České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů

Diplomová práce

Grafický editor konečných automatů a kaskády

Bc. Martin Adámek

Vedoucí práce: Ing. Josef Kufner

Studijní program: Otevřená informatika

Obor: Softwarové inženýrství

9. května 2015

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačové grafiky a interakce

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Martin Adámek**

Studijní program: Otevřená informatika
Obor: Softwarové inženýrství

Název tématu: **Grafický editor konečných automatů a kaskády**

Pokyny pro vypracování:

1 Nastudujte existující grafické editory konečných automatů.
2. Nastudujte framework pro vytváření webových aplikací postavený na kaskádě (acyklický orientovaný graf popisující provádění programu).
3. Navrhněte a implementujte grafický editor kaskády pro použití ve webové aplikaci. Editor bude realizován jako jQuery plugin nahrazující element textarea v běžném HTML formuláři popisující daný automat.
4. Rozšiřte vytvořený editor kaskády pro editaci konečných automatů.
5. Kvalitu a použitelnost uživatelských rozhraní editorů ověřte uživatelským testováním. Výsledky testování zdokumentujte a případné zjištěné nedostatky napravte.
6. Experimentálně zjistěte maximální velikost upravovaného grafu, pro kterou jsou vytvořené editory použitelné.

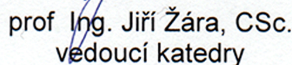
Seznam odborné literatury:

Kufner, Josef, and Radek Mařík. "Self-generating Programs–Cascade of the Blocks." Information and Communication Technology Springer Berlin Heidelberg, 2014. 199-212.

Kufner, Josef, and Radek Mařík. "State Machine Abstraction Layer" Information and Communication Technology Springer Berlin Heidelberg, 2014. 213-227

Vedoucí: Ing. Josef Kufner

Platnost zadání: do konce letního semestru 2015/2016



prof. Ing. Jiří Žára, CSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 25. 3. 2015

Poděkování

Děkuji všem, kteří mě podporovali při vytváření této práce, především pak vedoucímu práce, panu Ing. Josefu Kufnerovi, za poskytnuté rady a pomoc a mým rodičům za podporu po celou dobu mého studia.

Abstrakt

Tato práce se zabývá návrhem a implementací dvou grafických editorů – editoru kaskády bloků a editoru stavových automatů *Smalldb* – oba realizované jako JavaScriptové pluginy postavené na frameworku *jQuery*, tedy čistě klientské aplikace běžící v prohlížeči. Využívá moderních technologií HTML5 a CSS3, díky kterým se obejde bez závislostí na dalších knihovnách, které by ztěžovaly integraci do aplikací. U obou editorů se pak práce zaměří na přehlednou vizualizaci entit, se kterými pracuje, zejména pak na vykreslování hran grafu (spojnic mezi jednotlivými bloky, resp. přechodů automatu) bez kolizí a výpočet pozic uzlů grafu.

Klíčová slova

Grafický editor, stavový automat, kaskáda, Smalldb, blokové programování, JavaScript, jQuery, HTML5, CSS3, Bézierova křivka, Cardinal spline, JSON, detekce kolizí ve 2D, kreslení grafů.

Abstract

This thesis describes the design and implementation of two editors – one for cascade of blocks and the other for *Smalldb* state machines – both implemented as JavaScript plugins developed with *jQuery*, and therefore purely client-side applications running in a web browser. It uses advanced technologies HTML5 and CSS3, thanks to which it dispenses with the other dependencies. The work will then focus on clear visualization of the entities which both editors work with, especially on rendering graph edges (connections between blocks, state machine transitions) without collisions and calculating the positions for graph nodes.

Keywords

Visual editor, state machine, cascade, Smalldb, block programming, JavaScript, jQuery, HTML5, CSS3, Bézier curve, Cardinal spline, JSON, 2D collision detection, graph drawing.

Prohlášení

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

V Praze dne 9. května 2015. ....

Obsah

1 Úvod **17**
1.1 Motivace 17
1.2 Cíle a řešení 17
1.3 Komunikace editoru se serverem 18
1.4 Použité technologie 18
1.5 Členění textu 20

2 Teoretické základy – kaskáda **21**
2.1 Co je to kaskáda 21
2.2 Blok - základní kámen kaskády 22
2.3 Spojování bloků 22
2.4 Rostoucí kaskáda a jmenné prostory 23
2.5 Vyhodnocování 23
2.6 Vizualizace 23

3 Teoretické základy – Smalldb **25**
3.1 Co je to Smalldb 25
3.2 Nedeterministický konečný automat 25
3.3 Nedeterminismus 26
3.4 Vlastnosti automatu z pohledu editoru 26

4 State of the Art **29**
4.1 Framework postavený na kaskádě 29
4.1.1 Editor kaskády 29
4.1.2 Smalldb 29
4.2 Nástroje pro vizualizaci grafů 30
4.2.1 Graphviz 30
4.2.2 Canviz 30
4.2.3 Ti*k*Z/PGF 31
4.2.4 PSTrics 31
4.2.5 MetaPost 32
4.2.6 JointJS/Rappid 32
4.2.7 Dagre 33
4.3 Obecné editory grafů 34
4.3.1 yEd 34
4.3.2 Visual Graph Editor 34
4.4 Editory konečných automatů 36

4.4.1 UPPAAL 36
4.4.2 Yakindu SCT 37

5 Grafický editor kaskády 39
5.1 Analýza 39
5.1.1 Funkční požadavky 39
5.1.2 Nefunkční požadavky 40
5.1.3 Formát a schéma kaskády 41
5.1.4 Uživatelské role 42
5.1.5 Případy použití 43
5.2 Návrh 44
5.2.1 Diagram tříd 44
5.2.2 Reprezentace geometrických objektů 47
5.2.3 Uživatelské rozhraní 47
5.2.4 Kreslení spojnic 50
5.3 Implementace 51
5.3.1 Inicializace a konfigurace 52
5.3.2 Výpočet průsečíků 52
5.3.3 Výpočet úhlů pomocí atan2 53
5.3.4 Drag'n'drop 55
5.3.5 Úložiště prohlížeče 56
5.3.6 Přiblížení/oddálení 56
5.3.7 Historie 56
5.3.8 Schránka 57
5.3.9 Načítání palety bloků 57
5.3.10 Režim statického obrázku 57
5.4 Algoritmus pro vyhýbání se blokům 57
5.4.1 Naivní algoritmus 58
5.4.2 Nejkratší cesta v gridu 58
5.4.3 Nejkratší cesta v eukleidovském grafu 60
5.4.4 Srovnání a výběr řešení 60
5.5 Testování 63
5.5.1 Heuristická evaluace 63
5.5.2 Uživatelské testování 63
5.6 Experimenty 65
5.6.1 Konfigurace testovacího prostředí 65
5.6.2 Srovnání rychlosti implementovaných algoritmů 65
5.6.3 Maximální velikost grafu 66

6 Grafický editor konečných automatů 69
6.1 Analýza 69
6.1.1 Funkční požadavky 69
6.1.2 Nefunkční požadavky 70
6.1.3 Formát a schéma automatu 70
6.1.4 Uživatelské role 73
6.1.5 Případy užití 73
6.2 Návrh 74

6.2.1 Diagram tříd 74
6.2.2 Shodné vlastnosti s editorem kaskády 75
6.2.3 Uživatelské rozhraní 75
6.2.4 Kreslení spojnic 78
6.2.5 Výpočet bodu na spojnici 79
6.3 Implementace 80
6.3.1 Inicializace a konfigurace 80
6.3.2 Živá editace 80
6.3.3 Vlastnosti akce a přechodu 81
6.3.4 Výpočet bodu na elipse 81
6.3.5 Obarvování hran 81
6.4 Automatické rozmístění stavů 82
6.4.1 Silně souvislé komponenty 83
6.4.2 Algoritmus knihovny Dagre 83
6.4.3 Srovnání a výběr řešení 85
6.5 Možná budoucí rozšíření 86
6.5.1 Oprávnění 86
6.5.2 Hierarchické automaty 86
6.5.3 Obecný editor grafů 86
6.5.4 Minimalizace automatu 87
6.5.5 Desktopová verze 87
6.6 Testování 87
6.6.1 Heuristická evaluace 87
6.6.2 Uživatelské testování 88
6.7 Experimenty 90
6.7.1 Konfigurace testovacího prostředí 90
6.7.2 Srovnání rychlosti implementovaných algoritmů 90
6.7.3 Maximální velikost grafu 90

7 Závěr **93**

A Nielsenovy heuristiky **95**
A.1 Visibility of system status 95
A.2 Match between system and the real world 95
A.3 User control and freedom 95
A.4 Consistency and standards 96
A.5 Error prevention 96
A.6 Recognition rather than recall 96
A.7 Flexibility and efficiency of use 96
A.8 Aesthetic and minimalist design 96
A.9 Help users recognize, diagnose, and recover from errors 96
A.10 Help and documentation 96

B Referenční příručka – Editor kaskády **97**
B.1 Class: Block 97
B.2 Class: BlockEditor 99
B.3 Class: Canvas 100

B.4 Class: Editor 101
B.5 Class: Grid 102
B.6 Class: Line 102
B.7 Class: Palette 103
B.8 Class: ParentEditor 103
B.9 Class: Placeholder 104
B.10 Class: Point 106
B.11 Class: Spline 107
B.12 Class: Storage 107
B.13 Class: Toolbar 107

C Referenční příručka – Editor stavových automatů 109

C.1 Class: Action 109
C.2 Class: Canvas 110
C.3 Class: Editor 111
C.4 Class: Graph 112
C.5 Class: Line 112
C.6 Class: Node 113
C.7 Class: Point 113
C.8 Class: SmalldbEditor 113
C.9 Class: Spline 115
C.10 Class: Stack 115
C.11 Class: State 115
C.12 Class: Storage 117
C.13 Class: Tarjan 117
C.14 Class: Toolbar 117
C.15 Class: Transition 119

D Obsah přiloženého CD 121

Literatura 123

Rejstřík 127

Seznam obrázků

1.3.1 Schéma komunikace editoru se serverem 18

2.1.1 Celkový pohled na aplikaci (zdroj:[1]) 21
2.2.1 Grafická reprezentace bloku (zdroj: [1]) 22
2.3.1 Spojení více bloků do kaskády (zdroj: [1]) 22
2.4.1 Kaskáda před (vlevo) a po (vpravo) vykonání bloku A (zdroj: [1]) . . 23

3.3.1 Stavový diagram článku na blogu (zdroj: [4]) 26

4.1.1 Stávající podoba editoru kaskády (zdroj: [1]) 30
4.2.1 Ukázkový kód a jeho výstup – knihovna Ti*k*Z/PGF 31
4.2.2 Ukázkový kód a jeho výstup – knihovna PSTrics 32
4.2.3 Ukázkový kód a jeho výstup – knihovna MetaPost 32
4.2.4 Rappid (zdroj: [16]) 33
4.2.5 Ukázka výstupu Dagre (zdroj: [17]) 35
4.3.1 yEd (zdroj: [18]) 36
4.3.2 Visual Graph Editor (zdroj: [19]) 37
4.4.1 UPPAAL (zdroj: [20]) 38
4.4.2 Yakindu SCT (zdroj: [21]) 38

5.1.1 Vizualizace fragmentu z předchozí ukázky kódu 42
5.1.2 UML Use Case diagram editoru kaskády 43
5.2.1 Diagram tříd editoru kaskády 45
5.2.2 Diagram pomocných geometrických tříd 47
5.2.3 Mockup editoru kaskády – plátno s paletou a panelem nástrojů . . . 48
5.2.4 Mockup editoru kaskády – editor vstupu 49
5.2.5 Mockup editoru kaskády – editor nadřazeného bloku 50
5.2.6 Kubická spline křivka se znázorněnými kontrolními body 51
5.3.1 Výsledná podoba editoru kaskády 52
5.3.2 Výpočet průsečíku dvou přímek 53
5.3.3 Grafické znázornění hodnot funkce `atan(y,x)` (zdroj: [25]) 55
5.4.1 A* v gridu 59
5.4.2 Problém velkého počtu hran – A* v grafu (zdroj: [28]) 60
5.4.3 Srovnání algoritmů pro vyhýbání se blokům – bez vyhýbání 61
5.4.4 Srovnání algoritmů pro vyhýbání se blokům – naivní algoritmus . . . 61
5.4.5 Srovnání algoritmů pro vyhýbání se blokům – nejkratší cesta s A* v gridu, s vyhlazováním cesty 62

5.4.6 Srovnání algoritmů pro vyhýbání se blokům – nejkratší cesta s A* v gridu, bez vyhlazování cesty 62
5.6.1 Graf závislosti doby vykreslování na počtu bloků ve fragmentu kaskády 67
5.6.2 Fragment kaskády s padesáti bloky – algoritmus A* v gridu bez vyhlazování 67

6.1.1 Automat s větším počtem cyklů nad jedním stavem 70
6.1.2 Vizualizace entity z předchozí ukázky 73
6.1.3 UML Use Case diagram Smalldb editoru 73
6.2.1 Diagram tříd Smalldb editoru 76
6.2.2 Mockup Smalldb editoru – plátno s paletou a panelem nástrojů . . . 77
6.2.3 Mockup Smalldb editoru – tři režimy zobrazení editovacího panelu . 78
6.2.4 Kreslení přechodů mezi dvěma stavy 78
6.2.5 Aproximace Bézierovi křivky – vizualizace algoritmu *de Casteljau* (zdroj: [29]) 79
6.3.1 Výsledná podoba Smalldb editoru stavových automatů 80
6.3.2 Výpočet bodu na elipse (zdroj: [30]) 82
6.3.3 Automat reprezentující článek v redakčním systému – Smalldb editor 82
6.4.1 Křížení hran při špatně vypočítaném řazení 83
6.4.2 Automat vykreslený pomocí knihovny *Dagre* se zobrazenými kontrolními body křivek 84
6.4.3 Srovnání algoritmů pro kreslení automatu 86
6.5.1 Příklad hierarchického automatu (zdroj: [38]) 87
6.7.1 Graf závislosti doby výpočtu na počtu stavů 91
6.7.2 Automat o šedesáti stavech a sto dvaceti přechodech 91

Kapitola 1

Úvod

1.1 Motivace

Článek, uživatel nebo objednávka v internetovém obchodě. To vše jsou entity, se kterými vývojáři webových aplikací pracují na denní bázi. Každá z těchto entit reflektuje business logiku, kterou lze přirozeně zapsat pomocí stavového automatu (např. článek může nabývat stavů *Skrytý, Zveřejněný, Smazaný*). Pro snadnou integraci takových entit do webové aplikace lze využít frameworku Smalldb, postaveného právě na stavových automatech. Ten se postará o to, aby se nikdy entita nedostala do neplatného stavu. Každá entita je definována v souboru JSON[1], jeho ruční úprava ale není moc komfortní. To se snaží napravit grafický editor, jehož tvorbou se zabývá tato práce.

1.2 Cíle a řešení

Cílem této práce je vytvořit dva velmi si blízké grafické editory - jeden pro tvorbu kaskády[2] bloků a druhý pro tvorbu stavových automatů. Prvním cílem je navrhnout a vytvořit vizuální nástroje, které mohou být začleněny do administrační sekce webové aplikace, kterou bude pohánět framework postavený na kaskádě [1] (viz sekce 1.4, více pak v 2.1). Druhým cílem je oba editory uživatelsky otestovat a případné nedostatky opravit. Posledním cílem je experimentálně ověřit technické možnosti editorů, například maximální možnou velikost grafu, při které ještě dokáží plynule pracovat.

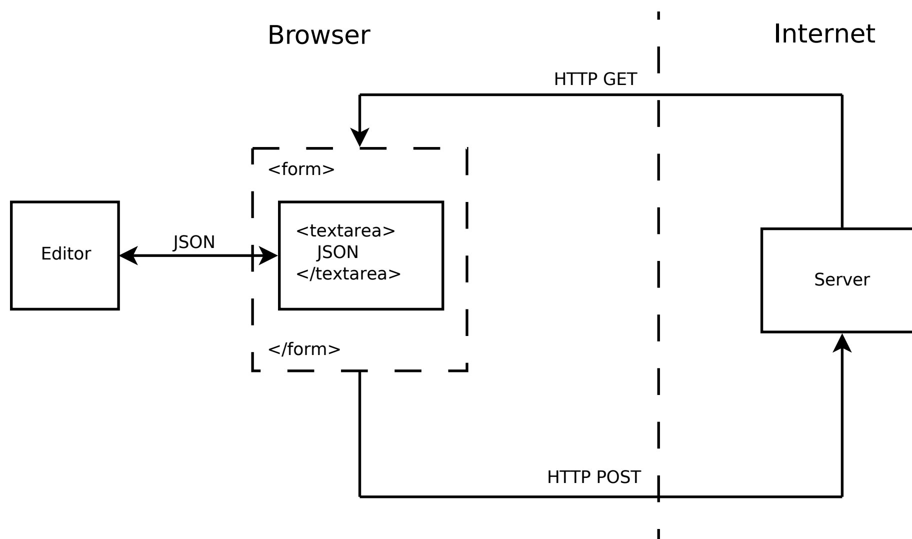
Oba editory budou provedeny jako JavaScriptové pluginy. Plugin umožní editaci libovolného stavového automatu, resp. fragmentu kaskády, které jsou definovány jako strukturovaná data ve formátu JSON uvnitř elementu `<textarea>`. Dále bude podporovat mód prostého zobrazování, ve kterém se bude chovat jako obrázek a veškerá interakce s uživatelem bude potlačena. Pro testování obou editorů bude vedle testu s uživatelem provedena i jejich heuristická evaluace na základě Nielsenových heuristik [2].

[1] Formát JSON: `http://json.org/`

[2] Kaskáda je dynamická struktura složená z navzájem propojených bloků, která slouží k řízeńı jejich postupného spouštění a předávání dat mezi nimi. Jednotlivé bloky si ze svých vstupů načtou data, zpracují je a výsledky nastaví na své výstupy.

1.3 Komunikace editoru se serverem

Role obou editorů začíná a končí hodnotou elementu `<textarea>`, nad kterým jsou spuštěny. Editor se nestará o ukládání dat, pouze nad nimi pracuje a aktualizuje je. Žije pouze v prohlížeči, tedy v klientské části, kde vizualizuje data uložená ve formátu JSON. Práce s daty zůstává na programátorovi, který editor do aplikace integruje. Přenos dat probíhá pomocí klasického HTML formuláře (element `<form>`), data do něj se načtou spolu s celou stránkou pomocí HTTP GET požadavku. Formulář pak na server odesílá data pomocí požadavku HTTP POST. Schéma komunikace editoru se serverem je znázorněno na obrázku 1.3.1.



Obrázek 1.3.1: Schéma komunikace editoru se serverem

1.4 Použité technologie

Při implementaci obou editorů bylo použito následujících technologií:

Kaskáda

Kaskáda [1, 3] je dynamická acyklická struktura sestavená z bloků, která nabízí nový pohled na programování webových aplikací. Jednotlivé bloky jsou propojeny do kaskády, která řídí i znázorňuje jejich spouštění a předávání dat mezi nimi. Bližší seznámení s kaskádou popisuje kapitola 2.

Smalldb

Smalldb [4] je framework pro implementaci modelu (ve smyslu MVC), který popisuje pomocí stavového automatu. Bližší seznámení se Smalldb popisuje kapitola 3.

JavaScript

JavaScript [5] je skriptovací jazyk určený primárně pro webové prohlížeče. V posledních letech ale JavaScript zažil velký *boom* s příchodem serverových frameworků jako je Node.js [6] nebo Angular.js [7]. JavaScript jako takový nemá nativní podporu pro koncept tříd. Objekty v JavaScriptu jsou ve skutečnosti neuspořádané množiny dvojic (klíč, hodnota). Klíč je vždy řetězec, hodnota může být cokoliv (jiný objekt, funkce, pole). Objekty se předávají jako reference, vytvoření nového objektu lze vynutit klíčovým slovem `new`.

JavaScript má dynamický typový systém, pro kontrolu typu objektu lze využívá přístup zvaný *duck typing*. Dle něj není nutné kontrolovat objekt podle jeho typu, ale pouze podle jeho vlastností a metod.

Dědičnost se dá v JavaScriptu docílit pomocí prototypování [8]. Každý objekt má odkaz na jiný objekt – zvaný *prototyp* – a ten má opět odkaz na svůj prototyp a tak dále, až se na konci objeví objekt `null`. Tomuto zřetězení objektů se říká *prototype chain*.

Framework jQuery

Jedinou závislostí pro oba editory bude framework jQuery [9]. Jedná se o knihovnu, sloužící k jednodušší práci s DOM[3]. Její hlavní přednost je v kompatibilitě mezi různými prohlížeči. Obsahuje také podporu pro jednoduchou práci s technologií AJAX[4]. jQuery je možné rozšířit pomocí pluginů, kterých dnes existuje velké množství. Se specifikací HTML5 ale přichází nové nativní možnosti (drag'n'drop, práce se soubory, multimédii, websockety, ...), díky kterými je dnes velká část těchto pluginů zbytečná.

HTML5

Finální verze HTML5 vyšla teprve 28. října 2014, ale už delší dobu fungovalo jako tzn. *working draft*. Dnes ho podporují všechny důležité prohlížeče. HTML5 nepřináší jen nové značky (například nové formulářové prvky `range`, `color`, `datetime`), ale také zároveň nové JavaScriptové API (například pro práci s formátem JSON). Přibyla i podpora pro ukládání dat v prohlížeči (`localStorage` a `sessionStorage`; více viz sekce 5.3.5) a pro nás nejdůležitější element `<canvas>`. Ten přináší podporu kreslení grafiky pomocí JavaScriptu, která je podstatně rychlejší než SVG[5] a je tudíž vhodnější pro vykreslování v interaktivních aplikacích.

[3] DOM: Document Object Model
[4] AJAX: Asynchronous JavaScript and XML
[5] SVG: Scalable Vector Graphics

Podpora prohlížečů

Díky využití moderních technologií bude pro úspěšné spuštění vyžadován moderní prohlížeč. Pro editor je klíčová podpora HTML5 elementu `<canvas>`[6] a dále podpora pro práci s formátem JSON[7]. Pro implementaci přibližování/oddalování (viz 5.1.1) je dále nutná podpora CSS3 transformací[8][9]. Podpora prohlížečů je znázorněna v tabulce 1.1.

Schopnost	Chrome	Internet Explorer	Firefox	Safari	Opera
`<canvas>`	1.0	9.0	1.8	2.0	9.0
`JSON API`	1.0	8.0	1.9.1	4.0	10.5
`CSS3 transform`	4.0	9.0	3.5	3.2	10.5
Celková podpora	4.0	9.0	3.5	4.0	10.5

Tabulka 1.1: Podpora prohlížečů

1.5 Členění textu

Druhá a třetí kapitola popisují nutné teoretické základy, na kterých tato práce staví – kaskádu a Smalldb. Čtvrtá kapitola nabízí přehled existujících řešení, která jsou pro tuto práci relevantní. Pátá kapitola se zabývá analýzou, návrhem a implementací editoru kaskády, jeho testováním a experimenty. Šestá kapitola nabízí stejný pohled na editor stavových automatů Smalldb.

První příloha popisuje deset heuristik dle Jacoba Nielsena, dle kterých je provedena heuristická analýza jednotlivých editorů. Druhá příloha obsahuje referenční příručku obou editorů generovanou na základě komentářů v kódu. Třetí příloha popisuje obsah přiloženého CD.

[6] `<canvas>`: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/canvas
[7] JSON: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON
[8] CSS3 transformace: http://www.w3schools.com/css/css3_2dtransforms.asp
[9] Uvedené verze prohlížečů implementují CSS3 transformace s prefixem

Kapitola 2

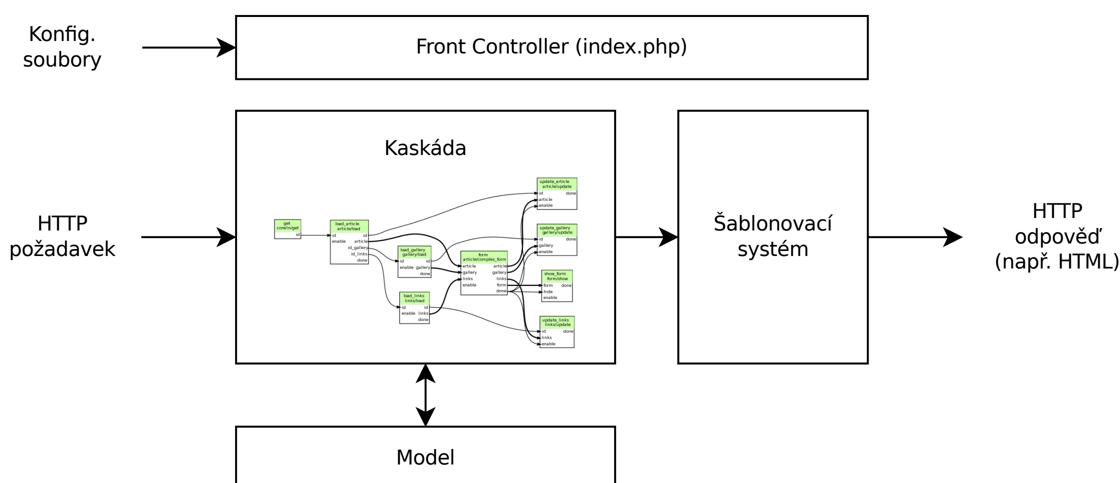
Teoretické základy – kaskáda

2.1 Co je to kaskáda

Kaskáda [3] je dynamická acyklická struktura sestavená z bloků. Je inspirována funkčními bloky, které se využívají pro programování PLC[1] a architekturou *Pipes and Filters*, na které stojí unixový shell[2]. Na rozdíl od nich se ale v kaskádě nemění data, ale pouze struktura. Jakmile je jednou blok vykonán, jeho výstup už se nikdy nezmění.

Na obrázku 2.1.1 je vidět, jakou roli má kaskáda v klasické webové aplikaci – zastává funkci kontroleru. Rozdíl oproti klasickému MVC spočívá v odstranění vazby mezi šablonovacím systémem a modelem. Navíc jsou všechny vazby, vyjma té mezi modelem a kaskádou, jednosměrné.

Běh aplikace je rozdělen do dvou etap: vyhodnocení kaskády a běh šablonovacího systému. Při vyhodnocování kaskády je plněn šablonovací systém objekty, které reprezentují jednotlivé části výsledné stránky a teprve po dokončení zpracování posledního bloku je šablonovací systém spuštěn a je vygenerován výstup dosazením dat z připravených objektů do šablon (zdroj: [1]).



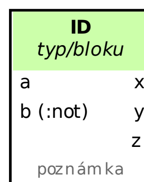
Obrázek 2.1.1: Celkový pohled na aplikaci (zdroj:[1])

[1] PLC – Programmable Logic Controller; průmyslové automaty [10].

[2] Unixový shell: textové uživatelské rozhraní používané v operačních systémech UNIX

2.2 Blok - základní kámen kaskády

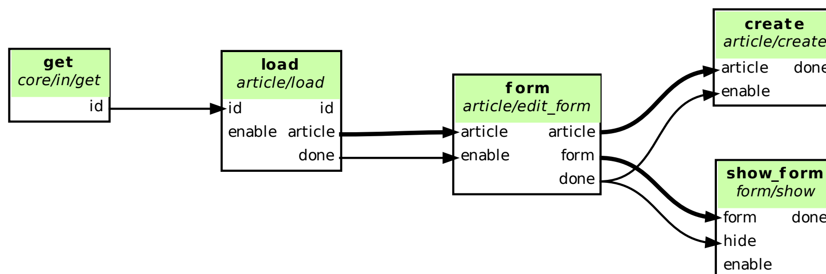
Blok je elementární entita kaskády. Každý blok (respektive jeho instance) je definován unikátním ID a typem (třídou). Blok má definované pojmenované vstupy a výstupy. Na obrázku 2.2.1 je vidět grafická reprezentace bloku využívaná ve frameworku postaveném na kaskádě, který ji vykresluje pomocí nástroje Graphviz[3]. Hlavička bloku obsahuje jeho ID na prvním řádku a typ na řádku druhém. Barva jejího pozadí odpovídá stavu bloku v daném čase (zelená značí již vykonaný blok). Tělo bloku obsahuje jednotlivé výstupy (pravá strana) a vstupy (levá strana). Některé bloky nemají pevně dané vstupy nebo výstupy, to je v editoru i v definici bloku naznačeno hvězdičkou místo možných vstupů, resp. výstupů. Jednotlivé vstupy a výstupy u takového bloku pak definuje až konkrétní instance bloku v kaskádě (a dalších bloků s ním spojených).



Obrázek 2.2.1: Grafická reprezentace bloku (zdroj: [1])

2.3 Spojování bloků

Jednotlivé bloky jsou spojeny do kaskády – každý spoj znázorňuje přenos dat z výstupu zdrojového bloku na vstup cílového bloku. Spojení jsou definována pouze na straně vstupní proměnné cílového bloku. Kaskáda stojí na důsledném zapouzdření bloků, blok tedy při vyhodnocování neví, od koho data na vstupu dostal, ani komu data na svých výstupech posílá. Příklad grafické vizualizace spojení pěti bloků pomocí programu Graphviz je znázorněna na obrázku 2.3.1. Tento fragment kaskády slouží k zobrazení a zpracování formuláře na editaci článku v redakčním systému.



Obrázek 2.3.1: Spojení více bloků do kaskády (zdroj: [1])

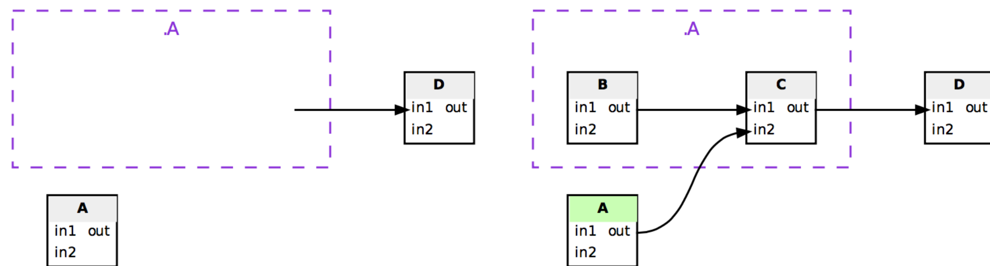
[3]Graphviz: http://www.graphviz.org/

2.4 Rostoucí kaskáda a jmenné prostory

Každý blok může při svém vykonávání dynamicky vkládat další nové bloky do kaskády. Aby nedošlo ke kolizi v identifikátoru bloku, nové bloky jsou vloženy do jmenného prostoru, který náleží vkládajícímu bloku. Na rozdíl od klasické implementace jmenných prostorů například v jazyce C je zde povoleno přistupovat do jmenného prostoru z venčí.

Jmenný prostor z pohledu kaskády je tedy její fragment, který ale vidí své okolí a může využívat vstupy a výstupy z bloků, které se nacházejí mimo něj samotný. Spojení na (v rámci fragmentu) neexistující bloky tedy mohou dávat smysl.

Editor bloků bude sloužit pro úpravu jednotlivých fragmentů. Ty budou pak při vyhodnocování vloženy do kaskády pomocí tzv. *proxy bloku*.



Obrázek 2.4.1: Kaskáda před (vlevo) a po (vpravo) vykonání bloku A (zdroj: [1])

2.5 Vyhodnocování

Vyhodnocování kaskády je proces, při kterém dojde k postupnému spouštění jednotlivých bloků v takovém pořadí, aby mohla být data požadovaná na vstupu každého bloku doručena – tzn. všechny zdrojové bloky všech vstupů daného bloku musí být vykonány před ním samotným. Toto pořadí je dané jednotlivými (jednosměrnými) spoji mezi bloky, které definují částečné uspořádání, ve kterém musí být spuštěny.

2.6 Vizualizace

K vizualizace kaskády ve frameworku postaveném na kaskádě se používá program Graphviz. Na obrázku 2.3.1 je vidět příklad kaskády sloužící k editaci článku – nejprve blok *get* načte z HTTP protokolu hodnotu *id*, tu pak předá bloku *load*, který článek načte a předá ho formuláři (blok *form*), a ten je nakonec vykreslen pomocí bloku *show_form*. Barva hlavičky bloků znázorňuje jejich stav – zelená značí stav *zombie*[4], tedy blok který úspěšně proběhl a jeho výstup je k dispozici; šedá znamená stav *queued* (dosud nevykonaný blok). V tomto případě je šedě podbarvená hlavička bloku *update*, je to právě proto, že se k jeho vykonání nedošlo (formulář nebyl odeslán).

Editor bloků bude zastávat opačnou funkci – umožní uživateli ručně sestavit kaskádu z již hotových bloků pomocí metody drag'n'drop, nakreslit mezi nimi spojení a nakonfigurovat jejich vlastnosti.

