



**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

BRNO UNIVERSITY OF TECHNOLOGY

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

FACULTY OF INFORMATION TECHNOLOGY

**ÚSTAV INTELIGENTNÍCH SYSTÉMŮ**

DEPARTMENT OF INTELLIGENT SYSTEMS

**WEBOVÁ APLIKACE PRO EDITACI HIERARCHICKÝCH  
BLOKOVÝCH SCHÉMAT A STAVOVÝCH STROJŮ**

WEB APPLICATION FOR EDITING HIERARCHICAL BLOCK DIAGRAMS AND STATE MACHINES

**BAKALÁŘSKÁ PRÁCE**

BACHELOR'S THESIS

**AUTOR PRÁCE**

AUTHOR

**SEBASTIÁN BENCSÍK**

**VEDOUCÍ PRÁCE**

SUPERVISOR

**doc. Ing. VLADIMÍR JANOUŠEK, Ph.D.**

BRNO 2025

## Zadání bakalářské práce



164952

Ústav: Ústav inteligentních systémů (UITS)  
Student: **Bencsík Sebastián**  
Program: Informační technologie  
Název: **Webová aplikace pro editaci hierarchických blokových schémat a stavových strojů**  
Kategorie: Web  
Akademický rok: 2024/25

### Zadání:

1. Prostudujte problematiku vizuálních modelovacích jazyků, zaměřte se na bloková schémata (diagramy funkčních bloků).
2. Seznamte se s formalismem DEVS (Discrete Event Systems Specification). Seznamte se s existujícími nástroji pro editaci blokových schémat (nejen pro DEVS), zvláštní pozornost věnujte nástrojům NodeRED a 4diac.
3. Navrhněte editor blokových schémat, který je konformní s formalismem DEVS a může běžet jako webová aplikace. Předpokládejte, že uložení modelů je na straně serveru a také kód definovaný blokovým schématem poběží na straně serveru, podobně jako je tomu u nástroje NodeRED. Vizuální stránka blokových schémat by měla odpovídat přístupu, který uplatňuje nástroj 4diac. Atomické bloky budou definovány stavovými stroji, podobně jako v případě 4diac. Vyšší úrovně hierarchie budou definovány blokovými schématy.
4. Editor implementujte vhodnými prostředky.
5. Vytvořenou aplikaci otestujte na několika příkladech vizuálně specifikovaného kódu.

### Literatura:

Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

První 2 body a část návrhu s prototypovou realizací dílčí funkcionality editoru.

Podrobné závazné pokyny pro vypracování práce viz <https://www.fit.vut.cz/study/theses/>

Vedoucí práce: **Janoušek Vladimír, doc. Ing., Ph.D.**

Vedoucí ústavu: Kočí Radek, Ing., Ph.D.

Datum zadání: 1.11.2024

Termín pro odevzdání: 14.5.2025

Datum schválení: 4.11.2024

## Abstrakt

Táto bakalárska práca predstavuje webový editor pre modelovanie a simuláciu systémov pomocou DEVS (Discrete Event System Specification) formalizmu. Hlavným cieľom práce bolo vytvoriť dostupný nástroj, ktorý umožňuje používateľom navrhovať a simulovať diskrétne udalostné systémy priamo v prehliadači. Implementované riešenie pozostáva z intuitívneho vizuálneho editora na strane klienta a simulačného jadra na strane servera. Systém podporuje tvorbu atomických a zložených modelov, ich hierarchickú kompozíciu a interaktívnu simuláciu. Výsledkom práce je funkčná webová aplikácia, ktorá sprístupňuje DEVS modelovanie širšiemu okruhu používateľov bez potreby inštalácie špeciálneho softvéru.

## Abstract

This bachelor thesis presents a web-based editor for modeling and simulation of systems using the DEVS (Discrete Event System Specification) formalism. The main goal was to create an accessible tool that allows users to design and simulate discrete event systems directly in the browser. The implemented solution consists of an intuitive visual editor on the client side and a simulation core on the server side. The system supports the creation of atomic and coupled models, their hierarchical composition, and interactive simulation. The result is a functional web application that makes DEVS modeling accessible to a wider audience without the need to install specialized software.

## Klíčové slová

DEVS, diskrétna simulácia, webový editor, vizuálne programovanie

## Keywords

DEVS, discrete simulation, web editor, visual programming

## Citácia

BENCSÍK, Sebastián. *Webová aplikace pro editaci hierarchických blokových schémat a stavových strojů*. Brno, 2025. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Ing. Vladimír Janoušek, Ph.D.

# Webová aplikace pro editaci hierarchických blokových schémat a stavových strojů

## Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána doc. Ing. Vladimíra Janouška, PhD. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....  
Sebastián Bencsík  
13. mája 2025

## Podakovanie

Týmto by som chcel poďakovať svojmu vedúcemu bakalárskej práce, doc. Ing. Vladimírovi Janouškovi, PhD., za jeho trpezlivý prístup a odborné usmernenie počas vývoja aplikácie a pri vypracovaní technickej dokumentácie. Zároveň vyjadrujem úprimné poďakovanie svojej rodine za neustálu podporu, povzbudenie a trpezlivosť počas celého štúdia.

# Obsah

<b>1</b>	<b>Úvod</b>	<b>4</b>
<b>2</b>	<b>Teoretické základy</b>	<b>5</b>
2.1	Úvod do diskretnej simulácie . . . . .	5
2.2	Formalizmus DEVS . . . . .	5
2.2.1	Matematická definícia . . . . .	6
2.2.2	Zložený DEVS model . . . . .	6
2.3	Stavové automaty v DEVS modelovaní . . . . .	7
2.4	Implementačné aspekty DEVS simulácie . . . . .	7
2.4.1	Architektúra DEVS simulačných systémov . . . . .	7
2.4.2	Reprezentácia DEVS modelov v programovacích jazykoch . . . . .	7
2.5	Webové technológie . . . . .	8
2.5.1	Architektúra klient-server . . . . .	8
2.5.2	Asynchrónna komunikácia klient-server . . . . .	8
2.5.3	Vizuálne programovanie pre DEVS modelovanie . . . . .	8
2.5.4	Serializácia a deserializácia modelov . . . . .	9
2.6	Existujúce nástroje . . . . .	9
2.6.1	Nástroje založené na DEVS formalizme . . . . .	9
2.6.2	Nástroje pre vizuálnu editáciu blokových schém . . . . .	9
<b>3</b>	<b>Návrh a implementácia</b>	<b>10</b>
3.1	Architektúra systému . . . . .	10
3.1.1	Komunikačná architektúra . . . . .	10
3.1.2	Autentifikácia a zabezpečenie . . . . .	11
3.1.3	Perzistencia dát . . . . .	11
3.2	Použité technológie . . . . .	11
3.2.1	Frontend (Editor) . . . . .	11
3.2.2	Backend . . . . .	11
3.3	Návrh komponentov . . . . .	12
3.3.1	Editor Core . . . . .	12
3.3.2	Simulačné jadro a adaptér modelov . . . . .	13
<b>4</b>	<b>Popis algoritmov</b>	<b>15</b>
4.1	Návrh interného modelu schémy . . . . .	15
4.1.1	Základné triedy a dátové štruktúry . . . . .	15
4.1.2	Validácia modelu . . . . .	16
4.2	Implementácia stavových uzlov . . . . .	16
4.2.1	Architektúra stavového uzla . . . . .	16

4.2.2	Algoritmus spracovania udalostí . . . . .	17
4.2.3	Algoritmus pre vykresľovanie prepojení v stavovom grafe . . . . .	17
4.2.4	Dynamická konfigurácia portov . . . . .	17
4.3	Generovanie exportu pre simulátor . . . . .	18
4.3.1	Transformácia modelu . . . . .	18
<b>5</b>	<b>Užívateľské rozhranie</b>	<b>19</b>
5.1	Hlavné rozhranie . . . . .	19
5.1.1	Základná štruktúra rozhrania . . . . .	19
5.1.2	Komponentná štruktúra . . . . .	20
5.1.3	Integrácia s Rete.js a React . . . . .	20
5.2	Práca s uzlami . . . . .	21
5.2.1	Pridávanie uzlov . . . . .	21
5.2.2	Typy uzlov a ich vizuálna reprezentácia . . . . .	21
5.2.3	Manipulácia s uzlami . . . . .	22
5.3	Editácia stavových uzlov . . . . .	23
5.3.1	Architektúra modálneho editora stavov . . . . .	23
5.3.2	Štruktúra stavového editora . . . . .	23
5.3.3	Komponenta modálneho okna . . . . .	23
5.3.4	Editácia prechodov . . . . .	24
5.3.5	Špecializované renderovacie komponenty . . . . .	24
5.4	Editor kódu . . . . .	25
5.4.1	Integrácia CodeMirror . . . . .	25
5.4.2	Validácia a chybové hlásenia . . . . .	26
5.5	Správa portov (sockets) . . . . .	26
5.5.1	Typy portov . . . . .	26
5.6	Pokročilé funkcie editora . . . . .	27
5.6.1	História a navigácia v editore . . . . .	27
5.6.2	Interaktívne prvky . . . . .	27
5.6.3	Simulácia a výsledky . . . . .	28
5.7	Typy selektora výberu uzlov . . . . .	29
5.8	Ukladanie a načítavanie modelov . . . . .	30
5.8.1	SaveFlowModal . . . . .	30
5.8.2	Modálne okno . . . . .	30
5.9	Zhrnutie užívateľského rozhrania . . . . .	31
<b>6</b>	<b>Testovanie systému a používateľského rozhrania</b>	<b>32</b>
6.1	Testovanie DEVS simulátora . . . . .	32
6.2	Testovanie používateľského rozhrania . . . . .	33
<b>7</b>	<b>Záver</b>	<b>34</b>
	<b>Literatúra</b>	<b>35</b>
	<b>A Štruktúra súborov</b>	<b>37</b>
	<b>B Diagramy</b>	<b>38</b>

# Zoznam obrázkov

3.1	Diagram architektúry systému . . . . .	10
3.2	Diagram tried hlavného editora . . . . .	13
5.1	Diagram používateľského rozhrania . . . . .	19
5.2	Hlavný editor s atomickým uzlom a modulom . . . . .	20
5.3	Editor stavového automatu s príkladom jednoduchého grafu stavov a prechodov	24
5.4	Editor kódu pre atomické uzly s ukázkou syntaxe a kontextovou nápovedou	25
5.5	Bočný panel pre konfiguráciu vstupných a výstupných portov uzla . . . . .	27
5.6	Kontextové menu po kliknutí na pracovnú plochu . . . . .	28
5.7	Kontextové menu po kliknutí na uzol. . . . .	28
5.8	Modálne okno so zobrazením výstupu simulácie v podobe textového logu. .	29
5.9	Použitie výberového nástroja (lasso) pre označenie viacerých uzlov v editore.	29
5.10	Modálne okno pre uloženie modelu na server. . . . .	30
5.11	Modálne okno pre načítanie modelu na server. . . . .	31
B.1	Diagram tried simulátora . . . . .	38
B.2	Sekvenčný diagram simulácie . . . . .	39

# Kapitola 1

## Úvod

V súčasnosti, keď digitalizácia preniká do všetkých oblastí spoločnosti a priemyslu, narastá aj potreba efektívnych nástrojov na modelovanie a simuláciu zložitých systémov. Jedným z etablovaných prístupov v tejto oblasti je formalizmus DEVS (Discrete Event System Specification), ktorý ponúka precízny a univerzálny rámec pre popis dynamiky systémov založených na diskretných udalostiach. Vďaka svojej modularite a matematickej presnosti je DEVS vhodný pre širokú škálu aplikácií – od akademického výskumu po inžiniersku prax.

Napriek silnému teoretickému zázemiu však DEVS v praxi často naráža na bariéru používateľskej zložitosti. Existujúce nástroje sú zväčša určené pre expertov s hlbokými znalosťami formálnych metód, čo obmedzuje jeho využitie širšou komunitou. Práve na túto výzvu reagujem v tejto bakalárskej práci, ktorej cieľom je navrhnúť a vytvoriť moderný webový editor pre vizuálne modelovanie DEVS systémov – tak, aby bol prístupný aj používateľom bez predchádzajúcich skúseností s DEVS formalizmom.

