TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Simulace obvodů v grafickém editoru číslicových obvodů

Diplomová práce

Studijní program: N2612 – Elektrotechnika a informatika
Studijní obor: 1802T007 – Informační technologie

Autor práce: **Bc. Tomáš Václavík**
Vedoucí práce: Ing. Martin Rozkovec, Ph.D.

Liberec 2017

TECHNICAL UNIVERSITY OF LIBEREC
Faculty of Mechatronics, Informatics
and Interdisciplinary Studies

Simulating digital circuits in HTML based schematic editor

Master thesis

Study programme: N2612 – Electrical Engineering and Informatics
Study branch: 1802T007 – Information Technology

Author: **Bc. Tomáš Václavík**
Supervisor: Ing. Martin Rozkovec, Ph.D.

Liberec 2017

Technická univerzita v Liberci
Fakulta mechatroniky, informatiky a mezioborových studií
Akademický rok: **2016/2017**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Tomáš Václavík**
Osobní číslo:	**M14000184**
Studijní program:	**N2612 Elektrotechnika a informatika**
Studijní obor:	**Informační technologie**
Název tématu:	**Simulace obvodů v grafickém editoru číslicových obvodů**
Zadávající katedra:	**Ústav informačních technologií a elektroniky**

Zásady pro vypracování:

1. Seznamte se stávajícím editorem číslicových obvodů. Osvojte si HTML5, Javascript a PHP.
2. Seznamte se způsoby diskrétní simulace obvodů.Navrhněte mechanismus pro interaktivní (na straně klienta) simulaci uživatelem vytvořeného obvodu.
3. Prozkoumejte možnosti offline (na straně serveru) simulace obvodu. Vytvořte mechanismus, který umožní automaticky vyhodnocovat správnost obvodů vytvořených uživatelem.
4. Daný systém implementujte a nasaďte na školní infrastruktuře.

Rozsah grafických prací: **Dle potřeby dokumentace**

Rozsah pracovní zprávy: **cca 40-50 stran**

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

[1] **CLIENT IO. JointJS - the HTML 5 JavaScript diagramming library. [online]. 2009 - 2014 [cit. 2015-10-13]. Dostupné z: http://jointjs.com/.**

[2] **PINKER, Jiří a Martin POUPA. Číslicové systémy a jazyk VHDL. 1. vyd. Praha: BEN - technická literatura, 2006, 349 s. ISBN 80-730-0198-5.**

Vedoucí diplomové práce: **Ing. Martin Rozkovec, Ph.D.**
Ústav informačních technologií a elektroniky

Datum zadání diplomové práce: **12. září 2016**
Termín odevzdání diplomové práce: **15. května 2017**

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

prof. Ing. Ondřej Novák, CSc.
vedoucí ústavu

V Liberci dne 12. září 2016

Prohlášení

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu TUL.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Diplomovou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Současně čestně prohlašuji, že tištěná verze práce se shoduje s elektronickou verzí, vloženou do IS STAG.

Datum: 1. 9. 2017

Podpis:

Poděkování

Děkuji svému vedoucímu práce, panu Ing. Martinu Rozkovcovi, Ph. D., za jeho čas, který mně věnoval při vedení mé diplomové práce, za přínosné rady a za trpělivost, kterou se mnou měl. Také bych rád poděkoval Ing. Jiřímu Jeníčkovi, Ph. D., za cenné rady a pomoc při konfiguraci serveru a za jeho správu. Rovněž děkuji svým rodičům a prarodičům, za jejich neutuchající podporu.

Abstrakt

Práce se zabývá rozšířením a upravením webové aplikace grafického editoru číslicových obvodů. Rozšířeními jsou interaktivní simulace obvodů (vizuální zviditelnění aktivních vodičů) na straně klienta. Testování a automatická kontrola vytvořených obvodů za využití programu Xilinx Vivado na straně serveru. Dále navržení a implementace uživatelského rozhraní pro správu a zadávání úkolů ve frameworku Backbone JS a nasazení celé aplikace na školní server.

Abstract

The thesis deals with the extension and modification of the web application of graphic digital circuits editor. The new features are interactive circuit simulation on the client side (to make active wires visually highlighted). Testing and auto-checking of created circuits using Xilinx Vivado on server-side. The thesis also describes the design and implementation of the user interface for managing and assigning tasks. And last: deploying the entire application to the school server.

Klíčová slova

Simulace obvodu, číslicové obvody, grafický editor, webová aplikace, PHP, Javascript, Backbone JS, Joint JS, Xilinx Vivado

Keywords

Circuit simulation, digital circuits, graphic editor, web application, PHP, Javascript, Backbone JS, Joint JS, Xilinx Vivado

Obsah

Seznam ilustrací

Seznam zdrojových kódů

Úvod

V rámci předchozí diplomové práce Jaroslava Řeháka [1] vznikl jednoduchý grafický editor číslicových obvodů, postavený na webové platformě za využití knihovny JointJS [2]. Jeho schopnostmi bylo vytvářet schéma číslicového obvodu umísťováním hradel na plátno a jejich propojování vodiči metodou táhni a pusť. K dispozici byla knihovna obsahující téměř 50 entit, ze kterých jste mohli obvod složit. Mezi nimi byly jak základní logické členy, tak i komplexní sekvenční obvody jako například paměti. Schéma vytvořené v tomto editoru bylo možné exportovat do VHDL souboru a tím ho uchovat pro další využití. Editor také umožňoval importovat schéma z VHDL, nicméně právě toto byla jediná možnost, jak schéma trvale uložit, jelikož po obnovení stránky prohlížeče byla všechna práce ztracena.

V rámci diplomového projektu jsem zmíněný editor rozšířil o další funkce a řadu z nich upravil. Zejména bylo potřeba doplnit zmíněné trvalé ukládání schémat, realizované v lokálním úložišti v prohlížeči a na serveru. Aby se toho dalo docílit, bylo potřeba přidat přihlašování uživatelů a uživatelské účty. Aplikace také prošla změnou uživatelského rozhraní, pro usnadnění jejího používání. Příkladem vylepšení za všechny je ukládání souřadnic umístění všech entit při exportu do VHDL. Díky tomu bylo možné je při opětovném načtení obnovit včetně jejich umístění.

Předchozí úpravy umožnili používání editoru v dlouhodobějším časovém horizontu, nicméně pro zamýšlené využití ve výuce číslicových obvodů nebyl editor vyhovující. Jednou z požadovaných nových funkcí je simulace obvodu. U klienta, ve webovém prohlížeči, jen vizuální. Na serveru potom simulace umožňující automatickou kontrolu správnosti obvodu. Aplikace je zamýšlená jako jednoduchý editor, ve kterém studenti vypracují své domácí úkoly a prostřednictvím aplikace je také odevzdají. Následně by jejich řešení mělo být automaticky zkontrolováno pomocí simulace na serveru. Toto si vyžádá úpravu a rozšíření uživatelského rozhraní o funkce od zadávání úkolů, přes správu účtů, až po zobrazení výsledků simulace. Rozšířit se musí také schéma databáze a jádro celé aplikace, která byla pouhou webovou stránkou s editorem. V cílové podobě bude spíše webovou aplikací.

1 Použité technologie

1.1 V prohlížeči a pro komunikaci

Podstatnou částí aplikace je klientská aplikace běžící v prohlížeči uživatele. Přirozeným programovacím jazykem pro web je Javascript (dále JS), který jsem využil pro vytvoření celé klientské části aplikace. Spolu s několika nezbytnými a užitečnými knihovnami a jedním frameworkem.

1.1.1 Knihovna JointJS

Základem editoru je knihovna pro vykreslování grafů a schémat JointJS, její funkce jsou dobře popsané v dokumentaci [3]. Umožňuje definovat a vykreslit v podstatě libovolný 2D graf. Hranami spojené uzly, přičemž hrany a uzly lze libovolně přidávat. Pro definici podoby grafů využívá vektorový formát SVG a umožňuje zacházet se vzhledem grafu také pomocí javascriptu. Dále poskytuje mechanismy pro manipulaci s grafem, například metodou táhni a pusť. Knihovně JointJS věnuji dále samostatnou kapitolu, která popíše, jakými prvky disponuje a jak se s knihovnou pracuje.

1.1.2 Framework Backbone

Doslova páteří aplikace, nejen částí editoru, je MVC framework Backbone.js. Knihovna JointJS je na něm založena a využívá jeho modelů, pohledů a událostí [3]. Backbone je starší open source knihovna, která díky struktuře tvořené z modelů využívajícího datového typu klíč-hodnota, podpoře vlastních událostí a vykreslování pomocí pohledů, přináší do aplikace ucelený formát [4]. Modely se dají slučovat do kolekcí, které nabízí rozhraní a funkce pro jejich procházení. Dále pohledy a šablony pro zobrazení dat. Silnou stránkou Backbone je podpora RESTful JSON komunikace se serverem. Poslední velmi využívanou částí je systém směrování (anglicky routing), díky kterému má každý pohled stránky vlastní adresu URI, respektive vlastní část adresy za znakem mřížky.

Backbone je dnes pro nové projekty typu interaktivní aplikace překonaný [5] a jeho vhodnou náhradou může být například React nebo Angular. Nicméně má své silné stránky: jednoduchost, volnost přístupu, směrování, podporu RESTful, pro které stojí za zvážení u specifických projektů. Pro tento projekt jsem ho zvolil jako vhodný z důvodu, že je již využíván zvolenou základní knihovnou JointJS pro vykreslení schéma v editoru. Dále pro jeho podporu RESTful komunikace a solidní stabilitu, která je dána léty vývoje.

1.1.3 Základní technologie a komunikace

Jelikož jde o webovou aplikaci, základní trojice technologií jsou HTML5, CSS3 a javascript. Poté jsem použil knihovnu jQuery, který je vhodná pro použití v kombinaci s Backbone a jeho pohledy (View), přičemž je Backbonem vyžadována verze alespoň 1.11.0. Bakbone také vyžaduje knihovnu Underscore v minimání verzi 1.8.3., kterou lze nahradit za kompatibilní Lodash, jenž je v podstatě její nadstavbou [4].

Komunikace serveru s klientem, probíhá v aplikaci až na výjimky asynchronně za využití REST JSON rozhraní operujícího přes HTTP, REST je pro potřeby této aplikace vyhovující komunikační architekturou. Je to dobře zvládnutá a rozšířená technologie a má jednak nativní podporu v knihovně Backbone a jednak ve zvoleném frameworku na serverové straně.

1.2 Server

Pro nasazení této aplikace na školní infrastrukturu jsem dostal k dispozici virtuální školní server s operačním systémem CentOS 6.9. Server nemám plně v režii, je spravovaný původním správcem a po určitou dobu tvorby této práce na něm hostovala další webová stránka.

Serverová aplikace v sobě spojuje několik technologií a knihoven. Webový server je Apache ve verzi 2.2.15, který byl na serveru nainstalován jeho správcem, stejně tak databáze MySql s verzí distribuce 5.1.73. K ní do kombinace je PHP verze 5.3.3. Tato konfigurace je do značné míry omezující a kód, respektive knihovny se musely těmto požadavkům přizpůsobit a vhodně zvolit, jako například Framework Slim.

1.2.1 Technologie aplikace

Serverová část aplikace se dá rozdělit do několika celků. Prvním je hlavní vstupní bod do aplikace, soubor index.php, který poskytuje stránku/aplikaci jako takovou se všemi jejími šablonami, styly a obrázky. Jak je patrné, je napsána v jazyce PHP. Aplikace je rozdělena po dílčích celcích do samostatných souborů a soubor index.php je výchozí bod. Zajišťuje také jednoduché směrování, pro které jsem využil malou lehkou knihovnu Slim ve verzi 2, která podporuje požadovanou verzi PHP 5.3.3 [6][9].

Druhou částí je REST API poskytující data z databáze (dále DB) a vyřizující dynamické požadavky aplikace. Je také napsané v jazyce PHP za využití knihovny Slim.

Za třetí část se dá považovat aplikace Xilinx Vivado (verze 2016.4), která byla na server nainstalována pro simulaci číslicových obvodů za účelem automatické kontroly schémat uložených v jazyce VHDL. Vivado je školou licencovaný software od společnosti Xilinx pro syntézu a analýzu návrhů ve VHDL. Vivado je ovládáno pomocí skriptovacího jazyka Tcl, celý postup popisuji v kapitole 6.4.4 TCL script.

1.3 Vývojové nástroje a automatizace

Řada činností při vývoji se dá automatizovat, nebo usnadnit využitím již hotových řešení. Pro vývoj v PHP a správu balíčků a knihoven využívám správce závislostí Composer, který se, kromě stažení požadovaného balíčku a všech jeho závislostí, postará o své automatické načtení do aplikace [7].

Obdobně ve vývoji klientské části používám nástroj Bower, který se stará o stažení balíčku a jeho závislostí zejména JS knihoven, ale i stylů a grafiky. Již však neumožňuje automatické načítání požadovaných scriptů do stránky [8].

Další užitečný nástroj je Grunt [9], automatizační nástroj, který umožňuje spouštět definované úlohy. Grunt pro svůj běh používá Node.js. Jeho typické použití je pro překlad například SASS stylů do čistého CSS, kontrola validity JS, minimalizace CSS pro zmenšení datového objemu a minimalizace a znepřehlednění JS kódu. U JS souborů, se provádí kromě vynechání prázdných a nevýznamných znaků také přejmenování proměnných na kratší, typicky jednoznakové názvy a tím se docílí ještě větší úspory místa. To vše je spouštěno automaticky při změně zdrojových souborů, o to se stará modul Watch, který kontroluje změny v souborech a poté vykonává definované operace.

Úlohy Gruntu se deklarují a konfigurují v souboru Gruntfile.js za využití jazyka JS. Každá úloha, kterou chcete provést musí mít svého vykonavatele, modul, který je potřeba nejprve stáhnout a nainstalovat pomocí balíčkovacího systému npm pro Node.js. Stažené úlohy jsou poté zaznamenány jako závislosti do souboru `package.json` a mohou být tak snadno předány dalším případným vývojářům, kteří by pokračovali v tomto projektu.

1.4 Backbone framework

Backbone je určen pro tvorbu tzv. jednostránkových aplikací (angl. Single-page), případně pro interaktivní části webu. Jak bylo nastíněno v krátkém představení frameworku (viz 1.1.2 Framework Backbone), Backbone využívá návrhový vzor MVC, respektive jeho odnož MVP (Model-View-Presenter). Základním stavebním prvkem je Model (písmeno M

z MVP), který reprezentuje data jako objekty. Tyto objekty jsou předávány pohledům (View, V z MVP), který má za úkol je zobrazit. Podoba dat je čistě závislá na pohledu a jeden model může být zobrazen různými pohledy. Pod třetím písmenem P je schovaný Presenter, který je trochu abstraktní. Modely jsou předávány pohledům při vytváření jejich instancí a pohled zajišťuje zpracování událostí v uživatelském rozhraní a změnu modelu. Změna modelu je poté automaticky (díky vyvolání události) promítnuta do všech pohledů, které ji využívají. Presenter je tak z části součástí pohledu. Část prezentační činnosti přebít další objekt Router, který definuje cesty (adresy) a pro ně jejich obslužné funkce. V obslužné funkci mohou být vytvořeny instance modelů a pohledů, případně jsou načtena data modelu, ale vždy je pohled alespoň vykreslen. Backone dále nabízí objekt kolekce (Collection) pro nakládání se sezamem modelů stejného typu. Je to alternativa k uchování modelů v poli, poskytuje navíc funkcionalitu podobnou jako u Modelu. Podobně může reagovat na změny a vyvolávat překreslení pohledu, či synchronizaci se serverem.

Ještě zmíním, že Backbone komunikuje se serverem nativně přes RESTfull rozhraní, modely a kolekce disponujíc funkcemi pro synchronizaci dat (sync, fetch, save atd.). Více o konkrétních částech v samostatných odstavcích.

1.4.1 Model

Ve frameworku Backbone jsou data reprezentována modely [4]. Modely je možné vyvářet, validovat, mazat a ukládat na server. Datové položky mohou být kdykoliv změněny a každá změna vyvolá spuštění události `change`. Na tuto změnu mohou být upozorněny všechny pohledy, které s daným modelem pracují, a tak na změnu patřičně zareagovat. Typicky překreslit daný obsah novými daty. Díky tomu není potřeba vytvářet kód navíc, který vyhledá prvek v DOM podle ID, jen aby mohl aktualizovat jeho HTML. Pokud se model změní, je pohled aktualizován automaticky.

Kromě datových položek v sobě Model obsahuje také logiku pro validaci, formátování, výpočty hodnot, či řízení přístupu. Vlastní modely se tvoří rozšířením třídy `Backbone.Model` pomocí funkce `extend`, která vytvoří nový objekt (model), kopírováním a rozšířením šablonového objektu (předka). Vlastní přidané metody se funkci `extend` předají v objektu jako parametr, viz Zdrojový kód 1. Potom instanci této třídy vytvoříme klasicky pomocí klíčového slova `new`.

```
var MujModel = Backbone.Model.extend ({
        initialize: function() {…},
});
var mujObjekt = new MujModel({
        param1:data1,
        param2:data2
});
```

Zdrojový kód 1:Rozšíření modelu Backbone

Model nabízí řadu metod pro manipulaci s daty, nejvyužívanější jsou metody `get`, pro získání hodnoty atributu, metoda `set` pro nastavení hodnoty atributu. Dále `escape`, která vrací hodnotu atributu jako metoda `get` s tím rozdílem, že se postará o ošetření výstupu HTML a přispívá tak k zamezení XSS útoku. Každá model má také unikátní identifikátor, automaticky přidělovaný **cid** a vlastní **id**, které je získáváno například z databáze, případně jiným způsobem. Název unikátního identifikátoru lze změnit například v závislosti na názvu sloupku v DB. Důležitá je také položka `defaults`, která umožňuje nastavit výchozí hodnoty pro atributy modelu.