[4] Označení stavu *zombie* převzato z unixu

Kapitola 3

Teoretické základy – Smalldb

3.1 Co je to Smalldb

Smalldb [4], neboli *State Machine Abstraction Layer*, je framework pro implementaci modelu (ve smyslu MVC). Jak název napovídá, jedná se o abstrakční vrstvu, ve které se chování jednotlivých entit popisuje pomocí stavového automatu. Přechody mezi jednotlivými stavy znázorňují uživatelské akce, které mění stav entity. Stejně jako u kaskády jsou zde jednotlivé entity popsány ve formátu JSON.

3.2 Nedeterministický konečný automat

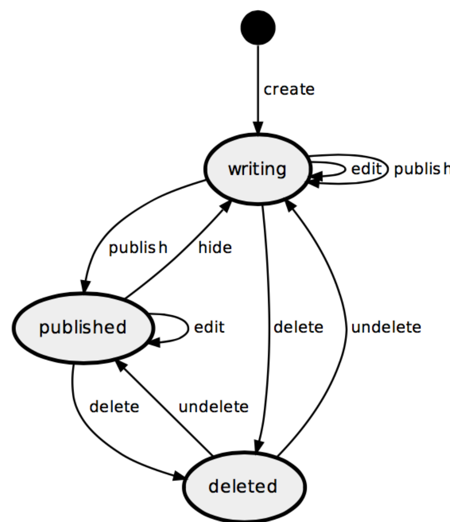
Smalldb využívá pro reprezentaci business logiky parametrický nedeterministický konečný automat. Tento automat je definován jako uspořádaná devítice $(Q, P, s, P_0, \Sigma, \Lambda, M, \alpha, \delta)$, kde:

- Q je konečná množina stavů.
- P je množina pojmenovaných vlastností. P^* je množina všech možných hodnot P. P^* nemusí být konečná. P_t je stav těchto vlastností v čase t. $P_t \in P^*$.
- s je stavová funkce $s(P_t) \mapsto q$, kde $q \in Q, P_t \in P^*$.
- P_0 je množina počátečních hodnot vlastností P, $P_0 \in P^*$.
- Σ je konečná množina parametrických vstupních událostí.
- Λ je množina parametrických výstupních událostí (volitelné)
- M je konečná množina akcí (metod): $m(P_t, e_{in}) \mapsto (P_t + 1, e_{out})$, kde $P_t, P_t + 1 \in P^*, m \in M, e_{in} \in \Sigma, e_{out} \in \Lambda$.
- α je aserční funkce: $\alpha(q_t, m) \mapsto Q_t + 1$, kde $q_t \in Q, Q_t + 1 \subset Q, e_{in} \in \Sigma$.
  - $\forall m \in M : s(P_t + 1) \in \alpha(s(P_t), m) \Leftrightarrow (\exists e_{in} : m(P_t, e_{in}) \mapsto (P_t + 1, e_{out}))$
- δ je přechodová funkce: $\delta(q_t, e_{in}, u) \mapsto m$, kde $q_t \in Q, e_{in} \in \Sigma, m \in M$, a u reprezentuje aktuální uživatelské oprávnění a případně další parametry vztahující se k uživatelskému sezení.

3.3 Nedeterminismus

Automat je definován jako nedeterministický, z daného stavu tedy dovoluje pomocí stejné akce přechod do více různých stavů. Ve skutečnosti však dojde jen k jednomu přechodu, kde cílový stav je určen výsledkem operace implementující přechod. Aserční funkce pak kontroluje, zda automat skončil v jednom z očekávaných stavů. Tato vlastnost lze snadno demonstrovat na příkladu entity článku na blogu znázorněném pomocí stavového diagramu na obrázku 3.3.1. V tomto konkrétním případě lze smazaný článek (stav *deleted*) obnovit pomocí akce *undelete*. Tato akce definuje ze stavu *deleted* přechody do stavů *waiting* a *published* – do kterého z nich automat přejde zjistíme až během provádění přechodu, kdy výsledek operace určí cílový stav. Nedeterminismus zde tedy reprezentuje nedostatek informací, které máme při vyvolání přechodu.

Dalším ukázkou nedeterministického chování je možnost selhání při vykonávání akce, například vlivem chyby při vykonávání změn v databázi. V takovém případě automat zůstane ve stejném stavu.



Obrázek 3.3.1: Stavový diagram článku na blogu (zdroj: [4])

3.4 Vlastnosti automatu z pohledu editoru

Formální definice automatu ze sekce 3.2 je poměrně obsáhlá. Pro editor je podstatná pouze malá část, kterou lze zjednodušit na tyto tři množiny:

1. Q – množina stavů – každý stav má unikátní jméno, pozici a barvu.
2. M – množina akcí – představujících funkci, která se provede při přechodu do cílového stavu. Jedná se tedy o skupinu hran.
3. T – množina přechodů náležících k dané akci – tedy jednotlivé hrany mezi stavy.

Tomu také odpovídá schéma, ve kterém jsou stavové automaty ukládány. Editor umožní práci s každou z těchto tří množin. Každý prvek těchto množin může mít vedle svých

základních vlastností i volitelné dynamické parametry. Žádná data nebudou ignorována, editor umožní úpravu všech parametrů, i těch pro něj neznámých.

Tato zjednodušená definice v podstatě odpovídá stavovému automatu typu Mealy (nebo Moore) [11, 12]. Základní rozdíl oproti klasickému automatu je v množině akcí, která představuje společné vlastnosti podmnožiny přechodů. Každý přechod může kteroukoliv z poděděných vlastností akce přepsat. Vlastnosti akce tedy slouží jako výchozí hodnoty vlastností každého jejího přechodu. Další odchylkou je absence vstupních a výstupních symbolů, které zde nahrazuje název akce – ten představuje metodu, která implementuje přechod daného stavu.

Editor by tedy bylo možné využít i pro modelování klasického stavového automatu, respektive jakéhokoliv (orientovaného i neorientovaného) grafu. Potřebné úpravy jsou však nad rámec této práce (více o nich najdete v sekci 6.5.3).

Kapitola 4

State of the Art

4.1 Framework postavený na kaskádě

Kaskáda popsaná v kapitole 2.1 je pouze teoretický koncept, který nám dává nástroj, jak popsat běh aplikace pomocí znovupoužitelných bloků. O implementaci kaskády se stará framework postavený na kaskádě[1] [1]. Jedná se o ucelený nástroj pro tvorbu webových aplikací, kde kaskáda generuje výsledné stránky.

Framework lze snadno rozšířit pomocí pluginů. Oba editory budou implementovány tímto způsobem.

4.1.1 Editor kaskády

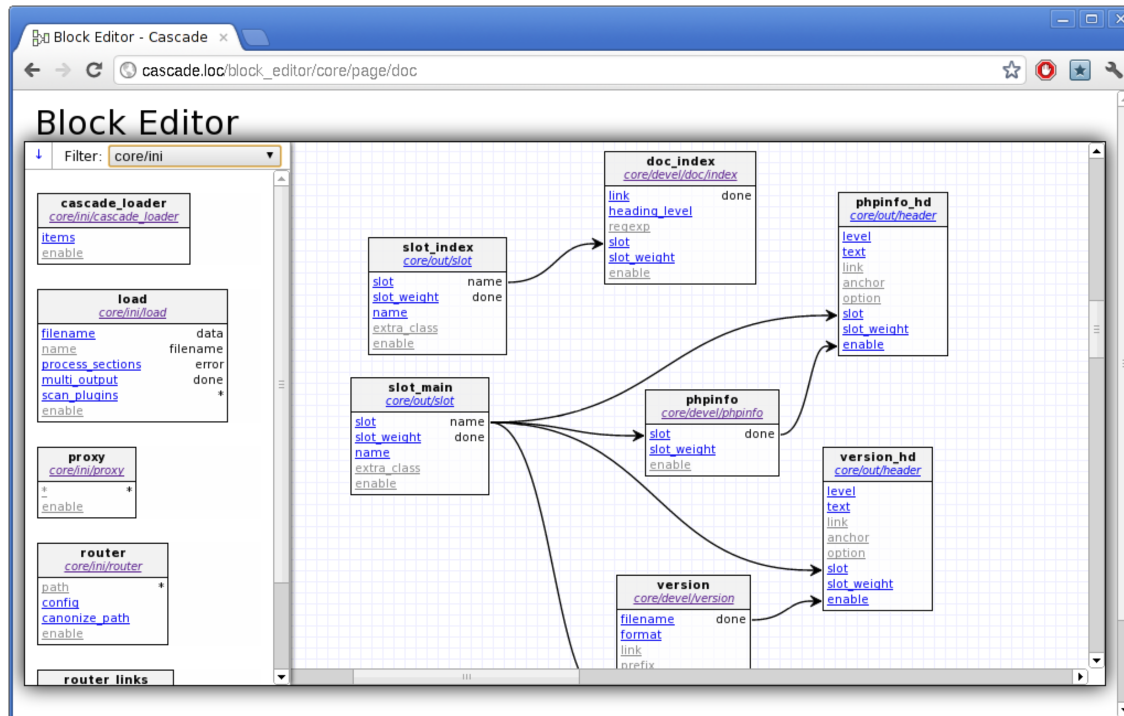
V současné verzi frameworku se již editor kaskády nachází (viz kapitola 7.3 *WYSIWYG editor bloků* v [1]). Jedná se pouze o základní implementaci (*proof of concept*), která podporuje přidávání nových instancí bloků do fragmentu kaskády pomocí metody drag'n'drop a správu jejich vlastností včetně spojení na jiné bloky. Spojení mezi bloky se dají vytvořit pouze zadáním jména bloku a jeho výstupní proměnné. Editor neumí pokročilejší operace s bloky jako je hromadný přesun po plátně, práce se schránkou nebo s historií. K vykreslování spojnic editor využívá knihovny Raphaël[2]. Ukázka editace jednoho složitějšího fragmentu kaskády je vidět na obrázku 4.1.1.

4.1.2 Smalldb

Další součástí frameworku je implementace Smalldb stavových automatů. Pro stavové automaty v současné době žádný grafický editor neexistuje, lze však částečně využít externího editoru *yEd* (viz 4.3.1), pomocí kterého lze přidávat nové stavy, upravovat jejich popisek a barvu, a vytvářet nová spojení. Popisek spojení reprezentuje název akce, ke které daný přechod náleží. *yEd* ale nezná koncept akcí jako podmnožin přechodů, a tak nezbývá než některé změny udělat ručně v textovém editoru. Práce s *yEd* je tedy velké usnadnění, nicméně se stále jedná o značně nepohodlné řešení.

[1] Framework postavený na kaskádě: http://cascade.frozen-doe.net/

[2] Raphaël: http://raphaeljs.com/



Obrázek 4.1.1: Stávající podoba editoru kaskády (zdroj: [1])

4.2 Nástroje pro vizualizaci grafů

4.2.1 Graphviz

Graphviz je balík nástrojů pro vizualizaci grafů. Pracuje se soubory v jazyce *DOT*[3] (přípony `*.dot` a `*.gv`). Obsahuje několik nástrojů pro vykreslování grafů na základě jejich typu – nejběžnější je program `dot` pro orientované grafy. Příklad grafu vykresleného pomocí programu `dot` je vidět na obrázku 3.3.1.

Framework postavený na kaskádě využívá *Graphviz* pro zobrazování ladících informací při zpracování kaskády. Jednotlivé bloky do něj generuje jako HTML tabulky, které *Graphviz* také dokáže vykreslit (viz například obrázek kapitole 2.2). Tuto roli by měl v budoucnosti nahradit editor kaskády, který bude podporovat zobrazení fragmentu v režimu statického obrázku. Díky tomu bude možné odstranit nutnost spouštění *Graphvizu* na serveru.

4.2.2 Canviz

Canviz[4] je JavaScriptová knihovna pro vizualizování grafů ve formátu, který využívá program `dot` (viz 4.2.1). *Canviz* vykresluje do elementu `<canvas>`, využívá tedy podobnou technologii jako naše editory. Nestará se však přímo o kreslení grafu (včetně tedy o výpočet pozic uzlů), pouze interpretuje (vykresluje) data uložená ve formátu *DOT*. Nezvládne tedy pracovat s čistou definicí grafu a potřebuje pomocí *Graphvizu* dopočítat jednotlivé pozice a cesty mezi uzly.

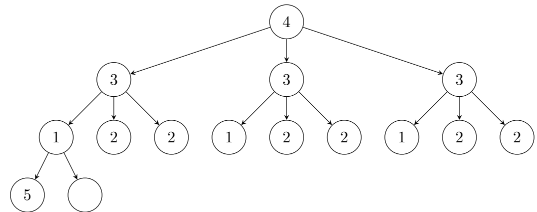
[3] Jazyk DOT: http://www.graphviz.org/doc/info/lang.html
[4] Canviz: https://code.google.com/p/canviz/

4.2.3 Ti*k*Z/PGF

PGF[5] [13] je nízkoúrovňový jazyk pro tvorbu vektorové grafiky na základě geometrického, respektive algebraického, popisu. Ti*k*Z[6] je soubor maker, definovaných nad tímto jazykem. Na obrázku 4.2.1 je vidět krátká ukázka kódu pro zobrazení jednoduchého kořenového stromu a jeho výstup. Ti*k*Z/PGF lze využít například v TEXu/LATEXu, pracovat s ním umí také *Inkscape*[7] nebo *MATLAB*[8].

Do formátu Ti*k*Z by se snadno dal vyexportovat diagram přímo z našich editorů, a následně vložit například do vědecké práce nebo jiného dokumentu psaného v TEXu, případně přímo pro generování PDF. Jedná se ale pouze o nízkoúrovňový jazyk pro kreslení, neřeší logiku toho, co kreslí (v našem případě stavových automatů, respektive orientovaných grafů).

```
\begin{tikzpicture}[>=stealth, every node
    /.style={circle, draw, minimum size
    =0.75cm}]
    \graph [tree layout, grow=down, fresh
        nodes, level distance=0.5in,
        sibling distance=0.5in] {
        4 -> {
            3 -> { 1 -> { 5, " " }, 2,2 },
            3 -> { 1, 2, 2 },
            3 -> { 1, 2, 2 }
        }
    };
\end{tikzpicture}
```



Obrázek 4.2.1: Ukázkový kód a jeho výstup – knihovna Ti*k*Z/PGF

4.2.4 PSTrics

PSTricks [14] je alternativou pro Ti*k*Z/PGF. Také se jedná o nízkoúrovňový jazyk pro tvorbu vektorové grafiky, určený pro vkládání PostScriptu[9] do TEXu/LATEXu. Opět jsou zde dostupné rozšíření například pro kreslení grafů funkcí. Na obrázku 4.2.2 je vidět ukázka kódu vizualizující orientovaný graf.

Stejně jako Ti*k*Z (viz 4.2.3), ani *PSTricks* se nehodí pro implementaci editoru, ale mohl by sloužit jako vhodný jazyk pro export.

[5]PGF: Portable Graphics Format

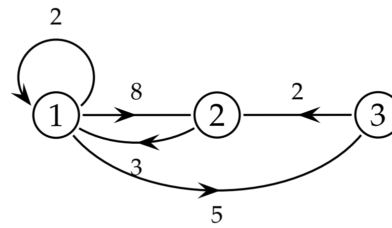
[6]Ti*k*Z: Ti*k*Z ist *kein* Zeichenprogramm

[7]Inkscape: https://inkscape.org/cs/

[8]MATLAB: https://www.mathworks.com/products/matlab/

[9]PostScript je programovací jazyk určený ke grafickému popisu tisknutelných dokumentů vyvinutý firmou Adobe Systems (http://www.adobe.com/products/postscript/)

```
\psmatrix[colsep=1.5cm,rowsep=1.5cm,mnode=
    circle]
    &3\\
    1&&2
    \ncline{2,1}{2,3}
    \ncarc[arcangle=-30]{2,3}{1,2}
    \ncarc[arcangle=-30]{1,2}{2,1}  \
        nccircle[nodesep=4pt]{->}{2,3}{.6
        cm}
\endpsmatrix
```



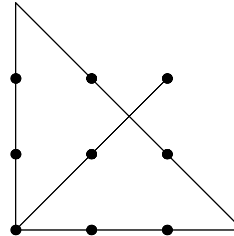
Obrázek 4.2.2: Ukázkový kód a jeho výstup – knihovna PSTrics

4.2.5 MetaPost

MetaPost [15] je dalším programovacím jazykem pro kreslení vektorové grafiky na základě geometrického nebo algebraického popisu. Vychází z knihovny *Metafont*, která slouží pro generování vektorových fontů v TEXu. Součástí *MetaPostu* je i jeho stejnojmenný interpretr, pomocí kterého lze kód přeložit do *PostScriptu* nebo SVG. Jednoduchá ukázka kódu a jeho výstup je na obrázku 4.2.3.

MetaPost je další alternativou k TikZ a *PSTricks* (4.2.3 a 4.2.4) použitelnou pouze pro export, ale ne pro samotnou implementaci editoru.

```
beginfig(2);
    u=1cm;
    draw (2u,2u)--(0,0)--(0,3u)--(3u,0)
        --(0,0);
    pickup pencircle scaled 4pt;
    for i=0 upto 2:
        for j=0 upto 2:
            drawdot (i*u,j*u);
        endfor
    endfor
endfig;
```



Obrázek 4.2.3: Ukázkový kód a jeho výstup – knihovna MetaPost

4.2.6 JointJS/Rappid

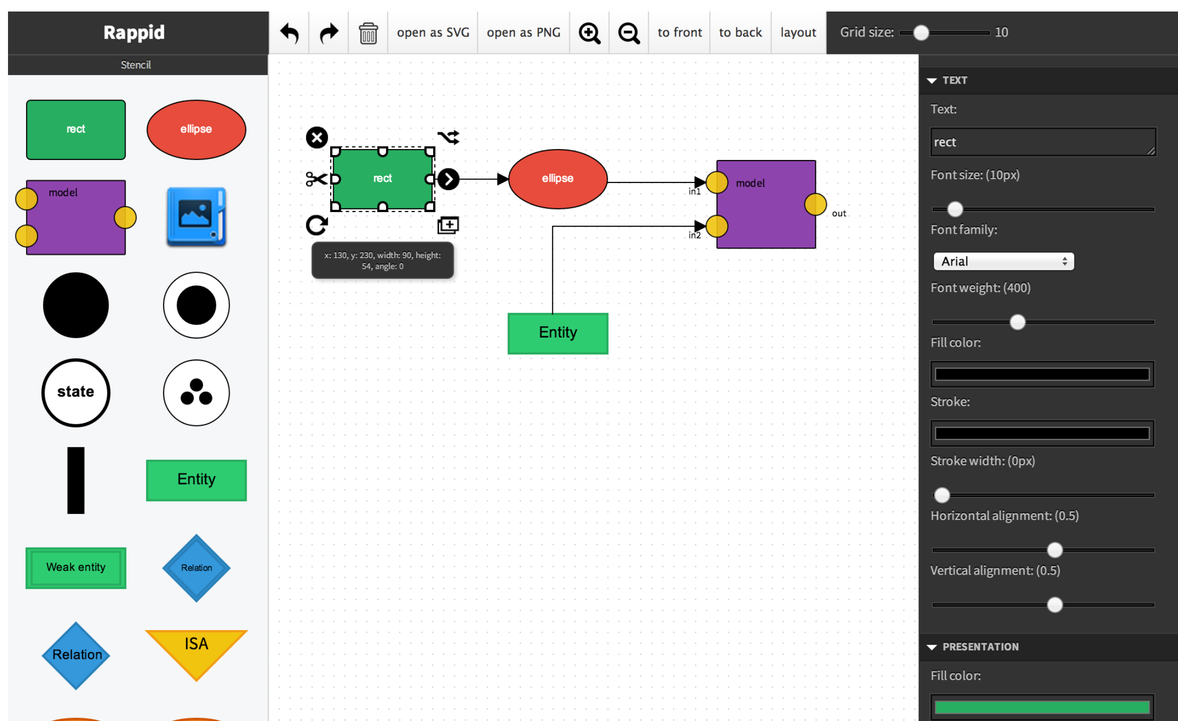
JointJS [16] je moderní JavaScriptová knihovna pro tvorbu diagramů využívající HTML5. Vedle vizualizace do statických obrázků má podporu i pro interakci s diagramy. Mezi jeho vlastnosti patří:

- Podporuje práci se standardními elementy UML diagramů i vlastními elementy v SVG.
- Vyhlazování spojnic pomocí Bézierovi interpolace.
- Hierarchické diagramy.
- Serializaci do formátu JSON.
- *Event-driven* architektura – možno pověsit vlastní *callback* na libovolnou událost.

- Podpora ovládání pomocí dotyku (na mobilních zařízeních).
- Rozšiřitelnost pomocí pluginů.

Knihovna *JointJS* neřeší logiku automatů (respektive grafů), stará se pouze o jejich kreslení. Pro naše editory se její využití nejeví jako dostatečně profitabilní, vzhledem k tomu, jak jednoduše lze dnes pomocí HTML5 pracovat s grafikou přímo prostřednictvím JavaScriptu.

Rappid je JavaScriptová aplikace postavená nad knihovnou *JointJS* (viz obrázek 4.2.4). Je zaměřená na obecné a UML diagramy a jejich kreslení, také neřeší jejich význam a logiku. Na rozdíl od samotné knihovny jeho licence neumožňuje další rozvoj a vlastní modifikace. Proto se nehodí pro integraci do dalších *open source* aplikací.



Obrázek 4.2.4: Rappid (zdroj: [16])

4.2.7 Dagre

Dagre [17] je JavaScriptová knihovna pro výpočet rozložení orientovaného grafu. Jedná se o implementaci, která vychází z algoritmu, který používá program dot (viz sekce 4.2.1 – *Graphviz*). Na obrázku 4.2.5 je vidět ukázka rozložení TCP stavového diagramu, které vypočítal *Dagre*.

Klíčové vlastnosti *Dagre*:

- Pouze klientský kód v JavaScriptu, žádné serverové závislosti.
- Vysoká rychlost vykreslování, možnost nastavit limit pro dobu výpočtu.

- Není závislý na způsobu vykreslování (stará se pouze o rozložení uzlů grafu a výpočet kontrolních bodů cest mezi nimi).

Podrobnějším rozborem algoritmu, který *Dagre* využívá pro výpočet pozic uzlů a cest mezi nimi, se zabývá sekce 6.4.2.

4.3 Obecné editory grafů

4.3.1 yEd

yEd [18] je obecný nástroj pro tvorbu diagramů a grafů. Je napsán v jazyce Java a je dostupný pro všechny tři hlavní platformy (Windows, Linux, OS X a další které podporují JVM[10]). Pomocí *yEd*u lze tvořit mnoho různých druhů diagramů – vývojové a síťové diagramy, UML[11] a BPMN[12] diagramy, myšlenkové mapy nebo ER[13] modely.

Vedle nástrojů pro kreslení grafů má *yEd* také podporu pro automatické rozmístění uzlů grafu (respektive elementů diagramu), a to pomocí pestrého výběru algoritmů, mimo jiné například:

- *Force-based*, tedy na základě fyzikálního modelu vzájemně působících sil.
- Hierarchický pro vývojové diagramy.
- Ortogonální pro UML diagramy tříd.
- Stromový, Kruhový, Náhodný.

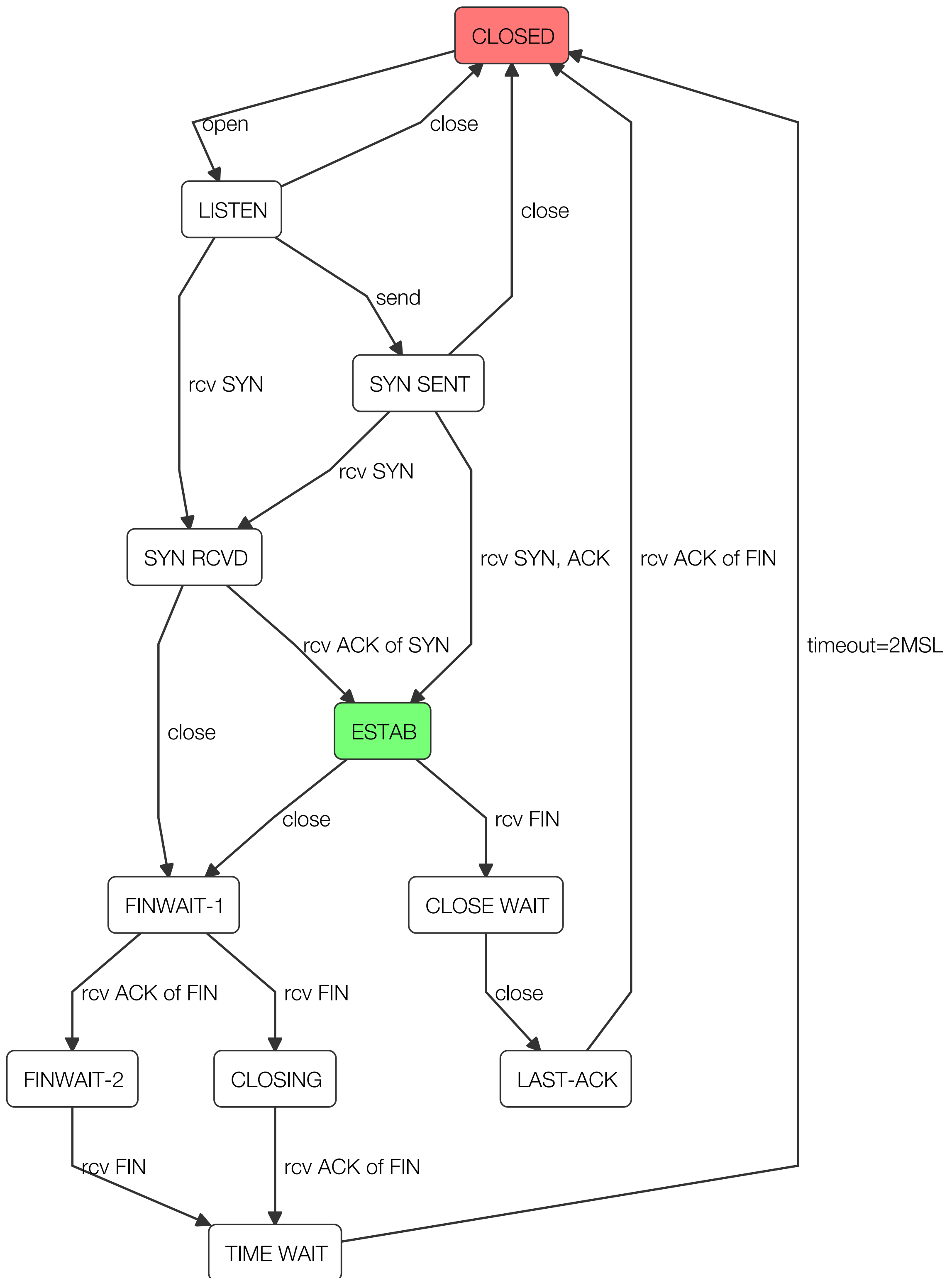
Data ukládá v *GraphML*[14], což je obsáhlý strukturovaný formát souboru založený na XML. Vedle popisu samotného grafu podporuje i vkládání informací specifických pro konkrétní graf (respektive aplikaci, kterou graf reprezentuje). Soubor v tomto formátu lze v současné době do Smalldb stavového automatu vložit pomocí parametru *includes* (viz sekce 6.1.3), a část modelování automatu provést právě pomocí *yEd*.

yEd podporuje import dat ve formátu *MS Excel* a dále pomocí XSLT transformace i jakákoliv data ve formátu XML. Exportovat pak umí rastrovou (formáty BMP, GIF, JPEG, PNG) i vektorovou (PDF, SVG) grafiku. Pro využití na webu se hodí možnost vygenerovat klikací obrázkovou mapu.

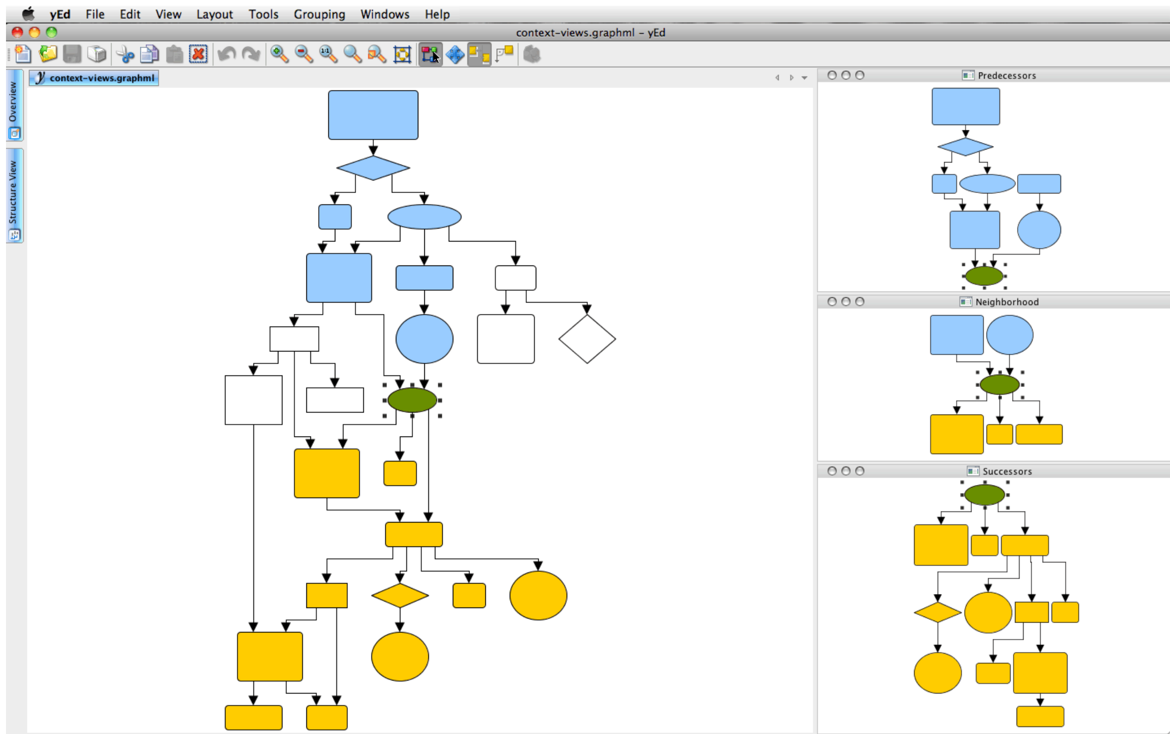
4.3.2 Visual Graph Editor

Visual Graph Editor [19] je jednoduchý editor grafů. Vedle nástrojů pro tvorbu grafu a jeho úpravu umí i analyzovat různé problémy spojené s grafovou teorií. Podporuje

[10]JVM: Java Virtual Machine
[11]UML: Unified Modeling Language
[12]BPMN: Business Process Model and Notation
[13]ERM: Entity-relationship model
[14]GraphML: http://graphml.graphdrawing.org/



Obrázek 4.2.5: Ukázka výstupu Dagre (zdroj: [17])



Obrázek 4.3.1: yEd (zdroj: [18])

úpravu grafu ze tří hlavních pohledů – čelní, boční a horní. Výsledek pak umí vizualizovat ve trojrozměrném módu. Je postaven na knihovně *Qt*[15], podporuje opět všechny tři hlavní platformy (Windows, Linux, OS X). Data uchovává v souborech vlastního formátu postaveném nad XML.

Visual Graph Editor je v podstatě jedinou desktopovou konkurencí k *yEd*, za kterým ale hodně zaostává. Pro naši potřebu dále není vhodný kvůli absenci podpory práce se stavovými automaty.

4.4 Editory konečných automatů

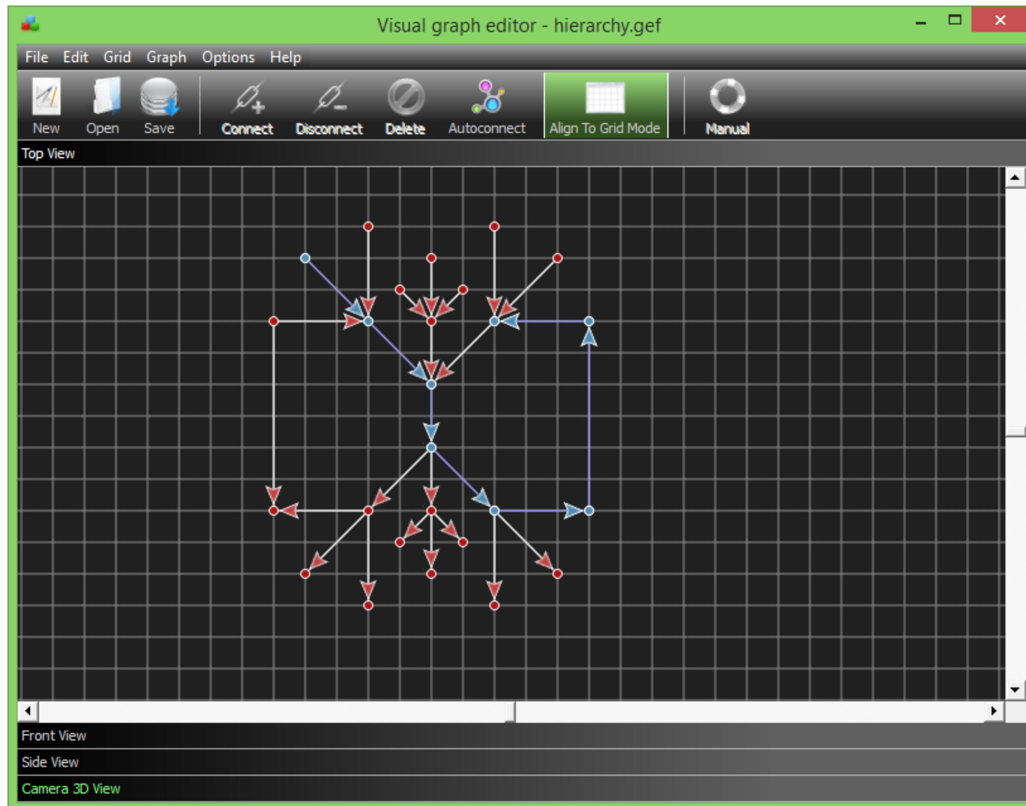
4.4.1 UPPAAL

UPPAAL [20] je integrované prostředí pro modelování, validaci a verifikaci systémů reálného času pomocí sítě časových[16] automatů. UPPAAL je zkratka sestávající z názvů dvou univerzit – *Upp*sala University ve Švédsku a *Aal*borg University v Dánsku – které ho společně vyvíjí. Podporuje všechny tři hlavní platformy – Windows, Linux a OS X.