Zvolený prístup kladie dôraz na intuitívne grafické rozhranie, ktoré podporuje hierarchické modelovanie, vizualizáciu stavových automatov a definovanie správania komponentov prostredníctvom programového zápisu. Výsledný nástroj som navrhol ako webovú aplikáciu s podporou simulácie modelov na strane servera a možnosťou ukladania projektov. Práca zahŕňa analýzu existujúcich riešení, návrh systémovej architektúry, výber vhodných technológií, ako aj samotnú implementáciu a testovanie.

Celý projekt som rozdelil do piatich kapitol. Po úvodnej časti nasleduje teoretický prehľad základov DEVS formalizmu a súvisiacich konceptov diskretnej simulácie 2. Tretia kapitola 3 opisuje návrh architektúry a použité technológie. Vo štvrtej kapitole 4 rozpracovávam spôsob implementácie kľúčových funkcií vrátane simulácie. Piata kapitola 5 sa zameriava na používateľské rozhranie a možnosti interakcie. Šiesta kapitola 6 sa zameriava na testovanie aplikácie, vrátane overovania funkcionality používateľského rozhrania a správnosti správania simulátora. Záver práce hodnotí dosiahnuté výsledky a ponúka návrhy na ďalšie rozšírenie a praktické využitie vytvoreného nástroja.

Cieľom tejto práce je navrhnúť a implementovať webový nástroj, ktorý kombinuje vizuálne programovanie s hierarchickým modelovaním a serverovou simuláciou, pričom umožní definovanie správania komponentov pomocou stavových automatov alebo kódu.

# Kapitola 2

## Teoretické základy

### 2.1 Úvod do diskkrétnej simulácie

Diskrétna udalostná simulácia predstavuje dôležitý prístup k modelovaniu systémov, kde sa stav systému mení v diskrétnych časových bodoch následkom výskytu udalostí. Tento simulačný paradigma sa odlišuje od spojitej simulácie tým, že modelový čas nepostupuje konštantným krokom, ale "skáče" od jednej udalosti k nasledujúcej. Medzi týmito bodmi sa stav systému nemení, čo umožňuje efektívnejšie využitie výpočtových zdrojov pri simulácii komplexných systémov [4].

Základnými konceptmi v diskkrétnej udalostnej simulácii sú:

- **Stav systému** - súbor premenných, ktoré popisujú systém v určitom časovom bode
- **Udalosti** - asynchrónne výskyty, ktoré menia stav systému
- **Časový plán** (event schedule) - dátová štruktúra uchovávajúca budúce udalosti zoradené podľa času výskytu
- **Simulačný čas** - virtuálna reprezentácia času v simulovanom systéme
- **Mechanizmus postupu času** - algoritmus určujúci, ako simulácia prechádza od jednej udalosti k ďalšej

Tieto princípy sú kľúčové pre pochopenie sofistikovanejších formalizmov vrátane DEVS, ktorý je ústredným predmetom tejto práce [18].

### 2.2 Formalizmus DEVS

Formalizmus DEVS (Discrete Event System Specification) vyvinul Bernard P. Zeigler ako matematický základ pre modelovanie a analýzu diskrétnych udalostných systémov [17]. DEVS poskytuje formálny rámec pre špecifikáciu hierarchických modulárnych modelov, čo umožňuje systematický prístup k definícii systémov s diskrétnym správaním. Významnosť tohto formalizmu spočíva v jeho teoretickej podloženosti, flexibilita a podpore pre kompozíciu modelov, čo ho robí vhodným pre široké spektrum aplikácií od technických systémov cez biologické procesy až po simuláciu sociálnych javov [15].

### 2.2.1 Matematická definícia

Atomický DEVS model je definovaný ako 7-mica [18]:

$$M = \langle X, Y, S, \delta_{ext}, \delta_{int}, \lambda, t_a \rangle \quad (2.1)$$

kde:

- $X$  je množina vstupných hodnôt
- $Y$  je množina výstupných hodnôt
- $S$  je množina stavov
- $\delta_{ext} : Q \times X \rightarrow S$  je externá prechodová funkcia, kde  $Q = \{(s, e) | s \in S, 0 \leq e \leq t_a(s)\}$  je úplný stav systému
- $\delta_{int} : S \rightarrow S$  je interná prechodová funkcia
- $\lambda : S \rightarrow Y$  je výstupná funkcia
- $t_a : S \rightarrow \mathbb{R}_0^+ \cup \{\infty\}$  je funkcia postupu času

Táto formálna definícia poskytuje presný matematický základ pre reprezentáciu správaní diskretných udalostných systémov a umožňuje ich analýzu a kompozíciu.

### 2.2.2 Zložený DEVS model

Zložený DEVS model umožňuje vytvárať hierarchické štruktúry spájaním jednoduchších komponentov. Je definovaný ako 8-mica [18]:

$$N = \langle X, Y, D, \{M_d\}, \{I_d\}, \{Z_{i,d}\}, Select \rangle \quad (2.2)$$

kde:

- $X$  je množina vstupných hodnôt
- $Y$  je množina výstupných hodnôt
- $D$  je množina mien (identifikátorov) komponentov
- $\{M_d | d \in D\}$  je množina DEVS komponentov
- $\{I_d | d \in D\}$  definuje vplyvy pre každý komponent
- $\{Z_{i,d} | i \in I_d, d \in D\}$  sú prepojovacie funkcie
- $Select : 2^D \rightarrow D$  je funkcia riešenia konfliktov

Táto definícia podporuje modulárny prístup k modelovaniu.

## 2.3 Stavové automaty v DEVS modelovaní

Stavové automaty predstavujú dôležitý koncept v DEVS modelovaní, keďže poskytujú intuitívny spôsob špecifikácie správania systémov [8]. Konečný automat je definovaný ako 5-tica:

$$A = \langle \Sigma, S, s_0, \delta, F \rangle \quad (2.3)$$

kde:

- $\Sigma$  je vstupná abeceda
- $S$  je konečná množina stavov
- $s_0 \in S$  je počiatkový stav
- $\delta : S \times \Sigma \rightarrow S$  je prechodová funkcia
- $F \subseteq S$  je množina akceptačných stavov

V kontexte DEVS modelovania sú stavové automaty rozšírené o časové aspekty a výstupné funkcie, čo umožňuje modelovať komplexnejšie správanie. Tento prístup kombinuje výhody formálneho popisu s intuitívnou vizuálnou reprezentáciou, čím uľahčuje tvorbu a analýzu modelov [14].

## 2.4 Implementačné aspekty DEVS simulácie

### 2.4.1 Architektúra DEVS simulačných systémov

Implementácia DEVS formalizmu do funkčného simulačného systému si vyžaduje vhodne navrhnutú architektúru, ktorá zachováva teoretické vlastnosti formalizmu a zároveň poskytuje efektívne prostredie pre vývoj modelov [15].

Klasické architektúry DEVS implementácií typicky rozlišujú tieto kľúčové vrstvy [18]:

- **Vrstva modelu** - implementuje formálnu špecifikáciu atomických a zložených DEVS modelov
- **Vrstva simulátora** - zodpovedá za správne vykonávanie modelov podľa DEVS sémantiky
- **Vrstva experimentu** - riadi priebeh simulácie a zber výsledkov

### 2.4.2 Reprezentácia DEVS modelov v programovacích jazykoch

Prevod matematického DEVS formalizmu do konkrétneho programovacieho jazyka predstavuje výzvu v zachovaní formálnej sémantiky a zároveň vytvorení použiteľného API. Wainer [15] a Zeigler [18] diskutujú viaceré prístupy:

- **Objektovo-orientovaná reprezentácia** - využíva dedičnosť a polymorfizmus na reprezentáciu DEVS hierarchie
- **Funkcionálne prístupy** - modelujú prechodové funkcie ako čisté funkcie bez vedľajších efektov

- **Deklaratívne prístupy** - špecifikujú model ako konfiguračnú štruktúru, ktorú interpretuje simulačný motor

V kontexte webových aplikácií sú populárne najmä:

- **JSON reprezentácie** - pre serializáciu modelov a ich prenos medzi klientom a serverom
- **Objektové grafy** - pre dynamickú manipuláciu s modelom v pamäti počas behu aplikácie

## 2.5 Webové technológie

### 2.5.1 Architektúra klient-server

Moderné webové simulačné prostredia typicky využívajú architektúru klient-server [14], kde:

- **Klient (prehliadač)** zodpovedá za:
  - Používateľské rozhranie pre tvorbu a editáciu modelov
  - Vizualizáciu modelov a simulačných výsledkov
  - Základnú validáciu vstupov používateľa
- **Server** zodpovedá za:
  - Uloženie modelov v databáze
  - Vykonávanie náročných simulačných výpočtov
  - Poskytovanie API pre klientsku aplikáciu

Táto architektúra prináša niekoľko výhod, ktoré Wainer [15] sumarizuje ako:

### 2.5.2 Asynchrónna komunikácia klient-server

Webové DEVS prostredie typicky využíva asynchrónnu komunikáciu medzi klientom a serverom [15]:

- **REST API** - pre základnú CRUD funkcionality nad modelmi
- **WebSockets** - pre real-time aktualizácie počas simulácie
- **Server-Sent Events** - pre streaming výsledkov simulácie

### 2.5.3 Vizuálne programovanie pre DEVS modelovanie

Vizuálne programovanie predstavuje intuitívny prístup k tvorbe modelov bez potreby písania kódu. Pre DEVS modelovanie sa typicky používajú postupy, ktoré Wainer [15] kategorizuje ako:

- **Node-based editory** - modely sú reprezentované ako grafy uzlov a prepojení
- **Stavové diagramy** - pre špecifikáciu stavov a prechodov atomických modelov
- **Blokové diagramy** - pre reprezentáciu zložených modelov a ich prepojení

Výhody vizuálneho programovania pre DEVS zahŕňajú intuitívnejšie pochopenie štruktúry a toku udalostí v modeli.

## 2.5.4 Serializácia a deserializácia modelov

V distribuovanom prostredí webových aplikácií je kľúčovou schopnosťou efektívna serializácia modelov [14] pre:

- Uloženie v databáze
- Prenos medzi klientom a serverom
- Export/import modelov

## 2.6 Existujúce nástroje

### 2.6.1 Nástroje založené na DEVS formalizme

V oblasti DEVS modelovania a simulácie existuje niekoľko významných nástrojov a knižníc, ktoré Wainer [14] a Zeigler [18] analyzujú:

- **DEVSJAVA** – Java implementácia DEVS formalizmu vyvinutá Zeiglerom a jeho tímom na Arizonskej univerzite.
- **PowerDEVS** – nástroj pre modelovanie a simuláciu hybridných systémov založený na DEVS.
- **PythonDEVS** – ľahká implementácia DEVS formalizmu v jazyku Python.

### 2.6.2 Nástroje pre vizuálnu editáciu blokových schém

Okrem nástrojov striktno vychádzajúcich z DEVS formalizmu existujú aj populárne platformy pre vizuálnu tvorbu blokových modelov, ktoré sa v praxi využívajú najmä v oblasti automatizácie, IoT a distribuovaných systémov. Medzi najvýznamnejšie patria Node-RED a 4diac.

#### Node-RED

Node-RED je open-source vizuálny editor postavený na technológii Node.js. Umožňuje jednoduché vytváranie dátových tokov prepájaním uzlov, ktoré reprezentujú vstupy, výpočty a výstupy. Používa sa predovšetkým v oblasti internetu vecí (IoT), pri spracovaní udalostí a pri návrhu jednoduchých automatizačných procesov. Hoci nevyužíva DEVS formalizmus, jeho blokový prístup a asynchrónne spracovanie správ ponúkajú podobné koncepty v oblasti modelovania správania systémov [12].