Pro synchronizaci dat se serverem je využíváno metody `fetch`, která načte data modelu ze serveru asynchronně, k tomu je potřeba aby model měl definovanou **url**. Tu lze modelu nastavit přímo na konkrétní hodnotu, nebo přiřadit funkci, která umožní adresu parametrizovat a dynamicky ji generovat a reflektovat v ní například ID aktuálního modelu, případně data z jiného. Adresu využije také metoda `save`, která v závislosti na tom, jestli je nastaveno ID, odešle data na server metodou **POST** pro vytvoření nové položky, nebo metodou **PUT**, která data pouze aktualizuje. Pokud ID není nastaveno je model považován za nový a bude vytvořen, jinak jeho data aktualizována.

1.4.2 Kolekce (Collection)

Pro manipulaci se seznamem modelů se nabízí třída `Backbone.Collection`. Kolekce je seřazená množina modelů. Událost **change** je vyvolána vždy, když dojde k přidání nebo odebrání prvku z kolekce (události **add** a **remove**). Stejně jako u modelu má kolekce metodu `fetch`, která pošle asynchronně požadavek na server. Události vyvolané na modelu v kolekci jsou vyvolány také na kolekci samotné. Na kolekce je možné aplikovat přímo funkce z knihovny **Underscore.js** a využít tak její možnosti například v procházení nebo filtrování. (V případě této aplikace jde o funkce z kompatibilní knihovny Lodash.) Kromě klasických metod pro přidávání (`add`), odebírání (`remove`) a například řazení obsahuje metodu `create`, která vytvoří nový objekt ze zadaných parametrů, uloží ho na server a přidá ho do kolekce. Prvky kolekce lze předat při inicializaci a naplnit ji tak před prvním použitím [4].

```
eco.Collections.GroupCollection = Backbone.Collection.extend({
    model: eco.Models.Group,
    initialize: function (models, options) {
        this.urlString = options.url;
    },
    url: function () {
        return this.urlString;
    }
});

var groups = new eco.Collections.GroupCollection(null, {
    url: '/api/groups/'
});
```

Zdrojový kód 2: Vytvoření a použití kolekce

Framework vždy předpokládá úspěšné provedení požadavku na server a kolekci upraví ještě před přijmutím odpovědi ze serveru. Pokud půjde o neúspěch, je na nás abychom změny vrátili.

1.4.3 Pohledy (View)

Backbone pomáhá oddělit aplikační logiku od uživatelského rozhraní, prezentace dat uživatelům je tak záležitostí pohledů. Pohledy (**View**) vykreslují UI a reagují na změny

Obrázek 1: Spolupráce modelu a pohledu

v modelu. Starají se o uživatelské vstupy a aktualizují data modelu. Obrázek 1 ukazuje typické vazby mezi modelem a pohledem, kde pohled je automaticky aktualizován při změněn modelu [4].

Pohledy v Backbone neurčují žádná pravidla pro podobu HTML kódu a dovolují využít libovolný javascriptový šablonovací sytém. Základním a mnou využívaným je však ten z knihovny Underscore.

Vlastní pohledy se konstruují podobně jako Modely či kolekce funkcí `extend` na objektu **Backbone.View** viz Zdrojový kód 3, který zobrazuje vytvoření pohledu s událostí a funkcí `render`. Každý pohled pracuje a je omezen na svůj kontejner, ve kterém je vykreslován a jsou v něm zachytávány události z DOM. Pokud není nastaven kontejner pohledu parametrem **el** při jeho inicializaci, například na jQuery objekt, tak je vytvořen automaticky a je ho následně třeba přidat do DOM stránky.

Název elementu definuje atribut `tagName` a jeho třídu atribut `classsName`. Metoda render zajišťuje změnu obsahu kontejneru typicky při změně v modelu, je to však na naší volbě, nastavení odposlouchávání událostí je v inicializační funkci. Šablona, která má být vykreslena je v ukázce Zdrojový kód 3 nastavena jako řetězec a v inicializaci je z HTML kódu vytvořena šablona.

1.5 JointJS

JointJS [3] Je grafická knihovna pro vizualizaci grafů a diagramů různých podob a tvarů. Umožňuje vytváření jak statických diagramů, tak interaktivní manipulaci s nimi. Dovoluje libovolné možnosti přizpůsobení vzhledu grafů pomocí SGV jak bylo zmíněno v úvodní kapitole 1.1.1 Knihovna JointJS.

JointJS je ideální pro vykreslování prvků číslicových obvodů. Používá moderní SVG grafiku, která umožňuje zobrazit téměř cokoli a libovolně to ovládat prostřednictvím Javascriptu. Knihovna zvládá vykreslit stovky až tisíce entit a vodičů v reálném čase. Ze základních tvarů a prvků jako je obdélník, kruh, text či obrázek nebo libovolná cesta lze hierarchicky poskládat složitější prvky jako jsou hradla číslicových obvodů.

JointJS pro tvorbu své struktury používá knihovnu Backbone.js, jeho dvě základní součásti Graph a Paper jsou rozšířením objektů modelu a pohledu z Frameworku Backbone. Model Graph obsahuje veškerá data o vytvořeném schéma, všechny entity, vodiče, jejich propojení a další uživatelská data, které lze libovolné přiřazovat.

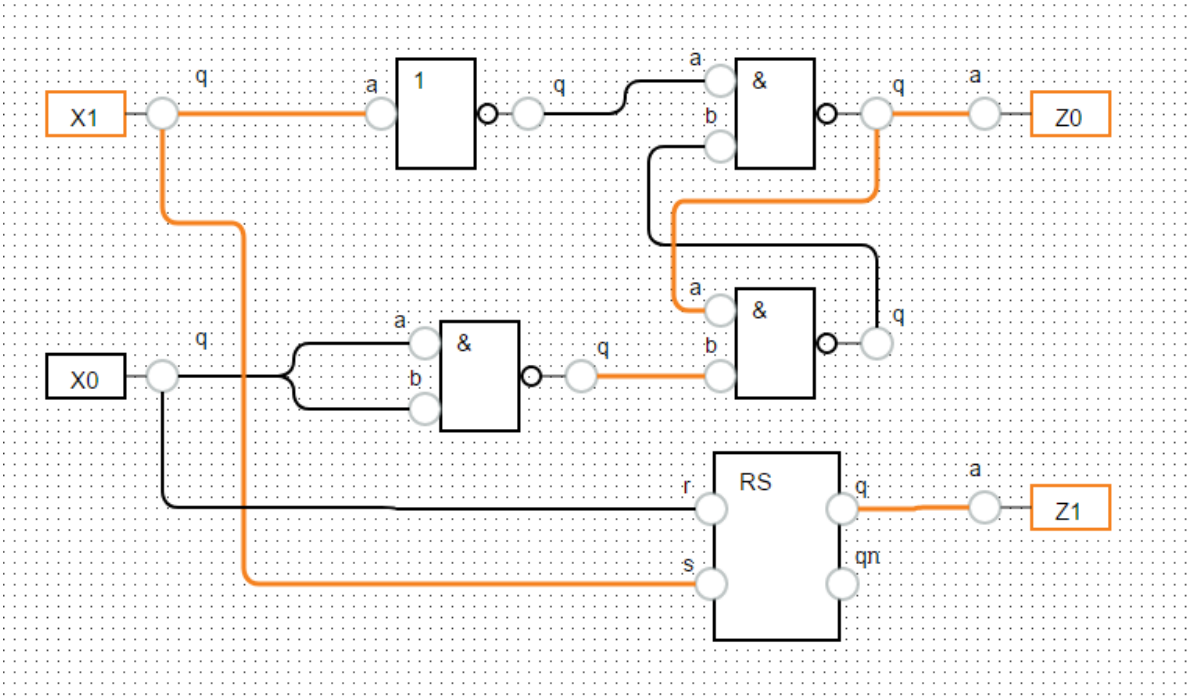
```
eco.Views.TableRow = Backbone.View.extend({
    tagName: 'tr',
    className: 'tableRow',
    initialize: function (opts) {
        this.template = _.template($(opts.template).html());
        this.model = opts.model;
        this.listenTo(this.model, 'change', this.render);
    },
    events: {
        'click .primaryAction' : 'primaryAction',
    },
    render: function () {
        var html = this.template(this.model);
        this.$el.html(html);

        return this;
    },
    primaryAction: function () {
        //Implementace v potomcích
    }
});
var row = new eco.Views.TableRow({
    model: someRow,
    tempalte: "#TableRow-tempalte"
});
$('body').append(row.render().$el);
```

Zdrojový kód 3: Ukázka vytvoření vlastního pohledu

Pohled Paper se poté stará o vykreslení předaného modelu. Pro jeho inicializaci mu stačí předat argumenty s výškou a šířkou plátna a modelem grafu a přiřadit element z DOM kam se plátno s grafem vykreslí.

JointJS má základní funkce a styly zobrazení entit a propojení, které je možné nakonfigurovat a jejich vzhled upravit podle požadavků tak, aby vypadali jako logická hradla a vodiče. Zvládá definici vstupních a výstupních portů, kterým je možné přiřadit omezení hlídající jaký port lze s čím připojit. V mém případě mám nadefinováno omezení, že z výstupního portu může vést více vodičů, ale do vstupního může vstupovat pouze jeden,

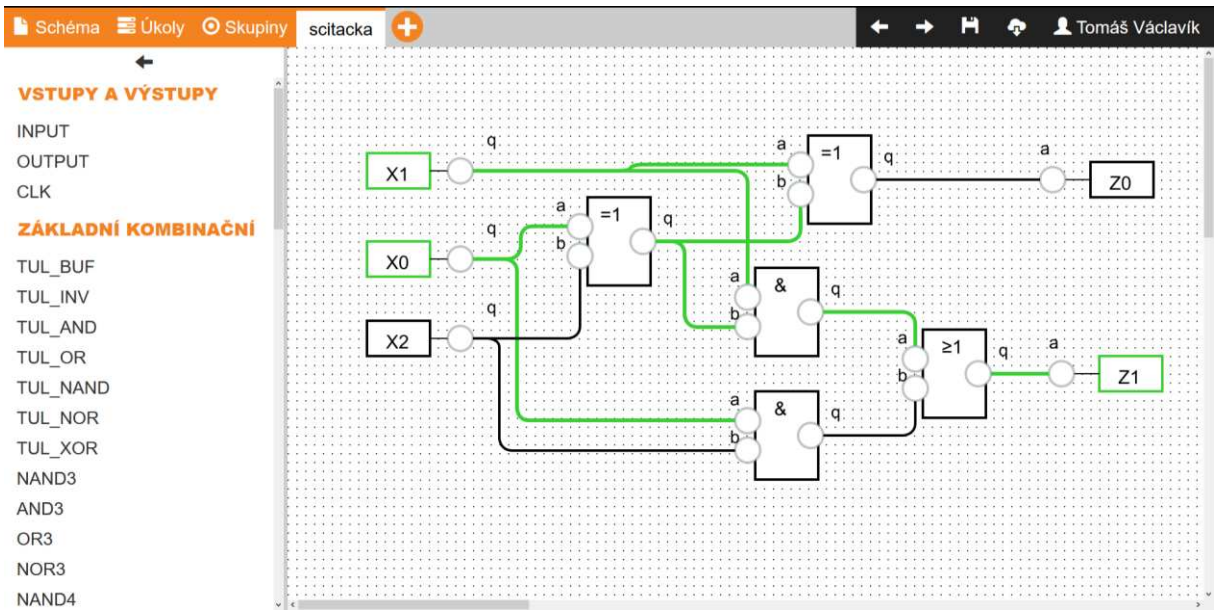
přičemž nelze spojit dva výstupní, nebo dva vstupní mezi sebou. Ukázka konkrétní definice hradla je v kapitole 3.1 Operační funkce entit ve Zdrojový kód 8.



Obrázek 2: Vizuální podoba simulace obvodu v editoru

2 Interaktivní editor se simulátorem

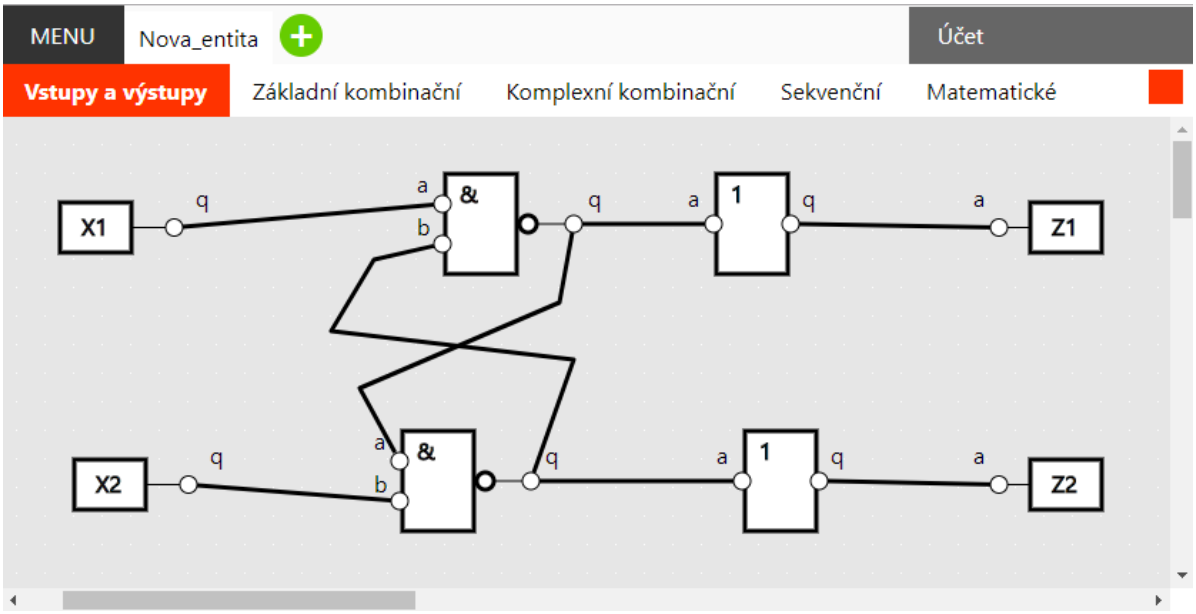
Mým úkolem bylo vytvořit vizuální simulaci navrženého schéma, která bude reagovat na změnu vstupů podle zadání uživatele. Jinými slovy uživatel může v reálném čase měnit hodnoty na vstupech a hned pozorovat změny v celém obvodu a na výstupech. Vizualizace simulace spočívá ve zvýraznění vstupů, výstupů a vodičů, které jsou ve stavu logické 1. Obrázek 3 ukazuje podobu editoru vizualizovanou simulací. Nahoře je umístěno hlavní menu aplikace (rozhraní je popsáno v kapitole 5 Uživatelské rozhraní) vlevo seznam entit, které zle umístit do schéma. Ve zbytku prostoru je zobrazeno schéma s interaktivní simulací a možností editování.



Obrázek 3: pohled studenta na editor s otevřeným schéma

2.1 Původní řešení a vizuální podoba editoru

Ve své práci navazuji na již vytvořený editor, který umožňoval podobným způsobem, metodou „táhni a pusť", vytvářet schéma. Z uživatelského a vizuálního hlediska měl několik nedostatků, které jsem zde také opravil. Zejména nevhodný způsob odstraňování hradel pomocí dvojkliku což nyní nebylo možné, protože u vstupních portů slouží právě kliknutí pro přepnutí logické úrovně a je tak nezbytné pro simulaci. Dvojklik by byl stále použitelný, nicméně by to neumožnilo rychle přepínat stavy na vstupech, nehledě na to, že by to bylo značně uživatelsky neintuitivní. Nový způsob odstranění je pomocí tlačítka s ikonou koše, které se zobrazí při najetí na entitu či jejím dotykem.



Obrázek 4: původní podoba editoru

Obrázek 4 demonstruje, jak aplikace vypadala, její vzhled prošel také řadou změn souvisejících s rozšířením uživatelského rozhraní, které je popsáno v kapitole 5 Uživatelské rozhraní. Co se týče samotného obvodu, tak jsem upravil vzhled portů, které jsou vizuálně téměř shodné se znakem negace a mohly být snadno zaměněny (zejména pokud jde o výstupní porty), také byly příliš malé pro ovládání na dotykových zařízeních.

2.1.1 Kolekce použitelných entit

Entity – hradla, vstupy, výstupy aj., která je možné do schéma přidat, jsou pro potřeby editoru definována v souboru **joint.shapes.mylib.js** jako modely odvozené od třídy Backbone modelu: **joint.shapes.basic.Generic**. V souboru je definována hierarchie entit v závislosti na podobnosti podle vzhledu, počtu a typu portů, nebo dle logické funkce. Byla

vytvořena základní třída **joint.shapes.mylib.Hradlo**, která obsahuje definici hradla, tj. obdélníkový blok, pozici a podobu popisku, umístění a CSS třídy vstupů a výstupů a defaultní nastavení chování vstupů a výstupů. Tuto definici jsem upravil o dříve popsané vizuální změny, například přidání tlačítka pro smazání. Od tohoto obecného hradla jsou odvozeny nové typy s konkrétním počtem portů a jejich umístění, popiskem a dalšími daty specifickými pro dané hradlo jako například vnitřní stav. Tím se dostávám k tomu nejzásadnějšímu rozšíření kolekce hradel, a to definice funkce operace hradla, která slouží pro simulaci.

2.2 Princip simulace

Realizované řešení simulace funguje na jednoduchém principu propagace signálu přes celý obvod postupně od vstupů. Při změně logické hodnoty na vstupu, kterou může uživatel přepínat, je tato změna propagována na všechny připojené výstupní vodiče. Při změně signálu na vodiči je vyvolána událost změny (change) a jsou vyhodnoceny všechny připojené entity. Entita reaguje na událost zavoláním propagační funkce. Následně pro vyhodnocení výstupního stavu zavolá funkci operace, která vykoná funkci dané entity (například NAND) a pro předané vstupní hodnoty vrátí výstupní ke všem svým výstupním portům. Nové hodnoty jsou dále propagovány na patřičné vodiče, které jsou připojeny na výstupní porty entity. Cyklus je opakován, dokud se nedojde na konec obvodu. Propagace končí samozřejmě buď na výstupu, nebo pokud není v entitě připojen žádný výstup. V případě zpětné vazby, pokud nedojde k ustálení, probíhá simulace stále dokola. Je totiž neustále volána propagační funkce. Tento přístup umožňuje vytvořit například astabilní klopné obvody, generátory pulzů a využívat zpětné vazby. Nicméně s obdobnými problémy jako u skutečných obvodů.