Je vhodný pro modelování systémů, které se dají rozdělit na vzájemně spolupracující nedeterministické procesy. Pro komunikace mezi jednotlivými automaty v rámci systému slouží komunikační kanály a sdílené proměnné. Pomocí nástroje pro verifikaci modelu lze pak nadefinovat tvrzení o běhu systému a následně je ihned ověřit.

[15] Qt: http://www.qt.io/

[16] Časový automat je stavový automat rozšířený o hodiny



Obrázek 4.3.2: Visual Graph Editor (zdroj: [19])

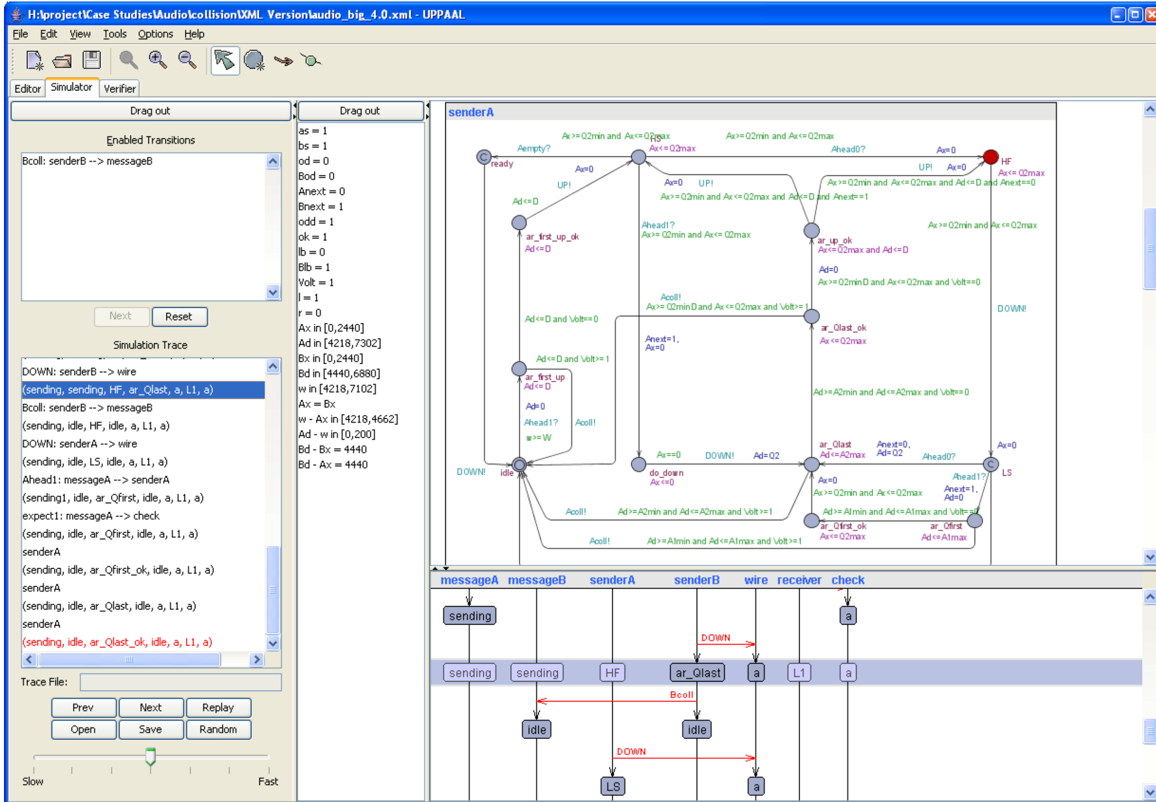
Formát, který *UPPAAL* využívá pro ukládání dat, není snadno importovatelný do Smalldb. Neobsahuje dostatečné množství metadat, které jsou pro Smalldb důležité. Jeho zaměření je hlavně na dokazování teoretických vlastností a tvrzení o modelovaných entitách, pro samotné modelování už tak vhodný není. Naopak export ze Smalldb automatu do formátu *UPPAAL* smysl má. *UPPAAL* by šlo využít pro modelování komplexního systému, jehož součástí by byly přímo definice entit ze Smalldb, a následné verifikaci tvrzení o tomto systému.

4.4.2 Yakindu SCT

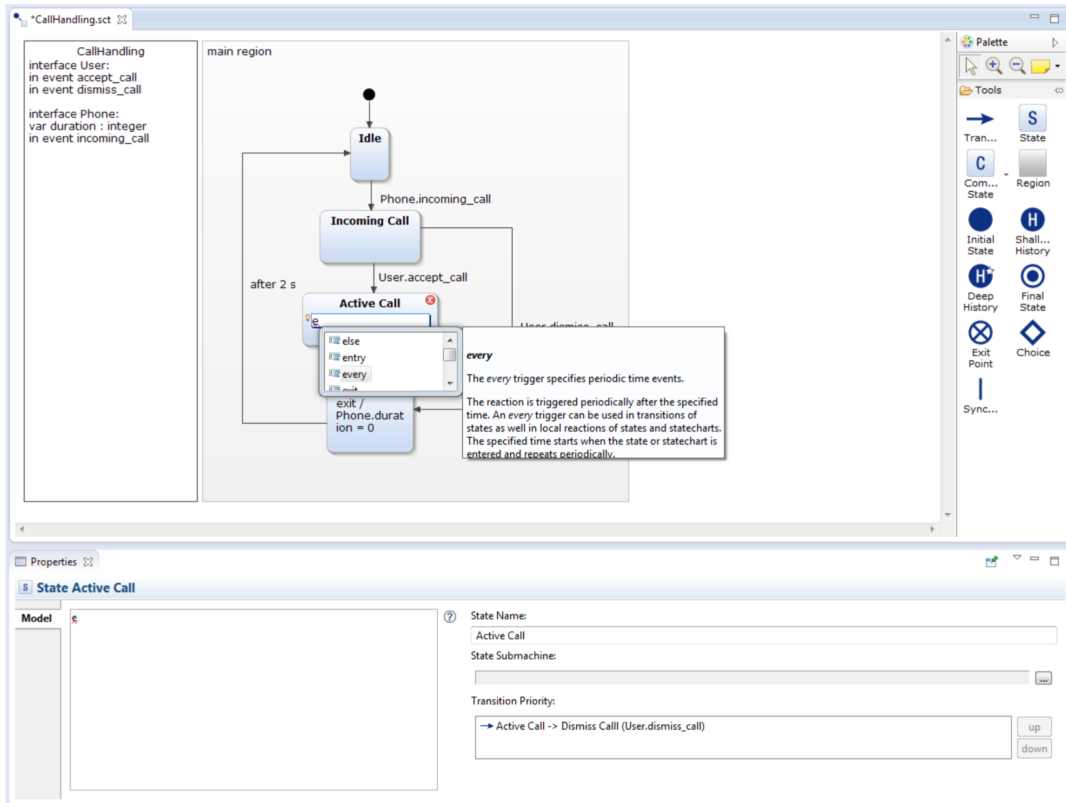
Yakindu SCT (*State Chart Tools*) je nástroj pro modelování stavových automatů. Vedle grafického editoru automatů obsahuje (stejně jako *UPPAAL*) nástroje pro syntaktickou i sémantickou validaci automatu a simulaci jeho běhu. Podporuje také generování zdrojového kódu reprezentující automat do jazyků Java, C a C++. To je ale jeho jediná výhoda oproti programu *UPPAAL* (viz 4.4.1), která pro Smalldb automaty není relevantní – Smalldb totiž interpretuje přímo graf reprezentující automat.

Yakindu SCT je implementován jako rozšíření editoru *Eclipse*[17], podporuje také všechny tři hlavní platformy – Windows, Linux a OS X.

[17] Eclipse: http://www.eclipse.org/



Obrázek 4.4.1: UPPAAL (zdroj: [20])



Obrázek 4.4.2: Yakindu SCT (zdroj: [21])

Kapitola 5

Grafický editor kaskády

5.1 Analýza

5.1.1 Funkční požadavky

Tato část popisuje funkční požadavky kladené na editor kaskády.

Kompatibilita s předchozí verzí editoru

Nová verze editoru kaskády musí být plně kompatibilní s původním editorem popsaným v části 4.1.1. Všechny stávající uživatelské funkce musí být zachovány.

Načtení aktuální palety bez obnovy prohlížeče

Při úpravě kaskády může dojít k situaci, kdy je potřeba přidat blok, který ve chvíli načtení editoru ještě nebyl implementován. Proto musí existovat možnost aktualizace palety bez nutnosti obnovit okno prohlížeče (a tím ztratit neuložená data).

Historie

Editor bude udržovat lokální historii, ve které bude možné procházet pomocí tlačítek *zpět* a *opakovat*.

Označení skupiny bloků

Každý blok půjde označit prostým kliknutím. Pomocí tahu myši bude možné označit skupinu bloků. Tahem zleva doprava se budou označovat bloky, které leží celé ve vybrané oblasti. Tahem zprava doleva se budou označovat i bloky, které do označené oblasti spadají jen výřezem. Podobnou možnost mají například uživatelé programu *AutoCAD* a podobných.

Tvorba spojení pomocí drag'n'drop

Pomocí tahu myši se stisknutou klávesou CTRL bude možné vytvořit nové spojení mezi bloky. Spojení bude možné začít jak od výstupu, tak od vstupu bloku.

Vyhýbání se blokům

Spojení mezi bloky musí při vykreslování kontrolovat kolize s ostatními bloky a pokud to je možné, vyhnout se jim.

Schránka

Označené bloky bude možné zkopírovat nebo vyjmout do schránky a poté je z ní vložit zpět na kreslící plátno.

Přiblížení/oddálení

Pro snazší orientaci v kaskádě bude editor podporovat přiblížení/oddálení kreslícího plátna. Dále bude přítomné tlačítko pro vynulování přiblížení (nastavení na 100%).

5.1.2 Nefunkční požadavky

Následují část popisuje nefunkční požadavky, jinak také požadavky na kvalitu, kladené na editor kaskády.

Závislost na dalších knihovnách

Kromě frameworku jQuery nesmí být editor závislí na žádné další knihovně třetí strany. Povoleny jsou pouze volitelné závislosti, bez kterých lze editor plnohodnotně používat.

Rychlost

Rychlost odezvy při práci s editorem (zejména při tvorbě nových spojení pomocí drag'n'drop) musí editor reagovat dostatečně rychle. Dle knihy *Usability Engineering* [2] by doba reakce neměla překročit zhruba 0.1 vteřiny. Z pohledu vykreslování to znamená rychlost minimálně 10 snímků za sekundu.

Podpora prohlížečů

Editor musí fungovat v prohlížečích Google Chrome, Mozilla Firefox a Internet Explorer. Minimální podporovaná verze jednotlivých prohlížečů je specifikovaná v tabulce 1.1.

Jazyk

Všechny řetězce, které se budou vypisovat uživateli, budou psány v anglickém jazyce. Tyto řetězce musí být označený pro překlad pomocí lokalizační značkovací funkce `_(string, params)`. Tato funkce bude prozatím implementována tak, aby pouze vracela svůj první parametr (řetězec k překladu) a dále umožní využití parametrizovaných překladů (náhrada nepřeložené hodnoty za znaky %s).

Verzování

Vývoj obou editorů bude verzován pomocí systému GIT[1], každý editor bude mít svůj vlastní repozitář, umístěný na serveru `https://git.frozen-doe.net`.

Kreslení spojnic

Spojnice mezi bloky musí být vykresleny co nejpřehledněji.

Klávesové zkratky

Všechny akce v panelu nástrojů musí jít vyvolat i pomocí klávesové zkratky.

5.1.3 Formát a schéma kaskády

JSON schéma

JSON schéma[2] je nástroj pro popis formátu dat uložených v souboru typu JSON. Samotné schéma je také platný kód ve formátu JSON, je tedy velmi snadno strojově čitelné. Schéma definuje kontrakt mezi aplikací (editorem) a soubory, se kterými pracuje; říká nám, jaké vlastnosti editor zná a umí zpracovat, a jakých typů musí být hodnoty těchto vlastností.

Následuje schéma fragmentu kaskády:

```
1  {
2      "title": "Schéma fragmentu kaskády",
3      "type": "object",
4      "properties": {
5          "blocks": {
6              "type": "object",
7              "properties": {
8                  "block": { "type": "string" },
9                  "force_exec": { "type": "boolean" },
10                 "in_val": {
11                     "type": "object"
12                     "items": { "type": "object" },
13                 },
14                 "in_con": {
15                     "type": "object",
16                     "items": { "type": "array" },
17                 },
18                 "x": { "type": "integer" },
19                 "y": { "type": "integer" }
20             },
21             "required": ["block"],
22             "additionalProperties": true
23         }
24     },
25     "required": ["blocks"],
26     "additionalProperties": true
27 }
```

Fragment kaskády je tedy objekt, který musí mít pod klíčem *blocks* uložený mapu bloků (jednotlivé klíče odpovídají identifikátoru bloku). Blok samotný je také objekt, má jednu povinnou vlastnost – *block* – tedy typ bloku (třída, jejíž je instancí). Hodnoty vstupních proměnných jsou uloženy v objektu *in_val*, spojení pak v objektu *in_con*.

[1] GIT: http://git-scm.com/

[2] JSON schéma: http://json-schema.org/

Pomocí vlastnosti *force_exec* lze vynutit provedení bloku (pokud není vynucené, proběhne pouze pokud je výstup bloku někde vyžádán).

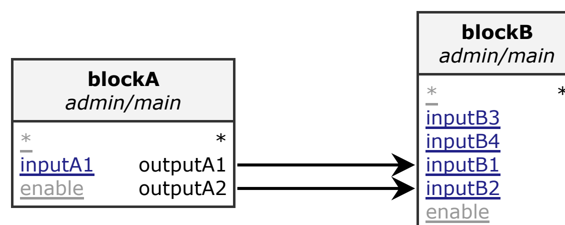
Ukázka fragmentu kaskády

Následující ukázka kódu popisuje fragment kaskády, sestávající ze dvou bloků (*blockA* a *blockB*), oba typu *admin/main*[3]. Mezi bloky jsou definována dvě spojení (řádek 11 – klíč *in_con*), obě na straně bloku *blockB* – směr je tedy do bloku *blockB* (vstupní proměnné *inputB1* a *inputB2*). Oba bloky mají dále definovány vstupní hodnoty *(inputA1, inputB3* a *inputB4* – klíč *in_val*).

Z pohledu kaskády jsou oba typy vstupů (vstupní hodnota *in_val* a vstupní spojení *in_con*) rovnocenné, rozdíl je pouze v zapojení. Hodnota *in_var* slouží jako konstanta, *in_con* lze pak chápat jako proměnnou.

```
1  {
2      "blocks": {
3          "blockA": {
4              "block": "admin/main",
5              "in_val": {
6                  "inputA1": "Value of inputA1"
7              }
8          },
9          "blockB": {
10             "block": "admin/main",
11             "in_con": {
12                 "inputB1": [
13                     "blockA", "outputA1"
14                 ],
15                 "inputB2": [
16                     "blockA", "outputA2"
17                 ]
18             },
19             "in_val": {
20                 "inputB3": "Value of inputB3",
21                 "inputB4": "Value of inputB4"
22             }
23         }
24     }
25 }
```

Na obrázku 5.1.1 je vidět vizualizace tohoto fragmentu pomocí editoru kaskády.



Obrázek 5.1.1: Vizualizace fragmentu z předchozí ukázky kódu

5.1.4 Uživatelské role

Z pohledu editoru existuje pouze jedna uživatelské role – **programátor**, který vytváří (nebo spravuje) webovou aplikaci pomocí frameworku postaveném na kaskádě. Není

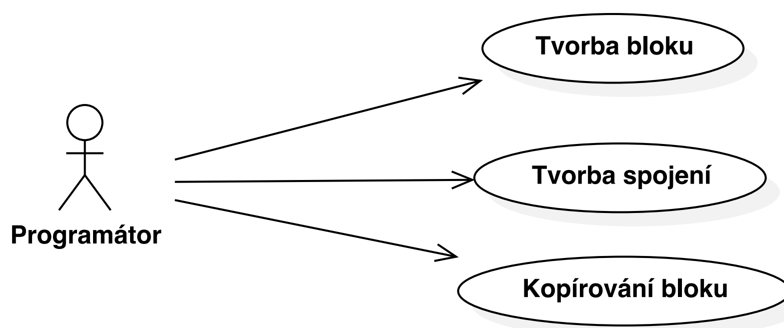
[3]Blok typu *admin/main* může mít libovolné vstupy i výstupy, viz 2.2

proto potřeba žádné ACL[4] vrstvy, všechny funkce editoru jsou vždy přístupné. Jedinou konfigurací z tohoto pohledu je příznak `block_storage_write_allowed` v konfiguraci frameworku, který určuje, jestli lze do úložiště bloků zapisovat, a tedy jestli může editor ukládat změny. Tento příznak je v produkčním režimu zakázán.

Programátor využívá editor k úpravě kaskády, kterou by jinak musel dělat ručně (pomocí textového editoru).

5.1.5 Případy použití

Zde jsou popsány tři typické případy použití editoru kaskády, které pokrývají základní práci s editorem.



Obrázek 5.1.2: UML Use Case diagram editoru kaskády

Tvorba bloku

1. Programátor otevře administraci webu s editorem kaskády, ve kterém je načtený příslušný fragment.
2. Přesunutím myši nad panel nástrojů zobrazí paletu bloků.
3. V paletě pomocí filtru vybere skupinu bloků *page*.
4. Do plátna přetáhne nový blok typu *page/hello*.
5. Ve vyskakovacím okně vyplní název nové instance bloku.
   (a) Pokud již existuje blok se stejným jménem, bude vyzván k zadání nového unikátního.

Tvorba spojení

1. Programátor otevře administraci webu s editorem kaskády, ve kterém je načtený příslušný fragment.

[4] ACL: Access control list

2. Se stisknutou klávesou CTRL stiskne tlačítko myši nad výstupem zdrojového bloku.

3. Se stisknutým tlačítkem myši přesune kurzor nad vstup cílového bloku a uvolní ho.

4. Pokud je vstup nebo výstup dynamický (označeno hvězdičkou), programátor do vyskakovacího okna vyplní názvy nově vytvořeného vstupu či výstupu.

5. Pokud do cílového vstupu už nějaké spojení vede, programátor do vyskakovacího okna vyplní vstupní funkci (*and, or, xor, ...*).

Kopírování bloku

1. Programátor otevře administraci s editorem kaskády, ve kterém je načtený příslušný fragment.

2. Se stisknutým tlačítkem CTRL stiskne tlačítko myši nad kreslící plochou (mimo bloky).

3. Tahem označí vybrané bloky.

   (a) Tažením zleva doprava označí bloky, které spadají celé do provedeného výřezu.

   (b) Tažením zprava doleva označí i jen z části překrývající bloky.

4. Pomocí tlačítka v panelu nástrojů nebo klávesové zkratky *CTRL + C* zkopíruje příslušné bloky do schránky.

   (a) Označené bloky lze i vyjmout (*CTRL + X*).

5. Otevře nový fragment kaskády, do kterého chce vložit obsah schránky.

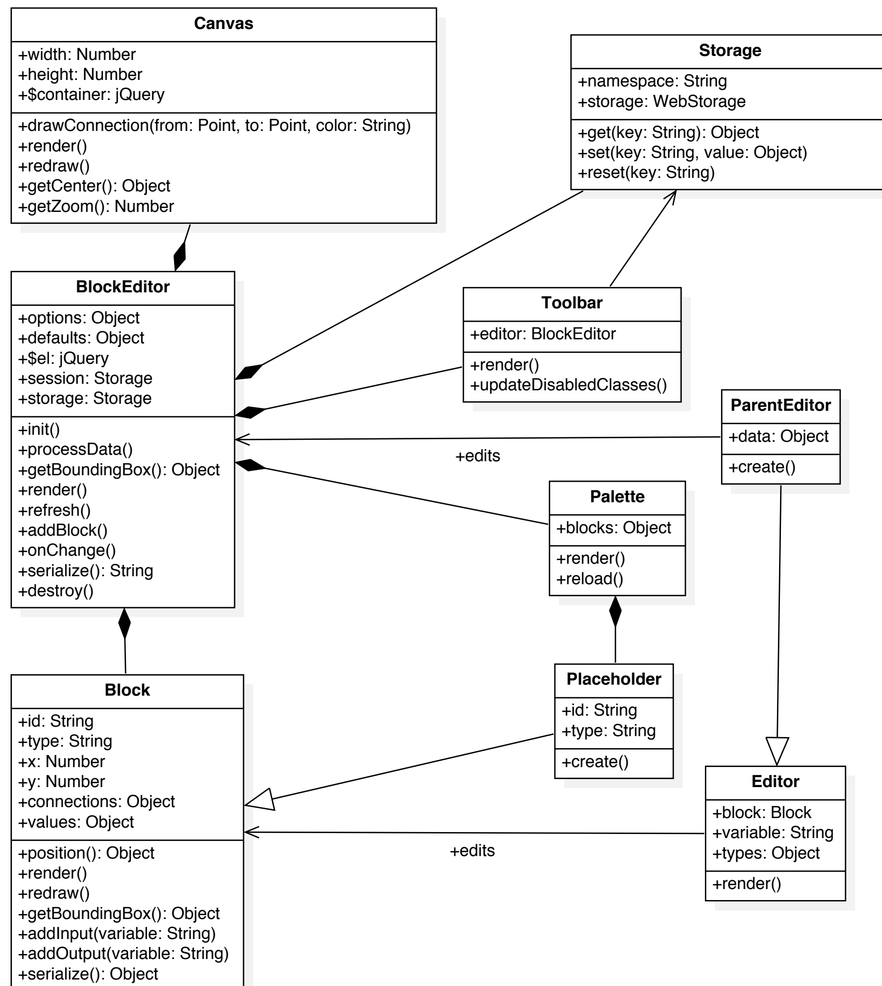
6. Pomocí tlačítka v panelu nástrojů nebo klávesové zkratky *CTRL + V* vloží bloky na střed plátna.

5.2 Návrh

5.2.1 Diagram tříd

Na obrázku 5.2.1 je znázorněn diagram tříd editoru kaskády a jejich vzájemná kompozice. Pro názornost byly vypuštěny nepodstatné atributy. Diagram obsahuje všechny třídy, kromě vykreslovacích primitiv popsaných v sekci 5.2.2. Následuje stručný popis odpovědností jednotlivých tříd.

BlockEditor (*Editor kaskády*) zapouzdřuje celý plugin. Stará se o načítání dat z elementu `<textarea>` a jejich zpětnou aktualizaci. Uchovává mapu aktuálně načtených bloků. Při inicializaci vytvoří ostatní objekty a spustí editor.



Obrázek 5.2.1: Diagram tříd editoru kaskády

Canvas (*Kreslící plátno*) se stará o kreslení spojnic mezi bloky. Na pozadí editoru vloží element `<canvas>`, do kterého kreslí.

Block (*Blok*) uchovává informace o instanci bloku – hodnoty vstupů, spojení a volitelně i jeho souřadnice.

Placeholder (*Zástupný blok*) je třída rozšiřující entitu *Block*, slouží pro vykreslení všech typů bloku v paletě. Po přetažení do kreslícího plátna instance zástupného bloku vytvoří nový blok odpovídajícího typu a sama se zničí.

Palette (*Paleta bloků*) slouží k vytváření nových instancí bloků pomocí metody drag'n'drop. Uchovává v sobě seznam všech typů bloků, které lze do kaskády vložit.

Editor (*Editor vstupů*) slouží pro úpravu vstupnů bloku. Editor vstupu se stará vždy pouze o jednu proměnnou, ne o celý blok. Vedle hodnoty proměnné lze nastavit i její typ. Jednotlivé typy vstupu bloku:

- *Connection* (spojení) – řetězec ve tvaru `id_bloku:název_vstupu`. Při vícenásobném spojení se každé vypisuje na vlastní řádek, dále je pak nutné na první řádku uvést vstupní funkci, která několik příchozích hodnot konvertuje do jedné (např. logické `:and`, `:or`, agregační funkce, konverze polí apod.).
- *Boolean* – pravda/nepravda.
- *Integer* – celé číslo.
- *String* – textový řetězec.
- *JSON* – objekt ve formátu JSON.

ParentEditor (*Editor nadřazeného bloku*) rozšiřuje třídu *Editor* a slouží k úpravě vlastností bloku, který tento fragment vkládá do kaskády. Umožní editaci všech vlastností, i těch neznámých.

Toolbar (*Panel nástrojů*) zapouzdřuje následující funkce:

- Zobrazení na celou obrazovku.
- Vlastnosti nadřazeného bloku – otevře modální okno.
- Obnova palety – načte pomocí AJAX nová data palety.
- Zpět a Vpřed v historii.
- Kopírovat/Vyjmout aktivní blok(y) do schránky.
- Vložit obsah schránky.
- Přiblížit/Oddálit zobrazení.
- Vynulovat přiblížení.

5.2.2 Reprezentace geometrických objektů

Pro výpočet průsečíků a kreslení zakřivené cesty slouží následující pomocné třídy (obrázek 5.2.2):

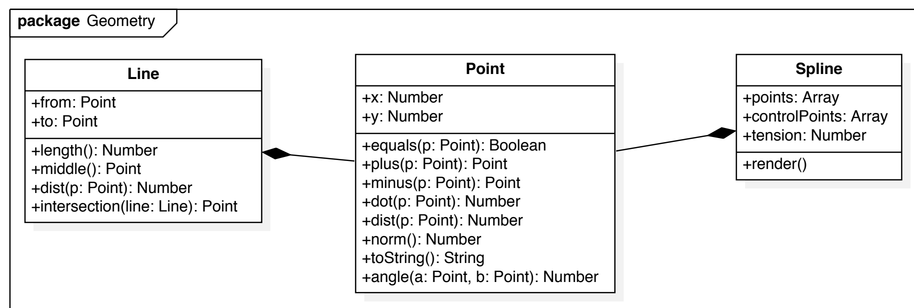
Point reprezentuje bod v eukleidovském prostoru. Třídu lze využít i pro reprezentaci vektoru, implementuje proto dvě důležité metody:

- `norm()` – pro výpočet normy (velikosti) vektoru.
- `dot(Point p)` – pro výpočet skalárního součinu s vektorem p.

Line reprezentuje úsečku ohraničenou dvěma objekty typu *Point*. *Line* implementuje tyto metody:

- `intersection(Line l)` – pro hledání průsečíku s jinou úsečkou.
- `dist(Point p)` – pro výpočet vzdálenosti bodu od úsečky. Tato metoda je důležitá pro schopnost označit kliknutím nakreslenou křivku, která se nejprve proloží několika úsečkami (více v kapitole ...).

Spline reprezentuje křivku procházející několika body v prostoru. Bližší popis způsobu vykreslování křivky najdete v 5.2.4.



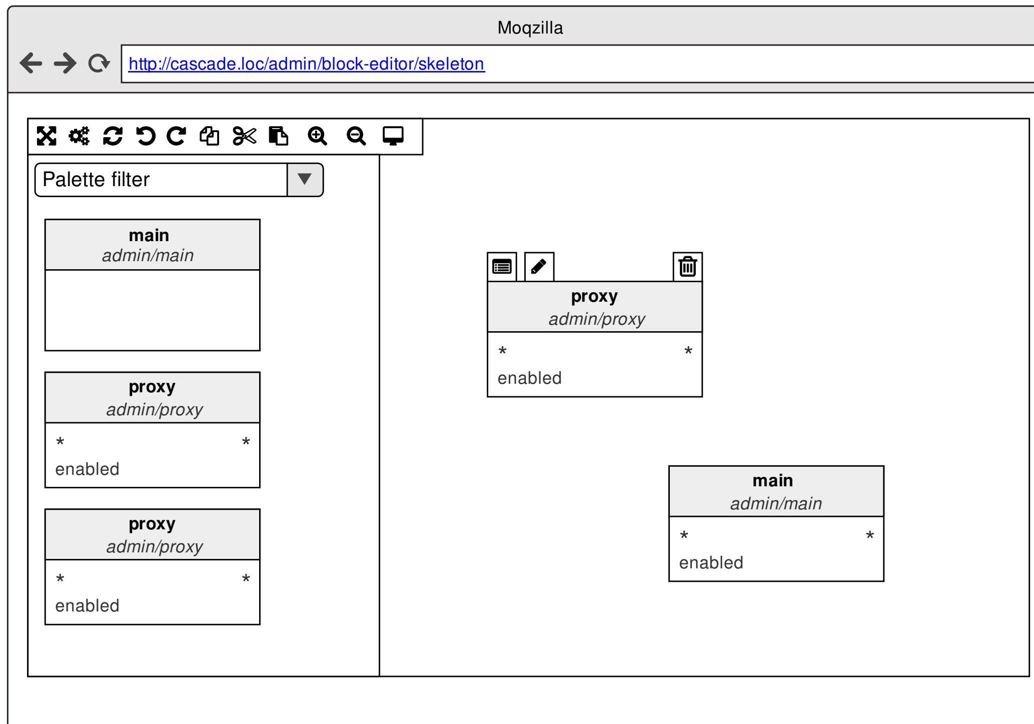
Obrázek 5.2.2: Diagram pomocných geometrických tříd

5.2.3 Uživatelské rozhraní

Uživatelské rozhraní vychází z původní verze editoru kaskády. Jeho návrh vytvořený pomocí online nástroje *Moqups*[5] je vidět na obrázku 5.2.3.

Kreslící plátno Na pozadí editoru je kreslící plátno, na kterém jsou zobrazeny jednotlivé instance bloků ve fragmentu kaskády. Bloky lze po plátnu libovolně pohybovat pomocí metody drag'n'drop, a to i hromadně.

[5]Moqups: https://moqups.com



Obrázek 5.2.3: Mockup editoru kaskády – plátno s paletou a panelem nástrojů

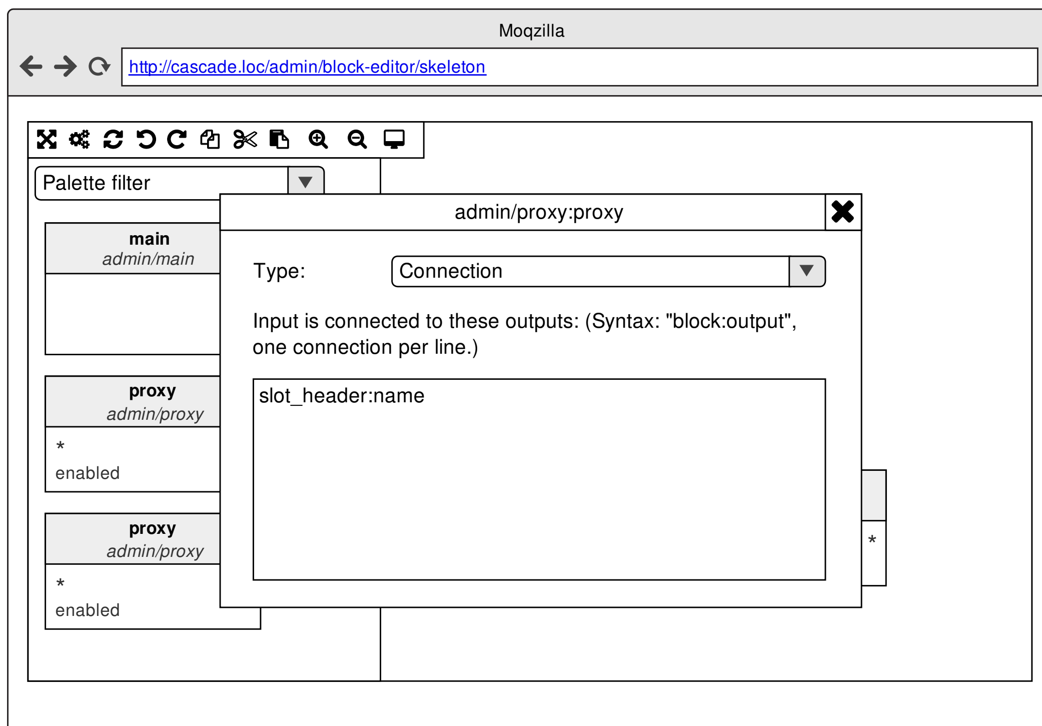
Blok Instance bloku je reprezentována stejně jako v původní verzi editoru kaskády – hlavička je podbarvena šedě, na první řádku je unikátní identifikátor bloku, pod ním pak typ bloku. Při dvojkliku na tyto položky bude vyvolán kontextový dialog pro jejich úpravu. Tělo bloku je rozděleno na dvě části – v levé polovině jsou vypsány vstupní proměnné, v pravé polovině pak ty výstupní. Spojení mezi bloky lze vytvořit jak pomocí editoru proměnných (viz dále), tak pomocí metody drag'n'drop tažením kurzoru se stisknutým tlačítkem CTRL z výstupní proměnné do vstupní proměnné nebo naopak. Po najetí kurzorem nad blok se zobrazí tři tlačítka. Po levé straně tlačítko s odkazem na dokumentaci bloku a tlačítko s odkazem na editaci bloku, které otevře do nového okna další instanci editoru kaskády s fragmentem, který reprezentuje daný blok. Tlačítko po pravé straně slouží pro smazání instance bloku z upravovaného fragmentu.