#### Eclipse 4diac

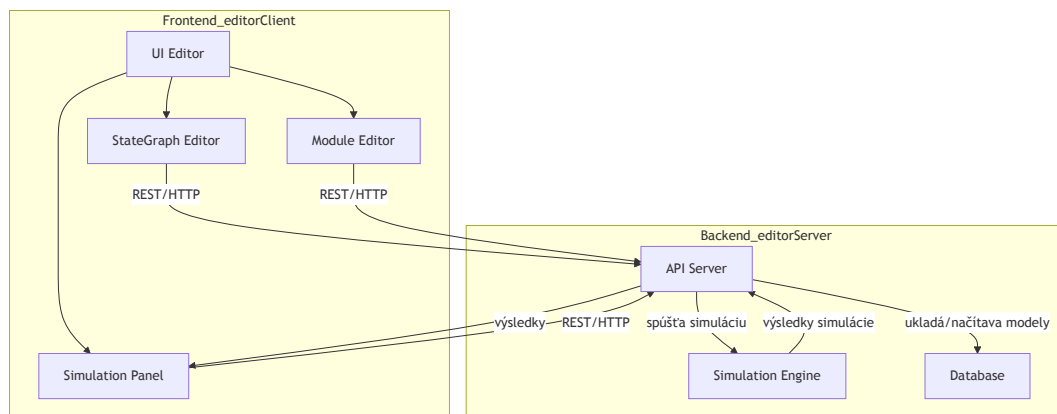
4diac je open-source riešenie pre návrh distribuovaných riadiacich systémov založené na štandarde IEC 61499. Poskytuje grafické rozhranie pre návrh funkčných blokov (Function Block Diagrams – FBD), ktoré komunikujú pomocou udalostí. 4diac sa využíva najmä v oblasti priemyselnej automatizácie. Hoci tiež nepracuje priamo s DEVS, jeho modulárny a udalostami riadený model je do veľkej miery kompatibilný s princípmi formálneho modelovania dynamických systémov [1].

# Kapitola 3

## Návrh a implementácia

### 3.1 Architektúra systému

System som postavil na klasickej klient-server architektúre. Diagram architektúry systému znázorňuje obrázok 3.1



Obr. 3.1: Diagram architektúry systému

#### 3.1.1 Komunikačná architektúra

Klient a server spolu komunikujú cez zabezpečené pripojenie pomocou protokolu HTTPS. Na prenos dát používam REST API, kde sú údaje odosielané a prijímané vo formáte JSON. Tento formát je ľahko čitateľný a jednoducho sa s ním pracuje na oboch stranách.

Server som nastavil tak, aby podporoval HTTPS spojenie. Na vývojové účely používam vlastné (self-signed) certifikáty, ktoré má server uložené v súboroch. Server načíta tieto certifikáty pri štarte a vytvorí zabezpečené spojenie cez HTTPS na špecifikovanom porte. Je dostupný cez všetky sieťové rozhrania (0.0.0.0). V praxi by sa ale mali použiť certifikáty podpísané dôveryhodnou autoritou, najmä kvôli dôveryhodnosti v prehliadačoch.

### 3.1.2 Autentifikácia a zabezpečenie

Každý prístup k API musí byť autorizovaný pomocou špeciálneho API kľúča. Tento kľúč sa odosiela v hlavičke požiadavky pod názvom `x-api-key`. Na strane servera som implementoval funkciu (middleware), ktorá tento kľúč overí. Ak kľúč chýba alebo je nesprávny, server požiadavku odmietne a vráti chybu 403 – zakázané. Tento spôsob zabezpečenia nie je dokonalý, ale poskytuje základnú ochranu pred neželaným prístupom.

### 3.1.3 Perzistencia dát

Na ukladanie modelov na serveri používam databázu SQLite. Ide o jednoduché riešenie, ktoré nevyžaduje samostatný databázový server, a zároveň umožňuje efektívne pracovať s uloženými dátami.

Modely ukladám do tabuľky s názvom `flows`. Táto tabuľka obsahuje napríklad ID modelu, jeho názov, popis, dátumy vytvorenia a úpravy, a najmä stĺpec `data`, kde sa nachádza samotný model vo formáte JSON. Takýto prístup mi umožňuje ukladať rôzne typy modelov bez potreby meniť štruktúru databázy.

K databáze som vytvoril plnohodnotné REST API, ktoré umožňuje vytvárať nové modely (cez POST), načítať zoznam existujúcich (GET), upravovať modely (PUT alebo PATCH) a odstraňovať ich (DELETE). Každá operácia vracia zodpovedajúci HTTP kód a chybové hlásenie v prípade problémov, aby bolo možné API jednoducho integrovať s klientskou aplikáciou.

## 3.2 Použité technológie

### 3.2.1 Frontend (Editor)

Frontend aplikácie som implementoval pomocou moderných webových technológií, ktoré zabezpečujú responzívne a interaktívne užívateľské rozhranie:

- **React + TypeScript:** Základom frontendu je knižnica React, ktorá umožňuje tvorbu komponentového užívateľského rozhrania s deklaratívnym prístupom [2]. Použitie TypeScriptu pridáva statické typovanie, čo zvyšuje robustnosť kódu a poskytuje lepšiu podporu vo vývojových prostrediach [9].
- **Retе.js:** Špecializovaná knižnica pre tvorbu vizuálnych node-based editorov, ktorá poskytuje infraštruktúru pre manipuláciu s uzlami a prepojeniami. Túto knižnicu som rozšíril o vlastnú funkcionálnu špecifickú pre DEVS modelovanie [7].
- **Ant Design:** Komponentová knižnica poskytujúca množstvo predpripravených UI elementov ako modálne okná, formulárové prvky, tlačidlá a ďalšie komponenty, ktoré urýchľujú vývoj a zabezpečujú konzistentný vzhľad aplikácie [3].
- **Vite:** Moderný build nástroj, ktorý zabezpečuje rýchlu kompiláciu, hot-reload funkcionálnu a efektívne balíčkovanie aplikácie pre produkciu [16].

### 3.2.2 Backend

Backend aplikácie som postavil na Node.js platforme s využitím Express frameworku pre tvorbu RESTful API:

- **Node.js + Express:** Kombinácia Node.js runtime prostredia a Express frameworku poskytuje efektívnu platformu pre vývoj serverových aplikácií v JavaScripte [5, 13]. Express zjednodušuje implementáciu API endpointov, sprostredkovateľ funkcií a ďalších serverových komponentov.
- **SQLite:** Databáza, ktorá nevyžaduje samostatný databázový server, čo zjednodušuje nasadenie aplikácie. SQLite poskytuje všetky potrebné funkcie pre perzistenciu dát v rámci jednoduchšieho projektu [6].
- **HTTPS s vlastnými certifikátmi:** Implementácia zabezpečenej komunikácie pomocou HTTPS protokolu s využitím self-signed certifikátov [11].
- **CORS podpora:** Sprostredkovateľ pre Express, ktorý umožňuje bezpečnú komunikáciu medzi frontendovou aplikáciou bežiacou na inej doméne a API serverom [10].

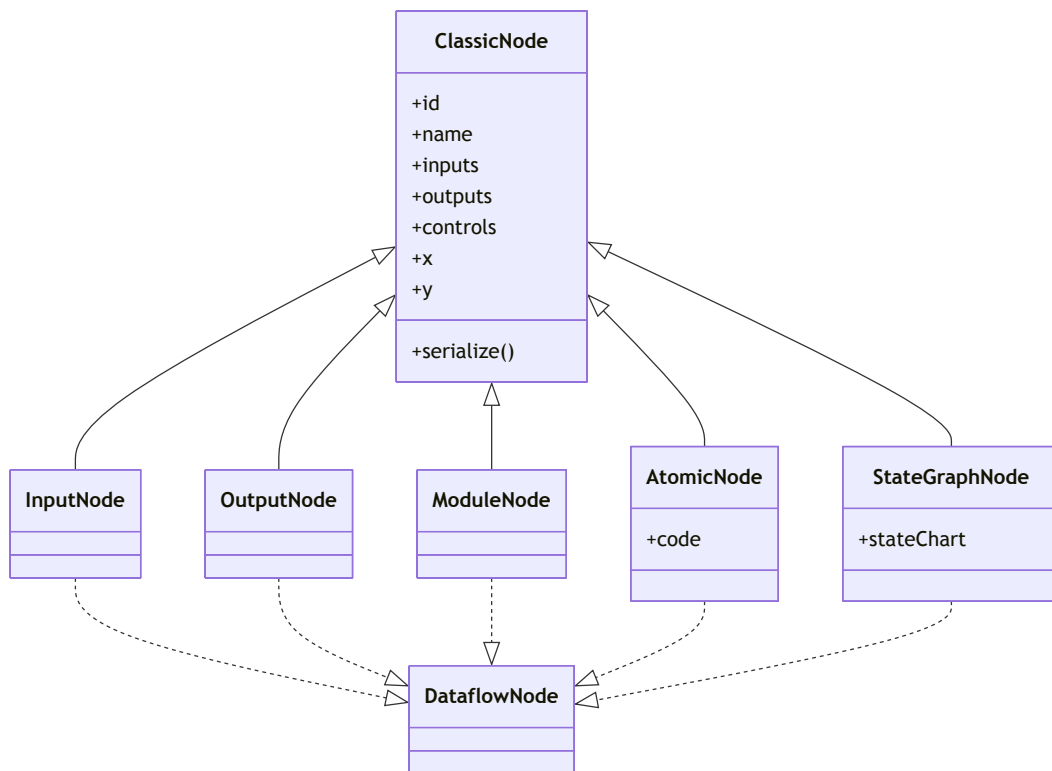
### 3.3 Navrh komponentov

#### 3.3.1 Editor Core

Jadro editora som postavil na knižnici `Rete.js`, ktorú som rozšíril o špecifickú funkcionálnosť pre DEVS modelovanie. Centrálnym prvkom implementácie je funkcia `createEditor`, ktorú som navrhol na inicializáciu a konfiguráciu editora pre konkrétne potreby aplikácie. Táto funkcia vytvára novú inštanciu `NodeEditor` a konfiguruje ju pomocou rôznych pluginov, ktoré poskytujú základnú funkcionálnosť pre prácu s vizuálnym editorom.

Medzi kľúčové pluginy patrí `AreaPlugin` pre správu pracovnej plochy, `ConnectionPlugin` pre vytváranie a správu prepojení, a `ReactRenderPlugin` umožňujúci použitie React komponentov pre vykresľovanie. Dôležitou súčasťou implementácie je úprava schémy modulu `CustomContextMenu`, ktorý poskytuje kontextovo závislé operácie podľa typu vybraného prvku v editore. Po inicializácii základných modulov funkcia `createEditor` registruje vlastné typy uzlov a konfiguruje správanie editora pri rôznych udalostiach, čím prispôbujem editor špecifickým potrebám DEVS modelovania.

Na nasledujúcom diagrame znázorňujem hlavné triedy súvisiace s uzlami a ich väzby na základné komponenty knižnice `Rete.js`.



Obr. 3.2: Diagram tried hlavného editora

### 3.3.2 Simulačné jadro a adaptér modelov

#### DEVSSimulator

Trieda `DEVSSimulator` tvorí základ simulačného systému a zabezpečuje beh simulácie podľa DEVS formalizmu. Obsahuje metódy na načítanie modelu, jeho simuláciu a sledovanie priebehu.

Metóda `loadModel` načíta model zo vstupu pomocou triedy `NodeFactory`, ktorá vytvára konkrétne uzly podľa ich typu. Uzly ukladám do mapy pre rýchle vyhľadávanie počas simulácie. Potom spracujem prepojenia medzi uzlami – pre každý spoj nájdem zdrojový a cieľový uzol a vytvorím medzi nimi spojenie.

Metóda `simulate` riadi celý priebeh simulácie. Na začiatku nastavím čas a pridám úvodné udalosti do fronty. Simulácia potom prebieha v cykle, kde sa postupne spracúvajú udalosti podľa ich času. Pri každej udalosti aktualizujem stav uzla a môžu vzniknúť nové udalosti, ktoré opäť zaradím do fronty. Počas simulácie zaznamenávam históriu, ktorá pomáha analyzovať výsledky.

#### ModelAdapter

Trieda `ModelAdapter` slúži na prevod modelu z formátu použitého v editore do formátu, ktorý používa simulátor. Hlavnú funkciu `adaptModelFromEditor` som navrhol tak, aby najprv skontrolovala vstupný model a podľa jeho typu zvolila správny postup konverzie.

Ak ide o modulárny model, použijem funkciu `adaptModuleStructure`. V prípade bežného modelu vytvorím novú štruktúru a postupne prevediem jednotlivé uzly a spojenia pomocou funkcie `adaptSingleNode`. Táto funkcia sa stará o správne spracovanie každého typu uzla – ak ide o uzol s vlastným kódom, vykonám aj jeho prevod na DEVS funkcie.

