových zkratek.

V práci použité webové technologie poskytly pohodlné vývojové prostředí k dosažení tohoto výsledku. Také umožnily vytvořit aplikaci, která je dostatečně plynulá při manipulaci s vykreslovaným automatem.

Dalším požadavkem aplikace byla implementace snadno rozšiřitelného importu a exportu automatů ze souborů. K tomu je využito modulů, které jsou dostatečně oddělené od zbytku aplikace, což jejich případné rozšíření činí velice snadné.

Požadavek na implementaci algoritmů pro automatické pozicování sdílí řešení v podobě modulů s importem a exportem. Pozicovací algoritmy dále poskytují snadné napozicování automatu do srozumitelné podoby.

Výsledná aplikace má navíc několik funkcí pro zpříjemnění kreslení automatu, jako jsou možnosti vracet se v historii, uložení rozpracovaného automatu a připnutí elementů na mřížku.

Autor práce vidí velký potenciál v možnostech dalšího rozšíření aplikace. Hotový editor slouží jako dobrý základ k rozšíření o komplikovanější funkcionalitu. Pokud se pomine rozšíření o další vstupní a výstupní formáty souborů a pozicovací algoritmy, tak existuje několik možností rozšíření, které sám autor plánuje postupem času přidat.

- Podpora tvorby hierarchických a paralelních stavových automatů.
- Využití WebAssembly pro optimalizaci rychlosti algoritmů pro pozicování.
- Dynamická vizualizace průběhu nakresleného automatu.
- Rozšíření o kreslení klasických grafů.
- Definice automatu pomocí formátů souborů přímo v aplikaci.

Literatura

1. *Xstate: State machines and statecharts for the modern web* [online] [cit. 2019-04-20]. Dostupné z: `https://github.com/davidkpiano/xstate`.
2. *Developer Survey Results 2019* [online]. 2019 [cit. 2019-04-20]. Dostupné z: `https://insights.stackoverflow.com/survey/2019\#technology`.
3. ŠESTÁKOVÁ, Eliška. Konečné automaty. In: *Automaty a gramatiky: Sbírka řešených příkladů*. 1. vydání. Praha: ČVUT, 2017, s. 22–23. ISBN 978-80-01-06306-4.
4. MAZETTI, Viktor; SÖRENSSON, Hannes. *Visualisation of state machines using the Sugiyama framework*. Göteborg, Sweden, 2012. Dostupné také z: `http://publications.lib.chalmers.se/records/fulltext/161388.pdf`. Diplomová práce. Chalmers University of Technology.
5. TAMASSIA, Roberto. *Handbook of Graph Drawing and Visualization*. 1st ed. Providence, Rhode Island: CRC Press, 2013. ISBN 978-1584884125.
6. KOBOUROV, Stephen. Force-Directed Drawing Algorithms. In: *Handbook of Graph Drawing and Visualization*. 1st ed. Providence, Rhode Island: CRC Press, 2013, s. 383–408. ISBN 978-1584884125.
7. ELLIOT, Eric. *What is Functional Programming?* [online] [cit. 2019-05-11]. Dostupné z: `https://medium.com/javascript-scene/master-the-javascript-interview-what-is-functional-programming-7f218c68b3a0`.
8. *ES6* [online] [cit. 2019-04-29]. Dostupné z: `https://www.w3schools.com/js/js_es6.asp`.
9. MATULA, Radek. *Grafický editor konečných automatů a kaskády*. Brno, 2009. Dostupné také z: `http://hdl.handle.net/10467/61693`. Diplomová práce. Vysoké učení technické v Brně.

10. *JQuery* [online] [cit. 2019-04-29]. Dostupné z: https://jquery.com/.

11. LESSNER, Petr. *Vizualizace konečných automatů*. Praha, 2012. Dostupné také z: https://alfresco.fit.cvut.cz/share/proxy/alfresco/api/node/content/workspace/SpacesStore/8f10f864-c3d8-4adf-904c-db33fd752c54. Bakalářská práce. České vysoké učení technické.

12. *Java applet* [online] [cit. 2019-04-29]. Dostupné z: https://docs.oracle.com/javase/tutorial/deployment/applet/index.html.

13. *JUNG* [online] [cit. 2019-04-29]. Dostupné z: https://github.com/jrtom/jung.