Panel nástrojů V levém horním rohu plátna je vykreslen panel nástrojů. Jedná se o horizontální lištu s jednotlivými akčními tlačítky. Každé tlačítko má svou klávesovou zkratku – ta je vidět v popisku při podržení kurzoru nad tlačítkem. Při použití klávesové zkratky bude na malou chvíli dané tlačítko podbarveno, aby simulovalo jeho stisk. Při najetí kurzorem nad panel nástrojů se zobrazí paleta bloků.

Paleta bloků se skrývá v levé části plátna. Je vykreslena po celé výšce kreslícího plátna editoru, navazuje na panel nástrojů, pomocí kterého se dá zobrazit. V horní části obsahuje jednoduchý filtrovací realizovaný elementem `<select>`. Ten při změně výběru okamžitě filtruje zástupné bloky, které jsou vykresleny pod ním. Panel s paletou bloků lze vertikálně scrollovat. Při zobrazení editoru na celou stránku je paleta zobrazena se zmenšenou šířkou, po najetí kurzoru se rozbalí do strany.

Zástupný blok je po vizuální stránce stejný jako instance bloku v kreslícím plátně. Reprezentuje typ bloku, které můžeme přetažením z palety do kreslícího plátna instancovat. Při přetažení se z něj stává instance bloku. Do fragmentu je možné vložit více instancí jednoho typu bloku, proto jsou v paletě vždy vypsány všechny dostupné typy bloků.

Editor vstupů Pro ruční editaci spojení a pro nastavení hodnot vstupních hodnot se využívá editor vstupů. Ten je realizován jako modální okno, které se vyvolá kliknutím na název vstupu. V kaskádě se informace ukládají pouze u vstupu, editor tedy nelze vyvolat kliknutím na výstup.



Obrázek 5.2.4: Mockup editoru kaskády – editor vstupu

Editor nadřazeného bloku

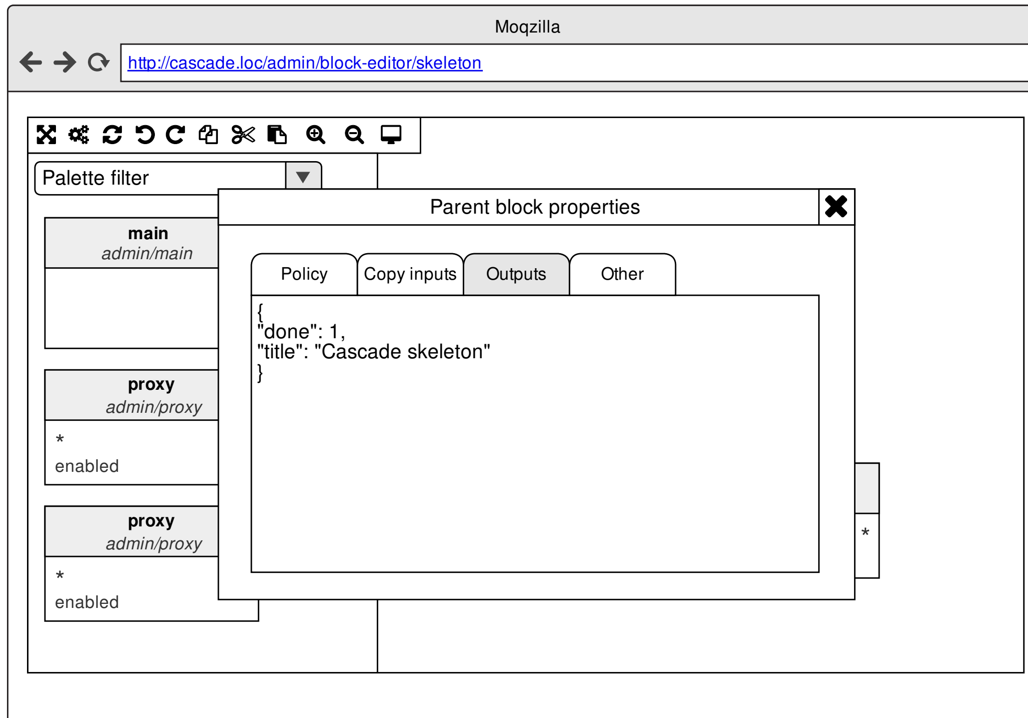
Editor nadřazeného bloku (obrázek 5.2.5) bude také realizován jako modální okno. Jednotlivé vlastnosti, které editor zná, budou vykresleny jako záložky. Zbytek neznámých vlastností pak bude vyčleněn do poslední záložky *Other*.

Editor zná tyto vlastnosti:

- *Policy* – nastavení oprávnění k přístupu k bloku, a jak reagovat, pokud uživatel přístup nemá (neřeší na úrovni entit).
- *Copy inputs* – které vstupy kopírovat na které výstupy (náhrada předání parametru do funkce).

- *Outputs* – nastavení výstupu fragmentu kaskády reprezentovaného instancí proxy bloku (náhrada návratové hodnoty funkce).
- *Forward outputs* – zastaralá implementace nahrazená vlastností *Outputs*.

První záložka, ve které bude nějaký obsah (tedy první známá vlastnost nadřazeného bloku), bude aktivována.



Obrázek 5.2.5: Mockup editoru kaskády – editor nadřazeného bloku

5.2.4 Kreslení spojnic

Spojnice mezi bloky musí být dle požadavků (viz sekce 5.1.1 a 5.1.2) kresleny co nejpřehledněji. Původní verze editoru (viz sekce 4.1.1) je kreslila jako jednoduché kubické Bézierovi křivky a neřešila kolize s ostatními bloky – cesta tedy vždy sestávala pouze ze zdrojového a cílového bodu.

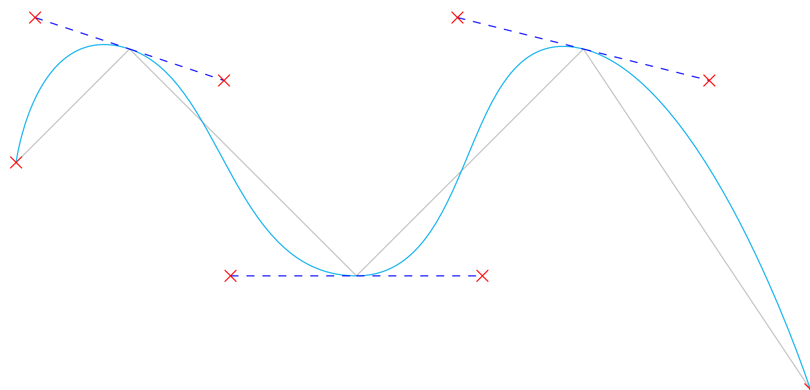
Výpočet cesty bez kolizí s dalšími bloky

Pro hledání cesty je možné použít několik přístupů. Podrobný rozbor této problematiky se nachází v sekci 5.4.

Kreslení křivky procházející body

Pro kreslení cesty spojnice se využívá křivka typu *Cardinal spline* [22] (někdy také *Canonical spline*). Jedná se o sekvenci individuálních křivek, které společně tvoří větší křivku. Křivka je definována pouze body, kterými prochází, a parametrem t, který

udává zakřivení. Ukázka takové křivky spolu se znázorněnými pomocnými body je vidět na obrázku 5.2.6.



Obrázek 5.2.6: Kubická spline křivka se znázorněnými kontrolními body

Jednotlivé sousední páry bodů reprezentují atomické kubické Bézierovi křivky. Na základě tří po sobě následujících bodů křivky se dle algoritmu 5.1 vypočítají kontrolní body těchto atomických křivek (pro každý bod dva kontrolní body). První a poslední křivka je pouze kvadratická, kontrolní body se tedy vypočítají pouze na základě dvojce bodů.

Parametr zakřivení lze chápat také jako délku tečny, která spojuje dvojce kontrolních bodů. Jeho hodnota musí být z uzavřeného intervalu $[0, 1]$. Pro $t = 0$ budou všechny tečny nulové (žádné zakřivení, na obrázku šedá křivka). Pro $t = 1$ vychází speciální typ křivky: *Catmull–Rom spline*.

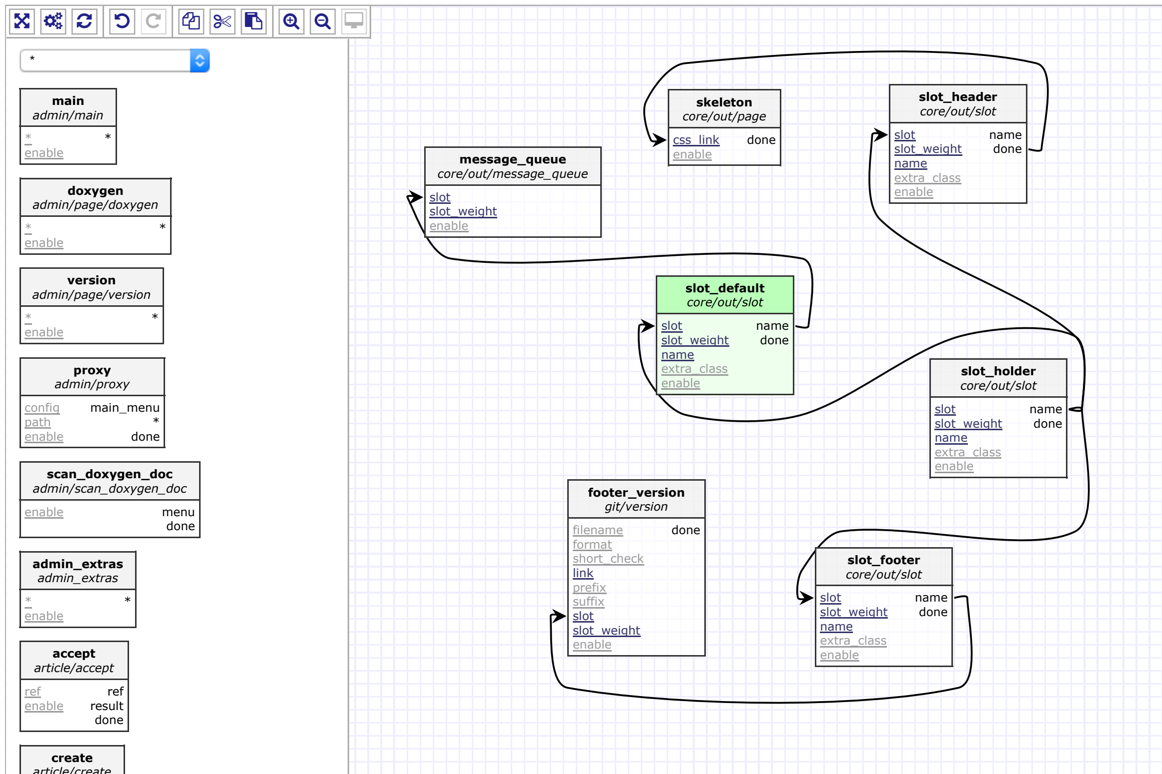
Algoritmus 5.1 Výpočet kontrolních bodů spline křivky

Data: po sobě následující body **A**, **B**, **C**, parametr zakřivení **t**
Result: kontrolní body pro bod **B**
$\mathbf{v}$ = vektor mezi body **A** a **C**;
$\mathbf{d_{ab}}$ = vzdálenost mezi body **A** a **B**;
$\mathbf{d_{bc}}$ = vzdálenost mezi body **B** a **C**;
$\mathbf{d_{ac}}$ = vzdálenost mezi body **A** a **C** přes bod **B**;

kontrolní body pak mají souřadnice:
$(B_x - v_x * t * \frac{d_{ab}}{d_{ac}}, B_y - v_y * t * \frac{d_{ab}}{d_{ac}})$
a
$(B_x + v_x * t * \frac{d_{bc}}{d_{ac}}, B_y + v_y * t * \frac{d_{bc}}{d_{ac}})$

5.3 Implementace

Na obrázku 5.3.1 je vidět výsledná podoba editoru kaskády. Tato sekce shrnuje některé jeho implementační detaily.



Obrázek 5.3.1: Výsledná podoba editoru kaskády

5.3.1 Inicializace a konfigurace

Výsledný editor byl realizován jako jQuery plugin, který se registruje do funkce: `$.blockEditor(options)`. Volá se nad elementem `<textarea>` a v parametru přebírá konfiguraci v objektu `options`, která přepisuje výchozí hodnoty uvedené v tabulce 5.1. Takto uvedenou konfiguraci lze ještě přepsat pomocí *data atributu* přímo na elementu. Na následující ukázce kódu je vidět inicializace editoru a následné přepsání konfigurace data atributem:

```
1  <!-- přepíšeme hodnotu uvedenou při inicializaci (250 -> 50) -->
2  <textarea data-block-editor-opts="{historyLimit: 50}">
3
4  <script type="text/javascript">
5      $(document).ready(function() {
6          $('textarea').blockEditor({
7              historyLimit: 250 // přepíše výchozí hodnotu (1000 -> 250)
8          });
9      });
10 </script>
```

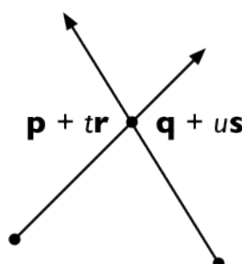
5.3.2 Výpočet průsečíků

Při hledání cesty pro vykreslení spojnice mezi bloky je nutné kontrolovat kolize s ostatními bloky. Blok je graficky reprezentován obdélníkem, cesta mezi bloky se zjednodušuje na přímku procházející uspořádanou množinou bodů. Hledáme tedy průsečík mezi dvěma přímkami. Pro jejich výpočet byla využita dvou dimenzionální varianta algo-

Klíč	Popis	Výchozí hodnota
paletteData	URL pro načtení dat palety	'...'
historyLimit	Maximum stavů v historii	1000
splineTension	Parametr zakřivení spline křivky	0.3
canvasOffset	Posunutí při kreslení	30 [px]
canvasExtraWidth	Přidaná šířka na levé a pravé straně plátna	1500 [px]
canvasExtraHeight	Přidaná výška na horní a dolní straně plátna	1500 [px]
canvasSpeed	Rychlost scrollování po plátně	2
viewOnly	Režim statického obrázku (viz 5.3.10)	false
scrollLeft	Iniciální horizontální posunutí kreslícího plátna	0 [px]
scrollTop	Iniciální vertikální posunutí kreslícího plátna	0 [px]

Tabulka 5.1: Konfigurace editoru kaskády

ritmu popsaného v článku *Intersection of two lines in three-space* [23]. Popis výsledného algoritmu 5.2 vychází z obrázku 5.3.2.



Obrázek 5.3.2: Výpočet průsečíku dvou přímek

5.3.3 Výpočet úhlů pomocí atan2

Spojnice mezi bloky se nehledá přímo z hraničních bodů bloků – za první a před poslední bod cesty se nejprve přidají po ose x mírně vychýlené pomocné body. Díky tomu pak vykreslená spojnice jasně ukazuje na vstup (respektive výstup) bloku.

Po nalezení správné cesty dojde k jejímu vyhlazení, v rámci kterého jsou vynechány přebytečné body (jejichž vynecháním nevznikne žádná nová kolize). Finální fáze vyhlazování nakonec upraví vertikální pozici pomocných bodů dle směru spojnice. K výpočtu úhlů je využita funkce `atan2(y,x)` [24], která je definována jako:

$$\operatorname{atan2}(y, x) = \begin{cases} \arctan \frac{y}{x} & x > 0 \\ \arctan \frac{y}{x} + \pi & y \geq 0, x < 0 \\ \arctan \frac{y}{x} - \pi & y < 0, x < 0 \\ +\frac{\pi}{2} & y > 0, x = 0 \\ -\frac{\pi}{2} & y < 0, x = 0 \\ \text{nedefinováno} & y = 0, x = 0 \end{cases}$$

Algoritmus 5.2 Výpočet průsečíku

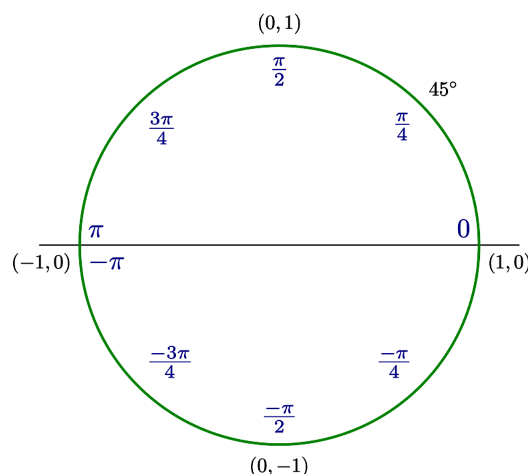
Data: dvě úsečky definované pomocí počátečního bodu a vektoru: $\mathbf{p} + \mathbf{r}, \mathbf{q} + \mathbf{s}$
Result: průsečík (pokud existuje)

1. Hledaný průsečík leží na souřadnicích odpovídající rovnici: $\mathbf{p} + t\mathbf{r} = \mathbf{q} + u\mathbf{s}$.
2. Definujeme vektorový součin pro dvě dimenze jako: $\mathbf{v} \times \mathbf{w} := \mathbf{v_x w_y} - \mathbf{v_y w_x}$.
3. Vektorově roznásobíme pomocí $\mathbf{s}$ zprava: $(\mathbf{p} + t\mathbf{r}) \times \mathbf{s} = (\mathbf{q} + u\mathbf{s}) \times \mathbf{s}$.
4. To lze díky $\mathbf{s} \times \mathbf{s} = 0$ zjednodušit na $t(\mathbf{r} \times \mathbf{s}) = (\mathbf{q} - \mathbf{p}) \times \mathbf{s}$.
5. A nakonec vydělíme: $t = (\mathbf{q} - \mathbf{p}) \times \mathbf{s}/(\mathbf{r} \times \mathbf{s})$.

Obdobným postupem lze dopočítat i parametr u. Mohou nastat tyto případy:

1. Pokud je $\mathbf{r} \times \mathbf{s} = 0$ a $(\mathbf{q} - \mathbf{p}) \times \mathbf{r} = 0$, pak jsou obě úsečky kolineární. Pokud navíc platí buď $0 \leq (\mathbf{q} - \mathbf{p}) \cdot \mathbf{r} \leq \mathbf{r} \cdot \mathbf{r}$ nebo $0 \leq (\mathbf{p} - \mathbf{q}) \cdot \mathbf{s} \leq \mathbf{s} \cdot \mathbf{s}$, pak se úsečky překrývají. Pokud ani jedna z dodatečných podmínek neplatí, úsečky se nedotýkají.
2. Pokud platí $\mathbf{r} \times \mathbf{s} = 0$, ale $(\mathbf{q} - \mathbf{p}) \times \mathbf{r} \neq 0$, pak se jedná o rovnoběžky (bez průsečíku).
3. Pokud platí $\mathbf{r} \times \mathbf{s} \neq 0$ a $0 \leq t \leq 1$ a $0 \leq u \leq 1$, úsečky se protínají v bodě $\mathbf{p} + t\mathbf{r} = \mathbf{q} + u\mathbf{s}$.
4. Pokud není splněna ani jedna z podmínek výše, úsečky nejsou paralelní, ale nemají žádný průsečík.

Slouží pro výpočet úhlu mezi vektorem definovaným jako (v_x, v_y) a osou x. Na rozdíl od klasické funkce $arctan(x)$ využívá obě souřadnice pro určení správného kvadrantu. Rozsah hodnot této funkce je $-\pi$ až π, některé její hodnoty jsou znázorněny na obrázku 5.3.3.



Obrázek 5.3.3: Grafické znázornění hodnot funkce `atan(y,x)` (zdroj: [25])

Způsob výpočtu úhlu mezi dvěma vektory (v_x, v_y) a (w_x, w_y) pomocí funkce `atan2(y,x)` znázorňuje algoritmus 5.3. Postup je jednoduchý, stačí vypočítat úhel mezi oběma vektory a osou x, a následně je mezi sebou odečíst. Výsledek je v intervalu $-\pi$ až π, je vhodné ho normalizovat do klasického intervalu 0 až π.

Algoritmus 5.3 Výpočet úhlu pomocí `atan2(y,x)`

Data: vektory **v** a **w**
Result: úhel ψ
$\psi = atan2(w_y, w_x) - atan2(v_y, v_x)$;
if $\psi < 0$ **then**
| $\psi = \psi + 2\pi$;
end

5.3.4 Drag'n'drop

Označování bloků, jejich přesun po plátně, tvorba nových spojení mezi bloky – to vše jsou funkce editoru, které fungují na základě metody drag'n'drop. Původní verze editoru pro snazší implementaci využívala knihovnu *jQuery UI*[6], nová verze ale nesmí mít kromě frameworku jQuery žádné další závislosti[7].

V první fázi implementace byla použita nativní podpora pro drag'n'drop[8], která je součástí specifikace HTML5. Ta ale bohužel nebyla dostatečně přizpůsobitelná, a bylo

[6] *jQuery UI*: https://jqueryui.com/
[7] *jQuery UI* je externí knihovna, není součástí frameworku *jQuery*
[8] Drag'n'drop: https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Drag_and_drop

tedy nutné naprogramovat vlastní implementaci, která využívá nativní JavaScriptové události *mousedown*, *mousemove* a *mouseup*. Na základě pozice kurzoru, která je v události přístupná v objektu *MouseEvent* jako *clientX* a *clientY*[9].

Při výpočtu relativní pozice kurzoru vůči kreslícímu plátnu bylo nutné zohlednit přiblížení (viz sekce 5.3.6) – použitá CSS3 transformace je totiž aplikována pouze na obsah editoru a nemá na hodnotu *clientX* a *clientY* vliv.

5.3.5 Úložiště prohlížeče

Při práci s editorem nastává potřeba ukládat dočasná data, jako například historii (viz sekce 5.3.7), schránku (viz sekce 5.3.8) nebo paletu bloků (viz sekce 5.3.9). Pro uchovávání dočasných dat se využívají dvě úložiště prohlížeče:

- `localStorage` – perzistentní úložiště, uchová data i po zavření prohlížeče. Centrální úložiště pro celý prohlížeč, umožňuje sdílet data mezi jednotlivými záložkami.
- `sessionStorage` – dočasné úložiště, uchová data jen po dobu co je otevřená záložka prohlížeče. Lokální úložiště, přístupné jen v rámci záložky.

Obě tato úložiště mají společnou výhodu oproti sušenkám (*cookies*): data se nepřenášejí s každým požadavkem na server. Úložiště je společné vždy pro celou doménu, je proto nutné klíče prefixovat, aby nedocházelo k jejich přepisování kvůli shodným názvům, například mezi editorem kaskády a Smalldb editorem.

5.3.6 Přiblížení/oddálení

Přibližování/oddalování kreslícího plátna bylo implementováno pomocí CSS3 transformace, která se aplikuje na vnořený `<div>` kontainer. Díky tomu lze snadno upravit velikost všech bloků, problém ale nastává při kreslení spojnic. Element `<canvas>`, do kterého se vykreslují, se sice poměrně zvětší (respektive zmenší), při výpočtu vertikální pozice počátečního a koncového bodu (který vychází přímo ze svislé stěny bloku) je ale nutné přiblížení zohlednit. Poměrně upravený blok má totiž jinou výšku, než jakou získáme pomocí JavaScriptu z DOMu.

Stejný problém nastává i u práce pomocí drag'n'drop, kde je nutné přiblížení zohlednit vůči získané pozici kurzoru.

Hodnota přiblížení se uchovává v dočasném úložišti prohlížeče `sessionStorage`, při otevření nového okna nebo záložky je vždy použito výchozí zobrazení (100%).

5.3.7 Historie

Každá akce, která mění fragment kaskády, zavolá na instanci editoru funkci `onChange()`. Tato funkce nejprve serializuje celý fragment, potom ho porovná s posledním záznamem v historii, a pokud se liší, přidá ho tam. Historie se ukládá do dočasného úložiště prohlížeče `sessionStorage`, po zavření okna prohlížeče je vždy samo promazáno.

[9] *MouseEvent.clientX*: http://www.w3schools.com/jsref/event_clientx.asp

Jednotlivé stavy historie jsou uchovávány jako JSON řetězce, jejich porovnávání je proto triviální a stav je vždy konzistentní. Vzhledem k možné zvýšené paměťové náročnosti je velikost schránky omezena na 1000 záznamů a starší záznamy se automaticky mažou. Tato hodnota lze upravit v nastavení pluginu při jeho inicializaci.

5.3.8 Schránka

Označené bloky lze kopírovat nebo vyjmout a posléze někam vložit. Schránka využívá úložiště `localStorage`, díky tomu umožňuje kopírovat bloky napříč otevřenými záložkami prohlížeče. Do schránky se bloky ukládají serializované jako JSON řetězce.

5.3.9 Načítání palety bloků

Paleta bloků v sobě udržuje seznam všech použitelných typů bloků, které programátor může v editoru použít. V současné době je skrze paletu přístupných 120 typů bloků, jedná se tedy často o řádově větší objem dat, než jaký představuje samotný fragment kaskády, který chceme editovat. Paleta bloků se proto cachuje do persistentního úložiště prohlížeče `localStorage`. Po načtení a spuštění editoru se pomocí AJAXu asynchronně načte aktuální podoba palety, porovná se se zacachovanou hodnotou a až v případě rozdílu se upraví DOM (což je z celého procesu nejpomalejší operace).

5.3.10 Režim statického obrázku

Editor lze spustit v režimu statického obrázku, který lze pak snadno pomocí prohlížeče vytisknout do PDF nebo vložit jako statický obsah do HTML stránky. Tento režim lze vynutit nastavením konfigurační direktivy *viewOnly* na *true*. Spojnice se vykreslují do elementu `<canvas>`, pokud je navíc do stránky nalinkovaná knihovna *Canvas2Svg*[10], vykreslí se pomocí SVG, tedy vektorově (vhodné při tisku do PDF). Tímto způsobem vznikly všechny snímky obrazovky s editorem kaskády použité v této práci.

Canvas2Svg je knihovna, která se zaregistruje namísto kreslícího 2D kontextu elementu `<canvas>`, a dále se nad ní volají totožné kreslící metody. Na konci kreslení se pak z knihovny vyexportuje SVG řetězec, který se vloží do DOM místo elementu `<canvas>`.

5.4 Algoritmus pro vyhýbání se blokům

Pro udržení dostatečné přehlednosti při vyšším počtu bloků v jednom fragmentu kaskády je důležité kreslit spojnice mezi nimi tak, aby neprocházeli pod jinými bloky. Tato operace musí být dostatečně rychlá, protože je potřeba přepočítat cestu například při pohybu blokem nebo při tvorbě nového spojení.

V této části jsou popsány tři implementované algoritmy pro hledání cesty bez kolizí s ostatními bloky. Sekce 5.6.2 se dále zabývá jejich srovnáním z pohledu rychlosti generování potřebných struktur a samotného hledání cesty.

[10]Canvas2Svg: https://github.com/gliffy/canvas2svg

5.4.1 Naivní algoritmus

Algoritmus 5.4 popisuje rychlý způsob jak předejít kolizím prvního řádu – tedy kolizí na přímé cestě mezi počátečním a koncovým bodem. Pokud takové kolize existují, přidá do cesty body, kterými se jim vyhne. Na nové cestě ale mohou vzniknout nové kolize s jinými bloky. Tyto druhotné kolize už ale tento algoritmus neřeší.

Algoritmus 5.4 Vyhýbání se blokům – naivní algoritmus

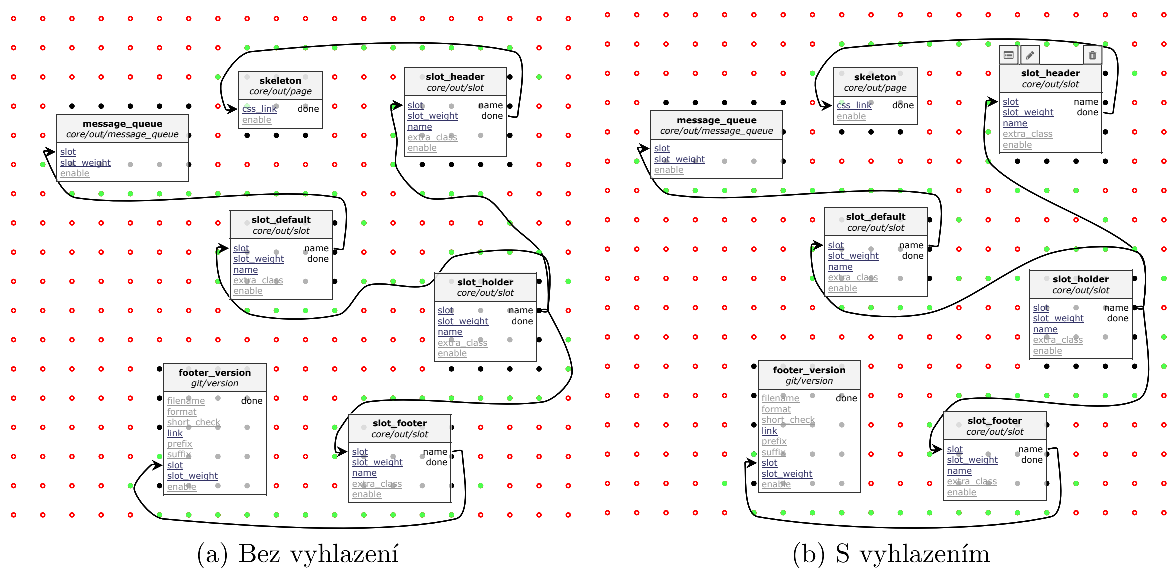
Data: počáteční a koncový bod, seznam bloků
Result: cesta

1. Nejprve zkusíme přímou cestu mezi počátečním a koncovým bodem. Na této cestě hledáme kolize s ostatními bloky.
2. Pro každou kolizi vybereme jeden nebo dva okrajové body kolidujícího bloku:
   (a) Pro každý průsečík vybereme nejbližší rohový bod.
   (b) Pokud nalezneme dva různé rohové body, zkontrolujeme jestli se nejedná o diagonálu. V takovém případě jeden roh vybereme a nahradíme za sousední.
   (c) Rohové body přidáme do cesty.
3. Řazení cesty (při přidávání rohových bodů není zaručeno správné pořadí):
   (a) Vypočítáme matici vzdáleností mezi jednotlivými body.
   (b) Seřadíme cestu tak, aby byla vždy využita nejkratší cesta mezi každými dvěma body.
4. Vyhlazení cesty:
   (a) Procházíme cestu, zkoušíme vynechat jednotlivé body a kontrolujeme vznik nových průsečíků.
   (b) Pokud vznikne jeden průsečík, přidáme ho do cesty místo vynechaného bodu.
   (c) Pokud nevznikne žádný nový průsečík, odebereme bod a zkusíme vynechat následující.
   (d) Opakujeme dokud neprojdeme celou cestu.