Tento spôsob prevodu mi umožňuje, aby editor a simulátor fungovali nezávisle, no zároveň spolupracovali. Vizuálnu časť modelu tak môžem vyvíjať samostatne, bez zásahov do simulačnej logiky, čo uľahčuje údržbu aj budúce rozšírenia systému.

# Kapitola 4

## Popis algoritmov

### 4.1 Návrh interného modelu schémy

Interný model systému som založil na princípoch DEVS (Discrete Event System Specification) formalizmu, ktorý poskytuje matematickú základňu pre modelovanie diskretných udalostných systémov. Tento formalizmus som zvolil pre jeho teoretickú podloženosť a schopnosť efektívne modelovať systémy s diskretným správaním v čase. Základnou stavebnou jednotkou modelu je hierarchická štruktúra uzlov a prepojení, ktorá mi umožňuje vytvárať komplexné systémy z jednoduchších komponentov.

#### 4.1.1 Základné triedy a dátové štruktúry

Návrh interného modelu som realizoval prostredníctvom sady tried, ktoré reprezentujú jednotlivé prvky modelovaného systému (pozri diagram tried v prílohe B.1). Základom tejto štruktúry je trieda `BaseNode`, ktorú som implementoval v module `baseNode.js`. Táto trieda definuje spoločnú funkcionálnosť pre všetky typy uzlov vrátane správy identifikátorov, názvov, vstupno-výstupných portov a vnútorného stavu. Konštruktor triedy inicializuje potrebné dátové členy, zabezpečuje nastavenie jedinečného identifikátora a východzieho prázdneho stavu.

Trieda `BaseNode` poskytuje tieto kľúčové metódy:

- `connectOutput` – vytvára spojenie medzi výstupným portom aktuálneho uzla a vstupným portom cieľového uzla
- `initialize` – nastavuje počiatočný stav pred spustením simulácie
- `getState` – poskytuje aktuálny stav pre účely simulácie a logovania
- `processEvent` – spracováva prichádzajúce udalosti a generuje odpovede

Prepojenia medzi uzlami predstavujú samostatné dátové štruktúry definujúce spojenie výstupného portu jedného uzla so vstupným portom druhého uzla. Každé prepojenie obsahuje kompletnú informáciu o zdroji a cieľi signálu vrátane identifikátorov príslušných uzlov a portov. Tento prístup som zvolil pre jednoznačnú definíciu toku signálov v systéme, čo mi umožňuje efektívne smerovanie počas simulácie.

Pre podporu hierarchických modelov som v simulátore vytvoril triedu `CoupledNode`, zatiaľ čo v editore som túto funkcionálnosť implementoval pomocou triedy `ModuleNode`. Táto rozdielna terminológia odráža odlišné zameranie komponentov – editor pracuje s vizuálnymi

modulmi, zatiaľ čo simulátor implementuje DEVS formalizmus s konceptom coupled modelov. Obe triedy slúžia na zapuzdrenie vnorených komponentov a poskytujú mechanizmy pre správu týchto komponentov, ich vzájomných prepojení a komunikáciu s externým prostredím. Implementáciu som realizoval pomocou mapy komponentov indexovanej podľa identifikátorov a poľa interných prepojení pre efektívnu správu hierarchickej štruktúry.

#### 4.1.2 Validácia modelu

V záujme zabezpečenia robustnosti a konzistencie modelov som do systému implementoval viacúrovňový proces validácie. Tento proces zahŕňa tri hlavné aspekty:

- Kontrolu existencie portov pri vytváraní prepojení
- Detekciu neplatných vnútorných prepojení v hierarchických štruktúrach
- Zabezpečenie typovej kompatibility pripájaných portov

Kontrola existencie portov prebieha v metóde `connectOutput`, ktorá najprv overuje dostupnosť výstupného portu na zdrojovom uzle. Pri neúspechu generuje informatívne chybové hlásenie s identifikáciou problému, vrátane identifikátora uzla a názvu neexistujúceho portu. Obdobný proces overenia aplikujem aj na vstupný port cieľového uzla.

Významným aspektom validácie je typová kontrola portov, ktorá rozlišuje medzi dátovými a udalostnými portami. Na tento účel som implementoval špecializované typy portov – `dataSocket` a `eventSocket`. Toto rozlíšenie zabezpečujem pre korektné prepojenie kompatibilných typov portov a bránim vytváraniu nekonzistentných prepojení medzi portami rôznych typov. Pri serializácii uzlov zahŕňam do výstupnej reprezentácie informáciu o type každého portu, čím zabezpečujem konzistenciu typov medzi vizuálnou reprezentáciou v editore a výpočtovým modelom v simulátore.

## 4.2 Implementácia stavových uzlov

Stavové uzly predstavujú kľúčový element pre modelovanie diskretných stavových automátov v systéme DEVS. Pri ich návrhu som sa zameral na vytvorenie intuitívneho rozhrania pre definíciu stavov a prechodov, s dôrazom na efektívnu implementáciu na strane simulátora.

### 4.2.1 Architektúra stavového uzla

Implementáciu stavového uzla som rozdelil do dvoch kooperujúcich vrstiev: užívateľskej reprezentácie pre vizuálny editor a serverovej logiky pre simulátor. Toto rozdelenie som zvolil pre jasné oddelenie prezentačnej a výpočtovej vrstvy systému, čo mi umožňuje nezávislý vývoj a optimalizáciu jednotlivých častí.

Na strane editora som implementoval triedu `StateGraphNode`, ktorá rozširuje základný typ uzla z knižnice `Rete.js` a implementuje rozhranie `DataflowNode`. Táto trieda spravuje komplexnú štruktúru stavového automatu prostredníctvom atribútu `stateGraph`, ktorý uchováva definíciu stavov, prechodov a ich vlastností. Pre manipuláciu s touto štruktúrou trieda poskytuje rozhranie na pridávanie a odoberanie stavov a prechodov. Používateľské rozhranie uzla je definované v komponentne `StateGraphNode`, ktorá zabezpečuje vizuálne zobrazenie stavu, jeho názvu a interakcií v rámci editora. Podrobnosti o štruktúre tohto komponentu uvádzam v sekcii 5.3.

Metóda `serialize` zabezpečuje prevod vizuálnej reprezentácie stavového automatu do formátu vhodného pre simulátor. Táto metóda transformuje vnútornú reprezentáciu na štruktúrovaný objekt.

Na strane simulátora som vytvoril zodpovedajúcu implementáciu, ktorá interpretuje definíciu stavového automatu a realizuje jeho správanie počas simulácie. Implementácia zahŕňa mechanizmy pre vyhodnocovanie podmienok prechodov, ich vykonávanie a generovanie výstupných udalostí.

#### 4.2.2 Algoritmus spracovania udalostí

Pre zabezpečenie efektívneho spracovania udalostí v stavovom uzle som navrhol algoritmus implementovaný v metóde `processEvent`. Pri návrhu som zvažoval dve alternatívne implementácie: rekurzívny prístup s postupným vyhodnocovaním všetkých možných prechodov, alebo optimalizovaný prístup založený na mapovaní stavov na dostupné prechody. Zvolil som druhý prístup, keďže poskytuje výrazne lepšiu časovú zložitosť  $O(1)$  pri vyhľadávaní dostupných prechodov z aktuálneho stavu, čo je kritické pre výkon pri komplexných modeloch s vysokým počtom stavov a prechodov.

Výsledkom spracovania je zoznam vygenerovaných výstupných udalostí pre ďalšie použitie v simulátore.

#### 4.2.3 Algoritmus pre vykresľovanie prepojení v stavovom grafe

Pri implementácii editora stavových automatov som narazil na obmedzenie knižnice `Rete.js`, ktoré sa prejavovalo pri vykresľovaní viacerých prechodov medzi tými istými uzlami. Všetky tieto prechody boli zobrazené ako jedna spoločná čiara, čo používateľovi znemožňovalo rozlíšiť jednotlivé prechody. Aby som tento problém odstránil, rozhodol som sa navrhnúť a implementovať vlastný algoritmus na výpočet trás prechodov, ktorý zabezpečí ich zreteľné a samostatné zobrazenie.

Základný princíp algoritmu spočíva v transformácii priamočiarych prepojení na krivky s ohybom, ktoré sú od seba vizuálne oddelené. V praxi to znamená, že namiesto priamky medzi dvoma bodmi sa vytvorí lomená čiara s medzi bodmi, ktoré ležia na kolmiciach k pôvodnej spojnici.

Prvou fázou algoritmu je detekcia viacnásobných prepojení. Pre každé prepojenie sa skontroluje, či medzi rovnakým párom stavov už existujú iné prepojenia. Ak sa zistí, že ide o jediné prepojenie, zostáva zachovaná priama čiara. V opačnom prípade sa určí poradové číslo prepojenia v množine všetkých prepojení medzi danými dvoma stavmi.

V druhej fáze sa vypočítava hodnota offsetu pre každé viacnásobné prepojenie. Tento výpočet berie do úvahy niekoľko faktorov: základný offset (štandardne 30 pixelov), poradie prepojenia a logický smer prepojenia. Algoritmus zabezpečuje, že offset alternuje medzi kladnými a zápornými hodnotami ( $-30$ ,  $+30$ ,  $-60$ ,  $+60$  atď.) a postupne sa zväčšuje s každým ďalším prepojením. Toto zabezpečuje vizuálnu symetriu a optimálne rozloženie prechodov. Ukážka sa nachádza na obrázku [5.3](#).

#### 4.2.4 Dynamická konfigurácia portov

Dynamickú konfiguráciu portov som implementoval vo všetkých typoch uzlov – `AtomicNode`, `StateGraphNode` aj `ModuleNode`. Pre viac informácií o vizuálnej implementácii a užívateľskej interakcii pozri [5.5](#).

## 4.3 Generovanie exportu pre simulátor

### 4.3.1 Transformácia modelu

Jadrom prevodu modelov z editora do simulátora je funkcia `adaptModelFromEditor`, ktorú som implementoval v module `modelAdapter.js`. Táto funkcia realizuje komplexnú transformáciu vizuálneho modelu na štruktúru optimalizovanú pre simuláciu.

Pre konverziu jednotlivých uzlov som vytvoril funkciu `adaptSingleNode`, ktorá realizuje prevod podľa typu uzla. Zvláštnu pozornosť som venoval prevodu prepojení, kde bolo potrebné explicitne definovať zdrojový a cieľový uzol a port pre každé prepojenie v simulačnom modeli. Výsledkom tohto procesu je kompletná reprezentácia modelu pripravená na simuláciu.

### Konverzia užívateľského kódu

Významnou súčasťou prevodu modelov je transformácia užívateľského kódu definovaného v atomických uzloch. Preto som implementoval funkciu `convertProcessFunctionToDEVS`, ktorá prevádza užívateľský kód na sériu funkcií požadovaných DEVS formalizmom.

Proces konverzie začína kontrolou validity vstupného kódu. V prípade neplatného vstupu generujem predvolenú implementáciu s pasívnym správaním, čím zabezpečujem odolnosť systému voči chybným vstupom. Pre platný kód vytváram kompletnú sadu DEVS funkcií vrátane inicializácie stavu, spracovania externých udalostí, a ďalších podporných funkcií.

### Spracovanie hierarchických štruktúr

Pre podporu komplexných hierarchických modelov som vytvoril funkciu `createCoupledNodeFromModule`, ktorá transformuje modulárne štruktúry z editora na vnožené reprezentácie v simulátore.

# Kapitola 5

## Užívateľské rozhranie

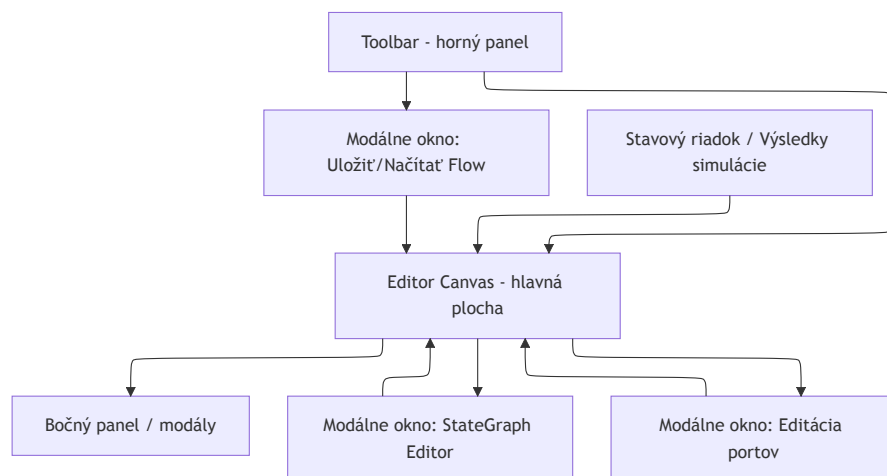
### 5.1 Hlavné rozhranie

Užívateľské rozhranie editora som vytvoril ako jednostránkovú React aplikáciu s využitím knižnice Rete.js pre diagramovú časť a Ant Design pre štandardné UI komponenty. Celý systém som riešil modulárne s oddelením logiky spracovania od prezentačnej vrstvy.