14. KVASNIČKA, Michal. *Vektorový grafický editor v prostředí webového klienta*. Praha, 2013. Dostupné také z: https://alfresco.fit.cvut.cz/share/proxy/alfresco/api/node/content/workspace/SpacesStore/95e3e3ff-d8a8-46f7-b351-f95042cd603a. Bakalářská práce. České vysoké učení technické.

15. *RaphaëlJS* [online] [cit. 2019-04-29]. Dostupné z: http://raphaeljs.com/.

16. *JQueryUI* [online] [cit. 2019-05-08]. Dostupné z: https://api.jqueryui.com/.

17. EVSEENKO, Iuliia. *Webový nástroj pro kreslení UML diagramů tříd*. Praha, 2018. Bakalářská práce. České vysoké učení technické.

18. KŮS, Jiří. *Nástroj pro vizualizaci a kreslení konečných automatů*. Brno, 2009. Bakalářská práce. Masarykova Univerzita.

19. MATULA, Radek. *Grafická reprezentace grafů*. Brno, 2009. Diplomová práce. Masarykova Univerzita.

20. *Graphviz: Graph Visualization Software* [online] [cit. 2019-04-20]. Dostupné z: https://www.graphviz.org/.

21. *DOT* [online] [cit. 2019-04-29]. Dostupné z: https://www.graphviz.org/doc/info/lang.html.

22. *Dagre: Directed graph layout for JavaScript* [online] [cit. 2019-04-20]. Dostupné z: https://github.com/dagrejs/dagre.

23. *Rappid* [online] [cit. 2019-04-20]. Dostupné z: https://www.jointjs.com/.

24. *Nomnoml* [online] [cit. 2019-04-29]. Dostupné z: https://github.com/skanaar/nomnoml.

25. *Canviz* [online] [cit. 2019-04-29]. Dostupné z: https://code.google.com/archive/p/canviz/.

26. *JFLPAP* [online] [cit. 2019-04-29]. Dostupné z: http://www.jflap.org/.

27. *Automata editor* [online] [cit. 2019-04-29]. Dostupné z: https://sourceforge.net/projects/automataeditor/.

28. *YEd Graph Editor* [online] [cit. 2019-04-29].
Dostupné z: https://www.yworks.com/products/yed.

29. *DRAW.IO* [online] [cit. 2019-04-29]. Dostupné z: https://www.draw.io/.

30. *Lucidchart* [online] [cit. 2019-04-29].
Dostupné z: https://www.lucidchart.com/pages/.

31. *Gliffy* [online] [cit. 2019-04-29]. Dostupné z: https://www.gliffy.com/.

32. *JavaScript* [online] [cit. 2019-04-29]. Dostupné z:
https://developer.mozilla.org/en-US/docs/Web/JavaScript.

33. *TypeScript* [online] [cit. 2019-04-20].
Dostupné z: https://www.typescriptlang.org/.

34. *TypeScript Documentation* [online] [cit. 2019-04-29]. Dostupné z:
http://www.typescriptlang.org/docs/handbook/basic-types.html.

35. *JavaScript versions* [online] [cit. 2019-04-29].
Dostupné z: https://www.w3schools.com/js/js_versions.asp.

36. *Angular* [online] [cit. 2019-04-29]. Dostupné z: https://angular.io/docs.

37. *Vue.js* [online] [cit. 2019-04-29]. Dostupné z: https://vuejs.org/v2/guide/.

38. *React: A JavaScript library for building user interfaces* [online]
[cit. 2019-04-20]. Dostupné z: https://reactjs.org/.

39. LEWIS, Paul. *Rendering Performance* [online] [cit. 2019-04-29].
Dostupné z: https://developers.google.com/web/fundamentals/
performance/rendering/.

40. *Virtual DOM* [online] [cit. 2019-04-29].
Dostupné z: https://reactjs.org/docs/faq-internals.html.

41. *React Component* [online] [cit. 2019-04-29].
Dostupné z: https://reactjs.org/docs/react-
component.html?utm_source=caibaojian.com.

42. ABRAMOV, Dan. *Presentational and Container Components* [online]
[cit. 2019-04-29].
Dostupné z: https://medium.com/@dan_abramov/smart-and-dumb-
components-7ca2f9a7c7d0.