5.4.2 Nejkratší cesta v gridu

Další algoritmus hledá nejkratší cestu v gridu, tedy v eulerovském grafu, ve kterém jsou jednotlivé uzly rozmístěny do mřížky. Vzdálenost mezi sousedními uzly je vždy 1 pro vodorovné a svislé hrany, a $\sqrt{2}$ pro diagonální hrany. Uzly, které na grafu překrývají

jednotlivé bloky, označíme jako zdi – těmito body pak cesta nesmí procházet. Dosažitelnost jednotlivých uzlů je pak vždy jen do sousedních uzlů, proto není při tvorbě gridu nutné generovat i jednotlivé hrany. Na obrázku 5.4.1 je vidět příklad fragmentu kaskády se znázorněným gridem. Červené tečky jsou uzly, kterými může cesta vést. Zeleně je vykreslena nalezená cesta mezi bloky. Černá barva pak značí zeď. Na pravém obrázku je spojnice před vykreslením vyhlazena – nepotřebné body jsou vynechány. Tento krok se dle měření (viz sekce 5.6.2) ukázal jako velmi náročný, již při počtu deseti bloků ve fragmentu přestává být pro interaktivní aplikaci použitelný. Výsledná spojnice ale bez vyhlazení nevypadá moc dobře, je proto vhodné vyhlazení provést alespoň jednou na konci překreslování, tedy po skončení práce uživatele.



(a) Bez vyhlazení (b) S vyhlazením

Obrázek 5.4.1: A* v gridu

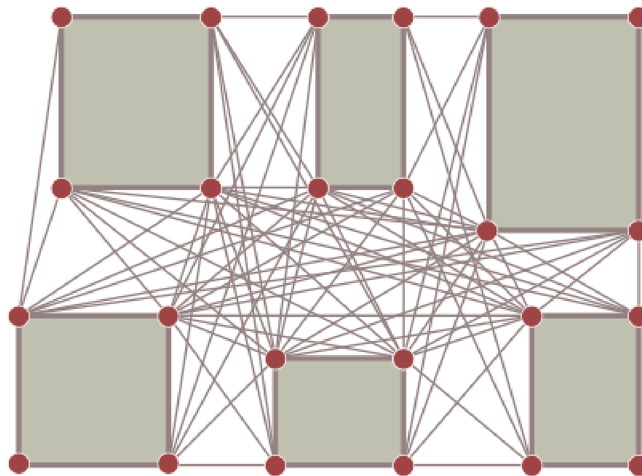
Algoritmus A*

Pro hledání nejkratších cest v gridu byl zvolen algoritmus A* [26]. Jedná se o rozšíření Dijkstrova algoritmu [27], které využívá pro zrychlení heuristiky určující směr prohledávání grafu. Postupně prochází graf a vybírá cestu, která má minimální vzdálenost $f(v) := d(s, v) + h(v, t)$, kde v je aktuální uzel, s a t jsou startovní a cílový bod, $d(s, v)$ značí dosavadní vzdálenost do bodu v a $h(v, t)$ značí heuristickou funkci. Tato heuristická funkce je dolním odhadem délky cesty z uzlu v do uzlu t. Díky tomu je zaručeno, že algoritmus najde optimální cestu.

V našem editoru využíváme heuristiku *diagonální vzdálenosti*, známou též jako *Chebyshevova vzdálenost* [28]. Ta je daná vzorcem $h(v, t) = max(|v - t|)$. Při implementaci je využita prioritní fronta, která uzly ukládá podle metriky $f(v)$ do binární haldy.

5.4.3 Nejkratší cesta v eukleidovském grafu

Třetí variantou bylo opět hledání nejkratší cesty pomocí algoritmu A*, tentokrát ale v eukleidovském grafu sestaveného z okrajových uzlů jednotlivých bloků. Počet uzlů se tím mnohonásobně zmenšil, problém ale nastal při hledání jednotlivých hran. Obrázek 5.4.2 znázorňuje příklad šesti bloků a všech platných hran mezi nimi. Na základě měření (viz sekce 5.6.2) se ukázalo již generování tohoto grafu jako příliš výpočetně náročné, a to nejen pro interaktivní překreslování, ale i pro dopočítání lepší cesty po skončení interakce.



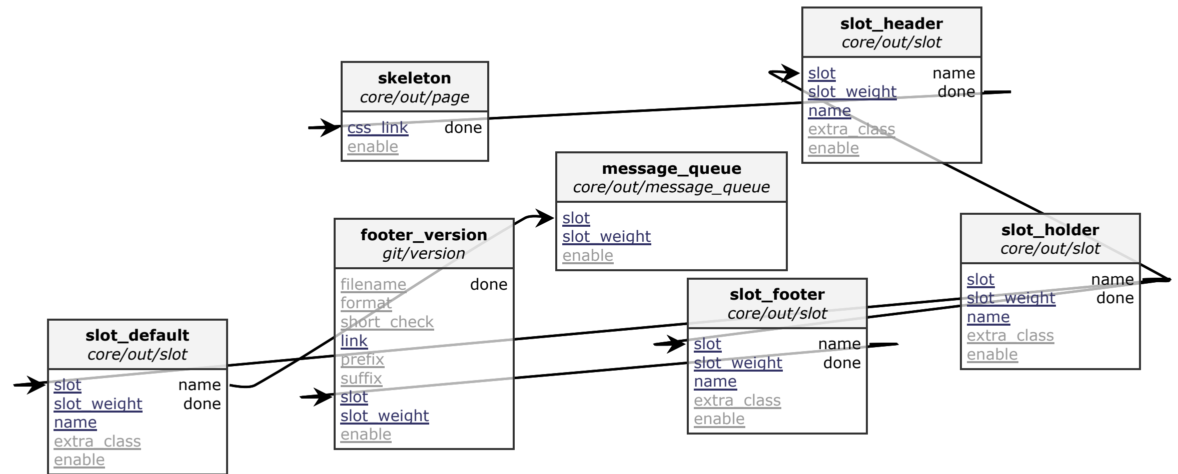
Obrázek 5.4.2: Problém velkého počtu hran – A* v grafu (zdroj: [28])

5.4.4 Srovnání a výběr řešení

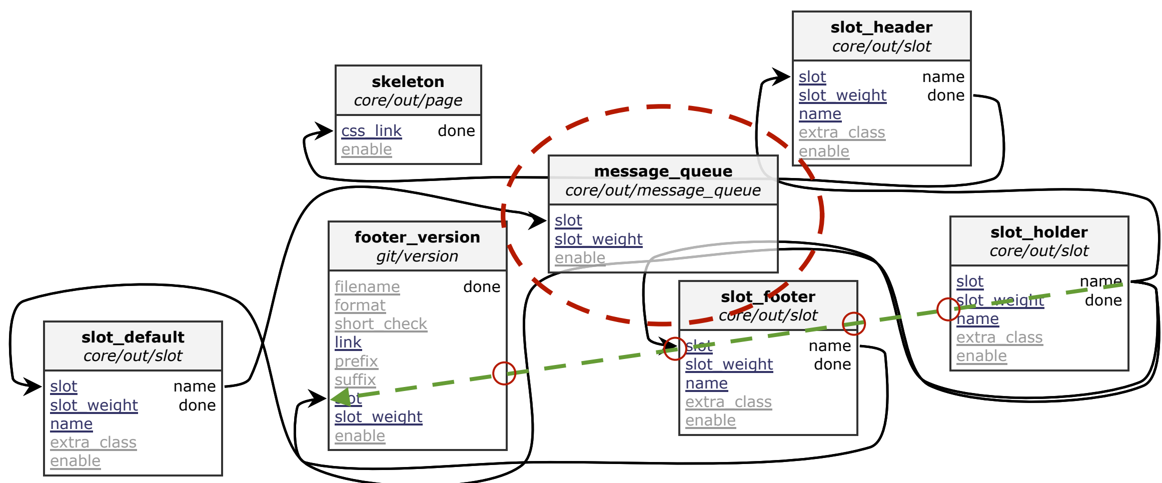
Na obrázcích 5.4.3 až 5.4.6 je vidět jeden fragment kaskády vykreslený pomocí všech implementovaných algoritmů. První obrázek kolize neřeší a pouze spojí zdrojový bod s cílovým. Na druhém obrázku je vidět naivní algoritmus, kde červeně znázorněný blok ukazuje na problém kolizí vyššího řádu – tedy kolizí, které nevzniknou při prostém spojení bodů úsečkou (na obrázku znázorněny zeleně). Třetí a čtvrtý obrázek ukazuje řešení pomocí algoritmu A* v gridu – variantu s vyhlazováním cesty i bez vyhlazování.

Naivní algoritmus v tomto konkrétním případě generuje na první pohled nejlepší výsledek, i když nedokáže předejít kolizím s prostředním blokem. Jeho rychlost se ale kvůli náročnému počítání průsečíků ukázala dle měření pro větší počet bloků příliš náročná (viz sekce 5.6.2). Vyhlazená varianta algoritmu A* se také při stoupajícím počtu bloků v kaskádě rychle stává pro interaktivní aplikaci nepoužitelnou. Navíc vyhlazená cesta v tomto konkrétním případě nevypadá tak dobře, jako varianta bez vyhlazení.

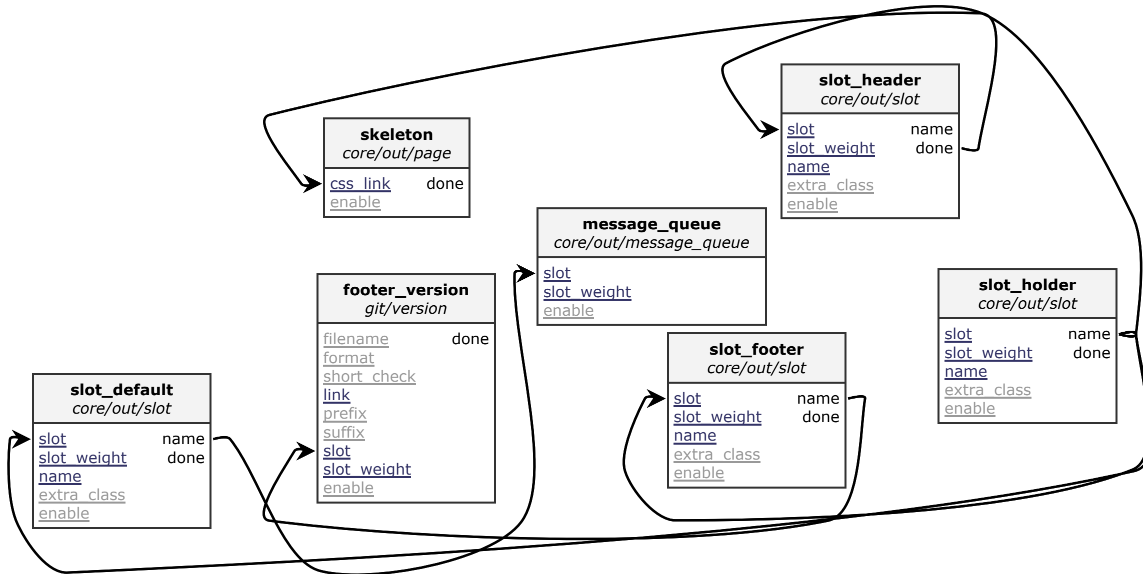
Vzhledem k vysoké výpočetní náročnosti hledání průsečíků byla jako výchozí implementace zvolena varianta algoritmu A* v gridu bez vyhlazování, která jako jediná tento výpočet nepoužívá.



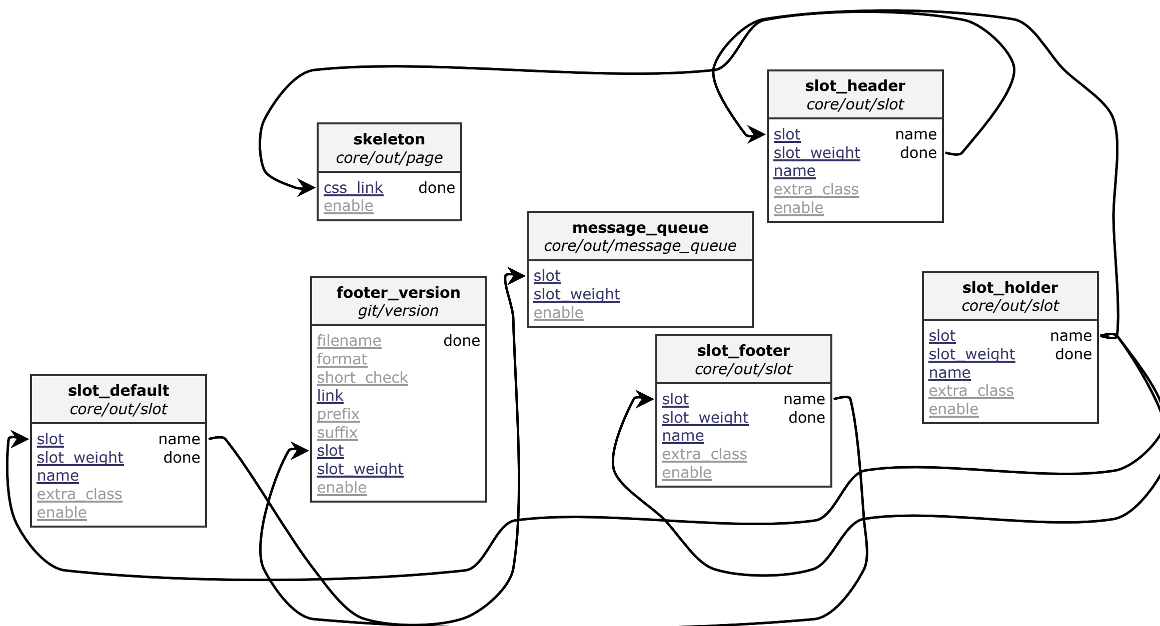
Obrázek 5.4.3: Srovnání algoritmů pro vyhýbání se blokům – bez vyhýbání



Obrázek 5.4.4: Srovnání algoritmů pro vyhýbání se blokům – naivní algoritmus



Obrázek 5.4.5: Srovnání algoritmů pro vyhýbání se blokům – nejkratší cesta s A* v gridu, s vyhlazováním cesty



Obrázek 5.4.6: Srovnání algoritmů pro vyhýbání se blokům – nejkratší cesta s A* v gridu, bez vyhlazování cesty

5.5 Testování

5.5.1 Heuristická evaluace

Heuristická evaluace je jednou z nejpopulárnějších metod pro testování použitelnosti. Její výhody jsou relativně snadná realizace, nižší časová náročnost a možnost aplikace jak teoretických tak praktických zkušeností testera. Její nevýhodou je pak nemožnost pokrytí všech nedostatků testované aplikace. Existují dva způsoby heuristické evaluace. V tom prvním se tester drží předem daného seznamu bodů, v tom druhém pak vychází pouze z vlastních zkušeností a teoretických znalostí – v takovém případě se jedná spíše o expertní posudek než o heuristickou analýzu.

Tato sekce popisuje problémy, které u editoru kaskády odhalila heuristická evaluace na základě Nielsenových heuristik definovaných v příloze A.

Vkládání bloků ze schránky (A.1) Při vkládání bloků ze schránky do kreslícího plátna editoru se zachovala jejich původní pozice. Pokud tedy uživatel vyjmul bloky na jednom místě, přesunul se po plátně na jiné místo a tam se je pokusil vložit, neviděl žádnou odezvu – původní pozice bloků totiž byla mimo viditelnou část obrazovky. Řešením tohoto problému bylo vkládat obsah schránky vždy doprostřed aktuálně viditelného pole.

Podobnost práce s bloky a ikonami na ploše (A.2) Práce s bloky (označování, pohyb po plátně, práce se schránkou) u uživatele evokuje práci s ikonami na ploše operačního systému. Původní implementace toto nerespektovala. Bylo nutné upravit způsob označování bloků a pozici nově vložených bloků ze schránky (viz výše).

Nekonzistentní chování klávesových zkratek v editoru vstupů (A.4) Editor vstupů bloku je proveden jako HTML formulář. V takovém formuláři se běžně používá pro označení následujícího elementu tabulátor, respektive pro krok zpět klávesová zkratka *Shift + Tab*. V editoru vstupů to ale takto nefungoval – z důvodu vlastních callbacků pověšených na události *keydown*, které zpracovávají klávesové zkratky.

Nápověda (A.10) Některé akce se v editoru kaskády dělají pouze pomocí klávesových zkratek nebo dvojkliku na plátno (přidání nového stavu). Navíc základní operace jako tvorba spojení se v původní verzi dala udělat pouze pomocí tažení myši spolu s klávesou CTRL. Noví uživatelé na to nebyli nijak upozorněni, pro řešení tohoto problému byla doplněna krátká uživatelská nápověda, kterou lze zobrazit pomocí tlačítka v panelu nástrojů.

5.5.2 Uživatelské testování

Pro uživatelské testování byla zvolena metoda testování s moderátorem, při testu dostal participant pokyny uvedené v testovacím scénáři. Test probíhal v přirozeném prostředí – přímo v kancelářích firmy, zabývající se vývojem webových aplikací.

Cílová skupina

Cílová skupina jsou programátoři webových aplikací, nemusí mít nutně zkušenosti s kaskádou.

Participanti

Uživatelské testování bylo provedeno na dvou participantech, oba ve věkové skupině 20-26 let, oba z prostředí firmy vyvíjející webové aplikace. Ani jeden z nich neměl zkušenosti s kaskádou, před samotným testem jim byla krátce představena.

Druhý participant dostal k otestování novou verzi editoru, opravenou na základě poznatků z testování s prvním participantem.

Testovací scénář

Pro testování byl využit následující scénář, který většinu možných interakcí s editorem:

Na počítači je otevřený prohlížeč a v něm dvě záložky s editorem kaskády, každá s jiným fragmentem.

1. Otevřete první záložku s editorem.
2. Zvětšete zobrazení na celou obrazovku (režim *fullscreen*).
3. Označte skupinu tří libovolných bloků.
4. Vyjměte tyto bloky do schránky.
5. Otevřete druhou záložku s editorem.
6. Přesuňte vložené bloky po plátně na libovolnou novou pozici.
7. Vraťte se o krok zpět (na původní pozici po vložení ze schránky).
8. Vytvořte nové spojení mezi libovolnými dvěma bloky.
9. Nastavte nějakému bloku vstupní hodnotu na řetězec „nová hodnota“. Použijte vstup, který je ve výchozím stavu (šedé písmo).
10. Nastavte nějakému bloku nové jméno (jeho identifikátor – první řádek v hlavičce).

Výsledky – participant 1

Při zvětšení zobrazení na celou plochu se zobrazila po levé straně automaticky zmenšená paleta – participanta to překvapilo, podvědomě mu překáží. Naopak při standardním zobrazení, kde paleta není vůbec vidět, by ho vůbec nenapadlo, kde ji hledat (původně byla přístupná při najetí nad panel nástrojů). Tlačítko pro zobrazení na celou obrazovku by hledal vpravo, ne v přímo v panelu nástrojů mezi ostatními akčními tlačítky. Při označování bloků napřed zkoušel označit tahem po plátně – původně k tomu ale bylo potřeba držet stisklou klávesu *Ctrl*, bez ní se dalo plátno posouvat. Při vkládání bloků do druhého editoru měl jeden z bloků ve schránce shodné jméno s jiným blokem, který již v editoru byl načtený – na participanta pak vyskočila hláška o zadání nového jména pro tento blok. Hláška byla ale příliš obecná, a tak ji nechápal. Při pohybu bloků po plátně by čekal změnu kurzoru.

Výsledky – participant 2

Druhý participant neměl během testu žádné problémy, objevil ale jednu implementační chybu. Při vkládání obsahu schránky do druhého editoru omylem místo *Vložit* vyvolal akci *Kopírovat*, která chybně uložila do schránky prázdné pole bloků a pak následné pokusy o vložení původního obsahu selhávaly.

Výsledky – souhrn

Na základě uživatelského testování byly provedeny tyto změny:

- Přidána stručná uživatelská nápověda.
- Pohyb po plátně se stisknutou klávesou *Ctrl*, bez ní označování bloků.
- Tvorba spojení bez nutnosti držet klávesu *Ctrl*.
- Paleta se skrývá i v režimu zobrazení na celou obrazovku, pro její zobrazení je nutné najet myší na levou část editoru, kde je znázorněno tlačítko s šipkou.

5.6 Experimenty

5.6.1 Konfigurace testovacího prostředí

Konfigurace počítače, na kterém byly prováděny experimenty:

Model: MacBook Pro 13“ 2013

Procesor: Intel Core i5 2,5 GHz

Paměť: 8 GB DDR3 RAM, 1600 MHz

Úložiště: Samsung SSD 840 Pro 256 GB

Grafická karta: Intel HD Graphics 4000, 1024 MB VRAM

Prohlížeč: Google Chrome 42

5.6.2 Srovnání rychlosti implementovaných algoritmů

Rychlost jednotlivých algoritmů pro hledání cesty bez kolizí popsaných v sekci 5.4 je srovnána v tabulce 5.2 a na grafu 5.6.1. Naměřené hodnoty odpovídají vždy průměru ze sta měření.

Jedinou použitelnou variantou pro fragmenty kaskády o více jak patnácti blocích se ukázal algoritmus A* s vypnutým vyhlazováním křivky. Fáze vyhlazování se ukázala jako velmi výpočetně náročná – problém je v kontrole nově vzniklých kolizí při každém zkrácení cesty o jeden uzel. Manuální kontrola kolizí s ostatními bloky pomocí hledání průsečíků je také problémem generování grafu pro poslední variantu algoritmu A* (viz obrázek 5.4.2). Pro každý pár uzlů je totiž nutné zkontrolovat jejich vzájemnou viditelnost a na základě toho mezi nimi vytvořit hranu. Při počtu bloků $N = 10$ tedy

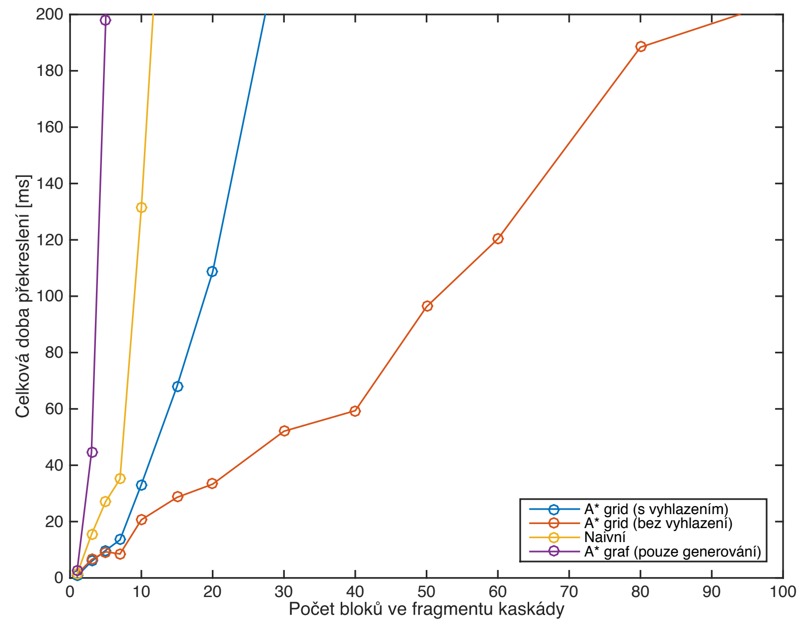
máme čtyřicet uzlů (každý blok je reprezentován čtyřmi rohovými body), a musíme zkontrolovat $(4N)^2$ možných hran, tedy 1600. Tato varianta byla na základě měření zavržena a nebyla dále implementována.

Počet bloků	Naivní [ms]	A* grid [ms]	A* grid [ms]	A* graf [ms]
vyhlazení	*ano*	*ne*	*ano*	*pouze generování*
1	0,7186	1,3680	1,4931	2,8321
3	5,8388	6,4811	15,3779	44,4723
5	9,7837	9,2884	27,1516	197,7208
7	13,5013	8,5044	35,0580	529,4121
10	33,1777	20,6456	131,3410	1525,9284
15	68,0045	28,5905	339,2848	-
20	108,5986	33,2923	-	-
30	232,5180	52,0732	-	-
40	-	59,3721	-	-
50	-	96,2569	-	-
60	-	120,3305	-	-
80	-	188,3718	-	-
100	-	205,0864	-	-

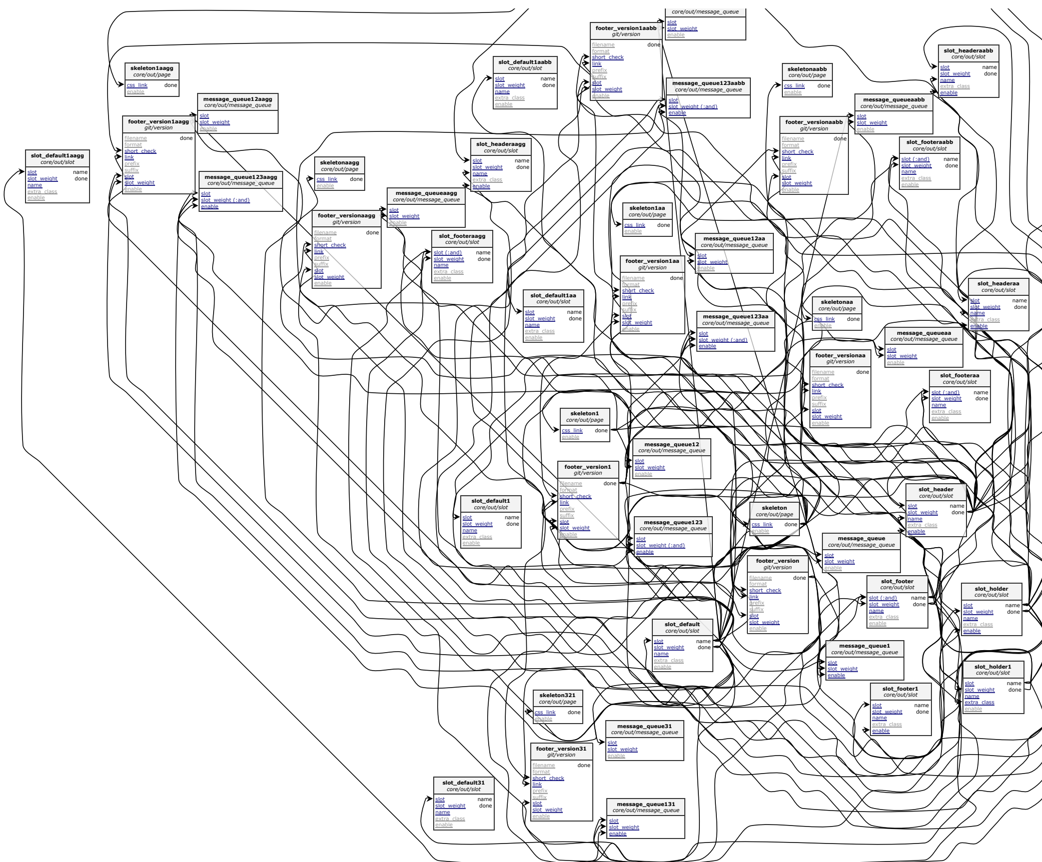
Tabulka 5.2: Doba vykreslování v závislosti na počtu bloků ve fragmentu kaskády

5.6.3 Maximální velikost grafu

Na obrázku 5.6.2 je vidět fragment kaskády s padesáti bloky. Jedná se o největší možný fragment, který bylo na testovací konfiguraci možné pomocí editoru relativně plynule ovládat. Při tomto množství bloků bylo nutné vypnout vyhlazování cest.



Obrázek 5.6.1: Graf závislosti doby vykreslování na počtu bloků ve fragmentu kaskády



Obrázek 5.6.2: Fragment kaskády s padesáti bloky – algoritmus A* v gridu bez vyhlazování

Kapitola 6

Grafický editor konečných automatů

6.1 Analýza

6.1.1 Funkční požadavky

Až na požadavek kompatibility s předchozí verzí editoru (který pro Smalldb neexistuje) a asynchronní načítání palety jsou funkční požadavky pro tento editor shodné se sekcí 5.1.1. Následuje výčet dalších funkčních požadavků kladených na editor konečných automatů.

Barevná interpretace

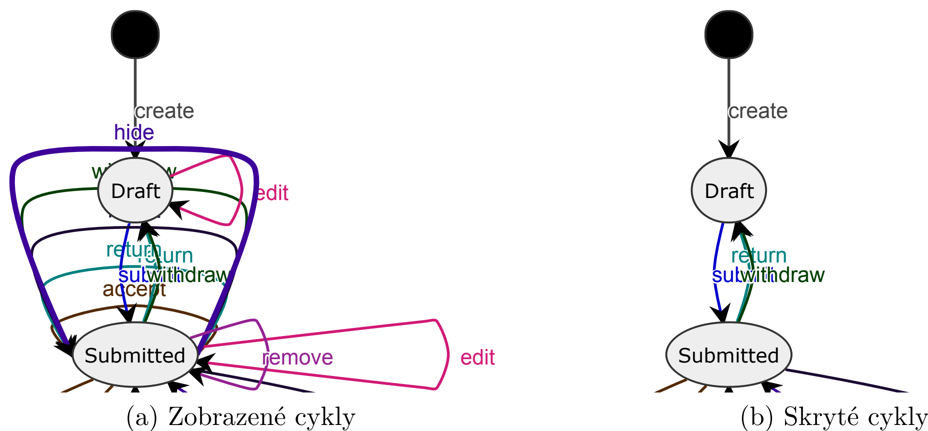
Stavy a přechody mohou obsahovat informaci o svojí barvě a popisku (*label*). Editor bude tyto informace respektovat při jejich vykreslování. Dále editor nabídne možnost automaticky obarvit všechny akce z předem připravené palety.

Skrývání cyklů

Při popisu business logiky v reálné aplikaci se velmi často vyskytují přechody do stejného stavu, tedy cykly. Na obrázku 6.1.1 je vidět část automatu s velkým počtem cyklů nad stavem *Submitted*. Pro snazší orientaci v diagramu je tedy nutné umožnit rychlé skrytí všech cyklů (a následně jejich zobrazení) pomocí tlačítka v panelu nástrojů.

Označování hran

Každá hrana ve vykresleném grafu představuje přechod a zároveň akci, ke které je tento přechod přiřazen. Každý přechod i akce mají vlastnosti, které musí jít prostřednictvím editoru upravovat. Proto je nutné, aby se dala hrana kliknutím označit.



(a) Zobrazené cykly

(b) Skryté cykly

Obrázek 6.1.1: Automat s větším počtem cyklů nad jedním stavem

Automatické rozložení stavů

Schéma, ve kterém je stavový automat ukládán, může obsahovat i jednotlivé pozice stavů v prostoru. Tato informace ale není povinná. Editor proto musí podporovat automatické rozmístění stavů tak, aby uživateli co nejvíce usnadnil orientaci ve vykresleném grafu.

6.1.2 Nefunkční požadavky

Nefunkční požadavky editoru konečných automatů jsou shodné se sekcí 5.1.2. Následuje výčet dalších nefunkčních požadavků kladených na editor konečných automatů.

Vzájemné a vícenásobné přechody (multihrany)

Při vykreslování vzájemné spojnice mezi dvěma různými stavy (tedy přechodu z A do B a druhého přechodu z B do A) musí být tvar spojnice jednosměrně vychýlen tak, aby oba přechody nesplývaly do jednoho.

Obdobné vychýlení musí být použito i při vykreslování vícenásobných přechodů mezi stejnými stavy (a se stejným směrem).

6.1.3 Formát a schéma automatu

JSON schéma

Stejně jako kaskáda (viz sekce 5.1.3), i Smalldb ukládá své entity ve formátu JSON:

```
1  {
2      "title": "Schéma Smalldb automatu",
3      "type": "object",
4      "properties": {
5          "actions": {
6              "type": "object",
7              "properties": {
8                  "color": { "type": "string" },
9                  "label": { "type": "string" },
10                 "transitions": {
11                     "type": "object",
```

```
12                      "properties": {
13                          "color": { "type": "string" },
14                          "label": { "type": "string" },
15                          "targets": {
16                              "type": "array",
17                              "items": { "type": "string" },
18                          }
19                      },
20                      "required": ["targets"],
21                      "additionalProperties": true
22                  }
23              },
24              "required": ["transitions"],
25              "additionalProperties": true
26          },
27          "states": {
28              "type": "object",
29              "properties": {
30                  "color": { "type": "string" },
31                  "label": { "type": "string" },
32                  "state": { "type": "string" },
33                  "x": { "type": "integer" },
34                  "y": { "type": "integer" }
35              },
36              "required": ["state"],
37              "additionalProperties": true
38          },
39          "includes": {
40              "type": "array",
41              "items": { "type": "string" }
42          },
43          "class": { "type": "string" },
44          "table": { "type": "string" },
45          "properties": { "type": "object" }
46          "virtualStates": { "type": "object" }
47      },
48      "required": ["actions", "states"]
49  }
```

Automat je tedy objekt, který má vlastnosti:

- *actions* – mapa akcí (jednotlivé klíče odpovídají identifikátoru akce). Akce je také objekt, pod klíčem *transitions* obsahuje jednotlivé přechody, které tuto akci implementují.
  - Přechod je opět objekt, pod klíčem *targets* musí mít uvedeny cílové stavy, zdrojový stav je uveden pod klíčem daného přechodu.
- *states* – mapa stavů (klíče opět odpovídají identifikátoru stavu)

Část definice automatu může být vygenerována pomocí externího editoru *yEd* (viz sekce 4.3.1) a uložena do souboru ve formátu *GraphML*. Tento soubor lze pak do definice vložit pomocí klíče *includes*.