Jelikož implementace simulace využívá principy jazyka JS, zejména jeho systém událostí pro propagaci signálu obvodem a reakci na změnu signálu na vodičích. Jsou tyto události obstarávány asynchronně. Nelze tak zajistit pořadí v jakém se funkce operací vyvolají. Hazardní stavy zde mají stejný projev jako u skutečných obvodů, nelze předem určit v jakém stavu výstup bude. Nicméně tento editor je primárně určený pro simulaci kombinačních obvodů a synchronních sekvenčních obvodů.

2.3 Struktura editoru a simulátoru

Editor číslicových obvodů je jednou z nejdůležitějších stránek celé webové aplikace. Je dostupný pod adresou `/#schemas/id`, kde id nahraďte unikátním identifikátorem schéma, které je otevřeno.

Pro chod aplikace je třeba několik javascriptových souborů a knihoven, které jsou vloženy do stránky v hlavičce nebo na jejím konci v závislosti na jejich nezbytnosti a času kdy jsou využívány.

Stěžejní soubory třetích stran pro samotný editor jsou zejména soubory frameworku Backbone, knihovny jointJS a jejich závislosti. Z vlastních souborů vytvořených přímo pro tento projekt to je definice entit používaných v editoru umístěná v souboru `/scripts/joint.shapes.mylib.js`. Dále soubor `/src/helpers/util.js` obsahující kolekci užitečných funkcí používaných v celé aplikaci. Soubor `/src/modules/simmulator.js`, který obsahuje kód modulu pro simulaci obvodu a samotný soubor `/src/modules/schema.js` s Backbone modelem schéma a pohledy pro jejich zobrazování, a to nejen pro potřeby editoru, ale celé aplikace. Poslední nezbytná část logiky je v hlavním souboru aplikace `application.js`.

2.4 Otevření schéma v editoru

Aplikace umožňuje uživateli mít otevřeno více schémat najednou, více instancí editoru, každý s vlastní simulací. Pro otevření schéma je třeba zobrazit stránku s adresou `/#schemas/id` s id požadovaného schéma. Tato adresa je interní adresa aplikace spravovaná objektem **Router** z Backbone. Tam je pro potřebnou adresu definována obslužná funkce, která má v tomto případě zobrazit editor.

Toto je jediný způsob, jak schéma otevřít, ale velmi jednoduchý a funkční. Adresa vždy jednoznačně určuje konkrétní schéma a odkazy na toto schéma se dají jednoduše vytvářet. Drobný problém by to znamenalo, pokud bychom chtěli zobrazit editor bez předchozího vytvoření schéma, ale to v mém případě není potřebné. Případně lze vždy vytvořit nové schéma automaticky s vygenerovaným jménem. Na server by se uložilo až při požadavku na uložení schéma od uživatele.

Právě otevřené schéma je přidáno do seznamu otevřených schémat, mezi kterými lze přepínat. Při přepínání mezi otevřenými schématy již nedochází k dotazování se serveru. Zobrazené může být v editoru pouze jedno schéma v jeden čas a ostatní jsou skryty. Odkazy na otevřená schéma jsou poté zobrazena nahoře na hlavní liště a uložena v kolekci `openedSchemas`. Aktuálně otevřené schéma je pro lepší dostupnost uloženo v proměnné `activeSchemaModel`.

```
if (isSchemaOpen(parseInt(id))) {
    showSchema(openedSchemas.get(id));
}
else {
    var schema = new eco.Models.Schema({id: id});
    schema.fetch({
        success: function () {
            showSchema(schema);
        },
        error: function () {
            router.navigate('schemas', {trigger: true, replace: true});
            showSnackbar('Požadované schéma nebylo nalezeno!');
        }
    });
}
```

Zdrojový kód 4: část funkce `showOpenSchema` pro otevření schéma

Zdrojový kód 4 ukazuje část funkce `showOpenSchema`, která otevírá schéma. V první části, která zde není uvedena je nastavení klávesových zkratek pro ukládání, export a procházení historie. Podstatná část uvedená v ukázce je zobrazení schéma, pokud jsou k dispozici jeho data (bylo již načteno/otevřeno), je zobrazeno pomocí funkce `showSchema`. Pokud tomu tak není, je poslán dotaz na server a následně je zobrazeno pomocí stejné funkce jako v předchozím případě. Pokud uživatel zadá ID neexistujícího nebo nedostupného schéma, je přesměrováno na seznam schémat a zobrazeno upozornění.

Úkolem krátké funkce `showSchema` je rozhodnout, zda schéma se zadanými daty je již otevřené a v tom případě pouze skryje ostatní a toto zviditelní nebo pokud nebylo schéma ani zobrazeno je zavolána funkce `openSchema`, viz Zdrojový kód 5, která ho zobrazí.

Funkce nejprve vytvoří pohledu **Paper**, který je z knihovny JointJS, poté instance vlastního **Counteru**, vlastního **simulátoru** a použitého **správce historie**. Poté jsou načtena data schéma ze serveru funkcí `loadGraph`. Jako callback je předána funkce, která se vykoná po načtení dat, byť neúspěšném. Ta zajistí inicializaci a nastaví dříve vytvořené instance

Counteru, simulátoru a historie. Na konec je pohledu Paper přidána obsluha pro přidávání entit do schéma metodou „táhni a pusť".

```
function openSchema(schema) {
    var paper = eco.createPaper(schema, schemaContainer);
    var counter = eco.Utils.createSetCounter(0);
    var sim = new eco.Models.Simulation({paper: paper});
    var undomanager = new Backbone.UndoManager();
    schema.set('undomanager', undomanager);
    schema.set('sim', sim);
    schema.loadGraph(function () {
        var graph = schema.get('graph');
        graph.set('counter', counter);
        openedSchemas.add(schema);
        addOpenedPaper(schema, paper);
        showSchemaPaper(schema);
        setSchemaActive(schema);
        eco.Utils.inicilizeCounterbyGraph(counter, graph);
        sim.startSimulation();
        undomanager.register(graph);
        undomanager.startTracking();

        paper.$el.droppable({
            drop: function (event, ui) {
            var entityId = parseInt($(ui.helper).attr('data-entityid'));
                var foundEntity = entities.get(entityId);
                var entityName = foundEntity.get('name');
                eco.Utils.addEntityToGraph(entityName, {
                    x: ui.offset.left - schemaContainer.offset().left,
                    y: ui.offset.top - schemaContainer.offset().top
                }, graph, foundEntity);
            }
        });
    });
}
```

Zdrojový kód 5: kód funkce openSchema pro vytvoření pohledu

Nyní je schéma načteno a zobrazeno, funguje na něm simulace a je možné ho editovat.

2.5 Počítadlo pro unikátní názvy

V předchozí ukázce kódu se objevuje proměnná counter. Jde o počítadlo, které používám pro zajištění unikátních názvů entit ve schéma. To je důležité pro export do VHDL a serverovou simulaci, kdy názvy musí být vždy unikátní. Jedno počítadlo generované funkcí `createSetCounter` dovoluje čítat různé entity seskupené podle zadaného klíče, kterým je v mém případě typ entity, například mylib.TUL_AND nebo mylib.INPUT atp. Počítá pouze s celými čísly, unikátní název je tak kombinace název entity a generovaného čísla. Díky němu jsou například vstupy označené znakem X a tímto generovaným pořadovým číslem.

Čítač si pamatuje všechny použité hodnoty, tím umožňuje znovu použít již přidělenou a následně smazanou hodnotu. Zdrojový kód 6 ukazuje metodu čítače `getNextEmpty`, která vrací další volnou hodnotu pro zadaný klíč. Tuto funkci hojně využívám pro výše popsaný problém. Pokud uživatel entitu smaže ze schéma, je potřeba její hodnotu obnovit, pro to slouží metoda `removeOne`, která hodnotu pod zadaným klíčem odstraní ze seznamu použitých.

```
getNextEmpty: function(key){
    var set = counts[key];
    if(set !== undefined && _.size(set)>0){
        var last = start;
        var result;
        _.forEach(set, function(value){
            if (last===value){
                last++;
            }else{
                counts[key].push(last);
                counts[key] = _.sortBy(counts[key]);
                result = last;
                return false;
            }
        });
        if(result === undefined){
            counts[key].push(last);
            result = last;
        }
        return result;
    }else{
        counts[key] = [start];
        return start;
    }
}
```

Zdrojový kód 6: metoda čítače pro získání další hodnoty

2.6 Historie editoru

Další malou novinkou je uživatelsky přívětivá funkce kroků vzad a vpřed v editoru. Kroky vytvoření a smazání entit a vodičů jsou zaznamenávány a lze se v nich vracet v rámci jednoho sezení. Historie je uchovávána pouze v operační paměti, proto je ztracena při opětovném načtení aplikace. Pro její implementaci jsem využil knihovnu **Backbone.Undo.js**, která je pro Backbone přímo určená. Objekt **undoManager** (manažer historie) sleduje změny na registrovaném modelu [10]. Tím je v mém případě objekt Graph uchovávající data schéma (entity a vodiče). Manažera stačí pouze inicializovat, zaregistrovat hlídaný model a zapnout sledování změn.

Poté poskytnout ovládací prvky pro pohyb v historii. Ty jsem umístil na hlavní lištu jako tlačítka a přidal klávesové zkratky Crtl+z pro zpět a Ctrl+Shift+z pro pohyb vpřed.

2.7 Export schéma do VHDL

Tato funkce je v editoru již od počátku, nicméně bylo potřeba ji optimalizovat, opravit chyby a přizpůsobit ji pro použití se simulací na serveru. Například změnit vytváření signálů a správně mapovat hodnoty na výstup bez nutnosti použít entitu Buffer před každým výstupem. Ve výsledku jsem vytvořil nový objekt **VhdExporter**, který obsahuje celý proces převodu schéma z datové podoby v javascriptu do VHDL, to vyžaduje alespoň základní znalost VHDL jazyka [11][12].

```
VhdExporter.prototype.exportSchema = function(c_name_en, c_name_arch,
graph) {
    var elements = this.getAllEntities(graph),
        inputs = this.getInputs(graph),
        outputs = this.getOutputs(graph),
        inputOutputs = this.portsToVHDL(inputs, outputs),
        signals = this.getSignals(graph, elements),
        gates = this.gatesToVHDL(graph, elements),
        outputsMap = this.getOutputMap(graph, outputs);
    var vhdl = "library IEEE;\r\nuse IEEE.std_logic_1164.all;" +
        "\r\n\r\nentity " + c_name_en + " is\r\n" +
        "\tport (\r\n" + inputOutputs +
        "\r\n\t);\r\nend entity " + c_name_en + ";\r\n\r\n" +
        "architecture " + c_name_arch + " of " + c_name_en + " is\r\n" +
        signals + "\r\n" +
        "begin" +
        gates + "\r\n\t" +
        outputsMap.join(';\r\n\t') +
        ((outputsMap.length > 0)?";":"") +
        "\r\nend architecture " + c_name_arch + ";\r\n" +
        "\r\n";
    return vhdl;
};
```

Zdrojový kód 7: výchozí funkce pro export schéma do VHDL

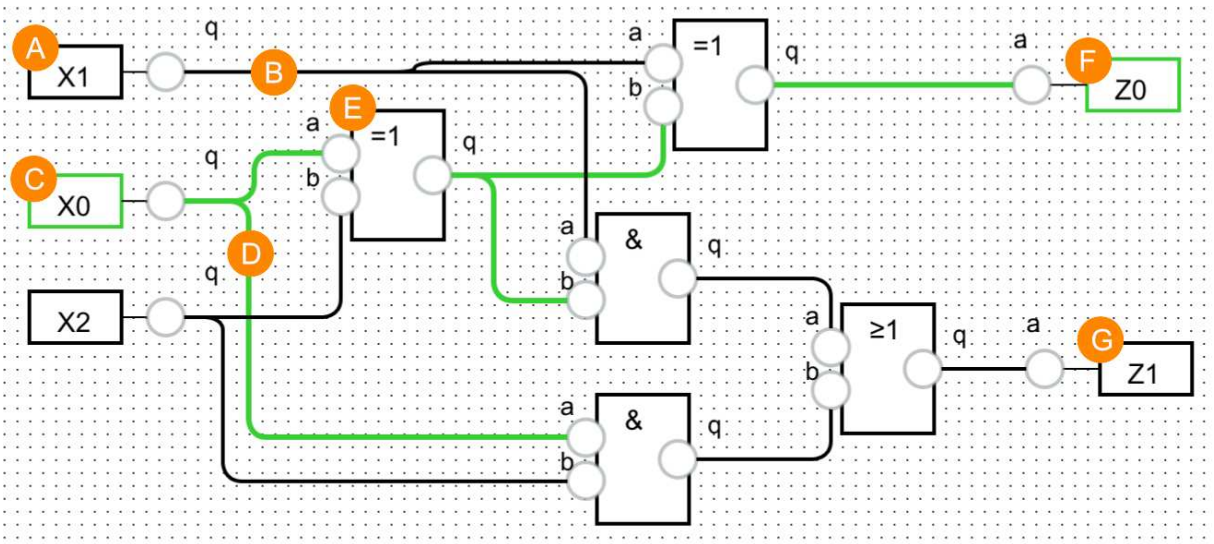
Export provede funkce `exportSchema`, viz Zdrojový kód 7, která vyžaduje název schéma (entity), název architektury (defaultně vždy **rtl**) a model **Graph** s daty schéma. Nejprve načtu do polí všechny elementy, vstupy a výstupy. Následně je použiji do dalších funkcí, které převedou jednotlivé části do VHDL: definici vstupů a výstupů, signálů, definici hradel a mapování jejich vstupů a výstupů a mapování signálů na výstupy. V následujícím řetězci je vše spojeno do struktury VHDL a řetězec je vrácen. Implementace jednotlivých metod není krátká, proto je zde nebudu uvádět. Najdete je v kompletních zdrojových souborech, které jsou k práci přiloženy v elektronické podobě.

3 Interaktivní simulace

Ke spuštění simulace je zapotřebí soubor `src/modules/simmulator.js` obsahující kód simulátoru, dále implementace požadovaných funkcí modely entit v knihovně `scripts/joint.shapes.mylib.js`. Knihovna obsahuje definici 46 entit od jednoduchých po komplexní, viz 3.2 Seznam knihovních entit.

Logika simulace s jejím nastavením a inicializací je koncentrována do jednoho modelu `eco.Models.Simulator` a je možné jí tak případně zaměnit za jiný model, s jinou implementací. Nicméně implementace funkce `operation`, kterou simulátor využívá, je součástí knihovny entit a je simulátorem vyžadována u všech entit.

Simulátor dále vyžaduje u ovladatelných vstupů implementovanou funkci `switchSignal` pro změnu signálu a funkci `isVisualyActive` pro informaci má-li být entita zvýrazněna (týká se vstupů a výstupů).



Obrázek 5: Podoba simulace s popisky

Obrázek 5 znázorňuje podobu simulace, zde jsou zeleně zvýrazněny vodiče, vstup a výstup se stavem logické 1.

A. vypnutý vstup ve stavu log. 0
B. Vodič ve stavu log. 0
C. Zapnutý vstup, ve stavu log. 1
D. Vodič ve stavu log. 1
E. Entita (hradlo XOR). U nich se stav nezvýrazňuje, mohou mít více výstupních portů

F. Výstup s log. 1

G. Výstup s log. 0

3.1 Operační funkce entit

Nejdůležitější funkcí pro simulaci je `operation`, která definuje logickou funkci entity. Jde v podstatě o mapování vstupů na výstupy. Hodnoty výstupů jsou vyhodnoceny definovanými funkcemi.

```
joint.shapes.mylib.TUL_OR = joint.shapes.mylib.Hradlo21.extend({
    defaults: joint.util.deepSupplement({
        type: 'mylib.TUL_OR',
        attrs: {
        '.label': { text: '≥1',ref: 'rect', 'ref-x': .3, 'ref-y': .1,
stroke: 'black'},
        '.jm': { text: 'a', ref: 'rect', 'ref-dx': -70, 'ref-dy': -70},
        '.jm2': { text: 'b', ref: 'rect', 'ref-dx': -70, 'ref-dy': -40},
        '.jm3': { text: 'q', ref: 'rect', 'ref-dx': 10, 'ref-dy': -60},
        }
    }, joint.shapes.mylib.Hradlo21.prototype.defaults),
    operation: function (p) {
        if (p['a'] < 0 || p['b'] < 0) return -1;
        return p['a'] || p['b'];
    }
});
```

Zdrojový kód 8: definice hradla OR

Funkce `operation` přebírá parametrem pole vstupů typu klíč hodnota, klíčem je název vstupního portu a hodnotou hodnota signálu na něm. Funkce může vracet buď přímo výstupní hodnotu, pokud má pouze jeden výstup nebo musí vrátit pole hodnot, kde klíčem je název portu a hodnotou hodnota signálu. Ke své činnosti může využívat jak datové položky objektu entity, tak její funkce.

Zdrojový kód 8 ukazuje definici hradla OR v souboru joint.shapes.mylib.js s kódem pro jeho vzhled v horní části (zůstává stejný tak, jak ho definoval můj předchůdce) a funkci `operation` dole. Zde je vidět jednoduchá implementace logické funkce hradla OR. U komplexnějších entit, jako klopných obvodů, využívám uchování vnitřních stavů v dodatečných proměnných a dalších pomocných funkcích definovaných u konkrétních entit. Entita hodin má například několik pomocných datových složek: `interval`, `signal`, `lastTime`, `tickOn`.