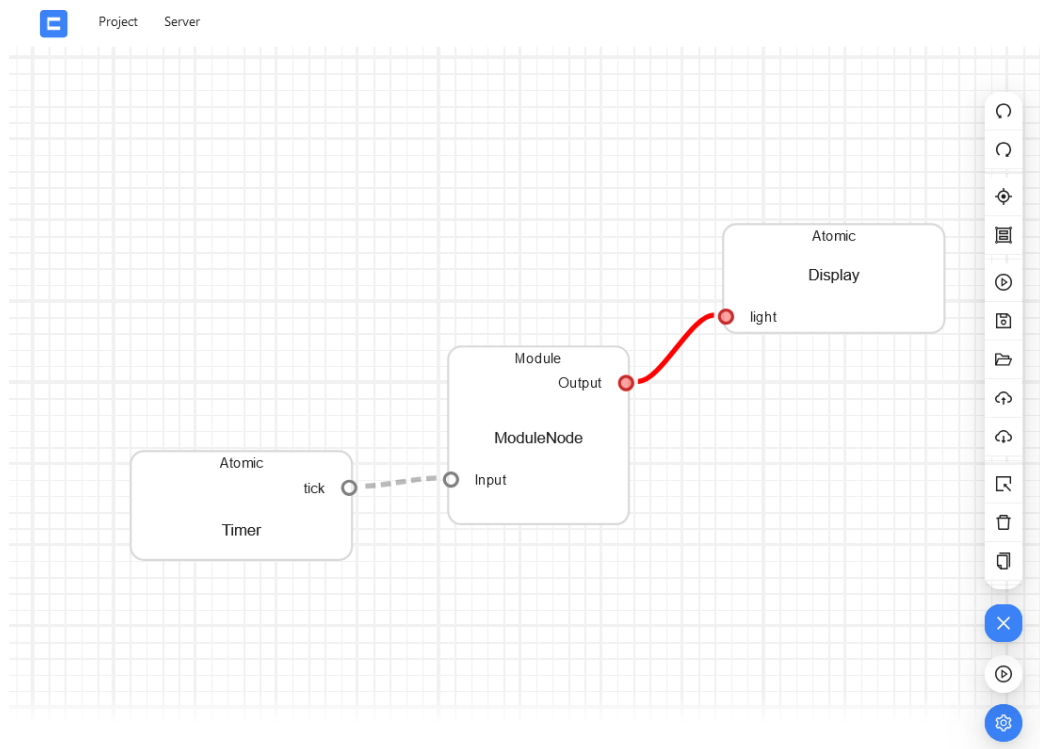
#### 5.1.1 Základná štruktúra rozhrania

Užívateľské rozhranie som rozdelil do niekoľkých hlavných častí. V hornej časti sa nachádza hlavné menu poskytujúce prístup k funkcionalitám ako správa súborov, editácia a nastavenia zobrazovania. Centrálnu časť tvorí editor samotný, implementovaný pomocou knižnice Rete.js, ktorý slúži na vizuálnu tvorbu a úpravu diagramov. V spodnej časti sa nachádza panel s dostupnými uzlami na výber a panel s detailami aktuálne vybraného uzla.

Návrh používateľského rozhrania znázorňujem na diagrame 5.1. Obsahuje hlavnú pracovnú plochu, bočný panel s blokmi a horné menu nástrojov.



Obr. 5.1: Diagram používateľského rozhrania



Obr. 5.2: Hlavný editor s atomickým uzlom a modulom

### 5.1.2 Komponentná štruktúra

Hlavnú aplikáciu som implementoval v komponente App, ktorá spravuje stav celej aplikácie a organizuje jednotlivé časti rozhrania. V rámci tohto komponentu som definoval nasledujúce kľúčové časti:

- **Stav projektu a editora** - systém implementuje funkcionality na výber projektu, jeho ukladanie a inicializáciu editora pomocou vlastných React hookov.
- **Stav modálnych okien a editácie uzlov** - aplikácia spravuje stavy pre rôzne modálne okná slúžiace na editáciu uzlov, vrátane editora kódu pre atomické uzly a editora stavových automatov.
- **Stav bočných panelov** - implementácia zahŕňa systém pre správu bočných panelov, najmä pre konfiguráciu portov uzla.

Celkovú štruktúru komponentu som navrhol tak, aby zahŕňala hlavný layout s hlavičkou obsahujúcou menu, centrálny editor a množstvo podradených komponentov pre editáciu a konfiguráciu, ako sú modálne okná pre úpravu kódu, stavových grafov a portov. Súčasťou rozhrania sú aj plávajúce tlačidlá poskytujúce rýchly prístup k často používaným funkciám.

### 5.1.3 Integrácia s Rete.js a React

Editor využíva React plugin pre Rete.js, ktorý sprostredkováva komunikáciu medzi React komponentami a Rete.js API. Táto integrácia je realizovaná pomocou špeciálneho renderovacieho modulu, ktorý umožňuje použitie React komponentov priamo v kontexte Rete.js.

Implementácia zahŕňa vytvorenie vlastného rendererového objektu, ktorý je nakonfigurovaný tak, aby používal prispôbené komponenty pre kontextové menu a ďalšie prvky rozhrania. Toto riešenie poskytuje niekoľko výhod:

- Možnosť využívať React komponenty priamo v Rete.js uzloch
- Obojsmerné prepojenie stavu medzi React a Rete.js
- Vlastné vykresľovanie a štylizáciu všetkých prvkov grafu

## 5.2 Práca s uzlami

### 5.2.1 Pridávanie uzlov

Uzly sa môžu pridávať do diagramu viacerými spôsobmi:

1. **Z kontextového menu** - užívateľ môže pravým klikom na plochu editora vyvolať kontextové menu, ktoré ponúka možnosti na pridanie rôznych typov uzlov. Implementácia zahŕňa funkcie na vytvorenie nového uzla na pozícii kurzora a jeho následné pridanie do editora.
2. **Z dock panelu** - druhým spôsobom je pretiahnutie už existujúceho typu uzla z panelu nástrojov. Táto funkcionálna je implementovaná pomocou `DockPlugin`, ktorý umožňuje definovať predpripravené uzly, ktoré môže užívateľ jednoducho pretiahnuť na pracovnú plochu.
3. **Klonovaním** - tretí spôsob je duplikovanie existujúceho uzla. Systém obsahuje funkcie na klonovanie uzlov rôznych typov, pričom zachováva všetky ich vlastnosti vrátane konfigurácie, kódu a špecifických nastavení.

### 5.2.2 Typy uzlov a ich vizuálna reprezentácia

#### Uzol definovaný kódom

Atomický uzol je základným prvkom pre implementáciu vlastnej logiky spracovania. Je reprezentovaný triedou `AtomicNode`, ktorá rozširuje štandardnú triedu uzla z knižnice `Rete.js` a implementuje rozhranie `DataflowNode`.

Atomický uzol poskytuje nasledujúce funkcionality:

- Definovanú šírku a výšku pre konzistentné zobrazenie
- Možnosť definície užívateľského kódu v JavaScripte
- Základné vstupné a výstupné porty pre dátové toky
- Ovládacie prvky pre úpravu názvu a otvorenie editora kódu

Užívateľ môže otvoriť editor kódu kliknutím na tlačidlo `Edit Code`. Otvorí sa modálne okno. Ukážka rozhrania sa nachádza na obrázku [5.4](#).

## Uzol stavového automatu

Uzol stavového automatu umožňuje modelovať správanie pomocou konečného automatu. Implementácia triedy `StateGraphNode` rozširuje štandardný uzol a obsahuje komplexnú vnútornú reprezentáciu stavového grafu.

Kľúčové vlastnosti tohto uzla zahŕňajú:

- Prispôsobené rozmery pre zobrazenie stavového grafu
- Vnútornú dátovú štruktúru pre reprezentáciu stavov a prechodov
- Ovládacie prvky pre editáciu názvu a otvorenie editora stavového grafu
- Podporu pre dynamické porty, ktoré sa môžu prispôsobiť požiadavkám užívateľa

Predvolená štruktúra stavového grafu obsahuje počiatočný a koncový stav s jednoduchým prechodom. Užívateľ môže tento graf rozšíriť a upraviť pomocou špecializovaného editora, ktorý sa otvorí po kliknutí na tlačidlo `Edit Graph`. Otvorí sa modálne okno. Ukážka rozhrania sa nachádza na obrázku 5.3.

## InputNode a OutputNode

Pre definíciu rozhraní v hierarchickom modeli som implementoval špeciálne typy uzlov:

- **InputNode** - uzol reprezentujúci externý vstup do modulu
- **OutputNode** - uzol reprezentujúci výstup z modulu

Tieto uzly slúžia na dynamické pridávanie portov v zložitom bloku modelu a definovanie hraníc a rozhraní medzi rôznymi úrovňami v modeli.

### 5.2.3 Manipulácia s uzlami

Editor poskytuje intuitívne spôsoby manipulácie s uzlami:

- **Pohyb uzlov** - užívateľ môže jednoducho presúvať uzly ťahaním pomocou myši
- **Zmena veľkosti** - veľkosť uzlov sa automaticky prispôbuje ich obsahu, najmä počtu portov a ovládacích prvkov
- **Mazanie** - uzly je možné odstrániť pomocou kontextového menu alebo klávesovej skratky Delete
- **Výber** - užívateľ môže vybrať uzol kliknutím alebo vytvoriť výber viacerých uzlov ťahaním obdĺžnika
- **Viacnásobný výber** - s použitím klávesy CTRL je možné postupne pridávať uzly do výberu

Implementáciu týchto funkcionalít som realizoval pomocou rozšírení poskytovaných knižnicou `Rete.js`, konkrétne `AreaExtensions`, ktoré poskytujú nástroje na prácu s výberom uzlov a interakciu s nimi.

## 5.3 Editácia stavových uzlov

### 5.3.1 Architektúra modálneho editora stavov

Editor stavových grafov je implementovaný ako modálne okno s vlastnou inštanciou `Rete.js`. Tento prístup umožňuje vytvoriť špecializovaný editor optimalizovaný pre prácu so stavovými automatmi. Editor využíva vlastné renderovacie komponenty a špecifické správanie pre tento typ diagramu.

V rámci inicializácie editora nastavujem nasledovné komponenty a moduly:

- Základný editor uzlov (`NodeEditor`)
- Modul pre správu pracovnej plochy (`AreaPlugin`)
- Modul pre správu prepojení (`ConnectionPlugin`)
- React modul pre vykreslenie komponentov (`ReactPlugin`)
- Vlastné komponenty pre zobrazenie stavov a prechodov
- Modul pre automatické usporiadanie uzlov (`AutoArrangePlugin`)

Špeciálne komponenty pre stavový graf zahŕňajú kruhové uzly reprezentujúce stavy, špeciálne porty pre prechody a vylepšené prepojenia s popisom udalostí. Automatické usporiadanie je nakonfigurované s parametrami optimalizovanými pre stavové diagramy.