Stav automatu se nejčastěji ukládá do relační databáze. Mapování jednotlivých vlastností automatu na sloupce tabulky v databázi probíhá automaticky, detekované názvy tabulky a sloupců lze upřesnit pomocí klíčů *table* a *properties*.

Pod klíčem *virtualStates* si editor ukládá informace o pomocných stavech – počátečním a koncovém.

GraphML

Smalldb v současné době podporuje načítání části konfigurace entity ze souboru ve formátu *GraphML* (viz sekce 4.3.1), který je odkázán z definice entity. Tento soubor lze pak upravit pomocí externího editoru, jako je například *yEd*. Smalldb editor by měl takový externí softwaru plně nahradit. Navíc *yEd* nedokáže rozlišit mezi akcí a přechodem, tudíž stále zůstává potřeba manuální úpravy JSON souboru s definicí automatu v textovém editoru.

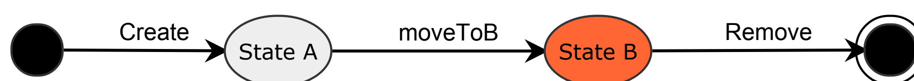
Ukázka entity

Následující ukázka entity automatu obsahuje dva stavy: *stateA* a *stateB*. Stav *B* má specifikovanou barvu, oba stavy mají specifikovaný popisek (*label*). Dále jsou definovány tři akce: *create*, *moveToB* a *remove*, každý s jedním přechodem. Přechod akce *create* vede z počátečního uzlu (značeno prázdným řetězcem "" na řádku 16), přechod akce *remove* vede do koncového stavu (značeno prázdným řetězcem "" na řádku 36). Akce *create* má dále definovaný výchozí popisek „*Create new entity*“ (řádek 22), který je ale u jejího přechodu přepsán hodnotou „*Create*“ (řádek 17).

Počáteční a koncový stav nejsou v poli *states* uvedeny. Jedná se o virtuální stavy, které nenesou žádnou informaci až na svoji polohu.

```
1  {
2      "states": {
3          "stateA": {
4              "state": "stateA",
5              "label": "State A"
6          },
7          "stateB": {
8              "state": "stateB",
9              "label": "State B",
10             "color": "#FF6633"
11         }
12     },
13     "actions": {
14         "create": {
15             "transitions": {
16                 "": {
17                     "label": "Create",
18                     "targets": ["stateA"]
19                 }
20             },
21             "heading": "New entity",
22             "label": "Create new entity"
23         },
24         "moveToB": {
25             "transitions": {
26                 "stateA": {
27                     "targets": ["stateB"]
28                 }
29             }
30         },
31         "remove": {
32             "transitions": {
33                 "stateB": {
34                     "label": "Remove",
35                     "targets": [""]
36                 }
37             },
38             "label": "Remove"
39         }
40     }
41 }
```

Na obrázku 6.1.2 je vidět vizualizace této entity pomocí Smalldb editoru. Počáteční a koncový stav jsou v příkladu reprezentovány pomocí prázdného řetězce "". Jsou reprezentovány jako dva unikátní stavy, ale ve skutečnosti se jedná pouze o jeden – reprezentovaná entita neexistuje. Proto jsou oba stavy ve schématu značeny jako prázdný řetězec. Do počátečního stavu nevede žádná hrana, a naopak z koncového stavu nevede žádná hrana – dle tohoto pravidla jsou při vizualizaci rozlišeny.



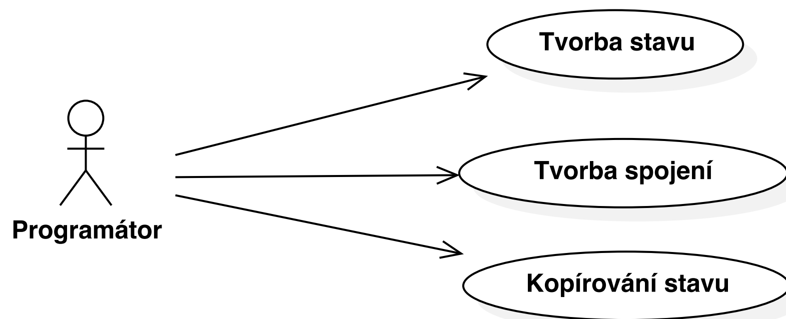
Obrázek 6.1.2: Vizualizace entity z předchozí ukázky

6.1.4 Uživatelské role

Jedinou rolí z pohledu Smalldb editoru je opět **programátor** definovaný v sekci 5.1.4. Programátor editor využívá k úpravě stavového automatu, kterou by jinak musel dělat ručně (pomocí textového editoru) nebo pomocí externího nástroje (například *yEd*, viz sekce 4.3.1).

6.1.5 Případy užití

Zde jsou popsány tři typické případy použití Smalldb editoru.



Obrázek 6.1.3: UML Use Case diagram Smalldb editoru

Tvorba stavu

1. Programátor otevře administraci webu se Smalldb editorem.
2. Kurzor myši přesune na vybrané místo na kreslícím plátně.
3. Dvojklikem vyvolá dialog nového přidání stavu.
4. Vyplní jméno nového stavu.

(a) Pokud již existuje stav se stejným jménem, bude vyzván k zadání nového unikátního.

Tvorba spojení (přechodu)

1. Programátor otevře administraci webu se Smalldb editorem.
   (a) Se stisknutým tlačítkem CTRL stiskne levé tlačítko myši nad zdrojovým stavem.
   (b) Se stisknutým tlačítkem myši přesune kurzor nad cílový stav a uvolní ho.
   (c) Vytvoří se přechod bez přiřazené akce.
   (d) Programátor v editovacím panelu vybere akci, ke které nově vytvořený přechod náleží.
      i. Stejným způsobem lze pro tento přechod vytvořit i novou akci.

Kopírování stavu

1. Programátor otevře administraci s editorem kaskády, ve kterém je načtený příslušný fragment.
2. Se stisknutým tlačítkem CTRL stiskne tlačítko myši nad kreslící plochou (mimo stavy).
3. Tahem označí vybrané stavy.
   (a) Tažením zleva doprava označí stavy, které spadají celé do provedeného výřezu.
   (b) Tažením zprava doleva označí i jen z části překrývající stavy.
4. Pomocí tlačítka v panelu nástrojů nebo klávesové zkratky *CTRL + C* zkopíruje příslušné stavy do schránky.
   (a) Označené stavy lze i vyjmout (*CTRL + X*).
5. V editoru otevře nový automat, do kterého chce vložit obsah schránky.
6. Pomocí tlačítka v panelu nástrojů nebo klávesové zkratky *CTRL + V* vloží stavy na střed plátna.

6.2 Návrh

6.2.1 Diagram tříd

Na obrázku 6.2.1 je znázorněn diagram tříd editoru stavových automatů Smalldb a jejich vzájemná kompozice. Pro názornost byly vypuštěny nepodstatné atributy. Diagram obsahuje všechny třídy, kromě vykreslovacích primitiv popsaných v sekci 5.2.2. Následuje stručný popis odpovědností jednotlivých tříd, které jsou specifické pro Smalldb editor.

SmalldbEditor (*Smalldb editor*) zapouzdřuje celý plugin. Stará se o načítání dat z elementu `<textarea>` a jejich zpětnou aktualizaci. Uchovává mapu aktuálně načtených stavů, akcí a jejich přechodů. Při inicializaci vytvoří ostatní objekty a spustí editor.

State (Stav) uchovává informace o stavu – název, popisek, barvu a volitelně i jeho souřadnice a další informace.

Action (Akce) uchovává informace o akci – název, popisek, barvu a volitelně i další informace.

Transition (Přechod) uchovává informace o jednotlivých přechodech – název, popisek, barvu a volitelně i další informace.

Editor (*Editovací panel*) slouží k úpravě vlastností stavu, akce i přechodu, podle toho co je v danou chvíli označené. Umožní editaci všech vlastností, i těch pro editor neznámých.

Tarjan (*Tarjanův algoritmus*) slouží k výpočtu silně souvislých komponent grafu, na základě kterých pak může být automat vykreslen. Pracuje s jednoduchou reprezentací grafu (třída *Graph* a třída *Node*) a zásobníku (třída *Stack*).

6.2.2 Shodné vlastnosti s editorem kaskády

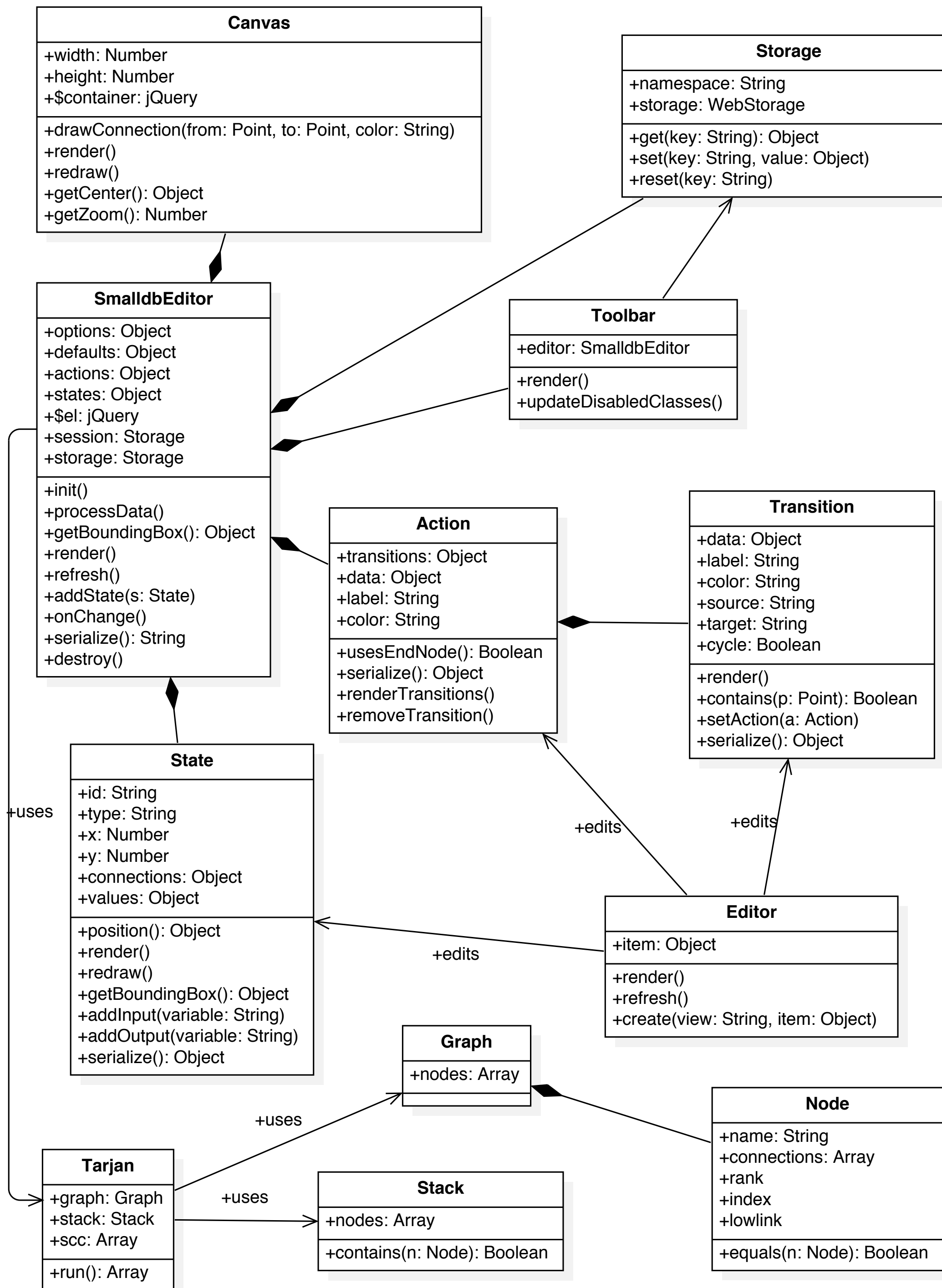
Velká část Smalldb editoru vychází z editoru kaskády. Kreslící plátno a pomocné geometrické objekty jsou shodné, liší se pouze podporou pro kreslení navzájem se vyhýbajících se přechodů a multihran (viz sekce 6.2.4). Další shodné části jsou podpora pro historii, přibližování, práci se schránkou, zobrazení na celou stránku nebo režim statického obrázku.

6.2.3 Uživatelské rozhraní

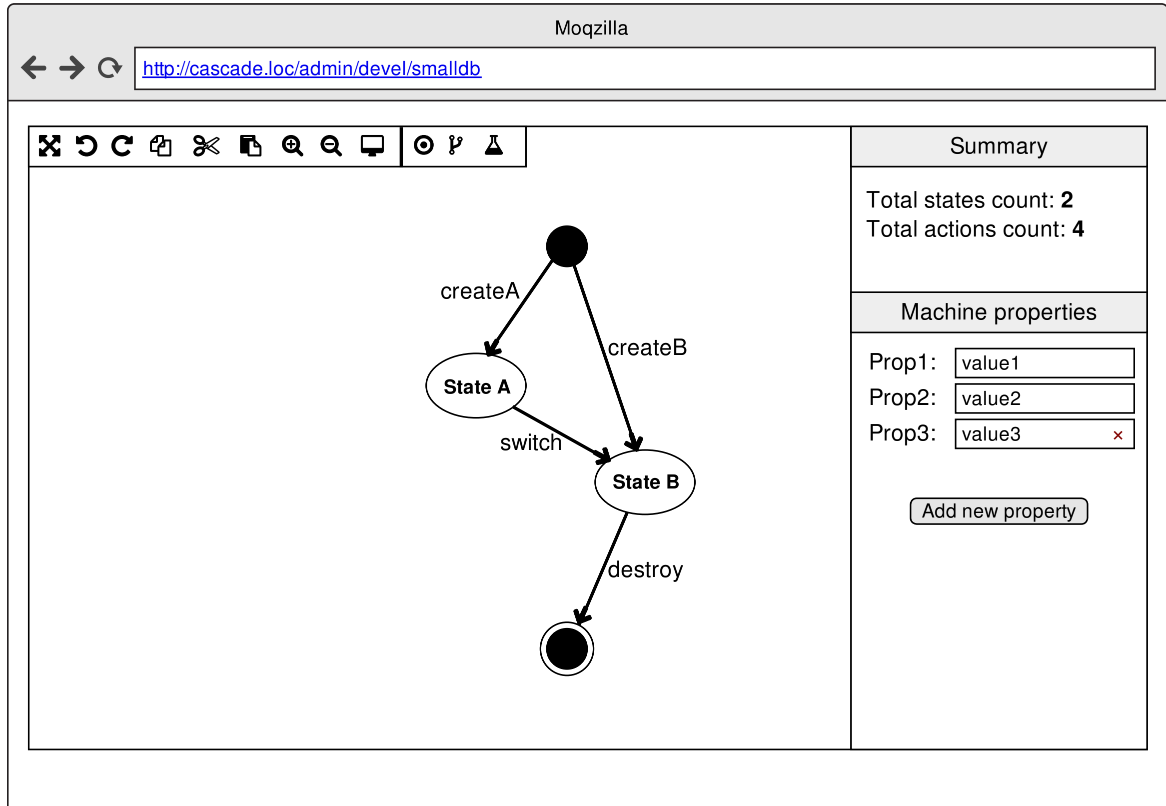
Uživatelské rozhraní Smalldb editoru vychází také z editoru kaskády (viz sekce 5.2.3). Na obrázku 6.2.2 je vidět jeho mockup vytvořený pomocí online nástroje *Moqups*. Kreslící plátno a panel nástrojů jsou shodné s editorem kaskády.

Stav je reprezentována pomocí elipsy, její výplň odpovídá barvě stavu (hodnotě jeho vlastnosti *color*), výchozí barva stavu je světle šedá (hexadecimálně `#EEEEEE`). Při dvojkliku na stav bude vyvolán kontextový dialog pro úpravu popisku stavu. Stav má vedle popisku i svůj unikátní identifikátor, ten se zobrazuje pouze v editovacím panelu pod klíčem *name*. Popisky být unikátní nemusí.

Přechod mezi stavy lze vytvořit pouze pomocí metody drag'n'drop tažením kurzoru se stisknutým tlačítkem CTRL ze zdrojového stavu do cílového. Po najetí kurzorem nad stav se zobrazí tlačítko pro jeho smazání z upravovaného automatu. Stav lze mazat



Obrázek 6.2.1: Diagram tříd Smalldb editoru



Obrázek 6.2.2: Mockup Smalldb editoru – plátno s paletou a panelem nástrojů

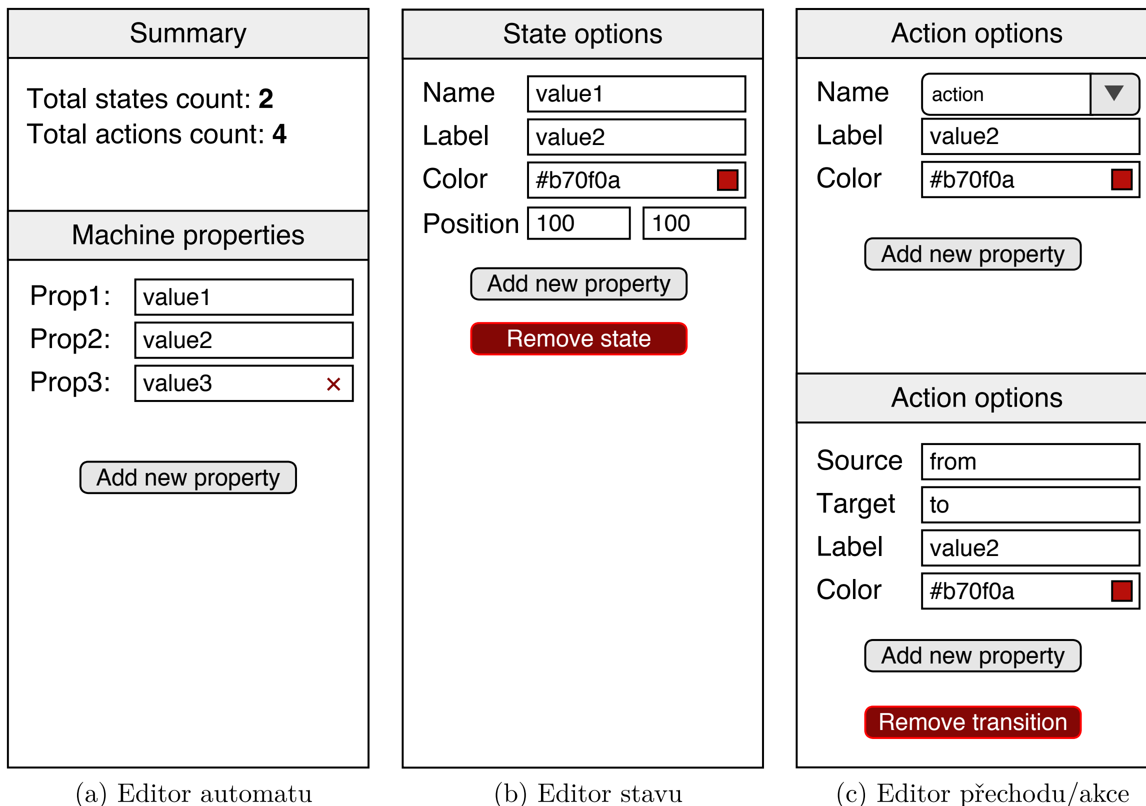
i pomocí tlačítka v editovacím panelu. Stavy lze opět hromadně označit a pohybovat s nimi po kreslícím plátně.

Editovací panel se zobrazuje vždy na pravé části kreslícího plátna, které překrývá. Je vykreslen s částečnou průhledností, aby zpřístupnil větší část kreslící plochy a tím usnadnil orientaci ve větších automatech. Při najetí kurzoru nad panel se jeho průhlednost dočasně potlačí. Editovací panel má tři režimy zobrazení, jejich porovnání je vidět na obrázku 6.2.3.

První režim je zobrazuje souhrnné informace o automatu a jeho dynamické vlastnosti. Pomocí tlačítka lze pak přidat novou vlastnost. Při najetí kurzorem nad pole s hodnotou vlastnosti se objeví tlačítko pro smazání dané vlastnosti. Tento režim je výchozí, je zobrazen vždy po kliknutí na kreslící plátno.

Druhý režim slouží pro správu stavu. Vedle dynamických vlastností zobrazuje svůj identifikátor, popisek, barvu a pozici. Tyto systémové vlastnosti nelze smazat, pouze upravit jejich hodnotu. U pole s nastavením barvy je vybraná barva vizualizována a po kliknutí na tuto vizualizaci je přístupný dialog pro její výběr.

Třetí režim zobrazuje v dolní části vlastnosti přechodu, který je právě aktivní, a v horní části i vlastnosti akce, ke které tento přechod náleží. Přechod i akce mají opět systémové vlastnosti – popisek, barvu, název akce, a zdrojový a cílový stav přechodu. Akce i přechod může mít také své vlastní dynamické vlastnosti.

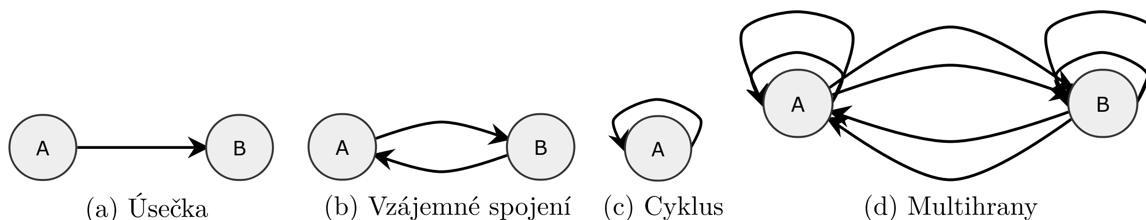


(a) Editor automatu (b) Editor stavu (c) Editor přechodu/akce

Obrázek 6.2.3: Mockup Smalldb editoru – tři režimy zobrazení editovacího panelu

6.2.4 Kreslení spojnic

Spojnice mezi jednotlivými stavy jsou vykreslovány z okrajových bodů stavu, který je vizualizován jako elipsa. Výpočtem bodů na elipse se zabývá sekce 6.3.4. Mohou nastat čtyři případy (viz obrázek 6.2.4):



(a) Úsečka (b) Vzájemné spojení (c) Cyklus (d) Multihrany

Obrázek 6.2.4: Kreslení přechodů mezi dvěma stavy

Úsečka

Pokud existuje mezi dvěma stavy pouze (jedno) jednosměrné spojení, je vykresleno jako úsečka.

Vzájemné spojení

Je-li spojení obousměrné, je nutné křivky vychýlit tak, aby se navzájem nepřekrývali. Spojení je vykresleno pomocí *Spline* křivky (viz sekce 5.2.4), která prochází jedním pomocným bodem.

Cyklus

Při kreslení cyklů se počáteční bod bere vždy z levé strany a koncový z pravé strany stavu. Spojení je opět vykresleno pomocí *Spline* křivky, tentokrát procházející čtyřmi pomocnými body.

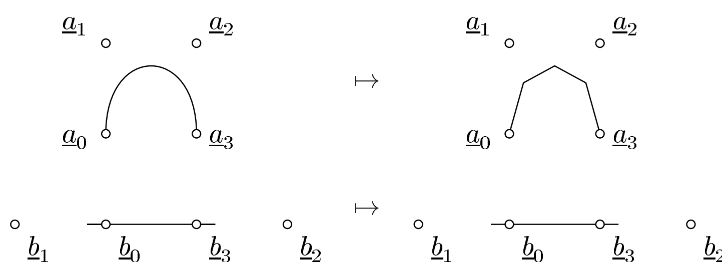
Multihrany

Na posledním obrázku (6.2.4d) je vidět příklad vícenásobného spojení mezi dvěma stavy. Multihrany se mohou objevit jak u klasických přechodů, tak u cyklů.

6.2.5 Výpočet bodu na spojnici

Ve Smalldb editoru je nutné vedle označování stavů označovat také jednotlivé přechody (hrany). Přechody jsou vždy kresleny pomocí *Spline* křivky (při kreslení jednoduchého spojení není využit žádný pomocný bod a *Spline* křivka se vykreslí jako přímka). Při kliknutí do kreslícího plátna je tedy nutné nějakým způsobem ověřit, zda na daném místě není vykreslen nějaký přechod.

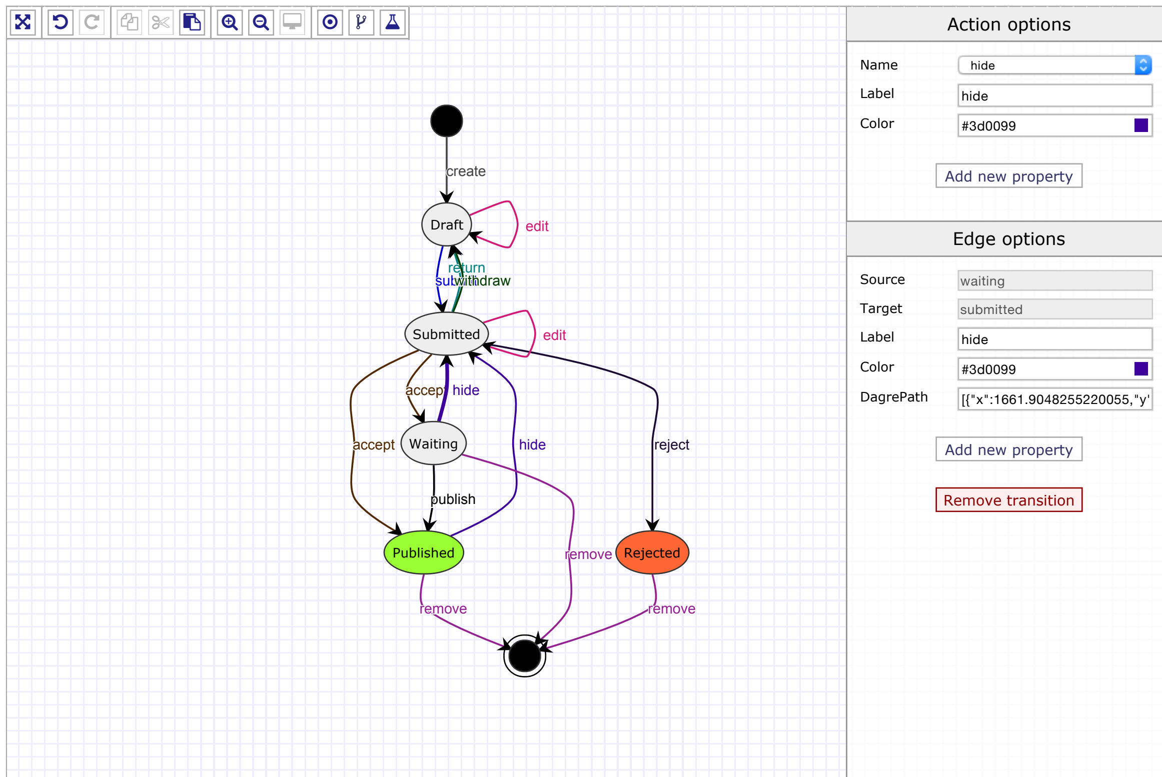
Vzhledem k malému počtu kontrolních bodů, které se při kreslení spojnic u Smalldb editoru využívají (jeden pro vzájemná spojení a čtyři pro cyklus), se dá křivka snadno proložit malým počtem přímek. Pro aproximaci *Spline* křivky se využívá zjednodušená varianta algoritmu *de Casteljau* [29], který původně sloužil pro kreslení Bézierovi křivky (rozkládá ji na přímky dokud není hladká). Pro naše potřeby je dostatečný jeden průchod algoritmu. Grafické znázornění běhu algoritmu je vidět na obrázku 6.2.5.



Obrázek 6.2.5: Aproximace Bézierovi křivky – vizualizace algoritmu *de Casteljau* (zdroj: [29])

6.3 Implementace

Na obrázků 6.3.1 je vidět výsledná podoba Smalldb editoru stavových automatů. Velká část implementace vychází z editoru kaskády (viz sekce 5.3 a 6.2.2). Tato sekce shrnuje implementační detaily, které souvisí pouze s editorem stavových automatů.



Obrázek 6.3.1: Výsledná podoba Smalldb editoru stavových automatů

6.3.1 Inicializace a konfigurace

Výsledný editor byl opět realizován jako jQuery plugin, který se registruje do funkce: `$.smalldbEditor(options)`. Volá se nad elementem `<textarea>` a v parametru přebírá konfiguraci v objektu `options`, která přepisuje výchozí hodnoty uvedené v tabulce 6.1. Takto uvedenou konfiguraci lze opět přepsat pomocí *data atributu* přímo na elementu (viz sekce 5.3.1).

6.3.2 Živá editace

Na rozdíl od editoru kaskády, kde editace probíhala pomocí modálního okna, u Smalldb editoru je v kreslícím plátně vidět jen minimum informací. Proto byl zvolen jiný přístup – živá editace pomocí bočního panelu. Ten je vidět vždy a pouze se mění jeho kontext na základě toho, jaký element (stav, přechod) je zrovna označený. Ve výchozím stavu je zobrazen stručný souhrn informací o automatu a dále jeho dynamické vlastnosti. Není

Klíč	Popis	Výchozí hodnota
`edgeClickOffset`	Tolerance při klikání na přechody	5 [px]
`historyLimit`	Maximum stavů v historii	1000
`splineTension`	Parametr zakřivení spline křivky	0.3
`canvasOffset`	Posunutí při kreslení	30 [px]
`canvasExtraWidth`	Přidaná šířka na levé a pravé straně plátna	1500 [px]
`canvasExtraHeight`	Přidaná výška na horní a dolní straně plátna	1500 [px]
`canvasSpeed`	Rychlost scrollování po plátně	2
`viewOnly`	Režim statického obrázku (viz 5.3.10)	false
`scrollLeft`	Iniciální horizontální posunutí kreslícího plátna	0 [px]
`scrollTop`	Iniciální vertikální posunutí kreslícího plátna	0 [px]

Tabulka 6.1: Konfigurace editoru stavových automatů

nutné žádné tlačítko na ukládání, změny jsou instantní. Vzhledem k implementované historii je tento přístup bezpečný, uživatel se kdykoliv může vrátit zpět.

Některé vlastnosti stavu, přechodu a akce, zejména popisek a barva, se při úpravě okamžitě promítnou do kreslícího plátna. Při živé editaci tedy bylo nutné selektivně překreslovat části plátna a zároveň aktualizovat editovací panel. Při úpravě akce je totiž nutné zároveň upravit i právě aktivní přechod.

6.3.3 Vlastnosti akce a přechodu

Ve Smalldb automatu jsou jednotlivé hrany spojující stavy reprezentovány pomocí přechodů, kde každý přechod náleží nějaké akci. Akci si tedy můžeme představit v podstatě jako šablonu pro všechny její přechody. Říká nám jaké výchozí hodnoty bude její přechod mít.

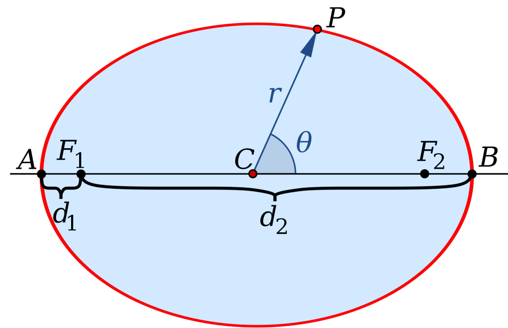
Při inicializaci editoru se pro zjednodušení všechny hodnoty z akce propagují přímo k jednotlivým přechodům. Při editaci akce se pak kontrolují všechny její přechody – pokud mají stejnou hodnotu jako je ta výchozí (uvedená u akce), přepíše se novou hodnotou.

6.3.4 Výpočet bodu na elipse

Spojnice mezi stavy se vykreslují z okrajových bodů stavu. Stav je vizualizován jako elipsa (viz obrázek 6.3.2), při kreslení tedy nejprve potřebujeme nalézt tyto body. Při hledání cesty je počáteční a koncový bod reprezentován středem elipsy (bod C). Pomocí funkce `atan2(y,x)` (viz sekce 5.3.3) zjistíme úhel θ mezi průvodičem r spojujícím tyto body a osou x. Nakonec středové body v cestě nahradíme za okrajové.