43. *Flux* [online] [cit. 2019-04-29].
Dostupné z: https://facebook.github.io/flux/docs/in-depth-
overview.html\#content.

44. *Context API* [online] [cit. 2019-04-29].
Dostupné z: https://reactjs.org/docs/context.html.

45. ABRAMOV, Dan. *Functional components* [online] [cit. 2019-04-29].
Dostupné z: https://overreacted.io/how-are-function-components-
different-from-classes/.

46. *React Hooks* [online] [cit. 2019-04-29].
Dostupné z: https://reactjs.org/docs/hooks-intro.html.

47. ABRAMOV, Dan. *React Hooks Benchmarks* [online] [cit. 2019-04-29].
Dostupné z: https://medium.com/@dan_abramov/this-benchmark-is-indeed-flawed-c3d6b5b6f97f.

48. *Redux: A predictable state container for JavaScript apps* [online]
[cit. 2019-04-20]. Dostupné z: https://redux.js.org/.

49. *React-redux* [online] [cit. 2019-04-29].
Dostupné z: https://github.com/reduxjs/react-redux.

50. *Redux: API reference* [online] [cit. 2019-04-29].
Dostupné z: https://redux.js.org/api/api-reference.

51. *Redux usage* [online] [cit. 2019-05-08].
Dostupné z: https://stackshare.io/reduxjs.

52. *Redux libraries* [online] [cit. 2019-05-08].
Dostupné z: https://github.com/xgrommx/awesome-redux\#react---a-javascript-library-for-building-user-interfaces.

53. *Redux dev tools* [online] [cit. 2019-04-29].
Dostupné z: https://github.com/reduxjs/redux-devtools.

54. *Redux undo* [online] [cit. 2019-04-29].
Dostupné z: https://github.com/omnidan/redux-undo.

55. *Canvas API* [online] [cit. 2019-04-29]. Dostupné z:
https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API.

56. SMUS, Boris. *SVG performance* [online] [cit. 2019-04-29].
Dostupné z: https://smus.com/canvas-vs-svg-performance/.

57. *SVG: Scalable Vector Graphics* [online] [cit. 2019-04-20].
Dostupné z: https://developer.mozilla.org/en-US/docs/Web/SVG.

58. *Material UI:*
React components that implement Google's Material Design. [online]
[cit. 2019-04-20]. Dostupné z: https://material-ui.com/.

59. *File saver* [online] [cit. 2019-04-29].
Dostupné z: https://github.com/eligrey/FileSaver.js.

60. *HTML5 Drag and Drop* [online] [cit. 2019-04-20]. Dostupné z:
https://www.w3schools.com/htmL/html5_draganddrop.asp.

61. *SVG Guidelines: Bezier Curves* [online] [cit. 2019-04-29].
Dostupné z: https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Paths\#Bezier_Curves.

62. DEMAINE, Erik; DEVADAS, Srini. *Introduction to Algorithms: Depth-first search (DFS), topological sorting* [online]. Massachusetts Institute of Technology: MIT OpenCouseWare, 2011 [cit. 2019-05-08]. Dostupné z: `https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/MIT6_006F11_lec14_orig.pdf`.

63. GÖTZ, Carlo. *Layered Graph Drawing: The Sugiyama Method* [online]. 2017 [cit. 2019-05-08]. Dostupné z: `https://blog.disy.net/sugiyama-method/`.

64. *SVG2 support* [online] [cit. 2019-04-29]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/SVG/SVG_2_support_in_Mozilla`.

65. *Blob* [online] [cit. 2019-05-08]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/API/Blob`.

66. *TypeDoc* [online] [cit. 2019-05-08]. Dostupné z: `https://github.com/TypeStrong/typedoc`.

67. *Jest* [online] [cit. 2019-05-08]. Dostupné z: `https://jestjs.io/docs/en/getting-started`.

68. MESZAROS, Gerard. *Unit test* [online]. 2009 [cit. 2019-05-08]. Dostupné z: `http://xunitpatterns.com/unit\%20test.html`.

Příloha A

Seznam použitých zkratek

API Application programming interface.

CSS Cascading Style Sheets.

DND Drag and drop.

DOM Document Object Model.

ES6 EcmaScript, verze 6.

HTML Hypertext Markup Language.

HTML5 Hypertext Markup Language, verze 5.

JSON JavaScript Object Notation.

PDF Portable Document Format.