Povinné funkce, které nejsou u konkrétní entity uvedeny jsou definovány v jednom z jejích předků. Pro zajímavost uvádím implementaci funkce `operation` u složitější entity, a to paměti ARAMx16, viz Zdrojový kód 9. Ta využívá datové položky `memory` pro uchování hodnot uložených v paměti.

```
operation: function(p) {
  var key = createRAMKey(p,['a3', 'a2', 'a1', 'a0']);
  if (inputsAreInvalid(p)) {return 1;}
  if (portIsHigh(p,'ce')){
    if(portIsHigh(p,'we')){
      this.get('memory')[key] = _.pick(p,['d0', 'd1', 'd2', 'd3']);
    }
  }
  this.clk = !this.clk;
  var index = 0;
  var result =  _.mapKeys(this.get('memory')[key], function(value, key){
    index++;
    return 'q'+(index-1);
  });
  return result;
}
```

Zdrojový kód 9: funkce `operation` entity ARAMx16

3.2 Seznam knihovních entit

Každá entita, která je v editoru dostupná musí mít reprezentaci jak v JS s funkcí operace, tak ve VHDL knihovně entit, která se používá při serverové simulaci. Tyto dvě různorodé definice sloužící jinému účelu jsou v samostatných souborech. Seznam entit je uveden ještě na třetím místě, a to v databázi. Tam jsou uloženy použitelné entity, které se zobrazí v nabídce v uživatelském rozhraní.

V databázi jsou uloženy také kategorie entit. Každá entita patří do jedné z kategorií a v menu se vypisuje pod ní, jak si můžete všimnout v levém panelu na Obrázek 3. Kategorií je celkem 6, seřazeny od těch nejzákladnějších to jsou:

- Vstupy a výstupy
- Základní kombinační
- Komplexní kombinační
- Sekvenční
- Matematické
- Komplexní sekvenční obvody

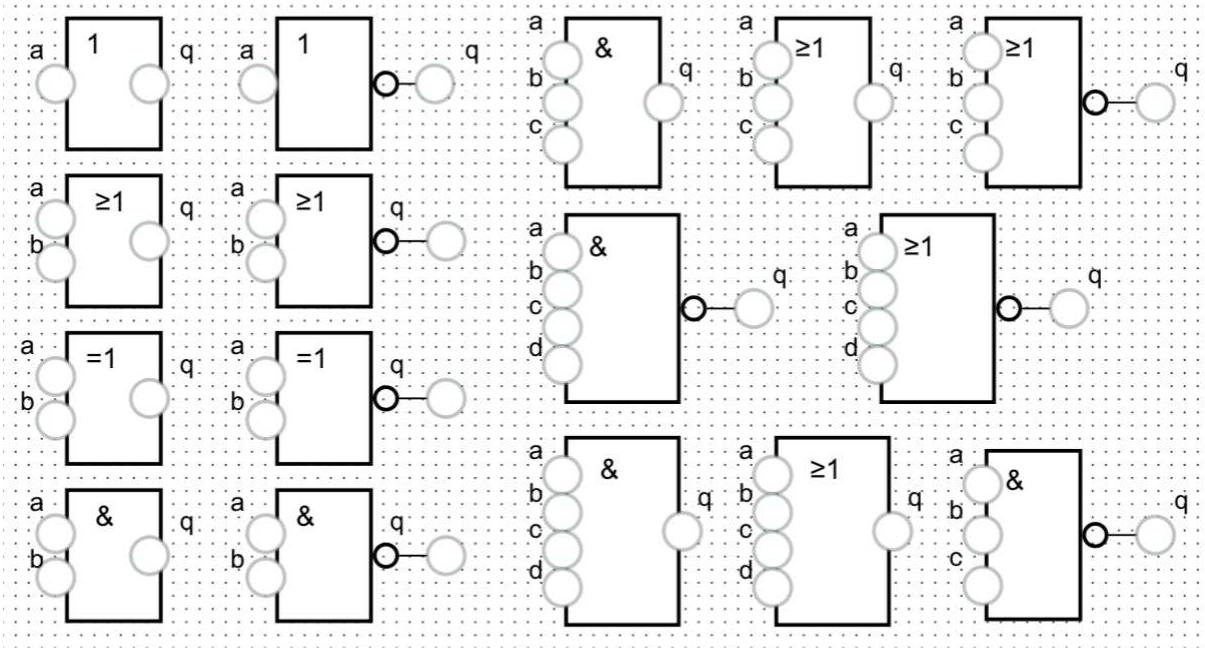
3.2.1 Vstupy a výstupy

Název této kategorie je všeříkající, obsahuje totiž vstup a výstup. Vstupu lze přepínat stav a výstup se podle stavu na něm zvýrazňuje.

3.2.2 Základní kombinační

Zde jsou zařazena jednoduchá kombinační hradla. Hradlo opakovač/zpožďovač (anglicky buffer, zkratka TUL_BUF) a invertor (logická negace). Hradla logického součinu, součtu a jejich negace jsou vyvedeny ve dvou, tří a čtyřvstupových provedeních. Exkluzivní logický součet je pouze dvouvstupový s další variantou s negovaným výstupem.

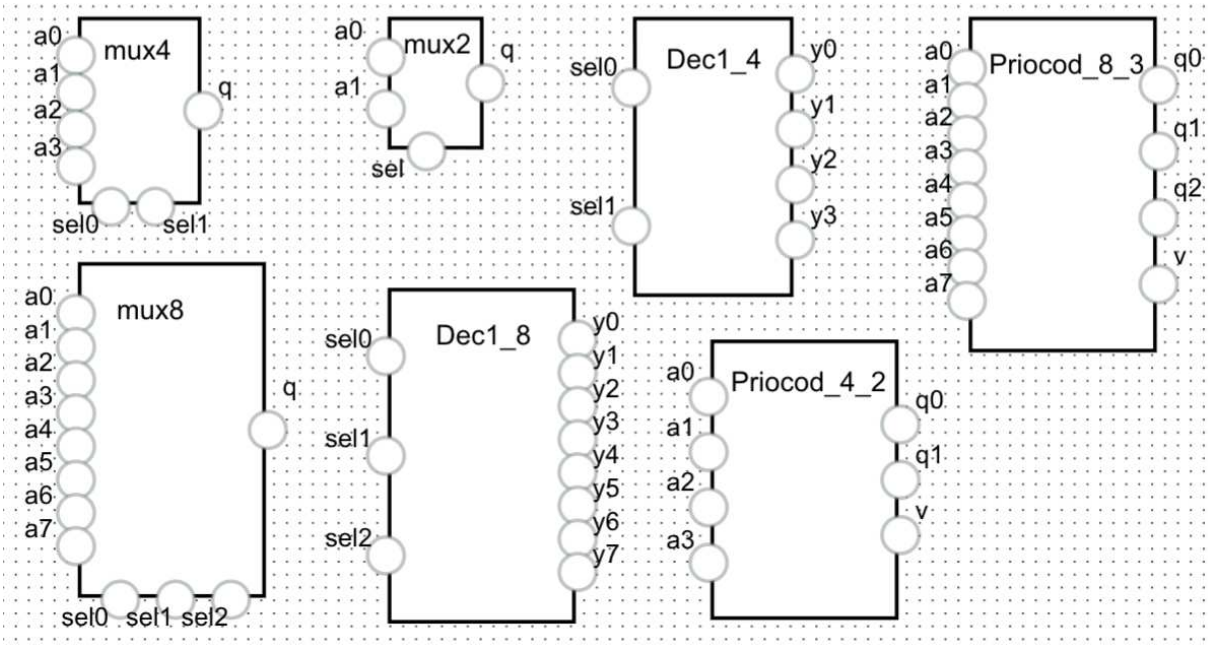
Základní dvouvstupová hradla mají v názvu předponu „TUL_“, to je z důvodu, aby jejich název nekolidoval s klíčovými slovy VHDL jazyka, kde „and“ realizuje logický součin. Implementace těchto hradel ve VHDL tato klíčová slova využívají. Více než dva vstupy jsou v názvu označena číslem, udávající počet vstupů. Celý seznam je: TUL_BUF, TUL_INV, TUL_AND, TUL_NAND, TUL_OR, TUL_NOR, TUL_XOR, TUL_XNOR, AND3, NAND3, OR3, NOR3, AND4, NAND4, OR4, NOR4.



Obrázek 6: přehled a podoba základních kombinační entity

3.2.3 Komplexní kombinační

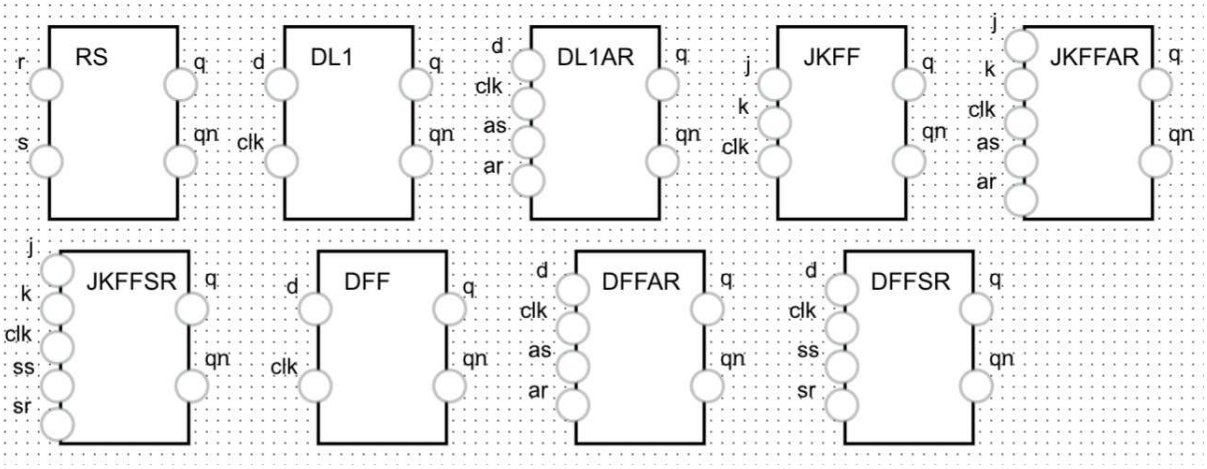
Tato kategorie obsahuje multiplexor, dekodér a prioritní kodér, konktrétně to jsou: MUX2, MUX4, MUX8, DEC14, DEC18, PRIOCOD42 a PRIOCOD83.



Obrázek 7: přehled a podoba komplexních kombinačních entit

3.2.4 Sekvenční

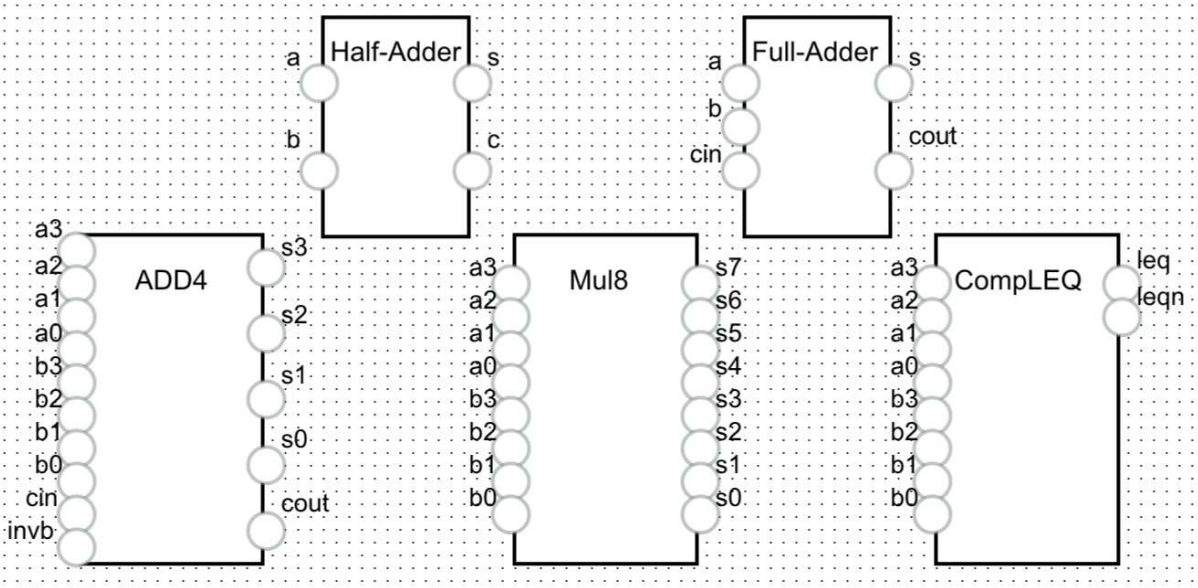
Mezi sekvenční patří klopné obvody: RS, DL1, DL1AR, JKFF, JKFFAR, JKFFSR, DFF, DFFAR a DFFSR. Jsou to klopné obvody reagující na hladinu (RS, DL1*) i na hranu (obsahují v názvu FF). Některá se synchronním (SR v názvu) a asynchronním resetem (AR v názvu).



Obrázek 8: přehled a podoba sekvenčních entit

3.2.5 Matematické

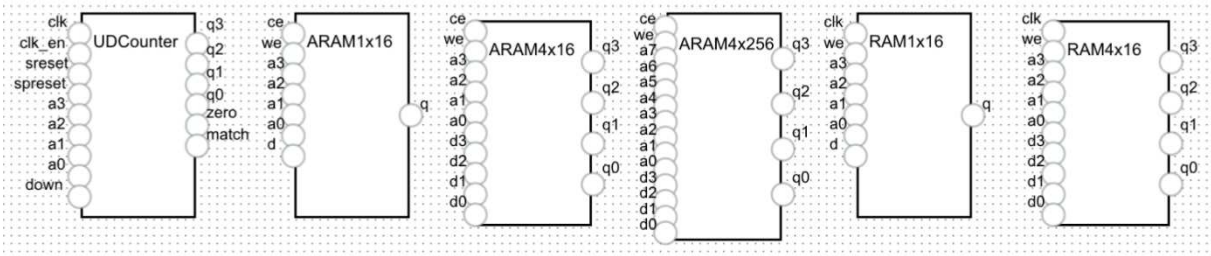
V této kategorii jsou sčítačky, násobička a komparátor menší rovno než: HALFADDER, FULLADDER, ADD4, MUL8 a COMPARATORLEQ.



Obrázek 9: přehled a podoba matematických entit

3.2.6 Komplexní sekvenční

Do této kategorie spadá čítač pulzů (obousměrný) a paměti, asynchronní a synchronní. Jsou to tyto: UPDOWNCOUNTER, ARAM1x16, ARAM4x16, ARAM4x256, RAM1x16, RAM4x16, RAM4x256. Paměti mají označení podle šířky slova 1 a 4 bity a počtu slov (16 a 256). Tomu odpovídá velikost adresy, viz



Obrázek 10: přehled a podoba komplexních sekvenčních entit

3.3 Inicializace simulace

Simulace je do aplikace integrována jako modul. Ke každému otevřenému schéma je při inicializaci přidána instance objektu `eco.Models.Simulation`. Ta vyžaduje předat

referenci na pohled **Paper**, který se stará o vykreslení schéma. Simulace může být poté zapnuta metodou `startSimulation`, případně zastavena pomocí `stopSimulation`. Opětovné zapnutí zachová nastavené hodnoty vstupů a stavů entit a naváže v simulaci.

Funkce `startSimulation` na začátku inicializuje signály všech entit a rozešle je celým obvodem pomocí funkce `initializeSignal`, viz Zdrojový kód 10. Poté nastaví obsluhu událostí pro reakci na změnu obvodu uživatelem. Konkrétně to jsou události: kliknutí na entitu vstupu (`cell:pointerclick`), změna vodiče (`change:source change:target`), odebrání entity či vodiče (`remove`) a změna signálu (`change:signal`).

```
initializeSignal: function (paper, graph) {
    var self = this;
    _.invoke(graph.getLinks(), 'set', 'signal', 0);
    paper.$el.find('.live').each(function () {
        V(this).removeClass('live');
    });
    var signalProducers = {
        INPUT: this.broadcastSignal,
        VCC: this.broadcastSignal,
        GND: this.broadcastSignal,
        CLK: this.startClock,
        Hradlo: this.startGate,
    };
    _.each(graph.getElements(), function (element) {
        var view = paper.findViewByModel(element);
        _.each(signalProducers, function (item, key) {
            (element instanceof joint.shapes.mylib[key])
            && item.call(self,element, {q: element.signal}, graph)
            && view.$el.toggleClass('live', element.isVisualyActive());
        });
    });
}
```

Zdrojový kód 10: Inicializace signálu

Při inicializaci je proveden nejprve reset signálů na všech vodičích nastavením do logické 0 a odebrání zvýraznění v celém obvodu. Dále je signál rozeslán na všechny výstupní vodiče vedoucí ze vstupů pomocí funkce `broadcastSignal`**,** kterou ukazuje Zdrojový kód 12 a vstup je podle signálu také zvýrazněn.

3.4 Stav, signál a jejich změna

Pro přenos výstupních stavů slouží vodiče, ty si uchovávají svou hodnotu stavu v proměnné **signal**, dále budu o nich mluvit jako o signálu, který je tak záležitostí vodičů. Nicméně je nastavován připojenými entitami. Signál na vodičích slouží k přenosu stavu na

další připojené entity a také se podle něj zvýrazňují vodiče s log. 1. Funkce inicializace signálu nastaví stavy všech entit ve schéma najednou pro počáteční nastavení a změní signál na všech připojených výstupních vodičích. O další propagaci signálu se již postará obsluha událostí změny signálu (`change:signal`).