6.3.5 Obarvování hran

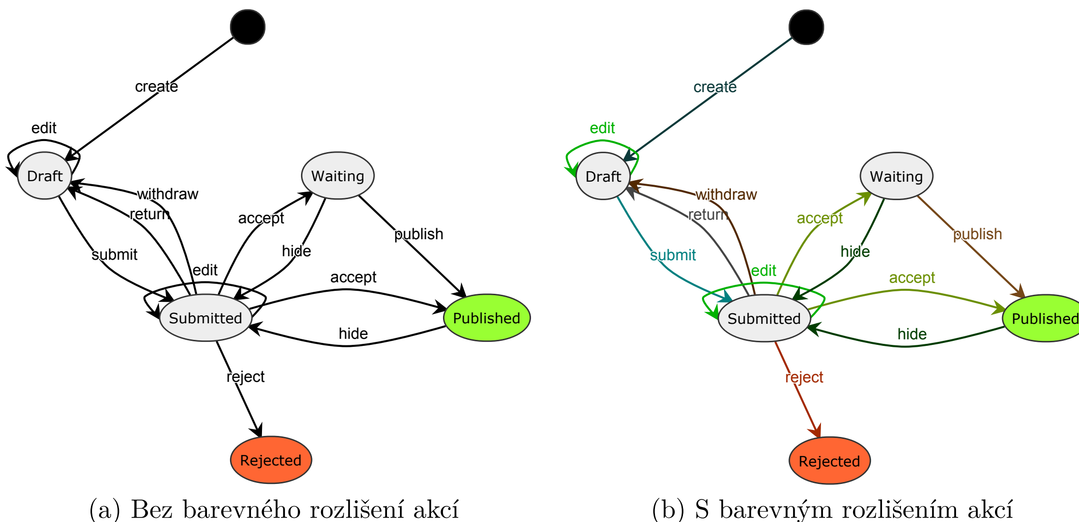
Se zvětšujícím se počtem přechodů začíná být orientace v automatu těžší a těžší. Pro snazší orientaci je pomocí vlastního tlačítka v panelu nástrojů umožněno automaticky obarvit všechny hrany pomocí předem vybrané škály barev. Obarvování funguje na základě akcí – pro každou akci jedna unikátní barva (dokud není počet barev vyčerpán,



Obrázek 6.3.2: Výpočet bodu na elipse (zdroj: [30])

ostatním akcím zůstane výchozí černá barva). Na obrázku 6.3.3 je vidět porovnání automatu reprezentující článek v redakčním systému *s* a *bez* barevného rozlišení akcí.

Pro manuální výběr barvy přechodu, akce či stavu byl využit nativní formulářový HTML5 element `<input type="color">`. Ten v současné době ale nelze dostatečně vizuálně přizpůsobit[1], bylo tedy nutné vykreslit ho průhledně a využít ho pouze pro vyvolání palety barev. Pod samotný element je pak vložen další element `<div>`, který zobrazuje aktuálně vybranou barvu.



(a) Bez barevného rozlišení akcí (b) S barevným rozlišením akcí

Obrázek 6.3.3: Automat reprezentující článek v redakčním systému – Smalldb editor

6.4 Automatické rozmístění stavů

Entita automatu ve své minimální konfiguraci (viz sekce 6.1.3) neuchovává žádnou informaci o poloze stavů. Pro první vykreslení automatu je proto důležité stavy automaticky rozmístit, jinak by se vykreslili všechny na stejné místo. Tato akce je pak

[1]Element `<input type="color">` se v různých prohlížečích vykresluje různě, problém je hlavně u jeho okraje, který nelze vždy potlačit.

kdykoliv přístupná pomocí tlačítka v panelu nástrojů. Uživatel se tedy může zaměřit pouze na přidávání stavů a přechodů, a posléze si nechat si automat znovu přehledně uspořádat.

Existuje mnoho způsobů jak kreslit orientované grafy [31, 32], následující sekce popisují dvě varianty vhodné pro použití u orientovaných grafů, které byly implementovány.

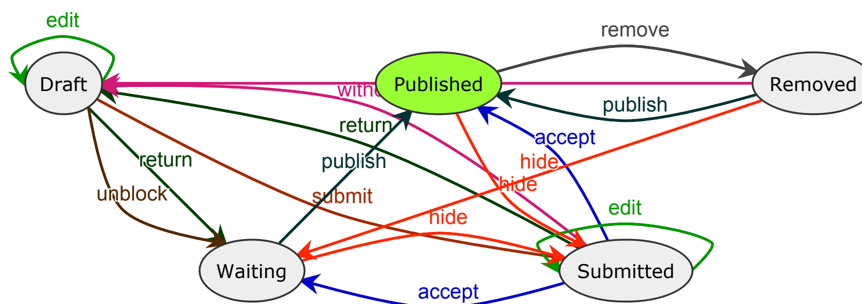
6.4.1 Silně souvislé komponenty

První – naivní – varianta využívá Tarjanův algoritmus [33] hledání silně souvislých komponent. Algoritmus jednotlivé komponenty a stavy v nich seřadí do topologického uspořádání. Stačí tedy jednoduše vypsat postupně všechny komponenty pod sebe.

Řazení uzlů v rámci komponenty

V rámci komponenty je vhodné stavy seřadit tak, aby se nejprve vykreslily ty, které mají největší počet spojení (dále jen *rank*). Jednotlivé stavy se vykreslují zleva doprava, střídavě jednou nahoře a jednou dole. Pro výpočet ranku je důležité ignorovat multihrany a cykly, dále je vypuštěna orientace hran. Zároveň jsou ignorovány přechody vedoucí mimo komponentu nebo z jiné komponenty.

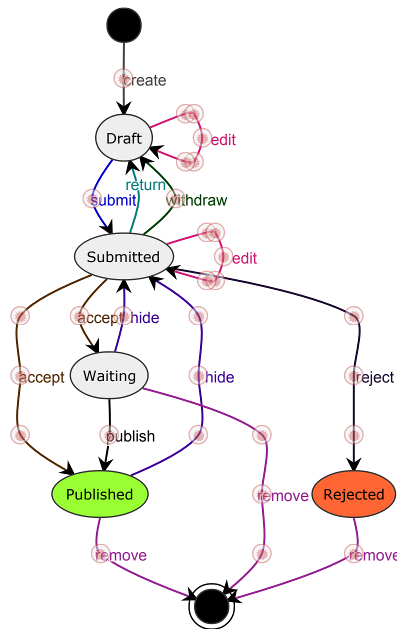
Tato jednoduchá heuristika je sice velmi rychlá, ale výsledné uspořádání stavů není při větším počtu přechodů moc přehledné. Neřeší vůbec navzájem křížící se hrany ani minimalizaci délky hran. Příklad nepřehledného uspořádání stavů v rámci komponenty pomocí tohoto algoritmu je vidět na obrázku 6.4.1. Vedle křížících se hran je na tomto obrázku vidět druhý problém – hrana procházející skrze jiný stav.



Obrázek 6.4.1: Křížení hran při špatně vypočítaném řazení

6.4.2 Algoritmus knihovny Dagre

Knihovna *Dagre* (viz sekce 4.2.7) používá pro výpočet rozložení uzlů a kontrolních bodů přechodů techniku zvanou *Layered graph drawing*, tedy kreslení grafů pomocí vrstev. Stejný způsob používá i program dot, který pro kreslení orientovaných grafů využívá knihovna *Graphviz* (viz sekce 4.2.1). Výsledné rozložení stavů pomocí knihovny *Dagre* je vidět na obrázku 6.4.2, kde jsou zároveň vykresleny pomocná tlačítka reprezentující kontrolní body jednotlivých přechodů.



Obrázek 6.4.2: Automat vykreslený pomocí knihovny *Dagre* se zobrazenými kontrolními body křivek

Kreslení grafu pomocí vrstev

Při kreslení grafu pomocí vrstev se jednotlivé uzly grafu rozdělí do horizontálních vrstev (řádků) tak, aby směr většiny hran byl směrem dolů [31, 32]. V ideálním případě by měl být vstupní graf planární (rovinný), tedy mělo by pro něj existovat rovinné zobrazení, ve kterém se žádné dvě hrany nekříží. Grafy automatů, které popisují reálné situace, ale často takové zobrazení nemají. Takový graf je proto nutné planarizovat. Při tomto procesu se vykreslí graf libovolnou metodou (například pomocí algoritmu BFS[2]), a do míst, kde se kříží hrany, se přidají nové pomocné uzly. Tím se křížící se cesty rozdělí na menší úseky bez kolizí. Při kreslení výsledného grafu jsou pak pomocné stavy nahrazeny kontrolními body spojnic, které se původně křížily [31, 34]. Planarizace grafu je ale NP těžký problém, pro větší grafy je proto nutné spolehnout se na sadu heuristik.

Fáze algoritmu

Kostra algoritmu knihovny Dagre vychází z článku *A Technique for Drawing Directed Graphs* [32], který popisuje čtyř fázový algoritmus využívaný v programu `dot`:

1. Hledání optimálního ranku jednotlivých uzlů pomocí síťové simplexové metody.

2. Řazení uzlů v rámci ranků tak, aby došlo k minimalizaci křížení hran.

3. Hledání optimálních souřadnic uzlů na základě vypočítaného ranku.

4. Výpočet kontrolních bodů spojnic.

[2]BFS: Breadth-first search (Prohledávání do šířky)

Pro první krok algoritmu využívá shodné techniky jako v tomto článku. Pro minimalizaci křížení hran využívá algoritmus popsaný v článku *2-Layer Straightline Crossing Minimization* [35], který se také zabývá srovnáním výkonnosti různých heuristik a exaktních algoritmů pro tento problém.

Algoritmus pro počítání kolizí hran mezi jednotlivými vrstvami vychází z článku *Simple and Efficient Bilayer Cross Counting* [36].

Pro třetí fázi algoritmu (přiřazování souřadnic jednotlivým uzlům) využívá vlastní algoritmus, který vychází z článku *Fast and Simple Horizontal Coordinate Assignment* [37].

Problémy při integraci

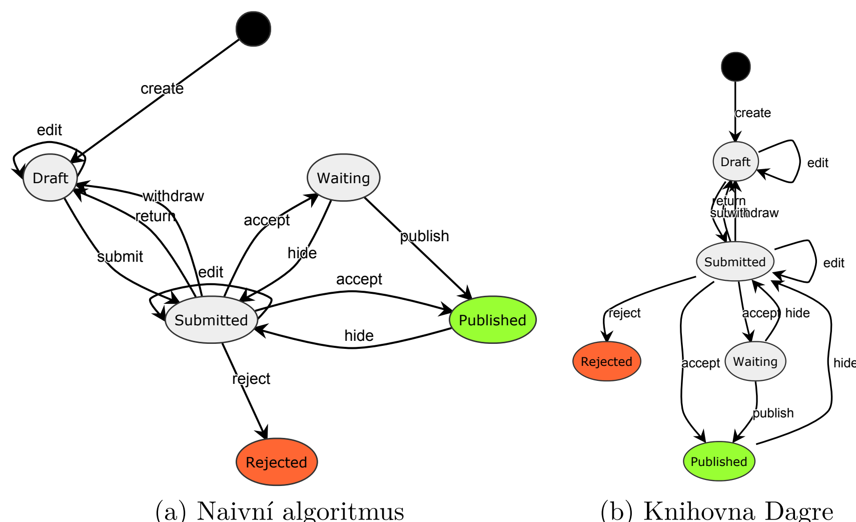
Při integraci tohoto řešení do editoru vyvstaly dva problémy:

Multihrany První problém je, že *Dagre* při výpočtu zahazuje multihrany. Je to vidět i na obrázku 6.4.3b u dvou zpětných přechodů ze stavu *Submitted* do stavu *Draft*. Cestu jednoho z nich *Dagre* nevrátí a tak je nutné ji dopočítat pomocí původního algoritmu. Pro řešení tohoto problému byla implementována možnost správy kontrolních bodů křivky. Uživatel si tak může jednotlivé kontrolní body cesty přemístit, smazat nebo přidat nové, a to nezávisle na tom, co *Dagre* vypočítá. Automat s opravenou multihranou a zobrazenými kontrolními body je vidět na obrázku 6.4.2.

Interaktivní změna pozice stavu Druhým problémem je vypočítaná cesta jednotlivých přechodů. Při přesunu stavu se dynamicky upraví pozice posledního bodu cesty, ostatní body zůstanou na původním vypočítaném místě. Při větším vychýlení stavu z původní pozice začne spojnice vypadat nepřehledně. Řešením této situace bylo opět přidání možnosti smazat nebo přemístit jednotlivé vypočítané body cesty.

6.4.3 Srovnání a výběr řešení

Na obrázku 6.4.3 je vidět srovnání výsledného rozložení stavů pomocí obou implementovaných algoritmů. Na takto malých grafech jsou výsledky obou způsobů přijatelné. Dle měření (viz sekce 6.7.2) se algoritmus knihovny *Dagre* dá použít i při vyšším počtu stavů. Jedná se o poměrně velkou knihovnu (84 KB), její přímé začlenění do aplikace tedy není vhodné. Místo toho je tato knihovna využita jako volitelná závislost. Editor kontroluje její přítomnost, a pokud ji najde, nabídne ji k použití.



(a) Naivní algoritmus (b) Knihovna Dagre

Obrázek 6.4.3: Srovnání algoritmů pro kreslení automatu

6.5 Možná budoucí rozšíření

V této sekci jsou popsány možná budoucí rozšíření, která jsou nad rámec této práce.

6.5.1 Oprávnění

Jednotlivé přechody v automatu mohou být podmíněné nějakou uživatelskou rolí nebo oprávněním. Editor by mohl na základě vybrané role zvýraznit možné cesty, kterými se daná role může vydat, a dosažitelnost jednotlivých stavů.

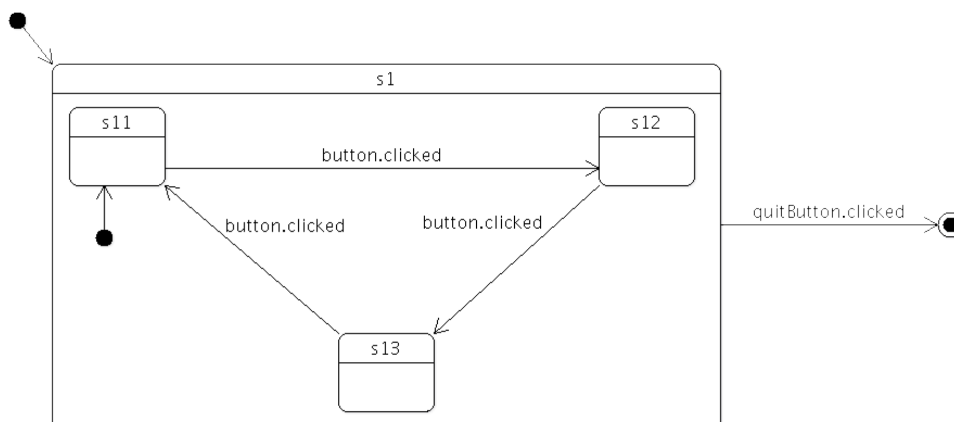
6.5.2 Hierarchické automaty

U stavových automatů se často vyskytuje rozšíření, které Smalldb neumí – jedná se o hierarchické automaty. Hierarchický automat si můžeme představit jako vnořený automat, který se v nějakém větším celku chová jako prostý stav. Příklad hierarchického automatu z knihovny *Qt* [38] je vidět na obrázku 6.5.1. Stav *s1* v sobě skrývá malý vnořený automat sestávající ze stavů *s11*, *s12* a *s13*. Tento vnořený automat má vlastní počáteční stav (*s11*) a vlastní přechody. Výstup z tohoto vnořeného automatu je nezávislý na jeho stavu.

Pro implementaci hierarchických automatů do Smalldb editoru by bylo nutné implementovat práci se skupinou stavů reprezentující vnořený automat. Ten by byl definován ve vlastním souboru, na který by nadřazený automat odkazoval. Editor by pak u vnořeného stavu nabízel jeho rozbalení, který by vnořený automat vykreslilo podobně jako na obrázku 6.5.1.

6.5.3 Obecný editor grafů

Automat, se kterým Smalldb editor pracuje (viz sekce 6.1.3), odpovídá až na množinu akcí klasickému nedeterministickému stavovému automatu. Mohl by tedy sloužit jako



Obrázek 6.5.1: Příklad hierarchického automatu (zdroj: [38])

základ pro obecný editor grafů a automatů, případně UML nebo BPMN diagramů. Jádro aplikace, které se stará o vykreslování stavů a přechodů by zůstalo v podstatě nezměněné. Takovýto obecný editor by pak mohl najít uplatnění například jako výuková pomůcka. Zároveň by mohl sloužit pro modelování business procesů, jejichž některé části by mohly být reprezentovány pomocí Smalldb automatů. Díky tomu by šlo snadno formálně ověřit správnost modelu nebo alespoň jeho části (viz sekce 4.4.1).

6.5.4 Minimalizace automatu

Ve světě klasických konečných automatů často chceme minimalizovat počet stavů automatu při zachování jazyka, který přijímá. Smalldb automaty ale popisují business logiku, minimalizace zde proto není žádoucí. Stavy automatu musí přesně odpovídat stavům business procesu, který modeluje. Minimalizace automatu (respektive kondenzace grafu) by ale mohla mít své místo v případném obecném editoru grafů (viz sekce 6.5.3).

6.5.5 Desktopová verze

Editor by šlo snadno upravit na desktopovou aplikaci, například pomocí frameworku AppJS [39]. Díky němu lze využít hotový JavaScriptový kód včetně CSS stylů a s minimálním úsilím vytvořit desktopovou aplikaci pro všechny tři hlavní platformy (Windows, Linux, OS X). Využívá Chromium a Node.js.

Výhodou desktopové verze by byla 100% kompatibilita s HTML5 a CSS3. Také by byl pravděpodobně zvýšen výpočetní výkon a namísto integrace do webové aplikace by mohla pracovat nad lokálními soubory.

6.6 Testování

6.6.1 Heuristická evaluace

Tato sekce popisuje problémy, které u editoru stavových automatů Smalldb odhalila heuristická evaluace na základě Nielsenových heuristik definovaných v příloze A. Vzhle-

dem ke společné implementaci některých částí obou editorů se zde objevili shodné problémy jako u editoru kaskády (viz sekce 5.5.1), zejména **vkládání stavů ze schránky** (A.1), **podobnost práce se stavy a ikonami na ploše** (A.2) a **chybějící dokumentace klávesových zkratek** (A.10). Následují další problémy specifické pro editor stavových automatů.

Potvrzení při úpravě metadat (A.1) Při editaci metadat v editovacím panelu se každá změna ukládá automaticky. Uživatel na to není nijak upozorněn, pouze u barvy a popisku se jejich změna promítne rovnou na kreslícím plátně. V aktuální verzi editoru je nyní při každém uložení na krátkou dobu podbarveno textové políčko zeleně.

Kontrolní body přechodů (A.3) Přechody mezi stavy jsou vykreslovány jako *Spline* křivky, procházející vypočítanou sadou bodů. Uživatel si v původní implementaci nemohl cestu křivky přizpůsobit.

Přechody do počátečního stavu a z koncového stavu (A.5) Smalldb zakazuje přechody do počátečního stavu, respektive přechody z koncového stavu. Editor v původní implementaci toto pravidlo nerespektoval. Nyní jsou při takových přechodech stavy zvýrazněny červeně a při pokusu takové spojení vytvořit se upraví jeho zdrojový (resp. cílový) bod tak, aby bylo spojení platné (např. spojení směrem do počátečního stavu bude přesměrováno do koncového).

6.6.2 Uživatelské testování

Uživatelské testování editoru automatů proběhlo na stejných participantech jako testování editoru kaskády (viz sekce 5.5.2). Před samotným testem byli participanti stručně uvedeni do problematiky stavových automatů Smalldb. Druhý participant opět dostal k otestování novou verzi editoru, opravenou na základě poznatků z testování s prvním participantem (včetně poznatků z testování editoru kaskády).

Testovací scénář

Pro testování byl využit následující scénář. Zaměřuje se na implementační rozdíly oproti editoru kaskády.

Na počítači je otevřený prohlížeč a v něm editor stavových automatů s načtenou entitou článku.

1. Vytvořte nový stav.
   (a) Nastavte stavu nějaké jméno a barvu.
   (b) Přidejte mu novou vlastnost a nastavte ji hodnotu.
2. Vytvořte přechod do tohoto stavu z libovolného jiného stavu.
   (a) Nastavte akci a parametry k novému spojení.
   (b) Přidejte druhé spojení opačným směrem, přiřaďte ho k jiné akci.

(c) Přidejte třetí spojení – cyklus nad novým stavem, pro tento přechod vytvořte novou akci jménem „cyklus“.

3. Označte nově vytvořený stav a přesuňte ho na jiné místo.

4. Označte libovolnou hranu, změňte ji popisek a upravte ji libovolnou další vlastnost.

5. Upravte cestu libovolného přechodu.

Výsledky – participant 1

První problém byl hned u prvního úkolu, participant by hledal tlačítko pro přidání stavu v panelu nástrojů. V původní verzi chyběla nápověda, byla nutná rada od moderátora. Při změně barvy označeného stavu se kvůli jeho špatnému zvýraznění změna neprojevila. V editovacím panelu hledal tlačítko na uložení nebo alespoň nějakou zpětnou vazbu, např. při stisku *Enter* by čekal přechod na další textové pole. Pro vytvoření nové akce také hledal vlastní tlačítko.

Při přesouvání stavů čekal změnu kurzoru, přechody se snažil vytvářet tahem od hrany stavu – tam se ale nedala moc dobře chytit a omylem pak posouval kreslící plátno (k tomu v původní verzi stačilo pouze pohyb myši se stisknutým levým tlačítkem, v upravené verzi je nutná klávesa *Ctrl*). Při označování hrany klikal na popisek, za ten ale původně přechod označit nešel. Také by čekal tučný popisek přechodu při jeho označení.

Výsledky – participant 2

Druhý participant měl během testu dva drobné problémy. Opět se jednalo o vkládání nového stavu – tentokrát už ale byla v panelu nástrojů možnost zobrazit krátkou nápovědu, ve které byla tato funkcionalita popsána. Při kliku pravým tlačítkem na kreslící plátno by čekal kontextové menu, kde bude tato možnost také. Druhý problém byla implementační chyba, kdy editovací panel povolil nastavení neplatného identifikátoru stavu (řetězec s mezerou) a kvůli tomu nebylo možné vytvořit přechod do tohoto stavu.

Výsledky – souhrn

Na základě uživatelského testování byly provedeny tyto změny:

- Přidána stručná uživatelská nápověda.
- Pohyb po plátně se stisknutou klávesou *Ctrl*, bez ní označování stavů.
- Tvorba přechodů bez nutnosti držet klávesu *Ctrl*, přesun stavu až po označení.
- Zpětná vazba při uložení nové hodnoty v editovacím panelu.
- Změna kurzoru při najetí nad stav.

6.7 Experimenty

6.7.1 Konfigurace testovacího prostředí

Konfigurace testovacího prostředí je shodná se sekcí 5.6.1.

6.7.2 Srovnání rychlosti implementovaných algoritmů

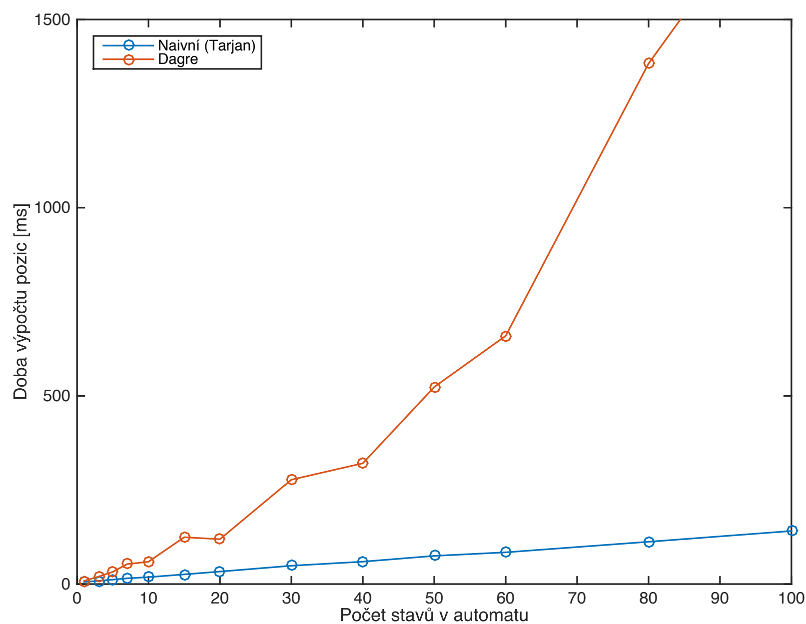
Tabulka 6.2 a graf 6.7.1 znázorňují srovnání rychlosti obou implementovaných algoritmů pro automaty do sta stavů, resp. dvou set hran. Naměřené hodnoty opět odpovídají průměru ze sta měření. Na grafu je vidět, že naivní řešení pomocí Tarjanova algoritmu má lineární složitost, zatímco algoritmus knihovny *Dagre* odpovídá zhruba kvadratické složitosti.

Počet stavů	Počet hran	Naivní [ms]	Dagre [ms]
1	3	5,4765	8,5623
3	6	8,4602	19,7063
5	10	11,6812	32,3439
7	15	15,2052	52,9208
10	20	18,3713	59,3401
15	30	25,7026	124,4174
20	40	33,1991	119,1901
30	60	48,7044	277,2577
40	80	59,5027	320,8411
50	100	74,8784	523,5911
60	120	84,2978	659,4621
80	160	112,3632	1382,9429
100	200	140,9947	1908,1467

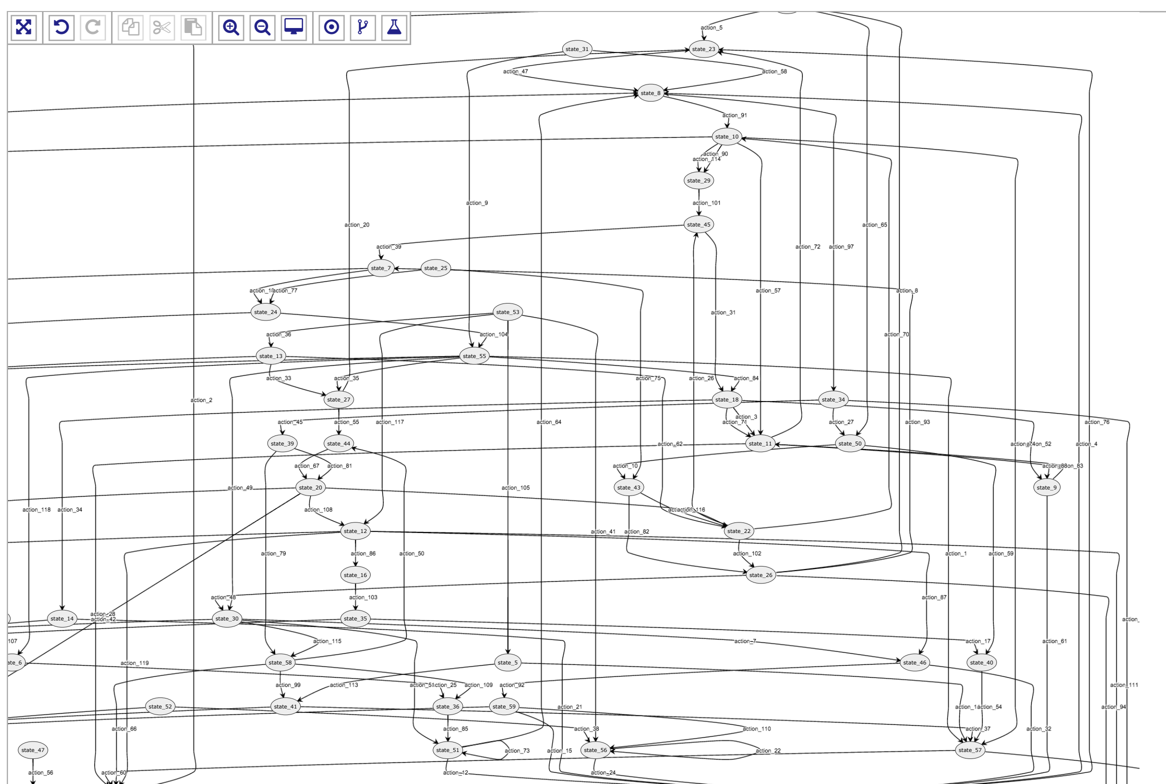
Tabulka 6.2: Srovnání rychlosti algoritmů pro kreslení automatu

6.7.3 Maximální velikost grafu

Hranice 100ms pro rychlost odezvy při interaktivních akcích, stanovená v nefunkčních požadavcích, je překročena až při více než šedesáti stavech. Na obrázku 6.7.2 je vidět takový automat s rozložením stavů vypočítaným pomocí knihovny *Dagre*.



Obrázek 6.7.1: Graf závislosti doby výpočtu na počtu stavů



Obrázek 6.7.2: Automat o šedesáti stavech a sto dvaceti přechodech

Kapitola 7

Závěr

Cílem této práce bylo navrhnout, implementovat a otestovat dva grafické editory – editor kaskády bloků (viz kapitola 5) a editor stavových automatů Smalldb (viz kapitola 6). Oba editory byly provedeny jako JavaScriptové pluginy pro framework jQuery, které pracují nad daty v elementu `<textarea>`.

Návrh uživatelského rozhraní editoru kaskády vychází z existujícího funkčního prototypu (viz sekce 4.1.1). Editor byl rozšířen o implementaci historie, práce se schránkou a označování více bloků najednou. Další důležitou funkcí je oddálení celého kreslícího plátna editoru a možnost zobrazit tak celý fragment kaskády, i když sestává z většího počtu bloků. Editor byl otestován pomocí heuristické evaluace a následného uživatelského testování. Obě tyto metody ukázaly na několik závažných nedostatků v použitelnosti (především nutnost použití klávesových zkratek, na které uživatel nebyl nikde upozorněn).

Byly implementovány tři různé algoritmy pro hledání cesty spojnic mezi bloky, která se vyhýbá kolizím s ostatními bloky. Tyto algoritmy byly srovnány jak po vizuální stránce, tak v rámci experimentů po stránce rychlostní. Jako jediný použitelný způsob pro hledání cesty se pro středně velké fragmenty kaskády (20-50 bloků) ukázal pouze jeden – hledání nejkratší cesty v gridu pomocí algoritmu A*.

Editor stavových automatů Smalldb z velké části vychází z implementace editoru kaskády. Staví na stejném základu kreslící plochy, na kterou vykresluje jednotlivé stavy a přechody mezi nimi. Entita automatu ve své minimální konfiguraci neudržuje informace o umístění stavu na plátně, před vykreslením je proto nutné stavy bez pozice umístit.

Přehledným vykreslením automatu se zabývá sekce 6.4. Popisuje dva algoritmy, jeden vykreslující automat po silně souvislých komponentách, a druhý využívající externí knihovnu *Dagre*, která pro kreslení grafu využívá planarizace a následné kreslení po vrstvách. Oba algoritmy byly opět vizuálně i experimentálně srovnány. Editor stavových automatů byl otestován stejně jako editor kaskády, i zde se objevily podobné nedostatky v použitelnosti.

Oba editory se při experimentech ukázaly jako dobře použitelné i pro středně velké entity o zhruba 50 stavech/blocích. Automatické rozložení stavů v editoru stavových automatů ale u takto velkých entit přestává být přehledné. V tom by mohla pomoci případná implementace hierarchických automatů.

Příloha A

Nielsenovy heuristiky

Heuristická evaluace je jednou z nejpopulárnějších metod pro testování použitelnosti. Její výhody jsou relativně snadná realizace, nižší časová náročnost a možnost aplikace jak teoretických tak praktických zkušeností testera. Její nevýhodou je pak nemožnost pokrytí všech nedostatků testované aplikace. Existují dva způsoby heuristické evaluace. V tom prvním se tester drží předem daného seznamu bodů, v tom druhém pak vychází pouze z vlastních zkušeností a teoretických znalostí – v takovém případě se jedná spíše o expertní posudek než o heuristickou analýzu.

Tato příloha obsahuje deset základních heuristik pro interaktivní design dle Jakoba Nielsena [2]. Jedná se o volný překlad, názvy heuristik ponecháváme v původním anglickém jazyce.

A.1 Visibility of system status

Systém by měl vždy uživatele informovat o tom, co se právě děje, pomocí adekvátní odezvy v rozumném čase.

A.2 Match between system and the real world

Systém by měl mluvit jazykem uživatele, slovy, frázemi a koncepty blízkými uživateli, raději než za pomoci systémově orientovaných termínů. Dodržujte konvence reálného světa, informace sdělujte v přirozeném a logickém pořadí.

A.3 User control and freedom

Uživatelé často spustí systémovou funkci omylem a budou tak potřebovat jasně označenou možnost návratu z tohoto stavu bez nutnosti procházet rozšířená dialogová okna. Podporujte historii – tlačítka *zpět* a *opakovat*.

A.4 Consistency and standards

Uživatelé by měli mít jasno v terminologii, kterou systém používá. Dodržujte konvence platformy.

A.5 Error prevention

Lepší než dobré chybové zprávy je pečlivý návrh, který zabraňuje vzniku problému. Eliminujte podmínky náchylné na chyby nebo si vyžádejte u takových akcí od uživatele potvrzení, že opravdu ví co dělá, dřív než to udělá.

A.6 Recognition rather than recall

Systém by měl jasně naznačit, v jakém stavu se nachází, a co bylo v předchozích krocích dané akce vybráno – uživatel by si tyto věci neměl sám nutně zapamatovat.