### 5.3.2 Štruktúra stavového editora

Stavový editor podporuje komplexnú prácu so stavovými automatmi:

- Vytváranie a editáciu stavov (reprezentovaných ako uzly)
- Definíciu prechodov medzi stavmi (reprezentovaných ako prepojenia)
- Konfiguráciu vstupných udalostí, strážnych podmienok a výstupných udalostí na prechodoch
- Špecifikáciu počiatočného stavu

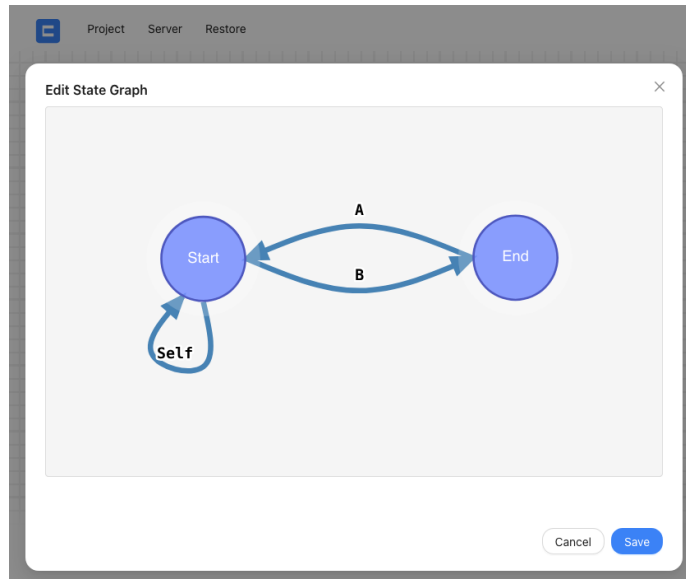
### 5.3.3 Komponenta modálneho okna

Komponenta `StateGraphModal` je React komponenta, ktorá vytvára modálne okno obsahujúce editor stavového grafu. Táto komponenta spravuje životný cyklus editora a zabezpečuje komunikáciu medzi hlavnou aplikáciou a editorom stavového grafu.

Implementácia zahŕňa nasledujúce kľúčové prvky:

- Referencia na DOM element pre vytvorenie editora
- Stav editora a správa jeho životného cyklu
- Stav pre zobrazenie modálneho okna na editáciu prepojení
- Spracovanie udalostí pre editáciu prechodov v stavovom grafe
- Funkcie na ukladanie zmien v grafe späť do hlavnej aplikácie

Pri ukladaní zmien sa z editora extrahujú informácie o uzloch (stavoch) a prepojeniach (prechodoch), ktoré sa následne transformujú do formátu vhodného pre hlavnú aplikáciu. Komponenta tiež obsahuje vnorené modálne okno pre editáciu vlastností prechodov.



Obr. 5.3: Editor stavového automatu s príkladom jednoduchého grafu stavov a prechodov

### 5.3.4 Editácia prechodov

Jednou z kľúčových funkcionalít stavového editora je možnosť konfigurácie prechodov medzi stavmi. Na tento účel slúži komponenta `ConnectionEditModal`, ktorá poskytuje rozhranie pre definovanie vlastností prechodu:

- Spúšťacia udalosť (*trigger event*)
- Strážna podmienka (*guard condition*)
- Výstupná udalosť (*output event*)
- Typ prechodu (externý alebo interný)

Komponenta je implementovaná ako formulár s poliami pre jednotlivé vlastnosti. Pri otvorení sa inicializuje aktuálnymi hodnotami vybraného prechodu a po uložení sa zmeny aplikujú späť na prechod v stavovom grafe.

### 5.3.5 Špecializované renderovacie komponenty

Pre optimálne zobrazenie stavového automatu som implementoval špeciálne renderovacie komponenty:

**Kruhovú uzly** Komponenta `CircleNode` vykresľuje stavy ako kruhové uzly s textom názvu stavu v strede. Implementácia zahŕňa spracovanie vlastností ako ID, štítok, rozmery a stav výberu. Uzol tiež obsahuje porty pre pripojenie prechodov.

**Výpočet pozícií portov** Pre estetické zobrazenie prechodov v kruhovom uzle som implementoval triedu `ComputedSocketPosition`, ktorá zabezpečuje rozloženie portov po obvode kruhu. Táto trieda počíta pozície portov na základe ich indexu a celkového počtu, čo umožňuje rovnomerne rozložiť pripojenia okolo obvodu uzla.

**Výpočet trasy prechodu** Aby boli jednotlivé prechody medzi rovnakými uzlami zreteľne odlišné, implementoval som vlastný algoritmus na výpočet ich trás. Podrobnosti o tomto algoritme sú uvedené v časti 4.2.3.

## 5.4 Editor kódu

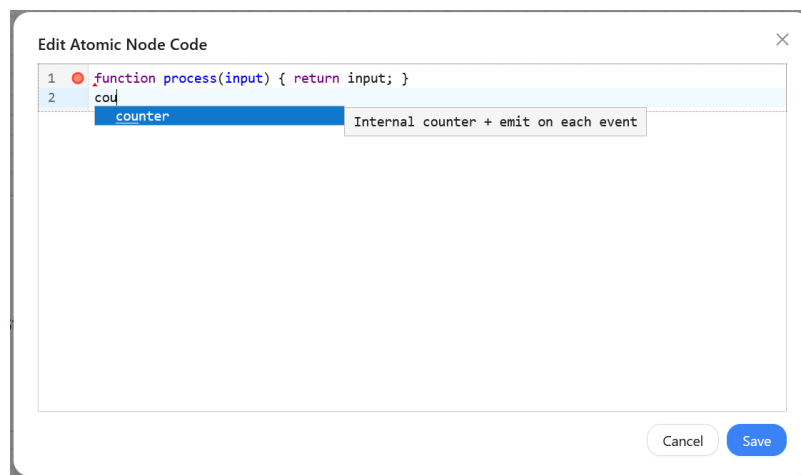
Pre atomické uzly systém poskytuje pokročilý editor kódu založený na knižnici CodeMirror 6, ktorý umožňuje užívateľom implementovať vlastnú logiku spracovania signálov.

### 5.4.1 Integrácia CodeMirror

Komponenta `AtomicCodeModal` vytvára modálne okno s editorom kódu pre atomické uzly. Implementácia zahŕňa:

- Inicializáciu a správu inštancie CodeMirror editora
- Detekciu preferovanej témy (svetlá/tmavá) pre konzistentný vzhľad
- Konfiguráciu špecializovaného systému automatického dopĺňania kódu
- Synchronizáciu obsahu editora s modelom aplikácie

Pre zlepšenie užívateľského zážitku editor implementuje vlastné automatické dopĺňanie, ktoré zohľadňuje kontext atomického uzla, vrátane dostupných vstupných a výstupných portov. Tento systém poskytuje kontextové návrhy pre premenné, funkcie a kódové fragmenty.



Obr. 5.4: Editor kódu pre atomické uzly s ukázkou syntaxe a kontextovou nápovedou

V používateľskom rozhraní komponentu sa zobrazujú informácie o dostupných vstupoch a výstupoch daného uzla, čo uľahčuje orientáciu používateľa pri implementácii logiky spracovania. Komponent zároveň obsahuje sekciu s nápovedou a ukázkami kódu, ktoré demonštrujú správne použitie API.

### 5.4.2 Validácia a chybové hlásenia

Editor kódu implementuje pokročilú validáciu pomocou linteru pre JavaScript. Tento nástroj vykonáva:

- Základnú kontrolu syntaxe JavaScriptu
- Overenie prítomnosti povinnej funkcie `process`
- Detekciu potenciálnych problémov a chýb v kóde

V prípade detekcie problému systém zobrazuje chybové hlásenia priamo v editore, s informáciou o type chyby a jej umiestnení v kóde. Pri syntaktických chybách sa extrahuje pozícia chyby z výnimky a chybové hlásenie sa umiestňuje na príslušný riadok a stĺpec.

## 5.5 Správa portov (sockets)

Kritickým aspektom užívateľského rozhrania je správa vstupných a výstupných portov uzlov, tzv. "socketov". Každý uzol môže mať rôzne typy portov, ktoré ovplyvňujú jeho správanie a kompatibilitu s inými uzlami.

Na úpravu portov uzlov slúži špeciálny bočný panel (drawer) implementovaný v komponente `SocketsNodeEditDrawer`. Tento komponent umožňuje:

- Zobrazenie a správu zoznamu vstupných portov
- Zobrazenie a správu zoznamu výstupných portov
- Pridávanie nových portov oboch typov
- Odstránenie existujúcich portov
- Úpravu názvov a typov portov
- Uloženie zmien a aktualizáciu konfigurácie uzla

Implementáciu som realizoval tak, aby zahŕňala správu stavov pre vstupné a výstupné porty, funkcie pre manipuláciu s týmito zoznamami a rozhranie pre zobrazenie a editáciu vlastností portov. Pri otvorení panela sa z uzla extrahuje aktuálna konfigurácia portov, ktorá sa potom zobrazí užívateľovi na editáciu.

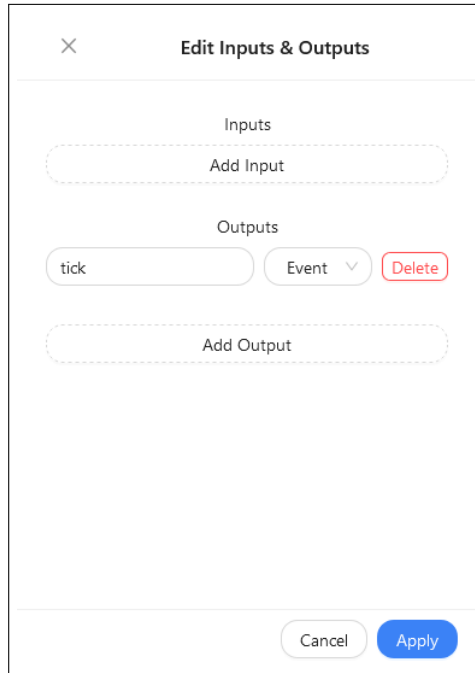
Užívateľské rozhranie bočného panela som organizoval pomocou záložiek, ktoré oddeľujú vstupné a výstupné porty. V rámci každej záložky je zoznam aktuálnych portov s možnosťou úpravy ich názvov a typov, ako aj tlačidlá na pridanie nových portov.

### 5.5.1 Typy portov

V systéme som definoval dva základné typy portov:

- **Dátový port** - určený na prenos dátových hodnôt medzi uzlami
- **Eventový port** - určený na prenos udalostí a signálov

Tieto základné typy sú rozšírené o ďalšie vlastnosti prostredníctvom dedičnosti.



Obr. 5.5: Bočný panel pre konfiguráciu vstupných a výstupných portov uzla

## 5.6 Pokročilé funkcie editora

### 5.6.1 História a navigácia v editore

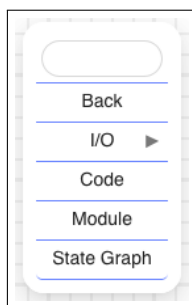
Editor poskytuje základné navigačné nástroje – napríklad modul `AreaPlugin` umožňuje používateľovi plynule posúvať a približovať (zoom) pracovnú plochu editora. Záznam histórie zmien sa spracúva cez `HistoryPlugin`, ktorý uchováva informácie o operáciách ako pridávanie, odstraňovanie či posúvanie uzlov, ako aj o pridávaní a mazaní prepojení medzi nimi. Funkcie späť (undo) a vpred (redo) potom využívajú tieto záznamy; štandardné klávesové skratky (napr. `Ctrl+Z` a `Ctrl+Y`) na aktiváciu týchto operácií možno jednoducho povoliť pomocou rozšírenia `HistoryExtensions.keyboard`. Tak je zabezpečené, že editor dokáže vrátiť posledné zmeny alebo opätovne obnoviť odvolané akcie podľa požiadaviek používateľa.

### 5.6.2 Interaktívne prvky

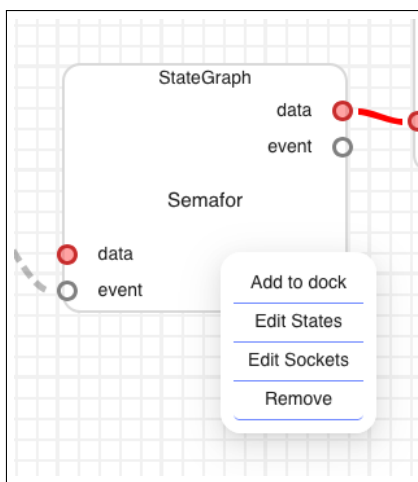
Editor obsahuje viaceré interaktívne prvky, ktoré zlepšujú ovládanie a prehľadnosť modelu:

- **Plávajúce tlačidlá (Floating Buttons):** kruhové tlačidlá s ikonou, ktoré sa zobrazujú nad ostatným obsahom (V pravom dolnom rohu obrázka 5.2), ponúkajú rýchly prístup k najčastejším akciám (napr. pridanie nového uzla alebo jeho úprava).
- **Kontextové menu:** pravým kliknutím na pracovnú plochu alebo na konkrétny uzol sa otvorí kontextová ponuka s relevantnými možnosťami. Pre plochu obsahuje zoznam preddefinovaných typov uzlov na vloženie (Obrázok 5.6), zatiaľ čo pre vybraný uzol sú k dispozícii akcie ako *Odstrániť* alebo *Duplikovať* (Obrázok 5.7).
- **Automatické usporiadanie (Auto-arrange):** špeciálny plugin využíva algoritmus knižnice `elkjs` na automatické rozmiestnenie uzlov v priestore. Po zapnutí tejto fun-

kcie plugin analyzuje štruktúru grafu a navrhuje optimálne usporiadanie uzlov tak, aby bol diagram čo najprehľadnejší.