3.4.1 Změna stavů

Vstupy jsou ovladatelné a obsahují funkci `switchSignal`, která změní patřičným způsobem jejich stav. Z log. 1 se stav změní na log. 0 a naopak. Změnit stav na vstupu lze kliknutím na entitu vstupu, což zajišťuje událost `cell:pointerclick` na objektu **Paper** a interně volá právě funkci `switchSignal` a poté propagaci signálu. Jediným dalším ovladatelným prvkem jsou hodiny, u kterých lze kliknutím zapnout nebo vypnout jejich funkci – generování signálu. Hodiny mají definovaný interval (frekvenci) se kterým mění výstupní signál automaticky.

```
window.setInterval(function () {
    if (clock.tryTick()) {
        self.broadcastSignal(clock, {q: clock.signal}, graph);
    }
}, clock.interval);
```

Zdrojový kód 11: tělo funkce pro zapnutí simulace hodin

3.4.2 Vyslání a propagace signálu

Funkce `broadcastSignal` (Zdrojový kód 12) je využívána při jakékoli změně stavu na entitě pro rozeslání signálu na připojené vodiče. Nejprve jsou nalezeny všechny výstupní

```
broadcastSignal: function (gate, signals, graph) {
    var outLinks = graph.getConnectedLinks(gate, {outbound: true})
        .map(function (x) {
            return x;
        });
    var result = _.reduce(outLinks, function (result, item) {
        var key = item.get('source').port;
        (result[key] || (result[key] = [])).push(item);
        return result;
    }, {});
    _.each(result, function (wires, key) {
        _.defer(_.invoke, wires, 'set', 'signal', signals[key]);
    });
    return true;
},
```

Zdrojový kód 12: vyslání signálu

vodiče. V druhém kroku jsou seskupeny do polí podle názvu výstupního portu. Následně je pro tyto vodiče nastavena nová hodnota signálu. Nové signály jsou této funkci předány jako argument `signals`, který obsahuje pole, kde klíčem je název portu a hodnotou je nový stav. Tyto nové stavy pochází z funkce `operation` konkrétní entity.

3.4.3 Změna signálu na entitě

Při změně hodnoty signálu na vodiči funkcí `broadcastSignal` dojde k vyvolání události `change:signal`. Ta je obsloužena funkcí, která vyhodnotí, co je k vodiči připojeno a podle toho signál zpracuje. V obsluze této události je nejprve nastaveno zvýraznění vodiče. Poté je získána entita, do které vodič vstupuje. Pokud se jedná o výstup, je také zvýrazněn podle hodnoty signálu a nic dalšího se neprovádí, neboť zde signál končí.

```
var ports = {};
_.each(gate.ports, function (x) {
    ports[x.id] = eco.Utils.notConnectedInputDefault;
});
_.chain(_.sortBy(self.paper.model.getConnectedLinks(gate, {inbound:
true}), function (x) {
    return x.get('target').port;
})).groupBy(function (wire) {
    return wire.get('target').port;
}).map(function (wires) {
    var inSignal = Math.max.apply(this, _.invoke(wires, 'get',
'signal'));
    ports[_.first(wires).get('target').port] = inSignal;
    return inSignal;
}).value();

var outputs = {};
var ops = gate.operation.apply(gate, [ports]);
if (_.size(ops) > 0) {
    _.each(ops, function (value, key) {
        outputs[key] = value;
    });
} else {
    outputs["q"] = ops;
}
self.broadcastSignal(gate, outputs, self.paper.model);
```

Zdrojový kód 13: funkce onSignal pro vyhodnocení změny signálu na entitě

Další zpracování se provádí pouze pro hradla, pro která je vyvolána funkce `onSignal`, viz Zdrojový kód 13. Ta provede zjištění hodnot všech vstupů a uloží je do pole, následně jej předá funkci `operation` patřičné entity a získá z ní pole výstupních stavů. To dále předá funkci `broadcastSignal`, která provede rozeslání.

Pro zjišťování vstupních signálů do entity jsem využil funkci `getConnectedLinks` objektu **Graph**, která pro zadaný element vrátí všechny jeho připojené vodiče, přičemž dokáže filtrovat podle typu: vstupní nebo výstupní. Tyto vodiče však vrací v poli seřazené podle pořadí připojení k entitě a nikoli podle portů, jak by bylo potřeba. Bez korekce by to vedlo k nekonsistentnímu chování i pokud by byl obvod stejně zapojen, jen v jiném pořadí zapojení. Tento problém jsem vyřešil využitím objektu pro předání vstupů a nastavením názvů portů jako klíče hodnot, místo obyčejného pole.

3.4.4 Zvýraznění vodiče

Zvýraznění vodičů je nastavováno pomocí CSS třídy **live**, která nastavuje barvu tahu (viz Obrázek 5, zelená barva). Funkce na následující ukázce nastavuje třídu live v závislosti na hodnotě signálu. Pokud je hodnota signálu větší než nula, je to považováno za log. úroveň 1 a vodiči, respektive vstupu, je přidána třída live a je tím zvýrazněn. Jinak je tato třída vodiči odebrána.

Speciální zvýraznění pomocí třídy invalid se použije, pokud je hodnota signálu menší než nula, ta je nastavována v ojedinělých případech jako zvýraznění zakázaného stavu, například u RS klopného obvodu. Zvýraznění je červené barvy a je definováno v CSS třídě invalid.

```
toggleLive: function (model, signal, paper) {
if (paper) {
  try {
  if (paper.findViewByModel(model))
  V(paper.findViewByModel(model).el).toggleClass('live', signal > 0);
  V(paper.findViewByModel(model).el).toggleClass('invalid', signal < 0);
  } catch (ex) {}
}},
```

Zdrojový kód 14: nastavení třídy pro zvýraznění

3.5 Výkonost simulace

Pro zjištění výkonnosti celé simulace, závislosti počtu hradel a propojení na rychlosti simulace, jsem provedl několik měření. Měřil jsem čas (v ms), který zabere propagace signálu od vstupu na výstup. Vždy jsem do schéma umístil jeden vstup a jeden výstup a mezi ně různý počet hradel. Celkem jsem vytvořil 10 schémat. Pro každé různé zapojení jsem opakoval měření šedesátkrát. Tabulka 1 zobrazuje maximální, minimální a průměrné naměřené hodnoty časů.

Začal jsem prvním měřením se zapojením jednoho hradla NOT, poté pěti hradly zapojenými sériově, následně 10 a 20. V dalších měřeních jsem použil stejné počty, ale zaměnil hradla negace za D klopný obvod. Jak můžete vidět v tabulce, časy jsou téměř shodné. U složitějšího klopného obvodu jsou vyšší maximálně o několik procent. Můžete si povšimnout, že ani rozptyly mezi maximem a minimem nejsou příliš veliké přesto, že se zvětšují s rostoucím počtem hradel. Zajímavý je také nárůst času při rostoucím počtu hradel. Pro 20 hradel negace je průměrný čas přepočítaný na jedno 15,5 ms, přitom naměřená průměrná hodnota pro jedno hradlo negace je 9,28 ms. Čas tak neroste lineárně, ale spíše geometrickou řadou.

Tabulka 1: naměřené časy propagace signálu od vstupu k výstupu (v ms)

	1 NOT	1 MUX	5 NOT	10 NOT	20 NOT	1 DL	5 DL	10 DL	20 DL	20 AND
maximum	20,04	14,50	40,32	109,26	386,80	17,02	54,56	131,76	374,56	165,48
minimum	5,76	5,68	24,46	75,54	262,70	6,02	25,66	80,06	265,92	53,20
průměr	9,28	9,01	31,00	90,55	309,97	9,40	33,35	98,63	315,70	110,76

Zajímavý poznatek vychází také z posledního testu, kde jsem použil 20 hradel AND, které byly připojeny paralelně. Hradla vytvořila 5 úrovní. Ze vstupu vedly vodiče do osmi dvouvstupových hradel AND z nich do 5 dalších, další úroveň tvořily 4 hradla, která se spojovala do dalších dvou a v poslední vrstvě zbývá jeden AND, který vede přímo do výstupu. Nejkratší cesta od vstupu k výstupu tak obsahuje 5 hradel. Průměrný čas zde je podobný času při měření 10 hradle NOT, či klopných obvodů zapojených sériově. Ale minimum je mnohem menší a maximum větší. Rozptyl časů mezi jednotlivými měřeními zde odpovídá rozptylu u 20 sériově zapojených hradel. Nicméně simulace takto vytvořeného obvodu (s 20 entitami) se jeví rychlejší.

Naměřené hodnoty se vztahují pouze k zobrazení simulace, celý editor pracuje stále plynule a uživatelské rozhraní nijak nezamrzá. Přesto, že u větších schémat (od 20 hradel výše) můžou být kroky a propagace signálu patrné, není to na škodu. Uživatel tak vidí, jak jsou hodnoty na vodičích přímo měněny. Zpoždění na hradlech je v reálných obvodech také přítomné, byť ne tak markantní. Editor se simulací je tak velmi dobře použitelný pro menší schéma a při paralelním zapojení se časová ztráta na jedno hradlo snižuje.

Výkon simulace závisí na konkrétním klientském stroji, já jsem testy prováděl na starším přenosném počítači s procesorem Intel Core i5-3230M, 16 GB RAM a operačním systémem Microsoft Windows 10. Jako prohlížeč jsem použil Mozilla Firefox ve verzi 57.

4 Webová aplikace

Nasazení na školní infrastrukturu a využití pro výuku vyžaduje rozšířit uživatelské rozhraní a celou aplikaci o řadu nových funkcí, pohledů (například výpis zadaných úkolů) a vytvoření databáze.

Přihlášený uživatel může vytvářet a upravovat vlastní schéma a ukládat je trvale na serveru. Pokud je uživatel student, je mu umožněno prohlížení jemu zadaných úkolů a přidávat k nim řešení, a tím je odevzdávat. Jako řešení bylo původně zamýšleno odevzdat schéma vytvořené v aplikaci, ale později jsem doplnil také možnost odevzdat kód VHDL ze souboru. Pro to, aby mohl student dostat úkol, musí být vyučujícím zařazen do konkrétní skupiny.

Uživatel s rolí vyučujícího má možnosti ještě rozsáhlejší. Může vytvářet skupiny studentů a spravovat jejich členy, zadávat studentům úkoly podle definovaných zadání a prohlížet odevzdaná řešení, případně vyhodnotit jejich správnost.

4.1 Platforma aplikace

Část aplikace je zpracovávána na serveru, zejména výchozí bod poskytující JS zdrojové soubory a API. Podstatná část však běží na klientské straně. Aplikace je nově postavená od začátku za využití frameworku Backbone.js (viz kapitola 1.4 Backbone framework). Z původních komponent využívá pouze základ editoru schémat a definici entit, které jsem ale upravil pro nové požadavky.

Jako webová aplikace, běží ve webovém prohlížeči u klienta a potřebná data ze serveru jsou buď součástí kódu samotné aplikace a načtena společně s ní, nebo načítána asynchronně. Tak jsou získávána uživatelská data jako například schéma, úkoly, zadání, skupiny a další. Pro poskytování těchto dat jsem vytvořil REST API na serveru, které je popsané v čísti 6.2 API.

4.2 Struktura webové aplikace

Hlavním vstupním bodem je sobor index.php, jde o krátký script, který provádí jednoduché směrování za využití PHP frameworku Slim, ale o něm v kapitole 6 Serverová část.

Důležité je že server poskytuje několik podstránek, a to hlavní aplikaci dostupnou pro všechny přihlášené uživatele, stránky přístupné pouze uživateli s rolí vyučujícího (pod

adresou /teacher), podstránka pro přihlášení (/login), respektive odhlášení (/logout) a stránky pro správu profilu. Podle adresy je tedy zvolen patřičný obsah. V podstatě je vybrána webová aplikace, která se uživateli zobrazí, přičemž některé JS soubory jsou společné pro více podstránek.

4.2.1 Adresářová struktura

Zdrojový kód 15 zobrazuje vybranou adresářovou strukturu aplikace včetně serverového API (ve složce api/). Scripty webové aplikace jsou ve složce src.

```
index.php
scripts
     joint.shapes.mylib.js
src
     helpers
          counters.js
          formaters.js
          mappers.js
          snackbar.js
          util.js
          vadidators.js
     modules
          entities.js
          files.js
          generic.js
          group.js
          homeworks.js
          main.js
          modal.js
          schema.js
          simulator.js
          student.js
          tasks.js
          users.js
          vhdlExporter.js
     application.js
     application_teacher.js
     structure.js
schemas
config
     config.php
commmon
     database.php
     mail.php
     task_validation.php
     test.php
     utils.php
templates
     templates
          admin.html
          groups.html
          homework.html
          schemas.html
          tasks.html
     footer.php
     head.php
     header.php
     homepage.php
     login.php
     profile.php
     register.php
     scripts.php
     teacher.php
     templates.php
vendor
api
     index.php
     routes
          auth.php
          entities.php
          files.php
          groups.php
          homeworks.php
          scehmas.php
          students.php
          teacher.php
          users.php
assets
```

Zdrojový kód 15: Adresářová struktura

4.3 Klientská aplikace

Server poskytne potřebný kód, JS soubory, které jsou umístěny v hlavičce, respektive patičce stránky a HTML kód šablon používaný pro vykreslování pohledy.

4.3.1 Datová struktura

První soubor structure.js definuje datovou strukturu Backbone aplikace, viz Zdrojový kód 16. Členění aplikace, které jsem zvolil vychází z použitého frameworku. Do této struktury jsou při načítání aplikace a během její činnosti ukládány používané objekty. Objekt také zavádí jmenný prostor „eco“ a zlepšuje přehlednost kódu.

```
eco = {
    Models: {},
    Collections: {},
    Views: {},
    Formaters: {},
    Validators: {},
    Mappers: {},
    Utils: {},
    basedir: '',
    ViewGarbageCollector: {},
    buttons: {},
    selectors: {},
    start: function () {}
    Router: null,
    activeSchemaModel: null,
};
```

Zdrojový kód 16: datová struktura aplikace

Veškeré definované modely jsou pod položkou `eco.Models`, kolekce pod `eco.Collections` a pohledy v `eco.Views`.

Položka `eco.Formaters` obsahuje sadu funkcí, které přebírají jako parametr model a vrací objekt s formátovanými položkami modelu, například datum vytvoření úkolu v českém formátu. Jsou využívány při předávání dat do šablony k vykreslení. Pod `eco.Mappers` jsou uloženy funkce pro získávání hodnot z DOM, typicky z formulářů a jejich uložení do konkrétního modelu. Položka `eco.utils` obsahuje pomocné funkce, které jsou využívané napříč celou aplikací. Položka `eco.basedir` obsahuje informaci o výchozím umístění aplikace na serveru a využívá se jako část adresy umístění API. Hodnota je inicializována z PHF. Díky tomu je umožněno jednoduše měnit umístění aplikace a jejího API v rámci serveru, čehož bylo v praxi využito.

Další položka `eco.ViewGarbageCollector` obsahuje objekt, který uchovává použité pohledy a dovoluje je ve správný okamžik, když přestanou být využívány, smazat.

`Buttons` a `slectors` obsahují CSS identifikátory pro nalezení tlačítek, respektive některých částí aplikace v DOM javascriptem. Proměnná `eco.activeSchemaModel` slouží pro uchování reference na aktuálně otevřené schéma napříč celou aplikací. `Eco.Router` je Backbone.Router objekt, který obsahuje definici všech **route** (cest/adres) a k nim definované obslužné funkce. Zdrojový kód 17 prezentuje routy ve studentské části aplikace.

Poslední položkou, která vše spouští je funkce `eco.start` a je přímo volána na konci stránky po načtení všech požadovaných scriptů.

```
'': 'home',
'schemas': 'showSchemas', //seznam schémat uživatele
'schemas/new': 'schemaCreateNew', //vytvoření nového schéma
'schemas/:id/vhdl': 'schemaExportVhdl', //exportuje schéma do souboru
'schemas/:id': 'openedSchema', //otevře schéma - editor
'schemas/:id/edit': 'showSchemaEdit', //upraví údaje schéma
'students/groups': 'showUserGroups', //seznam skupin studenta
'homeworks': 'showHwList', //seznam úkolů
'homeworks/:id': 'showHwDetail', //detail úkolů
'*path':  'defaultRoute', // defaultní: error 404
```

Zdrojový kód 17: směrování ve studentské části aplikace

4.3.2 Spuštění aplikace

V tuto chvíli je již připravena struktura tříd a objektů a je možné vytvářet jejich instance. O nalezení pohledu k zobrazení se postará směrovací (Router) systém z Backbone.

Obslužná funkce zpravidla připraví modely, pohledy a další pomocná data pro zobrazení požadované stránky. Data modelů v kolekcích jsou načtena pomocí funkce `fetch` asynchronně ze serveru přes serverové REST API. Poté jsou vytvořeny pohledy s konkrétním nastavením. Typicky je pohledu potřeba nastavit šablonu, objekt modelu, případně kolekci. Pohled poté metodou `render` převede data pomocí šablony do HTML kódu, který je následně přidán do DOM stránky. Kdykoli jsou poté data modelu načtena nebo změněna je obsah pohledu automaticky přepsán. Zdrojový kód 18 ukazuje jednoduchou obsluhu směrovací cesty (anglicky route) pro zobrazení seznamu schémat, které může uživatel otevřít. Nejprve je zobrazen blok, kam bude obsah umístěn, poté je vytvořena instance pohledu. Jako parametr jí je předán konfigurační objekt s šablonou seznamu

a šablonou řádku seznamu, formátovací funkce, kolekce schémat, která byla již načtena při startu aplikace a pole CSS tříd sloužící pro funkci filtrování seznamu. Nakonec je pohled vykreslen a přidán na stránku.

```
function showSchemas() {
    main_tab.show();

    var view = new eco.Views.SchemasOpenList({
        template: "#schemasOpenList-template",
        itemTemplate: "#schemasOpenItem-template",
        formater: eco.Formaters.SchemaSimpleFormater,
        collection: schemas,
        searchNames: [
            'list-name',
            'list-architecture',
            'list-created'
        ]
    });
    main.append(view.render().$el);
}
```

Zdrojový kód 18: obslužná funkce routy pro zobrazení seznamu uživatelových schémat

4.4 Univerzální pohledy

Spousta pohledů má společné prvky, proto jsem pro ně vytvořil společné předky. A jsou umístěny v souboru src/modules/generic.js. Soubor zahrnuje univerzální seznam (eco.Views.GenericList) a k němu univerzální prvek seznamu (eco.Views.GenericItem). Univerzální detail modelu (eco.Views.GenericDetail), který zobrazí data jednoho modelu. Nakonec univerzální formulář (eco.Views.GenericForm), který slouží pro vytvoření nového modelu a změnu dat existujícího.