A.7 Flexibility and efficiency of use

Systém by měl nabízet možnost využití klávesových zkratek a podobných urychlovačů práce (dále například uživatelská makra).

A.8 Aesthetic and minimalist design

Dialogy by neměli obsahovat zbytečné (v daném okamžiku irelevantní) informace. Každá nadbytečná informace v dialogu potlačuje relativní viditelnost těch podstatných.

A.9 Help users recognize, diagnose, and recover from errors

Chybová zpráva by měla být popsána prostým jazykem, ne kódem, a měla by přesně indikovat problém a konstruktivně nabídnout jeho řešení.

A.10 Help and documentation

I když by měl být systém použitelný bez dokumentace, je často nutné poskytnout nějakou nápovědu a dokumentaci. Ta by měla být snadno dohledatelná, zaměřená na popis problému z pohledu uživatele, měla by obsahovat výčet konkrétních kroků k vyřešení problému a neměla by být příliš velká.

Příloha B

Referenční příručka – Editor kaskády

Tato příloha obsahuje vývojářskou API dokumentaci editoru kaskády, vygenerovanou pomocí dokumentační knihovny JSDoc[1] na základě komentářů přímo v kódu editoru. Popisuje všechny třídy editoru a jejich metody. Jedná se pouze o zkrácenou verzi bez popisu jednotlivých parametrů, kompletní verze je dostupná v elektronické podobě na přiloženém CD.

B.1 Class: Block

new Block(id, data, editor) Creates new Block instance

Methods

(private) _changeId() → boolean Changes current block id, used as on click handler

(private) _changeType() → boolean Changes current block type, used as on click handler

(private) _create() Creates HTML container for current block

(private) _createHeader() → jQuery Creates HTML header element of this block

(private) _onClick(e) Click handler, sets active state to current block, or toggles it when CTRL pressed

(private) _onDragEnd(e) Drag end handler - used on mouseup event, saves new block position

[1] JSDoc: http://usejsdoc.org/

(private) _onDragEndFromInput(e, $target) Drag end handler - used on mouseup event, creates connection from output of source block

(private) _onDragEndFromOutput(e, $target) Drag end handler - used on mouseup event, creates connection from output of source block to target

(private) _onDragOver(e) Drag over handler - used on mousemove event, moves block over canvas

(private) _onDragOverFromInput(e, $target) Drag over handler - used on mousemove event, renders connection from output of source block to current mouse position

(private) _onDragOverFromOutput(e, $target) Drag over handler - used on mousemove event, renders connection from output of source block to current mouse position

(private) _onDragStart(e) Drag start handler - used on mousedown event, when CTRL is pressed, moves block on canvas, otherwise creates new connections

(private) _processConnections(connections) → Object Normalizes connections to internal format, where aggregation function is stored as pair ["", "func"] instead of [":func"]

(private) _remove() → boolean Removes current block, used as on click handler

(private) _renderConnection(id, source, x2, y2, color$_{\text{opt}}$) → boolean Renders single connection

(private) _toggleInputEditor(e) → boolean Toggles input variable editor, used as on click handler for input variables

activate() Activates current block

addConnection(source, target) → (nullable) boolean Adds connection to this block false when source block not present inside canvas or no name for wildcard variable provided

addInput(variable) Adds input variable to this block

addOutput(variable) Adds output variable to this block

deactivate() Deactivates current block

getBoundingBox() → (nullable) Object Gets current block container bounding box with bounding box points

getNewAggregationFunc() → (nullable) string Gets new aggregation function name

getNewId() → (nullable) string Gets new block id from user via window.prompt()

isActive() → boolean Is current block selected?

position() → (nullable) Object Gets current block container position inside canvas with top and left offset values, null when container not rendered

redraw() Redraw this block

remove() → Object Removes block from canvas Block data in JSON object

render() Renders block to canvas

renderConnections() Renders connections to this block

serialize() → Object Serializes current block to JSON object

toggle() Toggles active state of current block

updatePosition(dx, dy) Updates current block position

B.2 Class: BlockEditor

new BlockEditor(el, options$_{\text{opt}}$) Block Editor 2.0

Members

(static) __namespace

$el

defaults

Methods

__createContainer() Creates container

(private) __createHelp() Creates help modal window

(private) _init() Initialization, loads palette data via AJAX

addBlock(id, data) Adds new block to this editor instance

destroy() Removes editor instance

getBoundingBox(active$_{\mathbf{opt}}$) → Object Finds diagram bounding box

onChange() On change handler, propagates changes to

processData() Parses data and initializes parent block properties and child blocks

refresh() Refreshes editor based on data

render() Renders block editor

serialize() → string Serializes all blocks and parent block information to JSON string

B.3 Class: Canvas

new Canvas(editor) canvas class

Methods

(private) _create() Creates container and canvas element

(private) _drawArrow(x, y) Draws arrow pointing to the right

(private) _drawLine(fromX, fromY, toX, toY) Draws straight line to this canvas

(private) _findPointsToFollow(box, inters, from, to) → Array Finds points that should be followed to avoid box

(private) _getIntersections(id, line) → Array Does line intersect with given block?

(private) _improvePath(points) Removes useless points & adds extra points to smoothen line

(private) _onMouseDown(e) Move canvas or start making selection used as mouse down handler

(private) _onMouseMove(e) Moves canvas - used as mousemove handler

(private) _onMouseUp(e) Completes selection of blocks

(private) _onScroll(e) On scroll handler, used to save current center of viewport (used when zooming)

(private) _sortPoints(points) → Array Topologically sorts given points to path using modified Floyd Warshall algorithm preserves first and last point sorted path (array of points)

(private) _writeText(text, x, y) Writes text to canvas

(private) drawConnection(from, to, color$_{\mathbf{opt}}$) Draws connection line with arrow pointing to end

getCenter() → object Gets center of viewport

getZoom() → object Gets current zoom

redraw() Redraws canvas

render(box) Renders canvas and its container, computes width and height based on diagram bounding box

B.4 Class: Editor

new Editor(block, editor, target) Input editor

Methods

(private) _bind() Binds close handler to close button and ESC key

(private) _changeType(e) Changes type of current input, used as on click handler

(private) _close() → boolean Closes editor

(private) _create() → jQuery Creates input editor container - type element to focus when appended to DOM

(private) _fixTabs(e) → boolean Allows sending tabs to textarea

(private) _isValidJson(str) → boolean Checks whether string is valid JSON string

(private) _keydown(e) → boolean Keydown handler that allows adding tab keys and sending form with CTRL + enter

(private) _onDragEnd(e) Moves editor - unbinds move events

(private) _onDragOver(e) Moves editor

(private) _onDragStart(e) Moves editor - binds move events

(private) _save() → boolean Saves new input value, hides editor

getNewName($\mathbf{output_{opt}}$) → (nullable) string Gets new input name

render() Renders input editor

B.5 Class: Grid

new Grid(blocks, width, height, segment, offset) Grid representation for A* path finding

Methods

(private) _renderPoint(context, x, y, color) Renders single point, used for debugging

avoidBlocks(blocks) Sets walls in grid based on blocks' coordinates

getPoint(grid, x, y, clear) → * Gets grid node with given relative coordinates

getPointObject(x, y) → Point Creates canvas point instance based on relative grid coordinates

render(context) Renders grid control points, used for debugging

renderPath(context, path) Renders given path, used for debugging

toArray() → Array Returns this grid's array

B.6 Class: Line

new Line(from, to) Line segment representation in 2D space

Methods

(private) intersection(line) Do line segments intersect with each other?

length() → Number Computes line segment length

B.7 Class: Palette

new Palette(editor) Palette of blocks

Methods

(private) _createFilter() → jQuery Creates palette filter element

(private) _filter(e) → boolean Filters blocks inside palette

reload(callback) Reloads palette data via AJAX

render() Renders palette

B.8 Class: ParentEditor

new ParentEditor(editor) Parent block properties editor

Extends

Editor

Methods

(private) _bind() Binds close handler to close button and ESC key

(private) _changeTab(e) → boolean Changes active tab, used as on click handler

(private) _changeType(e) Changes type of current input, used as on click handler

(private) _close() → boolean Closes editor

(private) _create() Creates HTML container

(private) _fixTabs(e) → boolean Allows sending tabs to textarea

(private) _isValidJson(str) → boolean Checks whether string is valid JSON string

(private) _keydown(e) → boolean Keydown handler that allows adding tab keys and sending form with CTRL + enter

(private) _onDragEnd(e) Moves editor - unbinds move events

(private) _onDragOver(e) Moves editor

(private) _onDragStart(e) Moves editor - binds move events

(private) _save() → boolean Saves new variable value, hides editor

(private) _saveTextarea(type) → boolean Saves current variable

getNewName($\mathbf{output_{opt}}$) → (nullable) string Gets new input name

render() Renders input editor

B.9 Class: Placeholder

new Placeholder(id, data, editor) Creates new Block Placeholder instance, used inside Palette

Extends

Block

Methods

(private) _changeId() → boolean Changes current block id, used as on click handler

(private) _changeType() → boolean Changes current block type, used as on click handler

(private) _create() Creates HTML container

(private) _createHeader() → jQuery Creates HTML header element of this block

(private) _onClick(e) Click handler, sets active state to current block, or toggles it when CTRL pressed

(private) _onDragEnd() Moves block placeholder to editor canvas - create block instance from placeholder, used as mouseup handler

(private) __onDragEndFromInput(e, $target) Drag end handler - used on mouseup event, creates connection from output of source block

(private) __onDragEndFromOutput(e, $target) Drag end handler - used on mouseup event, creates connection from output of source block to target

(private) __onDragOver(e) Moves block placeholder to editor canvas, used as mousemove handler

(private) __onDragOverFromInput(e, $target) Drag over handler - used on mousemove event, renders connection from output of source block to current mouse position

(private) __onDragOverFromOutput(e, $target) Drag over handler - used on mousemove event, renders connection from output of source block to current mouse position

(private) __onDragStart(e) Moves block placeholder to editor canvas - binds move events, used as mousedown handler

(private) __processConnections(connections) → Object Normalizes connections to internal format, where aggregation function is stored as pair ["", "func"] instead of [":func"]

(private) __remove() → boolean Removes current block, used as on click handler

(private) __renderConnection(id, source, x2, y2, color$_{opt}$) → boolean Renders single connection

(private) __toggleInputEditor(e) → boolean Toggles input variable editor, used as on click handler for input variables

activate() Activates current block

addConnection(source, target) → (nullable) boolean Adds connection to this block false when source block not present inside canvas or no name for wildcard variable provided

addInput(variable) Adds input variable to this block

addOutput(variable) Adds output variable to this block

deactivate() Deactivates current block

getBoundingBox() → (nullable) Object Gets current block container bounding box with bounding box points

getNewAggregationFunc() → (nullable) string Gets new aggregation function name

getNewId() → (nullable) string Gets new block id from user via window.prompt()

isActive() → boolean Is current block selected?

position() → (nullable) Object Gets current block container position inside canvas with top and left offset values, null when container not rendered

redraw() Redraw this block

remove() → Object Removes block from canvas Block data in JSON object

render() Renders block to canvas

renderConnections() Renders connections to this block

serialize() → Object Serializes current block to JSON object

toggle() Toggles active state of current block

updatePosition(dx, dy) Updates current block position

B.10 Class: Point

new Point(x, y) Point representation in 2D space

Methods

(static) angle(a, b, c) Calculates the angle ABC (in radians)

dist(p) → Number Computes distance between to points in euclidean space

dot(p) → Number Calculates the cross product of the two points. cross product

equals(p) → Boolean Is given point equal to this point?

minus(p) → Point Subtracts points, returns new Point instance

plus(p) → Point Adds points together, returns new Point instance

B.11 Class: Spline

new Spline(points, tension, context) Smooth curved line, uses Cardinal Spline for rendering

Methods

(private) _controlPoints(p1, p2, p3) → Array Computes Bezier curve control points based on 3 following points

(private) _drawCurvedPath(cps) Internal rendering method

render() Renders curve to canvas

B.12 Class: Storage

new Storage(storage, namespace) Simple namespaced browser storage wrapper

Methods

(private) _key(key) → string Gets namespaced key name

(private) get(key, json) → Object Gets value for given key

reset(key) Resets variable key

set(key, value, json) → Object Sets variable key to value value

B.13 Class: Toolbar

new Toolbar(editor) Toolbar class

Methods

(private) _copy() → boolean Copies active block(s)

(private) _createButton(name, icon, title, enable$_{opt}$, letter$_{opt}$) → jQuery Creates button element

(private) _cut() → boolean Cuts active block(s)

(private) _keydown(e) → boolean Keydown handler, binds keyboard shortcuts

(private) __paste() → boolean Pastes blocks from clipboard

(private) __redo() → boolean Redo last reverted action

(private) __reloadPalette() Reloads palette data via ajax

(private) __toggleFullScreen() → boolean Toggles fullscreen mode

(private) __toggleHelp() Toggles help modal

(private) __toggleParentProperties() → boolean Toggles parent block properties editor

(private) __undo() → boolean Undo last action

(private) __zoomIn() Zooms in

(private) __zoomOut() Zooms out

(private) __zoomReset() Resets zoom

(private) __zoomTo(scale) Zooms to given scale

disableSelection(e_{opt}) Disables block selection, used as on click handler

render($container) → Array Renders toolbar - left and right toolbar jQuery objects

updateDisabledClasses() Updates disable state of all buttons inside toolbar

Příloha C

Referenční příručka – Editor stavových automatů

Tato příloha obsahuje vývojářskou API dokumentaci editoru stavových automatů, vygenerovanou pomocí dokumentační knihovny JSDoc[1] na základě komentářů přímo v kódu editoru. Popisuje všechny třídy editoru a jejich metody. Jedná se pouze o zkrácenou verzi bez popisu jednotlivých parametrů, kompletní verze je dostupná v elektronické podobě na přiloženém CD.

C.1 Class: Action

new Action(state) Creates new Action instance

Members

(static) colors :Array.<String> automatic colors Array.<String>

Methods

addTransition(source, transition) Assigns transition to this action

removeTransition(transition) Removes transition from this action

renderTransitions(states, index) Renders transitions to canvas

serialize() → Object Serializes current action to JSON object

usesEndNode() → Boolean Finds out whether this action uses end node is there any transition to ___end___ state?

[1] JSDoc: http://usejsdoc.org/

C.2 Class: Canvas

new Canvas(editor) canvas class

Methods

(private) _drawArrow(x, y, angle) Draws arrow pointing to the right

(private) _drawLine(fromX, fromY, toX, toY) Draws straight line to this canvas

(private) _drawPath(points, color$_{\text{opt}}$, highlight$_{\text{opt}}$) Draws given path

(private) _onDblClick(e) On double click handler, used to create new state

(private) _onDragEndCP() → void Drag end handler - used on mouseup event saves new control point position

(private) _onDragOverCP(points, point) → function Drag over handler - used on mousemove event moves control point over canvas

(private) _onDragStartCP(points, point) → function Drag start handler - used on mousedown event starts dragging of control point

(private) _onMouseDown(e) Move canvas or start making selection used as mouse down handler

(private) _onMouseMove(e) Moves canvas - used as mousemove handler

(private) _onMouseUp(e) Completes selection of states

(private) _onScroll(e) On scroll handler, used to save current center of viewport (used when zooming)

(private) _removeControlPoint(trans, points, point) → function Creates callback for removing control point from path

(private) _writeText(text, x, y, path, color$_{\text{opt}}$, highlight$_{\text{opt}}$, postpone$_{\text{opt}}$) Writes text to canvas

clickPosition(e, withoutOffset$_{\text{opt}}$) → Point Gets position of click event

(private) create() Creates container and canvas element

(private) drawConnection(from, to, index$_{opt}$, color$_{opt}$, bidirectional$_{opt}$, highlight$_{opt}$)
Draws connection line with arrow pointing to end

(private) drawCycleConnection(from, to, color$_{opt}$, index$_{opt}$, highlight$_{opt}$) Draws cycle connection line with arrow pointing to end

drawDagreConnection(trans, index, cycle) → Spline|Boolean

getCenter() → Object Gets center of viewport

getZoom() → Object Gets current zoom

redraw() Redraws canvas

render(box) Renders canvas and its container, computes width and height based on diagram bounding box

C.3 Class: Editor

new Editor(editor) Editor panel, with state, transition and machine summary views

Methods

(private) _addColorInputRow(key, label, value, object, cb$_{opt}$) → jQuery
Creates color input row with label and remove button - row object

(private) _addNewProperty(object) Creates new property

(private) _addTextInputRow(key, label, value, object, live$_{opt}$, cb$_{opt}$) → jQuery
Creates text input row with label and remove button - row object

(private) _bind() → void Binds close handler to close button and ESC key

(private) _changeAction(e) Change action handler

(private) _createChangeActionSelect() Creates change action

(private) _createEdgeView() Creates action & edge options view, called by create('edge')

(private) _createNewCP(trans, pos) Creates new control points, finds correct position in path

(private) _createSaveCallback(object, key, json, live$_{opt}$) → function Creates on change handler that instantly saves current value to given object

(private) _createStateView() Creates state options view, called by create('state')

(private) _createSummaryView() Creates summary view, called by create()

(private) _isValidJson(str) → Boolean Checks whether string is valid JSON string

(private) _keydown(e) → Boolean Keydown handler that allows adding tab keys and sending form with CTRL + enter

(private) _removeProperty(key, object) → function Creates handler for property removal

(private) _removeTransition() Removes active transition, used as onclick handler

countObject(obj) → Number Returns count of items defined in object

create(view$_{opt}$, item$_{opt}$, multiple$_{opt}$) Creates editor panel container

getNewName(text, value$_{opt}$, taken$_{opt}$) → (nullable) String Gets new state name

refresh() Refresh editor panel

render() Renders editor panel

C.4 Class: Graph

new Graph(nodes) Graph representation

C.5 Class: Line

new Line(from, to) Line representation in 2D space

Methods

dist(p) → Number Computes shortest distance from this line segment to given point

(private) intersection(line) Do lines intersects with each other?

length() → Number Computes line length

middle() → Point Computes middle point on this line

C.6 Class: Node

new Node(name) Graph node representation

Methods

equals(node) → Boolean Are nodes equal?

C.7 Class: Point

new Point(x, y) Point representation in 2D space

Methods

(static) angle(a, b) → Number Calculates the angle between two points using atan2 (in radians) angle in radians, normalized to interval [0;2pi)

dist(p) → Number Computes distance between to points in euclidean space

dot(p) → Number Calculates the dot product of the two points. dot product

equals(p) → Boolean Is given point equal to this point?

minus(p) → Point Subtracts points, returns new Point instance resulting vector

norm() → Number Computes norm of vector

plus(p) → Point Adds points together, returns new Point instance

toString() → string Returns string representation of this point

C.8 Class: SmalldbEditor

new SmalldbEditor(el) Smalldb Editor 1.0

Members

(static) _namespace

$el

defaults

Methods

__createContainer() Creates container

(private) __createHelp() Creates help modal window

(private) __sortComponent(component) → Array Sorts strongly connected component by its rank, preserve first item (stored in the end of array)

addState(id, data) Adds new state to this editor instance

dagre(force$_{opt}$) Places states to some position on canvas using dagre

destroy() Removes editor instance

getBoundingBox(active$_{opt}$, square$_{opt}$) → Object Finds diagram bounding box

getValue() Gets value

(private) init() Initialization

onChange(dontRefreshEditor$_{opt}$) On change handler, propagates changes to

placeStates(force$_{opt}$) Places states to some position on canvas; if dagre is loaded, use it, otherwise use tarjan

processData() Parses data and initializes machine properties, actions and states

refresh() Refreshes editor based on data

render() Renders Smalldb editor

rotate() Rotates whole entity in counter-clockwise direction

serialize() → string Serializes all states and parent state information to JSON string

setOptions(options) Initialize options map

setValue(value) Sets value

tarjan(force$_{\text{opt}}$) Places states to some position on canvas, uses tarjan's algorithm and renders states based on topological order

toggleHelp() Toggles help modal window

C.9 Class: Spline

new Spline(points, tension, context) Smooth curved line

Methods

(private) _controlPoints(p1, p2, p3) → Array Computes bezier curve control points based on 3 following points

(private) _debug() Renders points and control points for debugging

(private) _drawCurvedPath() Internal rendering method

render() Renders curve to canvas

C.10 Class: Stack

new Stack(nodes) Simple stack implementation

Methods

contains(node) → Boolean Is given node already on stack?

C.11 Class: State

new State(id, data, editor) Creates new state instance

Methods

(private) _changeLabel() → Boolean Changes current state label, used as on click handler

(private) _onClick(e) Click handler, sets active state to current state, or toggles it when CTRL pressed

(private) _onDragEnd(e) Drag end handler - used on mouseup event, saves new state position

(private) __onDragEndConnect(e) Drag end handler - used on mouseup event, creates connection from output of source state to target

(private) __onDragOver(e) Drag over handler - used on mousemove event, moves state over canvas

(private) __onDragOverConnect(e) Drag over handler - used on mousemove event, renders connection from this state to current mouse position

(private) __onDragStart(e) Drag start handler - used on mousedown event, when state is not active, creates new connections

(private) __renderConnection(target, $\mathbf{color_{opt}}$) → Boolean Renders single connection to target point

activate($\mathbf{multiple_{opt}}$) Activates current state

addConnection(target) Adds connection to this state

center() → (nullable) Point Gets current state container center position inside canvas

(private) create() Creates HTML container for current state

deactivate() Deactivates current state

getBorderPoint(other) → Point Gets point on ellipses

getBoundingBox() → (nullable) Object Gets current state container bounding box with bounding box points

getNewLabel() → (nullable) String Gets new state id from user via window.prompt()

isActive() → Boolean Is current state selected?

isConnected(target) → Boolean Is there a connection from this state to given target state?

position() → (nullable) Object Gets current state container position inside canvas with top and left offset values, null when container not rendered

redraw($\mathbf{noCanvasRedraw_{opt}}$) Redraw this state

remove() → Object Removes state from canvas state data in JSON object

removeConnection(target) Removes connection from this state

(private) removeHandler() → Boolean Removes current state, used as on click handler

render($\$parent_{opt}$) → this Renders state to canvas

serialize() → Object Serializes current state to JSON object

toggle($multiple_{opt}$) Toggles active state of current state

updatePosition(dx, dy) Updates current state position

C.12 Class: Storage

new Storage(storage, namespace) Simple namespaced browser storage wrapper

Methods

(private) _key(key) → string Gets namespaced key name

(private) get(key, json) → Object Gets value for given key

reset(key) Resets variable key

set(key, value, json) → Object Sets variable key to value value

C.13 Class: Tarjan

new Tarjan(graph) Tarjan's algorithm for finding strongly connected components

Methods

(private) _strongConnect(node) Finds strongly connected components for given node

run() → Array Find strongly connected components!

C.14 Class: Toolbar

new Toolbar(editor) Toolbar class

Methods

(private) __automaticEdgeColors() Assigns action colors automatically

(private) __automaticLayout() Assigns state positions automatically

(private) __copy() → Boolean Copies active state(s)

(private) __createButton(name, icon, title, enable$_{opt}$, letter$_{opt}$) → jQuery Creates button element

(private) __cut() → Boolean Cuts active state(s)

(private) __keydown(e) → Boolean Keydown handler, binds keyboard shortcuts

(private) __paste() → Boolean Pastes states from clipboard

(private) __redo() → Boolean Redo last reverted action

(private) __rotate() Rotates whole entity in counter-clockwise direction

(private) __toggleControlPoints() Toggles control points visibility

(private) __toggleCycles() Toggles cycles visibility

(private) __toggleFullScreen() → Boolean Toggles fullscreen mode

(private) __toggleHelp() Toggles help modal

(private) __undo() → Boolean Undo last action

(private) __zoomIn() Zooms in

(private) __zoomOut() Zooms out

(private) __zoomReset() Resets zoom

(private) __zoomTo(scale) Zooms to given scale

disableSelection(e$_{opt}$) Disables state selection, used as on click handler

render($container) → void Renders toolbar

updateDisabledClasses() Updates disable state of all buttons inside toolbar

C.15 Class: Transition

new Transition(action, data, source, target) Creates new Transition instance

Methods

(private) _renderDagrePath(states, s, index, bidirectional, cycle) Renders path computed by dagre

(private) _segmentize(points) → function Creates bezier curve split callback, uses de Casteljau's algorithm

activate() Activates current transition

contains(point) → Boolean Is given point on this transition's curve?

deactivate() Deactivates current transition

isActive() → Boolean Is current transition selected?

remove() Removes this transition

render(states, index) Renders transition to canvas

serialize() → Object Serializes current transition to JSON object

setAction(action) Attach this transition to different action

Příloha D

Obsah přiloženého CD

`adamema4-thesis.pdf` – Tento text ve formátu PDF.

`thesis/` – Zdrojové soubory pro tento text ve formátu pro editor LYX.

`block_editor/` – Zdrojové kódy pluginu „Editor kaskády".

`block_editor/api/` – Kompletní API dokumentace pluginu.

`block_editor/examples/` – Ukázky fragmentů kaskády v podobě HTML stránek.

`block_editor/external/` – Volitelné závislosti editoru kaskády.

`block_editor/js/classes/` – Zdrojové kódy jednotlivých tříd.

`smalldb_editor/` – Zdrojové kódy pluginu „Editor stavových automatů Smalldb".

`smalldb_editor/api/` – Kompletní API dokumentace pluginu.

`smalldb_editor/examples/` – Ukázky automatů Smalldb v podobě HTML stránek.

`smalldb_editor/external/` – Volitelné závislosti editoru automatů Smalldb.

`smalldb_editor/js/classes/` – Zdrojové kódy jednotlivých tříd.

Literatura

[1] Josef Kufner. Blokové programování webových aplikací, 2012. 15, 17, 18, 21, 22, 23, 29, 30

[2] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. ISBN 0125184050. 17, 40, 95

[3] Josef Kufner and Radek Mařík. Self-generating Programs – Cascade of the Blocks. *Information and Communication Technology, 199-212.*, 2014. 18, 21

[4] Josef Kufner and Radek Mařík. State Machine Abstraction Layer. *Information and Communication Technology. 213-227.*, 2014. 15, 18, 25, 26

[5] Mozilla Developer Network – JavaScript. URL `https://developer.mozilla.org/cs/docs/Web/JavaScript`. 19

[6] Node.js framework. URL `https://nodejs.org/`. 19

[7] Angularjs framework. URL `https://angularjs.org/`. 19

[8] Mozilla Developer Network – Inheritance and Prototype Chain in JavaScript. URL `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain`. 19

[9] jQuery framework. URL `https://jquery.com/`. 19

[10] Karl-Heinz John, Michael Tiegelkamp. IEC 61131-3: Programming Industrial Automation Systems, 1995. URL `http://www.dee.ufrj.br/controle_automatico/cursos/IEC61131-3_Programming_Industrial_Automation_Systems.pdf`. 21

[11] G. H. Mealy. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955. 27

[12] Edward F. Moore. Gedanken Experiments on Sequential Machines. In *Automata Studies*, pages 129–153. Princeton U., 1956. 27

[13] Andrew Mertz and William Slough. Graphics with TikZ. *The PracTEX Journal*, (1), 2007. 31

[14] Manjusha S. Joshi. Create Trees and Figures in Graph Theory with PSTricks. *The PracTEX Journal*, 1, 2007. 31

[15] John D. Hobby. A User's Manual for MetaPost. URL http://www.tug.org/tutorials/mp/mpman.pdf. 32

[16] JointJS – JavaScript Diagramming Library. URL http://www.jointjs.com/. 15, 32, 33

[17] Chris Pettitt. Dagre. URL https://github.com/cpettitt/dagre/wiki. 15, 33, 35

[18] yEd. URL http://www.yworks.com/en/products/yfiles/yed/. 15, 34, 36

[19] Visual Graph Editor 2. URL https://code.google.com/p/vge2/. 15, 34, 37

[20] UPPAAL. URL http://www.uppaal.org/. 15, 36, 38

[21] Yakindu State Chart Tools. URL http://www.statecharts.org/. 15, 38

[22] Cardinal Splines, 2012. URL https://msdn.microsoft.com/en-us/library/ms536358.aspx. 50

[23] Ronald Goldman. Intersection of two lines in three-space. *Graphics Gems*, page 304. 53

[24] Gregory G Slabaugh. Computing Euler angles from a rotation matrix. *Retrieved on August*, 6(2000):39–63, 1999. 53

[25] Qef. Values of atan2 on the unit circle, 2009. URL http://commons.wikimedia.org/wiki/File:Atan2_circle.svg. 15, 55

[26] Xiao Cui and Hao Shi. A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1):125–130, 2011. 59

[27] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959. 59

[28] Amit Patel. Amit's Thoughts on Pathfinding. URL http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html. 15, 59, 60

[29] Kaspar Fischer. Piecewise Linear Approximation of Bézier Curves, 2000. URL https://hcklbrrfnn.files.wordpress.com/2012/08/bez.pdf. 16, 79

[30] zinka. A diagram of the polar form (relative to center) of an ellipse, 2010. URL http://en.wikipedia.org/wiki/File:Ellipse_Polar_center.svg. 16, 82

[31] Ioannis Tollis, Peter Eades, Giuseppe Di Battista, and Loannis Tollis. *Graph drawing: algorithms for the visualization of graphs*, volume 1. Prentice Hall New York, 1998. 83, 84

[32] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem phong Vo. A Technique for Drawing Directed Graphs. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 19(3):214–230, 1993. 83, 84

[33] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972. 83

[34] Roberto Tamassia. *Handbook of Graph Drawing and Visualization (Discrete Mathematics and Its Applications)*. Chapman & Hall/CRC, 2007. ISBN 1584884126. 84

[35] Michael Jünger and Petra Mutzel. 2-layer straightline crossing minimization: performance of exact and heuristic algorithms. *J. Graph Algorithms Appl*, 1:1–25, 1997. 85

[36] Wilhelm Barth, Michael Jünger, and Petra Mutzel. Simple and Efficient Bilayer Cross Counting, 2002. 85

[37] Ulrik Brandes and Boris Köpf. Fast and Simple Horizontal Coordinate Assignment, 2002. 85

[38] *Qt Documentation*. URL `http://doc.qt.io/qt-5/statemachine-api.html`. 16, 86, 87

[39] Appjs framework. URL `http://appjs.com/`. 87

Rejstřík

A
A*, 59
akce, 75
atan2, 53
automat, 25

B
Bézierova křivka, 50
Blok, 46, 48

C
Canonical spline, 50
Cardinal spline, 50
Catmull–Rom spline, 51
Cíle a řešení, 17
Cílová skupina, 64
cyklus, 69, 79

D
Dagre, 33, 83
de Casteljau, 79
diagram tříd, 44
`dot`, 30
Drag'n'drop, 55

E
Editor kaskády, 29
Editor nadřazeného bloku, 46
Editor vstupů, 46
Editovací panel, 75
etapy, 21
Eukleidovský graf, 60

F
Formát, 41
framework, 25
Funkční požadavky, 39, 69

G
GIT, 41
GraphML, 34
Graphviz, 30
grid, 58

H
Heuristická evaluace, 63, 87, 95
Hierarchické automaty, 86
Historie, 56
HTML5, 19

I
inicializace, 52, 80

J
jmenné prostory, 23
JointJS, 32
jQuery, 19
JSDoc, 97
JSON schéma, 41, 70

K
Kaskáda, 18
kaskáda, 21
klávesové zkratky, 63
kolize, 50, 57
konečný automat, 25
konfigurace, 52, 80
Kreslení spojnic, 50
Kreslící plátno, 46, 47

L
localStorage, 56, 57

M
Maximální velikost grafu, 66, 90
mockup, 47
motivace, 17
multihrana, 70
Multihrany, 85
MVC, 21, 25

N
Načítání palety bloků, 57

Nápověda, 63
návrh, 44
Nedeterminismus, 26
Nefunkční požadavky, 40, 70
Nejkratší cesta, 58, 60

O
Oprávnění, 86
Označování hran, 69

P
paleta bloků, 46, 48
Panel nástrojů, 46, 48
planarizace grafu, 84
Podpora prohlížečů, 20
přechod, 75
Přiblížení/oddálení, 56
případy užití, 43
průsečík, 52

R
Rappid, 32
Režim statického obrázku, 57
rozšíření, 86

S
schéma, 41
schránka, 57, 63
sessionStorage, 56
Silně souvislé komponenty, 83
Smalldb, 18, 25, 29
spline křivka, 47, 50
Spojování bloků, 22
Srovnání, 60, 85
srovnání, 65
stav, 75

T
Tarjan, 75, 83
technologie, 18
testovací prostředí, 65
testování, 63, 88

U
Úložiště prohlížeče, 56
UPPAAL, 36
use case, 43
Uživatelské role, 42

V
Vizualizace, 23
Vlastnosti automatu, 26
vyhodnocení kaskády, 21
Vyhodnocování, 23

W
wireframe, 47
workflow, 18

Y
yEd, 34

Z
Zástupný blok, 46