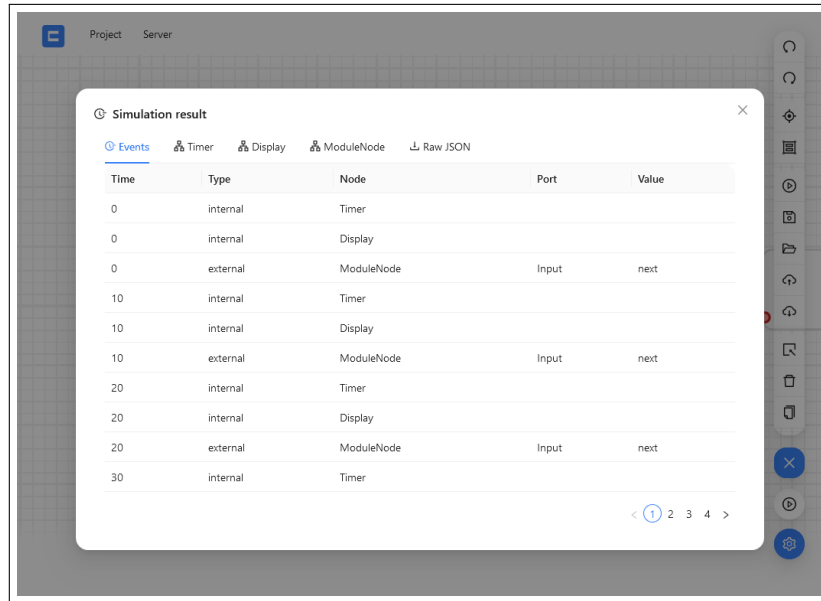
Obr. 5.6: Kontextové menu po kliknutí na pracovnú plochu



Obr. 5.7: Kontextové menu po kliknutí na uzol.

### 5.6.3 Simulácia a výsledky

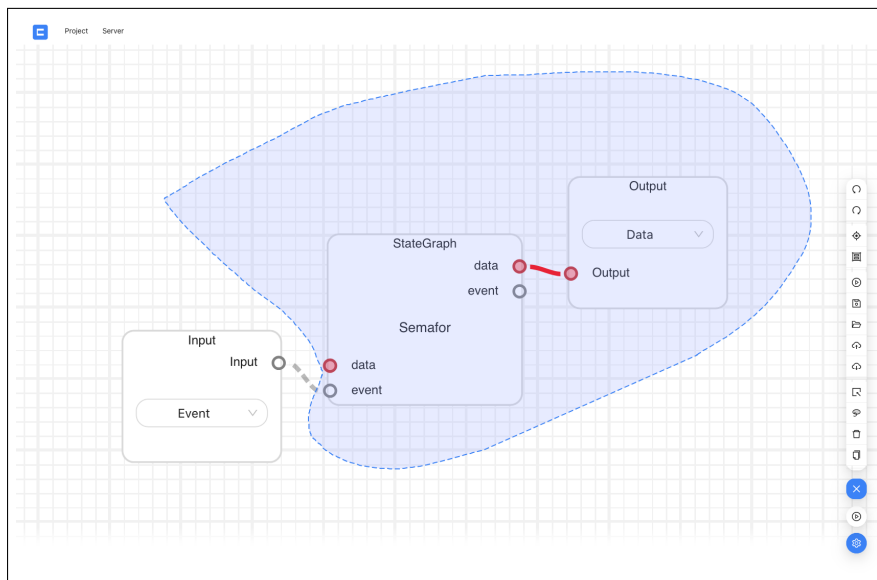
Editor ponúka možnosť spustenia simulácie vytvoreného modelu priamo z používateľského prostredia, ako je znázornené na sekvenčnom diagrame v prílohe B.2. Po inicializácii simulácie sa model odošle na simulačný server (napr. formou HTTP požiadavky) a priebežné dáta simulácie (napr. aktuálny stav modelu či generované udalosti) sa asynchrónne prenášajú späť do klienta. Výsledky simulácie sa zobrazujú vo vyhradenom komponente používateľského rozhrania, ktorý formou tabuliek zobrazuje numerické dáta, v logu prezentuje chronologický zoznam udalostí (napr. spustenie či ukončenie procesov) a vizualizuje priebeh simulácie (napr. pomocou progress baru alebo indikátora stavov). Tento výsledkový komponent sa dynamicky aktualizuje na základe prichádzajúcich dát zo servera, čím poskytuje používateľovi prehľad o priebehu a výsledkoch simulácie.



Obr. 5.8: Modálne okno so zobrazením výstupu simulácie v podobe textového logu.

## 5.7 Typy selektora výberu uzlov

Editor umožňuje používateľovi vyberať uzly niekoľkými spôsobmi. Okrem štandardného výberu kliknutím je k dispozícii aj selekcia ťahaním výberového obdĺžnikového rámcu (rect) alebo voľným ťahaním myšou (lasso). Uzly, ktoré pretínajú nakreslený výberový tvar, sú zahrnuté do výberu. Používateľ môže medzi týmito režimami (obdĺžnikový rámec, voľný tvar, jednoduchý klik) prepnúť pomocou príslušného ovládacieho prvku v rozhraní (napr. prepínača módu selektora), čím si jednoducho zvolí preferovaný spôsob výberu viacerých uzlov.



Obr. 5.9: Použitie výberového nástroja (lasso) pre označenie viacerých uzlov v editore.

## 5.8 Ukladanie a načítavanie modelov

### 5.8.1 SaveFlowModal

Na ukladanie modelov slúži modálne okno `SaveFlowModal`, ktoré poskytuje rozhranie pre zadanie metadát ukladaného modelu. Implementácia zahŕňa:

- Stav pre názov modelu a detailný popis
- Inicializáciu stavov pri otvorení modálneho okna
- Formulár s poľami pre zadanie názvu a popisu
- Funkcie na spracovanie zmien v poliach a odosielanie dát

Po potvrdení formulára sa názov a detaily odovzdávajú späť do hlavnej aplikácie, ktorá zabezpečí uloženie modelu spolu s týmito metadátami. Ukážka používateľského rozhrania sa nachádza na obrázku [5.10](#)



Obr. 5.10: Modálne okno pre uloženie modelu na server.

### 5.8.2 Modálne okno

Na načítavanie modelov zo servera slúži modálne okno `LoadFlowModal`, ktoré poskytuje rozhranie pre výber z uložených modelov. Implementácia zahŕňa:

- Vyhľadávanie v zozname uložených modelov
- Výber modelu zo zoznamu
- Zobrazenie detailov o modeloch vrátane názvu, popisu a času poslednej aktualizácie
- Funkcie na potvrdenie výberu a načítanie vybraného modelu

Rozhranie obsahuje vyhľadávacie pole, ktoré umožňuje filtrovať zoznam modelov podľa názvu, a zoznam dostupných modelov s možnosťou ich výberu. Pri každom modeli sa zobrazujú detaily, po výbere modelu a potvrdení sa tento model načíta do editora. Ukážka používateľského rozhrania sa nachádza na obrázku [5.11](#)



Obr. 5.11: Modálne okno pre načítanie modelu na server.

## 5.9 Zhrnutie užívateľského rozhrania

Užívateľské rozhranie editora poskytuje komplexné prostredie na vytváranie a simuláciu DEVS modelov s týmito kľúčovými vlastnosťami:

1. **Interaktívny diagram** - grafická reprezentácia modelov s možnosťou ťahania a úpravy uzlov a spojení
2. **Hierarchický model** - podpora modulárnosti a zapuzdrenia cez vnorené komponenty a stavové automaty
3. **Špecializované editory** - editor kódu pre atomické uzly a editor stavového automatu pre stavové uzly
4. **Flexibilná konfigurácia** - rozširiteľná architektúra s možnosťou pridávania nových typov uzlov a portov
5. **Pokročilé funkcie** - história, automatické usporiadanie, možnosť výberu typu selektora uzlov, validácia modelu, simulácia s vizualizáciou

Užívateľské rozhranie som navrhol s ohľadom na intuitívnosť, pričom kombinuje vizuálny prístup s možnosťou detailnej špecifikácie správania prostredníctvom kódu a konfigurácie.

## Kapitola 6

# Testovanie systému a používateľského rozhrania

Testovanie softvéru je nevyhnutnou súčasťou vývoja každej komplexnej aplikácie. V kontexte tejto práce má testovanie kľúčový význam pre overenie správnosti implementácie a rýchlu detekciu chýb. Rozlišujeme dve hlavné oblasti testovania: backendovú (simulačnú) časť a používateľské rozhranie. Tieto časti sa líšia nielen svojím technickým zameraním, ale aj použitou metodikou. Zatiaľ čo testovanie DEVS simulátora sa vykonáva prevažne automatizovane a zameriava sa na korektnosť logiky samotnej simulácie, testovanie používateľského rozhrania sa sústreďí na intuitívnosť, konzistenciu a správnosť vizuálnej prezentácie.

### 6.1 Testovanie DEVS simulátora

DEVS simulátor predstavuje serverovú (simulačnú) časť aplikácie, v ktorej sa spracovávajú modely a vykonávajú diskkrétne udalosti. Keďže ide o jadro systému, jeho spoľahlivosť a konzistencia sú kritické. Testovanie tejto časti preto prebieha automatizovane pomocou predpripravených testovacích súborov vo formáte JSON. Každý takýto súbor obsahuje špecifikáciu modelu alebo testovací scenár, ktorý má simulátor spracovať. Počas testovania systém spúšťam s určitým testovacím parametrom (napríklad príkazovým riadkom `-test`), ktorý umožňuje automatické spustenie všetkých definovaných testovacích scenárov.

Pre účely testovania som definoval viacero typov testovacích prípadov:

- **Základné modely:** Jednoduché príklady s minimálnymi komponentami, ktoré overujú základnú funkčnosť simulátora. Napríklad atómový model s jedným vstupným a jedným výstupným portom kontroluje, či simulátor spracuje elementárne prvky systému správne.
- **Stavové automaty:** Testovacie scenáre zahŕňajú atómové modely definované prostredníctvom stavových automatov s viacerými prechodmi a zdržaniami. Týmto spôsobom sa overuje, či simulátor korektne mení stavy a odosiela správy podľa navrhnutých prechodov.
- **Modulárne modely:** Zložitejšie scenáre predstavujú kompozitné (vnorené) modely skladajúce sa z viacerých atómových komponentov. Cieľom je overiť správnu synchronizáciu a komunikáciu medzi jednotlivými modulmi (coupled models), ako aj konzistentné výsledky pri kombinácii viacerých modelov.

- **Hraničné prípady:** Testy extrémnych alebo neštandardných situácií (napr. prázdny model, neplatne definované porty či nadmerne veľké parametre). Tieto scenáre zabezpečujú, že simulátor správne reaguje aj v neočakávaných situáciách bez zrútenia.

Výsledky testov vyhodnocujem podľa úspešnosti dokončenia simulácie bez runtime chýb. Namiesto priameho porovnávania číselných výstupov sledujem, či simulácia úspešne dobehla do konca. Tento prístup je nevyhnutný vzhľadom na rozsah a dynamiku dát generovaných simuláciou – konkrétne hodnoty môžu byť rozsiahle a ťažko porovnateľné. Predpokladám deterministické správanie simulátora, takže pri rovnakých vstupoch simulácia vždy skončí konzistentne. Automatizovaný mechanizmus zaznamenáva prípadné výnimky alebo chyby, ktoré vzniknú počas behu simulácie.