PHP Hypertext Preprocessor.

SVG Scalable Vector Graphics.

SVG2 Scalable Vector Graphics, verze 2.

UML Unified Modeling Language.

Virtual DOM Virtual Document Object Model.

XML Extensible Markup Language.

Příloha B

Obsah přiloženého CD

- `app` ..... zdrojové soubory implementace
  - `src` ..... zdrojový kód
- `build` ..... spustitelná forma implementace
  - `index.html` ..... webová aplikace
- `docs` ..... dokumentace zdrojových kódů
  - `index.html` ..... dokumentace
- `thesis` ..... zdrojové soubory práce
  - `BP_Svoboda_Petr_2019.tex` ..... práce ve formátu LaTeX
- `readme.txt` ..... stručný popis obsahu CD
- `thesis.pdf` ..... text práce ve formátu PDF

Příloha C

Uživatelská příručka

C.1 Spuštění aplikace

Aplikace je zkompilovaná a ke spuštění stačí otevřít soubor `build\index.html`

C.2 Zkompilování aplikace

Pro zkompilování je třeba nainstalovat několik aplikací.

- NodeJS, verze 12.2.0
- yarn, verze 1.15.2

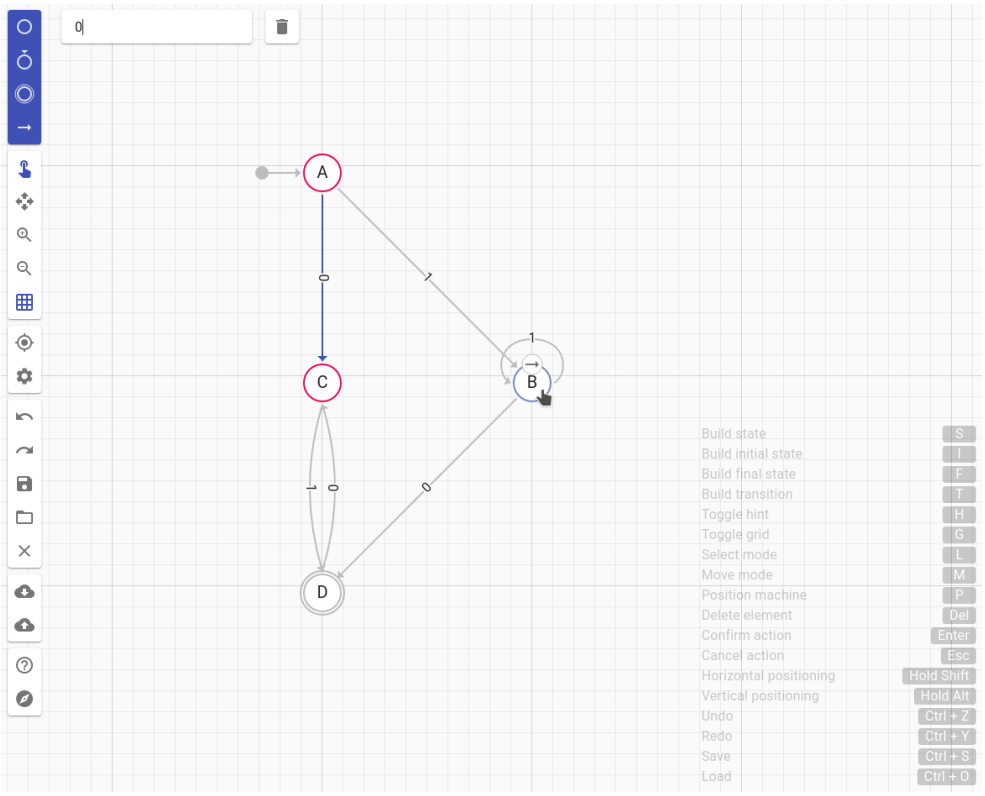
Po nainstalování se ze složky `app` spustí příkaz `yarn build`.

Pro spuštění vývojového serveru s automatickým načtení změn v souborech se použije příkaz `yarn start`. Aplikace je pak přístupná na adrese `http://localhost:3000/`.

Testy aplikace se spustí příkazem `yarn test`.

Příloha D

Ukázka aplikace



Obrázek D.1: Ukázka celé aplikace s nakresleným automatem a označeným přechodem 0 mezi stavy A a C