Pro ukázku jsem zvolil seznam, který byl použit jako předek také pro pohled v předchozí ukázce (Zdrojový kód 18Zdrojový kód 18). Zdrojový kód 19 představuje strukturu univerzálního seznamu, jsou tam vidět definice datových složek a funkce, Implementace funkcí je příliš rozsáhlá pro uvedení v této ukázce.

Tento pohled zajistí vykreslení kolekce modelů jako seznam podle šablony, typicky jako tabulku. V šabloně jsou uvedeny názvy sloupků v záhlaví tabulky. Pro jednotlivé položky je použita šablona pro prvek (`itemTemplate`). Dále objekt zajišťuje potvrzovací dialog pro mazání položek ze seznamu a filtrování seznamu podle zadaného klíčového slova.

```
eco.Views.GenericList = Backbone.View.extend({
initialize: function (opts) {
  this.title = opts.title || "";
  this.noRecordsMessage = opts.noRecordsMessage
    || 'Zatím zde nejsou žádné záznamy.';
  this.template = _.template($(opts.template).html());
  this.itemTemplate = opts.itemTemplate;
  this.itemView = opts.itemView || eco.Views.GenericItem;
  this.formater = opts.formater || eco.Formaters.GenericFormater;
  this.collection = opts.collection;
  this.searchNames = opts.searchNames || ['list-one'];
  this.deleteConfirm = _.merge({}, {…}, opts.deleteConfirm);
  this.renderLoading();
  this.listenTo(this.collection, 'sync', this.render);
  this.afterInitialization();
  this.vent = opts.vent;
  this.uniqueId = opts.uniqueId || '';
},
renderLoading: function () {},
afterInitialization: function () {},
events: {},
renderOne: function (item) {},
render: function () {},
deleteItem: function (event) {},
});
```

Zdrojový kód 19: struktura pohledu univerzálního seznamu `genericList`

5 Uživatelské rozhraní

Grafické rozhraní aplikace je rozděleno v klasickém pohledu do dvou částí, horního panelu pro menu a samotného obsahu. Horní panel zůstává pro různé pohledy stejný, mění se obsahová část stránky. Výjimku tvoří tlačítka specifická k otevřenému schéma, ta jsou viditelná, pokud je aktivní pohled se schéma. Jde o uložení a export do VHDL a ovládání historie změn. Lišta je také rozdělena na 3 části, vlevo je hlavní menu, které se liší podle role přihlášeného uživatele. Vpravo je informace o přihlášením uživateli a volby pro přihlášení/registraci atp. Ve zbývajícím prostoru uprostřed jsou umístěny ouška otevřených schémat. Specifická tlačítka pro schéma jsou poté umístěna vpravo, vedle informace o účtu.

Obsahová část je omezena na šířku kontejnerem a je centrována. Pohled pro schéma je trochu specifický, jelikož je na celou šířku okna prohlížeče a rozděluje obsahovou část ještě na dvě části. Vlevo je nabídka entit a vpravo samotný editor. Celé to znázorňuje Obrázek 3.

Podle role uživatele se přístup do aplikace dělí na tři úrovně, obyčejný uživatel, student a vyučující. Speciální úrovní je administrátor, který má pravomoc nastavovat role vyučujícího.

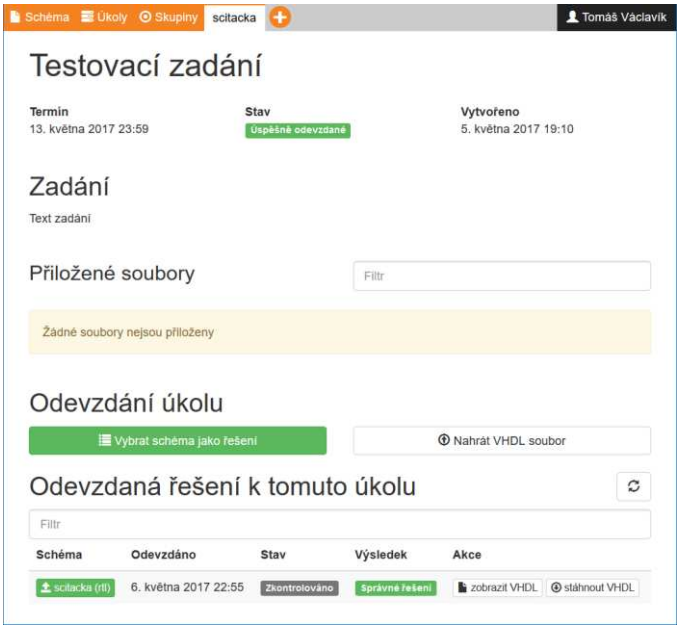
5.1 Uživatel

Má možnost pouze vytvářet a ukládat schéma na serveru. Má k dispozici omezenou nabídku uživatelského rozhraní. Může zobrazit seznam svých schémat viz Obrázek 11, vytvořit nové schéma, upravit údaje existujícího, otevírat, mazat, ukládat a editovat vlastní schéma. Také schéma exportovat do VHDL souboru.

Obrázek 11: Uživatelské rozhraní – seznam schémat a přidání nového – student

5.2 Student

Studen má stejné možnosti pro zobrazení schémat jako běžný uživatel. Student může schéma také smazat, ale pouze pokud již nebylo odevzdáno jako řešení úkolu. Oproti běžnému uživateli má studen navíc v nabídce přehled skupin, ve kterých je členem a stránku s úkoly, kde má seznam zadaných úkolů. Ke každému úkolu si může prohlédnout jeho zadání, odevzdat své řešení a vidí stav a výsledek automatické kontroly odevzdaných řešení.

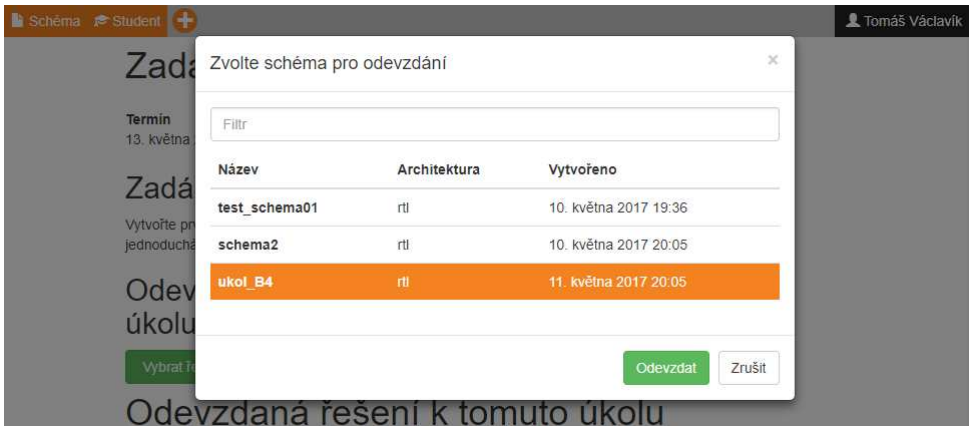


Obrázek 12: Detail úkolu studenta

5.2.1 Odevzdání úkolu

Další položka menu je seznam zadaných úkolů. Studen vidí název a datum odevzdání úkolu, údaj o skupině, ke které patří a stav odevzdání úkolu. V detailu (Obrázek 12) je navíc vidět seznam již odevzdaných schémat jako řešení. Každá změna ve schéma má-li být odevzdána a zkontrolována, musí být samostatně odevzdána, přičemž jako konečné řešení se bere první správné, respektive poslední odevzdané řešení. Znemožnění přidávat řešení po úspěšném odevzdání je nastavitelné v konfiguraci serveru.

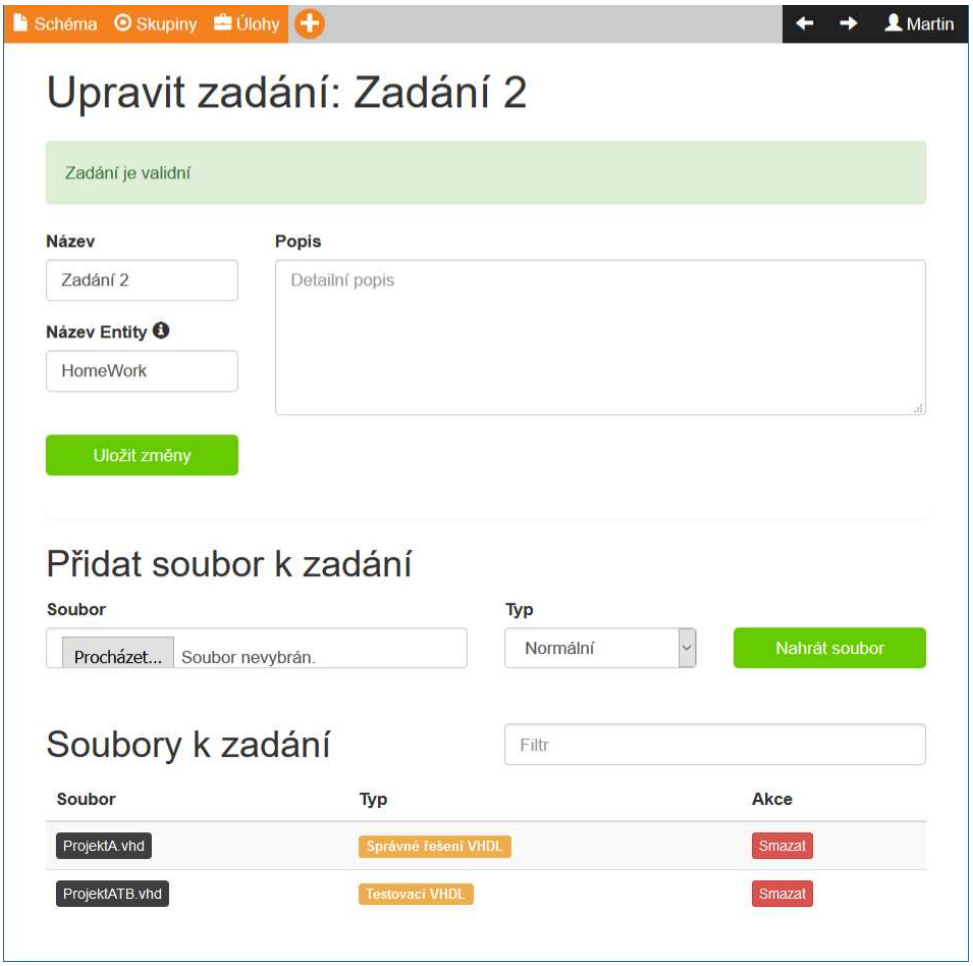
Pro odevzdání schéma jako řešení musí student kliknout na tlačítko vybrat řešení v detailu úkolu a zobrazeném okně zvolit schéma (vybrané je podbarveno oranžově), následně potvrdit odevzdání tlačítkem odevzdat. Dokud není odevzdané řešení automaticky zkontrolováno, lze ho ještě smazat, nicméně kontrola se spouští téměř okamžitě.



Obrázek 13: výběr schéma k odevzdání

5.3 Vyučující

Vyučující má, kromě zobrazení schémat jako běžný uživatel, podstránku /teacher, kde má možnosti v nabídce úlohy a skupiny. V úlohách (adresa /teacher#tasks) je seznam zadání a formulář pro vytváření nových. Tato zadání vyučující přiděluje studentům jako úkoly. Může



Obrázek 14: detail zadání

tak díky tomu používat stejné zadání vícekrát a jedno zadání přiřadit více studentům. V seznamu je uveden pouze název zadání a datum jeho vytvoření. Detail zadání je pod adresou /teacher#tasks/:id (s konkrétním id zadání. Vyučující může své zadání smazat, pokud ještě nebylo nikomu přiřazeno.

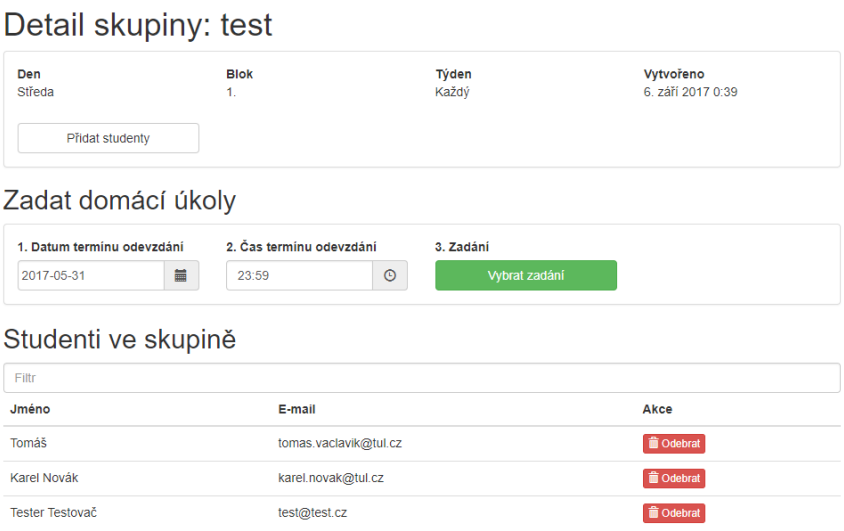
5.3.1 Detail zadání

V detailu lze upravit parametry zadání, přidávat k němu soubory a je zde zobrazena informace o validitě. Přidávané soubory lze označit jako normální, které vidí student a může si je stáhnout. Mohou to být doplňující dokumenty k zadání v PDF nebo jakékoli další datové soubory. Dalším typem jsou soubory, které student nevidí a slouží ke kontrole řešení na serveru.

Aby bylo možné úlohu zadat studentovi vyžaduji, aby zadání bylo validní a kompletní. Zadání musí obsahovat dva soubory důležité pro automatickou kontrolu, bez jejich vložení nemůže být úkol automaticky zkontrolován. Jsou to správné řešení ve VHDL a testovací obvod ve VHDL, které musí vyučující vytvořit podle zamýšleného zadání. Jejich využití při testování je popsán v kapitole 6.4 Simulace na serveru. Soubory musí být patřičně označeny, aby se studentovi nezobrazili. Také je potřeba zadat název entity, která je v souborech použita.

5.3.2 Skupiny a studenti

Vyučující si pro zadávání úkolů nejprve musí vytvořit skupiny do kterých přidá studenty. Skupiny by měly odpovídat skutečným předmětům a cvičením, takže mají kromě



Obrázek 15: Detail skupiny studentů

názvu i informace o dni, bloku a týdnech, kdy se předmět vyučuje. Název je nicméně libovolný. Pokud skupina nemá zadané žádné úkoly, lze ji smazat. Tím se ztratí pouze informace o skupině a jejich studentech. U každé skupiny lze zobrazit seznam úkolů zadaných konkrétním studentům a jejich detail.

5.3.3 Detail skupiny a přidání studentů do skupiny

V detailu lze přidat studenty do skupiny. K tomu slouží tlačítko „přidat studenty", které zobrazí okno se seznamem studentů. Kliknutím na tlačítko přidat u studenta je tento student přidán do skupiny a ihned se zobrazí v seznamu studentů dole na stránce, viz Obrázek 15.

Studenta je možné kdykoli odebrat, skupiny slouží pouze vyučujícímu pro jednodušší orientaci ve studentech a jejich úkolech.

5.3.4 Zadání úkolu

Postup pro zadání úkolu je v uživatelském rozhraní očíslován, nejprve je potřeba vybrat studenty. Studenty je možné vybrat ze seznamu kliknutím na řádek s jeho jménem, ten je zvýrazněn podbarvením. Opětovným kliknutím je vybrání zrušeno. Lze vybrat jednoho, či více studentů pro hromadné zadání úkolů.

Jakmile jsou studenti vybráni následují očíslované kroky. Zadání data a času do kdy má být úkol odevzdán. V posledním kroku vybrat zadání ze seznamu z otevřeného okna. Výběr zadání funguje podobně jako výběr studentů. Zadání lze vybrat i více. Pokud je vybráno pouze jedno, je přiřazeno všem studentům. Pokud jich je vybráno více jsou zadání opakována, přičemž je dodrženo pořadí, v jakém byli vybrány (označeny). První zadání je přiřazeno prvnímu vybranému studentovi, druhé druhému. Pokud byste vybrali tři studenty a pouze dvě zadání, je třetímu přiřazeno opět první vybrané zadání atd. Díky tomu lze hromadně přiřadit zadání, a přesto přesně určit komu se jaké přiřadí.

5.3.5 Kontrola úkolu

Vyučující může zkontrolovat řešení úkolu, když si zobrazí jeho detail ze seznamu úkolů. Uvidí tam stav úkolu, výsledek automatické kontroly a může si studentovo schéma otevřít v editoru a sám se podívat. Také má možnost si přímo zobrazit odevzdané řešení v podobě VHDL, a to si i stáhnout jako soubor. Tento pohled je podobný detailu zadání, které vidí student, pouze chybí možnost odevzdat úkol.

5.4 Registrace uživatele

Nový uživatel se může zaregistrovat pomocí volby Účet v menu. Po zvolení položky registrovat se zobrazí registrační formulář. V něm je potřeba vyplnit e-mail pro identifikaci a heslo. Pokud uživatel zadá školní e-mail je automaticky nastaven jeho účet s rolí student. Roli vyučujícího může přiřadit pouze administrátor ve speciálním pohledu pro nastavování oprávnění. Po registraci je uživateli zaslán e-mail pro ověření jeho vlastníka, které se provede otevřením dočasného odkazu.

6 Serverová část

Aplikaci jsem vytvořil tak, aby se bez obtíží dala nasadit na vhodný server (kompatibilní s popsaným v 1.2 Server). Nasazení se provede pouhým zkopírováním zdrojových souborů, upravením konfigurace a spuštěním instalačního scriptu pro vytvoření DB. Poté by měla být webová část aplikace funkční. Instalaci balíku Vivado je třeba provést manuálně administrátorem serveru. Stejně tak předpokládám správné nastavení serveru administrátorem pro běh aplikace.

Aplikace je nasazená na školním serveru (viz kapitola 1.2 Server), který je přístupný na adrese https://rlabu.ite.tul.cz/eco. Zkratka „eco“ znamená editor číslicových obvodů.

6.1 Konfigurace serveru

Konfigurační soubor je nezbytný pro uložení přístupových údajů pro připojení k DB, ale neslouží pouze k tomu. Definuje cesty k souborům a adresářům, adresář projektu, cestu ke složce se spustitelnými scripty atp. Tyto cesty jsou se liší v produkčním a vývojovém prostředí. Různá konfigurace tak umožnuje běh v obou prostředí bez změny kódu.

Dále obsahuje administrátorské heslo, které se využívá pro ověření při instalaci databáze a správě učitelských rolí. Heslo pro šifrování tokenů pro ověřování e-mailových adres a dobu platnosti tokenu. Dále parametry odesílaných e-mailů, jako je adresa odesílatele, předmět a URI aplikace.

Poslední část konfigurace obsahuje nastavení serverové simulace pod položkou `settings`. Zde se dá nastavit maximální limit odevzdaných řešení ve stavu čekající pro jednoho studenta (`maxWaitingSolutions`). Hodnotou limitu je číslo, pokud je rovné nule, tak je tato kontrola vypnuta a limit je nekonečný. Pokud je limit překročen, nemůže student odevzdávat další řešení. Další dvě volby zapínají, nebo vypínají, omezení odevzdávat řešení ke kontrole. První z nich (`disableAfterHomeworkDone`) omezuje odevzdání, pokud je již úkol úspěšně odevzdán. Druhá (`disableAfterDeadline`) nedovolí odevzdat řešení po vypršení zadaného termínu.

Pro dostupnost konfigurace v celé aplikaci jsem vytvořil třídu `Config`, která obsahuje statickou proměnnou s výchozí definicí nastavení a dvě statické funkce pro přístup ke konfiguraci. Konfigurace je uložená v proměnné typu asociativního pole. Každá položka obsahuje buď přímo hodnotu, nebo další pole, což umožní seskupit nastavení týkající se

jednoho celku. Například nastavení adresy, uživatele a hesla do databáze jsem umístil do pole pod klíčem „`db`".

Pro přístup k takto uložené konfiguraci lze použít funkci `getConfig()`, která vrátí celé pole. Z něj lez poté pomocí klíčů hodnoty vyčíst. Před tím, než tato funkce vrátí konfiguraci, ověří, jestli je načtená uživatelská konfigurace z doplňkového souboru a případně ji načte. Tento doplňkový soubor používám právě pro specifikaci rozdílných nastavení na serveru a lokálním vývojovém prostředí. Konfigurace z tohoto souboru přepíše tu výchozí. Pro usnadnění nalezení položky v hierarchii polí jsem přidal funkci `getKey($key,$delimiter='/')`, která přebírá klíč jako řetězec určující cestu, kde jsou jednotlivé klíče odděleny lomítkem. Takže pro získání hesla do databáze celý klíč vypadá následovně: `db/password`.

6.2 API

Vlastní serverové API má za úkol zejména poskytovat data webové aplikaci a přijímat požadavky na jejich změnu. API využívá architekturu REST [13] a Klient s ním komunikuje asynchronně pomocí JS. Pro implementaci API na serveru jsem použil knihovnu Slim ve verzi 2, která usnadňuje tvorbu REST API. Slim poskytuje funkce pro zpracování HTTP požadavků základních HTTP metod. Konkrétně je to metoda GET, která je určená pro získání dat (implementace často využívá databázový příkaz SELECT), metoda POST pro vytvoření nových záznamů (typicky využívá databázový příkaz CREATE), dále je to metoda PUT, která slouží pro změnu dat (UPDATE v DB) a poslední DELETE, pro smazání záznamů. HTTP protokol definuje několik dalších metod [14], které ve své práci nevyužívám, nicméně Slim framework je umí také zpracovávat [15]. Každá z těchto metod přebírá jako první parametr cestu, na kterou reaguje. Dalším parametrem může být takzvaný middleware, nepovinný kód, který lze takto modulárně přiřazovat. Posledním parametrem je obsluha této cesty. Ještě zmíním, že klientský JS využívá Backbone, který při synchronizaci sám volí správnou metodu, podle toho, co se s modelem dělo.

Definice API se nachází v hlavní souboru `/api/index.php` a je velmi jednoduchá. Nejprve je vytvořena instance třídy Slim, která reprezentuje aplikaci a obsahuje směrovací systém. Poté jsou definovány cesty pro patřičné metody spolu s obslužnými funkcemi. Zdrojový kód 20 ukazuje část definice API, konkrétně vytvoření instance objektu Slim na první řádku. Na dalším řádku je definice výchozí cesty, která pouze odešle status 200, bez

jakéhokoli výstupu. Následují řádky pro manipulaci se schéma, využil jsem zde všechny čtyři zmíněné metody. První je metoda `get`, která požaduje seznam schémat, obslouží ji funkce `schemas`. Obslužné funkce lze definovat více způsoby. Buď přímo vložit anonymní funkci jako to je v prvním případě nebo se na ní odkázat pomocí jejího jména jako textového řetězce. Ten může obsahovat také identifikátor třídy, pokud funkce je statická a nějaké třídě náleží. Další řádek obsahuje stejnou adresu, ale reaguje na metodu post, obsluhou je funkce `schemaCreate`, která má za úkol vytvořit nové schéma. Třetí definici s metodou `put` obsluhuje funkce `schemaUpdate` a jak název napovídá, stará se o změnu schéma. Poslední definice používá funkci `schemaDelete` a schéma odstraňuje, respektive označuje za smazané.

```
$app = new \Slim\Slim(array("settings" => $slim_config));

$app->get('/', function() use($app) {$app->response->setStatus(200);});
$app->get('/schemas', 'schemas');
$app->post('/schemas', 'schemaCreate');
$app->put('/schemas/:id', 'schemaUpdate');
$app->delete('/schemas/:id', 'schemaDelete');
```

Zdrojový kód 20: ukázka definice části API

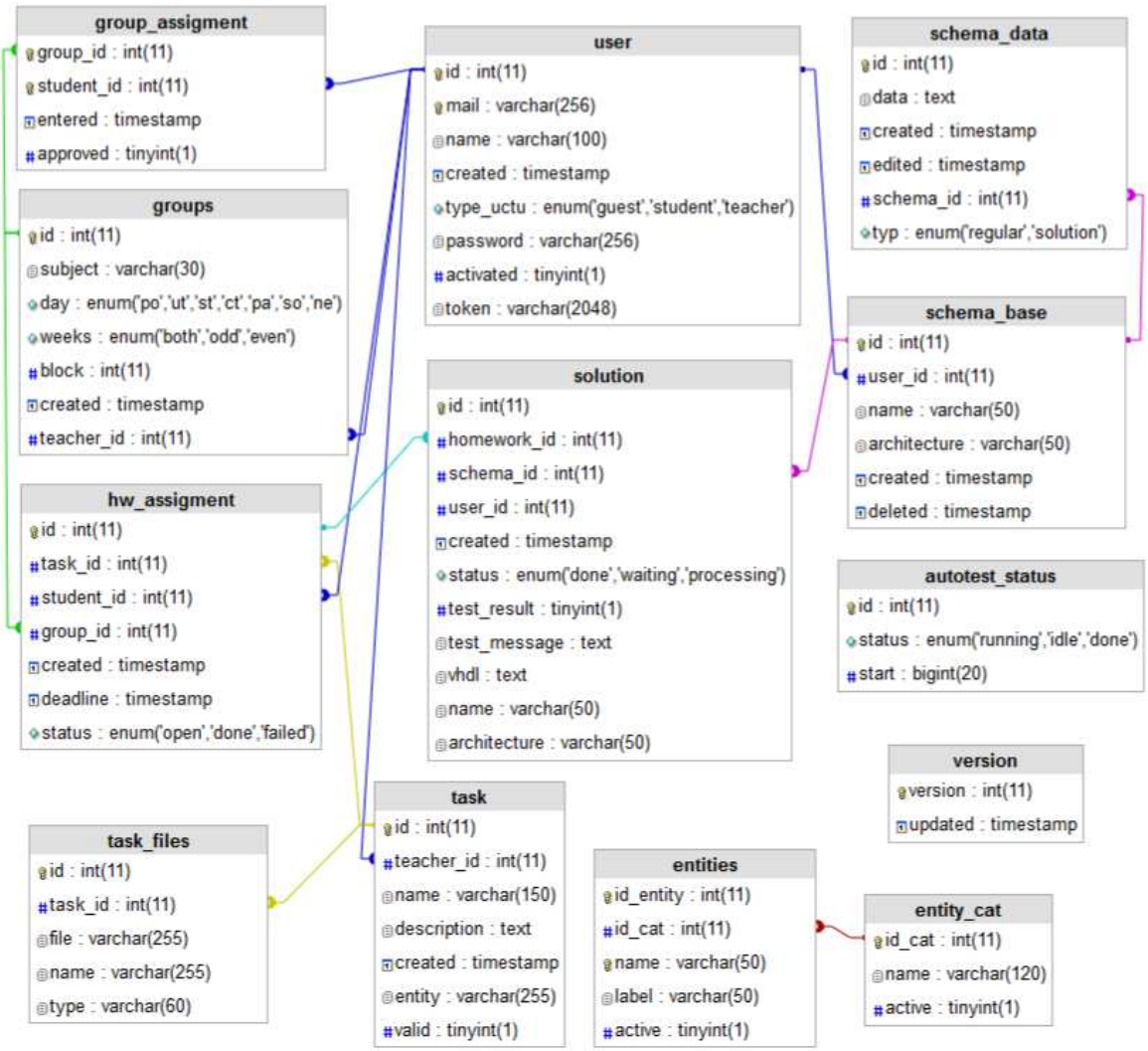
Rozhraní nabízí přístup pouze k nezbytným zdrojům, které jsou vyžadovány webovou aplikací a většina z nich je neveřejná a přístupná pouze pro přihlášené uživatele disponující patřičnou uživatelskou rolí.

Obslužné funkce jsou pro větší přehlednost rozděleny do souborů podle toho, k jaké části patří. Část je v definicích cesty uvedena v první části a odpovídá objektům, ke kterým dotaz logicky náleží. Například zjištění úkolů studenta je v souboru students.php byť vrací kolekci úkolů. Většina těchto funkcí jsou přímo využívána jako obsluha v REAST API, a proto vypisují určitý JSON objekt, který je posílán zpět na klienta jako odpověď. Buď to jsou vybraná data, nebo zprávy o úspěchu, či neúspěchu dané operace.

6.3 Databáze

Databáze není příliš rozsáhlá, obsahuje 13 tabulek, viz Obrázek 16. Pro přístup k databázi využívám rozšíření PDO, které v PHP poskytuje přehledné a jednoduché rozhraní pro přístup k databázi [16]. Jako ovladač využívám PDO_MYSQL, neboť databáze je MySql. Verze distribuce nainstalované DB na školním serveru je 5.1.73.

Referenční integritu zajišťují primární a cizí klíče. Téměř všechny vazby mezi tabulkami jsou typu 1:N, za výjimku by se dala považovat vazba mezi uživatelem (tabulka user) a skupinou (tabulka groups), kdy jeden uživatel může patřit do více skupina a skupina může mít samozřejmě více uživatelů (studentů). Tabulka group_assingment je tak jejich vazební. Nicméně vazby mezi těmito třemi tabulkami jsou na této úrovni realizované jako 1:N.



Obrázek 16: databázové schéma

6.3.1 Instalace databáze

Instalaci databáze a její aktualizaci provádí script `install.php`, který je zabezpečen heslem z konfigurace. Přihlašovací údaje k databázi se rovněž načítají z konfiguračního souboru a je potřeba je v něm předtím nastavit.

Po ověření připojení k databázi administrátor může vybrat jednu ze dvou možností. První je čistá instalace databáze, kdy jsou všechna dosavadní data vymazána a druhá je aktualizace. Při instalaci je nejprve zpracován hlavní script `eco_schema_mysql.sql` ze složky `schemas`, který obsahuje definice databázových tabulek. Následně jsou vložena výchozí data definována v souboru `eco_data_mysql.sql`. Poté je zkontrolována verze databáze a jsou případně nainstalovány aktualizace uložené ve složce `versions`.

Aktualizace databáze vynechá všechny kroky před kontrolou verze a instaluje pouze nové verze.

Soubory verzí jsou číslované od jedné výše a zpracovávají se postupně. V každém souboru je také příkaz pro změnu verze databáze. Je to jednoduchý způsob, díky němuž lze databázi měnit a tyto změny mít zaznamenány. Tento způsob je však celkem primitivní a nezajišťuje proveditelnost aktualizace databázového schéma. Případné problémy způsobené uloženými daty je potřeba vyřešit manuálně. Nejprve se zjistí verze, v jaké databáze právě je a s instalací se začne až od souboru s názvem o jedno větším.

6.4 Simulace na serveru

Na serveru má simulace sloužit jinému cíli než v klientské části aplikace. Není požadována interaktivita a kooperace s uživatelem, ale je potřeba prostřednictvím simulace vyhodnotit správnost navrženého obvodu podle zadání. Tomu musí odpovídat úroveň simulace. V javascriptu šlo o vyvolávání událostí a volání funkcí při změnách signálu. Tento postup ale neprovádí žádné porovnávání, pouze vizualizuje. Na serveru jde o to, jestli navržené schéma dává správný výstup.

6.4.1 Navržené přístupy simulace

Možností, jak k tomu přistoupit je několik. Nejtriviálnější by bylo porovnat vygenerovaný VHDL kód schéma se správným VHDL. Přičemž by se hledělo pouze na podstatnou část entit a jejich zapojení. Nicméně obvod je považován za správný, jestliže dává požadovaný výstup a nezáleží na tom, jak je toho docíleno. Tento přístup by neumožnil různou, byť správnou implementaci a je proto absolutně nepoužitelný.

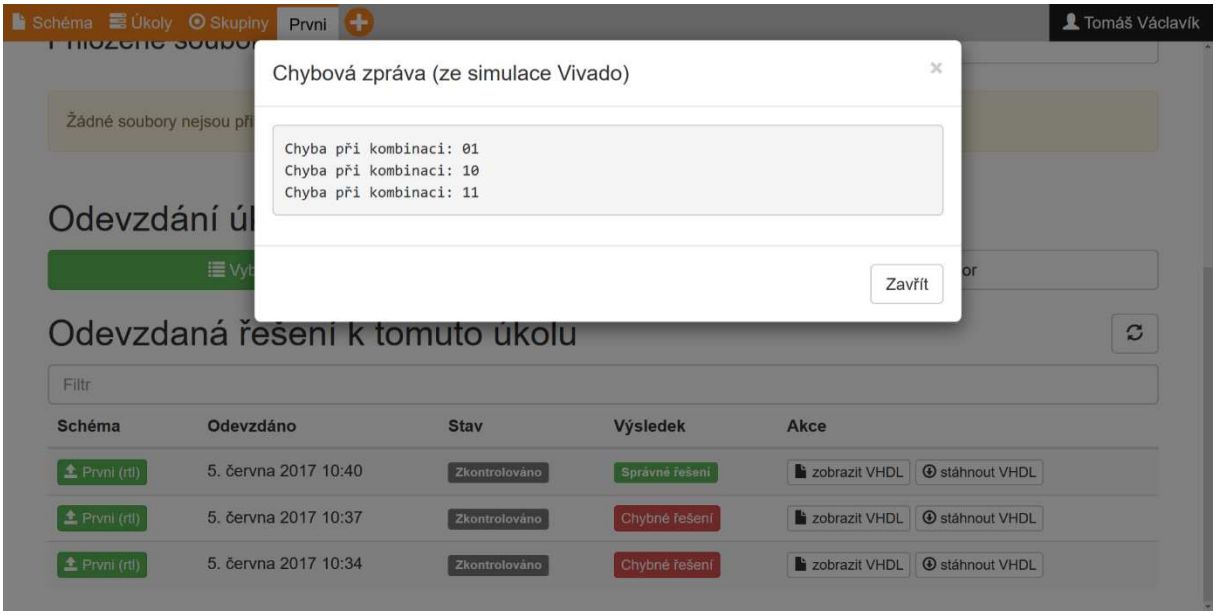
Zbývá obvod simulovat. Vytvořit vlastní implementaci simulace v PHP zjednodušenou a ořezanou na entity používané v editoru. Zde přichází otázka, z jaké reprezentace schéma obvod simulovat. Editor ukládá schéma na serveru jako řetězec JSON a jde o reprezentaci modelu `Graph` z JoitJS. Tam jsou uloženy všechny prvky jako parametry

objektu. Vodiče jsou s entitami propojeny parametry zdroje a cíle určené podle jejich id. To není ideální reprezentace pro simulaci, funkce entit zůstaly definovány v javascriptu a jsou na serveru nepoužitelné. Lepší možností by bylo uložit na server data schéma exportované do formátu VHDL, jazyku přímo určeném pro popis a simulaci hardwaru. V druhém případě by to znamenalo vytvořit simulátor jazyka VHDL, ale ty již existují a domnívám se, že v daleko lepší kvalitě, než které bych mohl dosáhnout sám.

Finální možností, kterou jsem se shodou s vedoucím práce zvolil je využití balíku Vivado disponující simulací a syntézou hardwaru popsaného pomocí VHDL. Export schéma do VHDL existuje v aplikaci již od počátku, po jeho drobné úpravě a opravení chyb se dá dobře využít. Vivado bylo zvoleno na úkor konkurence díky tomu, že je to již školou licencovaný software a je při výuce zajišťované ITE používaný.

6.4.2 Princip simulace ve Vivado

Klasický přístup testování obvodu v programu Vivado je vytvoření tzv. testbenche. Testbench je VHDL soubor obsahující entitu, která ve své implementaci (architecture) používá testovanou entitu jako komponentu a buzením jejich vstupů a porovnáním výstupů pomocí mechanismu assert provádí testování. Assert funguje obdobě jako při testování funkcí v jiných jazycích. Podmínkou stanoví očekávání a pokud není naplněno, je vypsána definovaná chybová hláška [17].



Obrázek 17: vyhodnocená domácí úloha s výpisem chyb

Ukázka Zdrojový kód 21 obsahuje část testu (z testbench souboru) vstupní kombinace obou vstupů v log. 0, kde a, b, hw_q a etalon_q jsou signály. Pokud se výstupy testovaného (hw_q) a správného (etalon_q) řešení nerovnají, vypíše se chybová hláška, kde je uvedeno, jaká kombinace je nesprávná. Tyto informace jsou poskytnuté studentům. Díky tomu mají přesnější informace a na jejich základě mohou schéma opravit, viz Obrázek 17.

```
a <= '0';
b <= '0';
wait for 10 ns;
assert hw_q = etalon_q report "Chyba pri kombinaci: 00" severity error;
```

Zdrojový kód 21: příklad testovací podmínky v testbench VHDL souboru

Testbench může používat více než jednu komponentu. Toho využívám pro testování tím způsobem, že testbench nedefinuje správný výstup sám, ale porovnává výstupy testované entity s výstupy entity správného řešení. Právě tento přístup jsem použil.

Vytvoření souboru testbench a správného řešení je úkolem vyučujícího, jelikož musí být vytvořeny na míru zadání. Aby bylo co testovat, musí vyučující vytvořit v aplikaci zadání domácí úlohy (podobu a možnosti popisuje kapitola 5.3.1 Detail zadání a ilustruje Obrázek 14). Správné řešení je k zadání domácí úlohy přidáno vyučujícím, jde o VHDL soubor, ve kterém je definována a implementována entita, která správně realizuje požadovanou funkci. Při testování jsou porovnávány výstupy tohoto správného řešení s řešením odevzdaným studentem. Získání jeho VHDL kódu popisuje následující kapitola.

6.4.3 Příprava VHDL pro testování

Testbench se odkazuje na porovnávané entity jejich názvem. Název entity je uveden uvnitř VHDL v části definice entity. Jméno samotného souboru nemá v tomto případě žádný význam. Pro simulaci je potřeba, aby název entity ve vygenerovaném VHDL souboru odpovídal entitě použité v testbench.

Při odevzdání je uložen VHDL kód odevzdaného schéma do DB. Ten je vygenerován Javascriptem ještě na klientské straně a poslán společně s požadavkem na serverové API. Jako název entity při exportu není použit název odevzdávaného schéma, protože to může student pojmenovat téměř libovolně podle svého uvážení. Místo toho je entita pojmenována podle explicitně zadaného názvu, který je uveden u zadání a uložen v DB. Tento název tedy určuje vyučující při vytváření zadáni, stejně jako testovací soubory a je tedy jediný, kdo zná jejich správný název.

Díky tomuto mechanismu si studen může pojmenovávat schéma libovolným způsobem, a přesto je možné jej testovat. Zbytek exportu do VHDL probíhá beze změny a je shodný například s tím, které se používá pro uložení schéma.

Pro kontrolu, jestli je název entity zadán správě je každé zadání validováno, při tom je kontrolován výskyt entity se zadaným názvem v testovacím souboru. Pokud není nalezen, zřejmě je použitý název jiný a simulace by nešla dokončit, zadání je tak nevalidní. Snadno nás může napadnou brát název entity přímo z testovacího VHDL, to ale není dobře proveditelné, protože v tomto souboru může být použito více než jedna entita a nelze určit, která je ta testovaná. Možností by bylo označit takovou entitu smluveným komentářem přímo ve VHDL. To ale stále spoléhá na zadání vyučujícím, přičemž je komplikovanější získat její název a nelze ani zaručit, že takto bude označena. Já jsem tedy zvolil předchozí přístup.

6.4.4 TCL script

Vivado používá pro ovládání přes konzoli skriptovací jazyk TCL [18]. Pro spuštění simulace jsme připravili krátký script, ve kterém je nejprve vytvořen Vivado projekt, následně jsou do něj přidány potřebné soubory příkazem `add_files`. Vyžadován je testbench a entita správného řešení, které byly jako soubor přiloženy k zadání. Poté soubor lib.vhd s knihovnou entit a soubor s VHDL testovaného řešení, které bylo studentem odevzdáno. Následuje nastavení simulátoru a výchozího souboru pro testování (testbench). Poté je již příkazem `launch_simulation` spuštěna simulace. Tento script je generován dynamicky, neboť umístění přidávaných souborů je závislé na tom, jaké zadání je testováno. Každé zadání má potřebné soubory uloženo v samostatné složce a příkaz pro přidání souborů požaduje uvedení celé cesty.

6.4.5 Script pro spuštění Vivado

Složka pro Vivado projekty, které jsou vytvářeny pro testování je umístěna na serveru v adresáři cgi-bin, kde je místo odkud je povoleno spouštět shell scripty.

Aby bylo možné Vivado přes příkaz spustit, nestačí použít v PHP příkaz `exec` a zadat jeho název a předat mu TCL script. Je potřeba nastavit proměnné prostředí. Také je třeba přesunout TCL script ze složky ve které byl vygenerován do složky cgi-bin. Protože PHP nemá z bezpečnostních důvod oprávnění k zápisu do složky cgi-bin a Tcl script je potřeba generovat dynamicky.

O toto se stará bash script eco.sh, který provádí výše popsané úkony. Jeho celý kód představuje Zdrojový kód 22.

```
#!/bin/env bash
. /etc/profile.d/xil_ISE14.6_settings64.sh
. /etc/profile.d/xil_Vivado2016.4_settings64.sh
HOME="/var/www/apache_fake_home"
export HOME

cp /var/www/html/eco/test.tcl /var/www/cgi-bin/test.tcl

/opt/Xilinx/Vivado/2016.4/bin/vivado -mode batch -source
/var/www/html/eco/test.tcl 2>&1
```

Zdrojový kód 22: eco.sh script pro nastavení a spuštění Vivado

6.4.6 Příprava a spuštění simulace z PHP

Impuls pro start simulace je dán z PHP, když student odevzdá řešení. Aby nedocházelo k přepisování generovaných souborů je možné mít spuštěnou pouze jednu simulaci současně. Nicméně script test.php, který se o její spuštění stará je zavolán vždy. Proto jsem zavedl frontu řešení čekajících na kontrolu.

Script, který je spuštěn nejprve ověří, jestli neběží nějaký jiný, to je zaznamenáno v DB v tabulce `autotest_status`. Při startu scriptu je do tabulky vložen záznam s přesným časem v mikrosekundách. Poté je z tabulky zjištěn nejstarší záznam a pokud to je záznam patřící této instanci, je v testování pokračováno. V opačném případě je test ukončen ještě před začátkem a záznam je z tabulky stavu odstraněn.

Každé řešení má v DB uložen stav: čekající/zpracovávaný/hotový a přesný čas kdy bylo vytvořeno. Podle času vytvoření jsou řešení řazena a pro testování jsou brána od nejstaršího.

Jedna kontrola simulací trvá přibližně půl minuty, zdlouhavá je zejména inicializace programu Vivado a tuto dobu tak není možné příliš zkrátit. Aby uživatel nezaplnil frontu na delší dobu dopředu, má omezení odevzdat pouze určité množství řešení, která jsou ve stavu čekající. Tato hodnota je na serveru konfigurovatelná, ale přednastavená hodnota jsou 3 řešení. Jakmile je řešení zkontrolováno jeho stav se změní a uživatel může odevzdat další.

Během činnosti simulace mohou být odevzdána další řešení nebo mohou čekat řešení jejichž spouštěcí script nemohl být vykonán, protože byl spuštěn jiný. Nicméně řešení je potřeba zkontrolovat, proto test.php obsahuje smyčku, která spouští více testů po sobě,

dokud jsou k dispozici čekající řešení nebo není překročen definovaný limit (ve výchozím nastavení 30).

Každá iterace smyčka obsahuje tyto kroky:

1. Dotázání se databáze na další čekající řešení ve frontě, pokud žádné není, je smyčka ukončena a testování zastaveno.
2. Kontrola, jestli je zadání validní a obsahuje všechny potřebné soubory. Je nutné to provádět i před testováním, protože zadání, které bylo v době přidělení jako úkol validní, nemá vyučující zakázáno následně měnit. Pokud validní není je přeskočeno (může být testováno znovu) ale je u něj uložena poznámka o této chybě. Tu si může student přečíst a informovat případně vyučujícího.
3. Je vytvořen unikátní název pro složku projektu k simulaci
4. Jsou získány cesty k testbench souboru a správnému řešení z DB
5. VHDL odevzdaného řešení je uloženo do souboru, který je pro tento účel vytvořen.
6. Je vytvořen TCL script pro Vivado. Jako argumenty jsou předány cesty k získaným a vytvořeným souborům v předchozích krocích.
7. TCL script je uložen do souboru. Do kořenové složky stránky, kam má PHP právo zápisu.
8. Ze záznamu v konfiguračním souboru je vytvořen příkaz s cestou ke spouštěcímu scriptu eco.sh a ten je spuštěn PHP funkcí `exec( $config["cgipath"]. "eco.sh 2>&1", $outputLines)`. Jako parametr je předán text příkazu a proměnná pro uložení výstupu. Chybový výstup je přesměrován na standardní výstup a čeká se, než je simulace Vivadem dokončena.
9. Po skončení simulace je analyzován výstup.
10. Výsledek je uložen do DB
11. Smazání vytvořených dočasných souborů TCL scriptu a složky s Vivado projektem

6.4.7 Vyhodnocení výsledků simulace

Správnost řešení určuji na základě konzolového výstupu z programu Vivado, který je po skončení simulace uložen ve výstupní proměnné. Vivado vypisuje informace o celém procesu simulace, ale nás zajímá výskyt chybových zpráv, které začínají slovem „error“. Pokud se ve výpisu vyskytne, je řešení považováno za nesprávné. K řešení je potom do databáze uložena poznámka s textem nalezených chyb. Díky tomu si student může přečíst kde a proč k chybě nastalo.

Původcem chyby by měl být příkaz `assert` v testbench souboru, který informuje o špatném řešení při určité kombinaci vstupů a vnitřního stavu entity. Vivado nicméně může produkovat další chyby, které jsou také zachyceny a zobrazeny studentovi. K nim by však nemělo docházet, pokud je testbench správně vytvořen vyučujícím. Jednou z podmínek je, aby testovaná entita měla správný název odpovídající entitě použité v testbenchi, což zajišťuje přejmenování entity při odevzdání popsané v kapitole 6.4.3 Příprava VHDL pro testování.

7 Závěr

V rámci této práce jsem se seznámil s možnostmi simulace číslicových obvodů jak v prohlížeči, tak na serveru. Tato různá prostředí vyžadují jiný přístup k simulaci, neboť simulace slouží jiným účelům. Na klientské straně jsem docílil interaktivity simulace její vizualizací. Zvýrazněním výstupů a vodičů nesoucích signál s log. úrovní 1. Dále pak díky možnosti přepínat hodnotu signálu na vstupech a tím obvod ovládat. Editor obvodů tak získal nový modul, který simulaci provádí a uživatel si může přímo při návrhu obvodu intuitivním způsobem vyzkoušet jeho funkci a funkci jednotlivých entit. Díky tomu mohou studenti lépe pochopit principy fungování logických obvodů.

Při měření výkonu simulace jsem došel k zjištění, že editor zvládá simulovat obvody s desítkami až stovkami hradel bez větších problémů. Záleží však na konkrétním zapojení. Při rostoucím počtu zapojených entit se zvyšuje čas potřebný pro projití signálu celým obvodem. Zpoždění je patrné již při sériovém zapojení 20 entit, kdy čas, za jaký se projeví změna na výstupu, se při změněně vstupu pohyboval v průměru kolem 300 ms. Nicméně pokud část entit zapojíme paralelně, doba se výrazně zkrátí. Ve všech případech nemělo zatížení simulací vliv na plynulost editoru.

Simulace na serveru slouží primárně pro automatickou kontrolu studentských schémat. Tento úkol vyžadoval přesnější simulaci než na klientské straně a šlo hlavně o zkontrolování výstupů studentských schémat podle zadání. To jsem splnil za pomoci programu Vivado, který simuluje entity exportované do VHDL. Díky navrženému přístupu, student po simulaci vidí, jestli jeho řešení je správné, případně při jaké kombinaci vstupních hodnot došlo k chybě.

Tato kombinace vlastností v jedné webové aplikaci je jedinečná, nicméně vždy je co zlepšovat. Je možné jednoduše rozšířit knihovnu entit o nové, případně umožnit uživatelům definovat si vlastní komponenty. Uživatelské rozhraní samotného editoru se dá také vylepšovat, například hromadným výběrem a úpravou entit. Jedním z dalších vylepšení, které by zlepšilo použitelnost by mohlo být využití poskytovatelů identit pro přihlášení, například pomocí Shibboleth, Facebook, nebo Google účtu.

Celou aplikaci jsem nasadil na školní server, který mně byl pro tento účel poskytnut, společně s nainstalovaným balíkem Vivado. Aplikace je veřejně dostupná pro registrované uživatele na adrese https://rlabu.ite.tul.cz. Pro potřeby správy schémat, úkolů, jejich

odevzdávání a dalších úkonů, jsem vytvořil potřebné uživatelské rozhraní, webovou aplikaci, která tyto požadavky řeší.

Díky všem realizovaným částem tato webová aplikace umožňuje veřejnosti vytvářet vlastní schéma, uložit si je na serveru pro pozdější práci nebo je exportovat do VHDL, které nabízí další možnosti zpracování. Poskytuje jim možnost vytvořený obvod interaktivně simulovat a vyzkoušet si jeho funkce. Studentům navíc umožňuje realizovat zadané domácí úlohy v editoru, odevzdávat je prostřednictvím webové aplikace a nechat si je automaticky zkontrolovat vytvořeným serverovým simulátorem. Pro tyto účely jsem v aplikaci vytvořil patřičná uživatelská rozhraní, výpisy zadání, detail konkrétního zadání s výsledky testů atd. Vyučující prostřednictvím aplikace může nejen vytvářet úlohy a zadávat je studentům, ale má připravené rozhraní pro celkovou správu úkolů. Nástroje pro vytvoření skupin, do kterých může přidávat studenty a jim poté zadávat úlohy k vypracování. Formuláře pro přípravu zadání s nahráváním klíčových souborů. Nástroj pro hromadné zadávání úkolů. Přehledné zobrazení odevzdaných a otestovaných řešení, která lze otevřít v editoru nebo zkontrolovat výsledky automatického testu a studenta na jejich základě ohodnotit.

Byť tato rozhraní nebyla explicitně požadována, právě díky nim tato aplikace umožnuje kontrolu schémat na serveru a může se stát užitečným nástrojem pro výuku logických obvodů na Technické univerzitě v Liberci.

Seznam referencí

[1] Jaroslav, Řehák. Grafický editor číslicových obvodů v HTML5. Diplomová práce. Liberec : TUL, 2014.

[2] CLIENT IO S.R.O. JointJS [online]. [cit. 2017-07-03]. Dostupné z: https://www.jointjs.com/opensource

[3] Joint API [online]. [cit. 2017-07-03]. Dostupné z: http://resources.jointjs.com/docs/jointjs/v2.0/joint.html

[4] Backbone.js [online]. [cit. 2017-07-02]. Dostupné z: http://backbonejs.org/

[5] Benmccormick.org: The Sad State of the Backbone Ecosystem [online]. [cit. 2017-07-03]. Dostupné z: https://benmccormick.org/2016/03/07/the-sad-state-of-the-backbone-ecosystem/

[6] *Slim Framework v2: Get started* [online]. [cit. 2017-08-05]. Dostupné z: http://docs.slimframework.com/start/get-started/

[7] *Composer: Introduction* [online]. [cit. 2017-08-05]. Dostupné z: https://getcomposer.org/doc/00-intro.md

[8] *Bower: A package manager for the web* [online]. [cit. 2017-08-10]. Dostupné z: https://bower.io/

[9] *Getting started - Grunt: The JavaScript Task Runner* [online]. [cit. 2017-08-09]. Dostupné z: https://gruntjs.com/getting-started

[10] *Backbone.Undo.js: A simple Backbone undo-manager for simple apps* [online]. [cit. 2017-08-21]. Dostupné z: https://osartun.github.io/Backbone.Undo.js/

[11] PINKER, Jiří a Martin POUPA. *Číslicové systémy a jazyk VHDL*. Praha: BEN - technická literatura, 2006, 349 s. ISBN 80-730-0198-5.

[12] *VHDL Reference Guide: Contents* [online]. [cit. 2017-07-26]. Dostupné z: http://www.ics.uci.edu/~jmoorkan/vhdlref/

[13] *REST: architektura pro webové API: Zdroják* [online]. [cit. 2017-08-15]. Dostupné z: https://www.zdrojak.cz/clanky/rest-architektura-pro-webove-api/

[14] *HTTP request methods: HTTP | MDN* [online]. [cit. 2017-07-26]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

[15] *Custom Routes: Slim Framework v2* [online]. [cit. 2017-08-05]. Dostupné z: http://docs.slimframework.com/routing/custom/

[16] *PHP: PDO: Manual* [online]. [cit. 2017-05-03]. Dostupné z: http://php.net/manual/en/class.pdo.php

[17] *VHDL reference guide: Assert* [online]. 2009 [cit. 2017-07-26]. Dostupné z: http://www.ics.uci.edu/~jmoorkan/vhdlref/assert.html

[18] *Vivado Design Suite Tcl Command Reference Guide: UG835 (v2014.4)* [online]. Xilinx, 2014 [cit. 2017-07-26]. Dostupné z: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_4/ug835-vivado-tcl-commands.pdf

Obsah přiloženého CD

- Text diplomové práce
  - Diplomová práce – VÁCLAVÍK.pdf
- Zdrojové soubory webové aplikace
  - ECO-source-DP-Vaclavik.zip