## 6.2 Testovanie používateľského rozhrania

Testovanie používateľského rozhrania (frontend) vykonávam manuálne a zameriavam sa na kvalitu interakcie používateľa so softvérom. Na rozdiel od automatizovaných testov simulátora ide predovšetkým o subjektívne hodnotenie použiteľnosti a konzistencie rozhrania. Táto fáza testovania je nevyhnutná pre interaktívne aplikácie, pretože odhaľuje chyby, ku ktorým pri automatizovanom testovaní nemusí dôjsť.

Metodológiu testovania užívateľského rozhrania som rozdelil do viacerých kategórií:

- **Prieskumné testovanie (Exploratory testing):** Systematicky skúmam celé používateľské rozhranie bez vopred definovaných scenárov. Pri tom experimentujem s rôznymi vstupmi vrátane neplatných alebo netypických hodnôt (napríklad chybné priradenie portov) a sledujem reakciu aplikácie. Tento prístup mi umožňuje identifikovať neočakávané chyby v ovládaní, nedostatočnú validáciu vstupov a iné nekonzistencie v správaní aplikácie.
- **Scenárové testovanie (Scenario testing):** Vytváram realistické používateľské scenáre simulujúce bežné úlohy. Napríklad testujem postupnosť vytvorenia modelu – od definovania atómových komponentov cez spájanie portov až po spustenie simulácie a analýzu výsledkov. Pri týchto testoch overujem správnu postupnosť krokov a konzistenciu stavov v užívateľského rozhrania (napríklad indikátory priebehu simulácie) a sledujem aj výkon aplikácie pri náročnejších operáciách (ako načítanie rozsiahlych modelov), čo mi umožňuje identifikovať a odstrániť prípadné spomalenia.
- **Testovanie použiteľnosti (Usability testing):** Táto časť sa sústreďuje na pohodlie a efektívnosť pre koncového používateľa. Testovanie zahŕňa posúdenie prehľadnosti rozhrania, zrozumiteľnosti ovládacích prvkov a informatívnosti chybových hlásení.
- **Testovanie kompatibility (Compatibility testing):** Kontrolujem správne fungovanie aplikácie v rôznych prostrediach. Testujem spustenie vo viacerých webových prehliadačoch (Chrome, Firefox, Edge) a na rozličných operačných systémoch. Zvlášť dbám na testovanie pri rôznych rozlíšeniach a veľkostiach obrazovky, aby bol dizajn responzívny. V tejto fáze som odhalil drobné problémy s rozložením používateľského rozhrania, ktoré sa prejavili len za špecifických podmienok (napríklad orezané prvky pri netradičnom rozlíšení) a následne som ich upravil pre lepšiu použiteľnosť.

# Kapitola 7

## Záver

Táto bakalárska práca sa sústreďovala na návrh a vývoj webového editora určeného na modelovanie a simuláciu diskretných udalostných systémov vychádzajúcich z DEVS formalizmu. Ambíciou bolo sprístupniť tento typ modelovania aj používateľom bez hlbších znalostí formálnych metód, a to prostredníctvom intuitívneho vizuálneho prostredia dostupného priamo vo webovom prehliadači.

Navrhnutý systém využíva architektúru klient–server s dôrazom na modulárnosť a rozširiteľnosť. Používateľské rozhranie je implementované v Reacte s použitím knižnice Rete.js na vizuálnu tvorbu modelov, zatiaľ čo serverová časť v Node.js zabezpečuje spracovanie simulácie na základe DEVS špecifikácie a ukladanie modelov do databázy SQLite. Kľúčovými súčasťami systému sú editor pre definíciu stavových a atomických uzlov, hierarchické zoskupovanie komponentov a základná vizualizácia výsledkov simulácie.

Počas implementácie som vyriešil viaceré technické výzvy vrátane synchronizácie portov pri dynamických zmenách uzlov, podpory viacnásobných prepojení v stavových automatoch či prevodu modelov medzi internou a vizuálnou reprezentáciou. Systém zároveň poskytuje API rozhranie so základnou podporou autentifikácie a bezpečnej výmeny dát.

Z pohľadu budúceho vývoja by bolo prínosné integrovať podporu pre interaktívne simulácie v reálnom čase. Takáto funkcionálna by mohla zahŕňať krokovanie simulácie, zvýrazňovanie aktívnych uzlov a animáciu toku správ, čím by sa výrazne zvýšila didaktická hodnota nástroja. Ďalšie rozšírenia môžu zahŕňať podporu distribuovaných a paralelných simulácií, nové modelovacie prístupy ako Petriho siete alebo agentovo orientované systémy, a rozšírené možnosti vizualizácie výsledkov vo forme grafov alebo štatistických prehľadov.

Vytvorený nástroj predstavuje funkčný základ pre webové modelovanie DEVS systémov, ktorý môže nájsť uplatnenie nielen vo výučbe, ale aj pri vývoji a testovaní modelov v praxi. Vďaka kombinácii prístupnosti, vizuálnej interakcie a podpory formálneho jadra môže tento editor uľahčiť prácu so zložitými systémami širokému spektru používateľov bez potreby inštalácie špecializovaného softvéru.

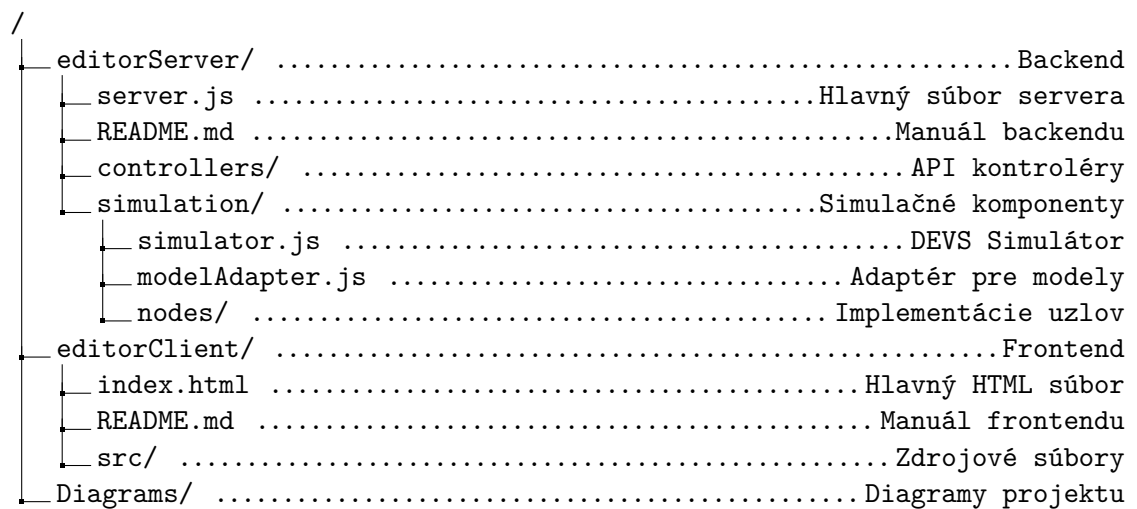
# Literatúra

- [1] ECLIPSE FOUNDATION. *Eclipse 4diac: Open source infrastructure for distributed industrial process measurement and control systems*. 2025. Dostupné z: <https://eclipse.dev/4diac/>. [cit. 2025-05-13].
- [2] FACEBOOK, I. *React – A JavaScript library for building user interfaces*. 2013. Dostupné z: <https://reactjs.org/>. Accessed: 2025-05-12.
- [3] FINANCIAL, A. *Ant Design – A design system for enterprise-level products*. 2016. Dostupné z: <https://ant.design/>. Accessed: 2025-05-12.
- [4] FISHMAN, G. S. *Discrete-event simulation: modeling, programming, and analysis*. 1. vyd. Springer Science & Business Media, 2013. ISBN 978-1-4757-3803-9.
- [5] FOUNDATION, O. *Node.js*. 2009. Dostupné z: <https://nodejs.org/>. Accessed: 2025-05-12.
- [6] HIPPI, D. R. *SQLite - Self-contained, high-reliability, embedded, full-featured SQL database engine*. 2000. Dostupné z: <https://www.sqlite.org/>. Accessed: 2025-05-12.
- [7] HLOPOV, N. *Rete.js – JavaScript framework for visual programming*. 2017. Dostupné z: <https://rete.js.org/>. Accessed: 2025-05-12.
- [8] HOPCROFT, J. E.; MOTWANI, R. a ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation*. 2. vyd. Addison-Wesley, 2001. ISBN 978-0-201-44124-6.
- [9] MICROSOFT. *TypeScript: JavaScript With Syntax For Types*. 2012. Dostupné z: <https://www.typescriptlang.org/>. Accessed: 2025-05-12.
- [10] NETWORK, M. D. *CORS - Cross-Origin Resource Sharing*. N.d. Dostupné z: <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>. Accessed: 2025-05-12.
- [11] NETWORK, M. D. *HTTPS - HyperText Transfer Protocol Secure*. N.d. Dostupné z: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview>. Accessed: 2025-05-12.
- [12] O'LEARY, N. a CONWAY JONES, D. *Node-RED: Low-code programming for event-driven applications*. 2025. Dostupné z: <https://nodered.org/>. [cit. 2025-05-13].
- [13] STRONGLOOP, I. *Express - Fast, unopinionated, minimalist web framework for Node.js*. 2010. Dostupné z: <https://expressjs.com/>. Accessed: 2025-05-12.
- [14] WAINER, G. a MOSTERMAN, P. J. *Introduction to Discrete-Event Modeling and Simulation*. 1. vyd. CRC Press, 2011. ISBN 978-1-4398-1625-9.

- [15] WAINER, G. A. *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. 1. vyd. CRC Press, 2009. ISBN 978-1-4200-5337-1.
- [16] YOU, E. *Vite – Next Generation Frontend Tooling*. 2020. Dostupné z: <https://vitejs.dev/>. Accessed: 2025-05-12.
- [17] ZEIGLER, B. P. *Theory of modeling and simulation*. 1. vyd. John Wiley & Sons, 1976. ISBN 978-0-12-778455-7.
- [18] ZEIGLER, B. P.; PRAEHOFER, H. a KIM, T. G. *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. 2. vyd. Academic press, 2000. ISBN 978-0-12-778455-7.

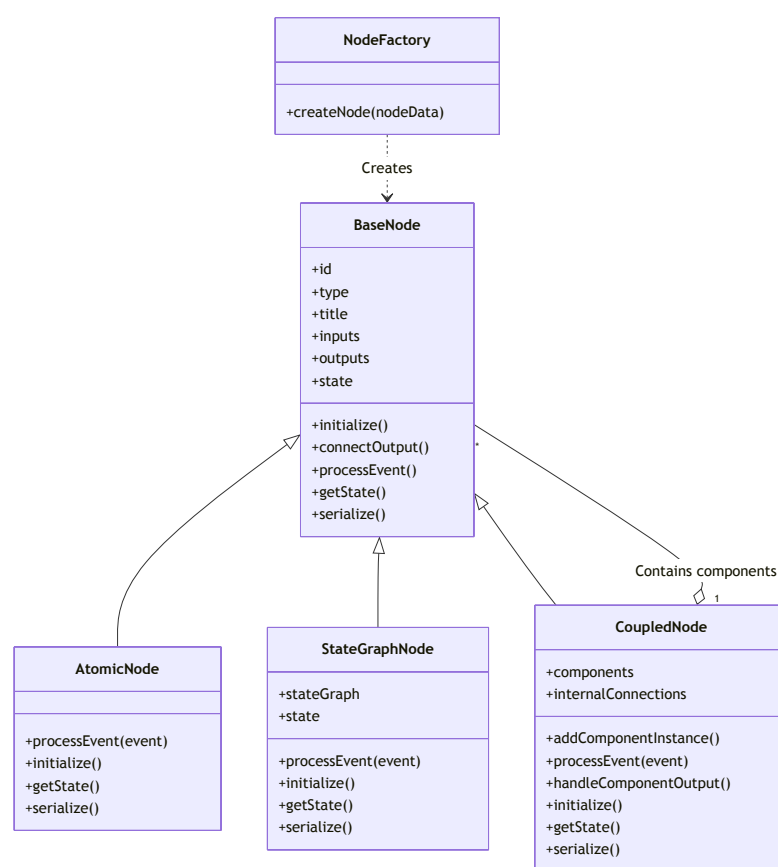
# Príloha A

## Štruktúra súborov

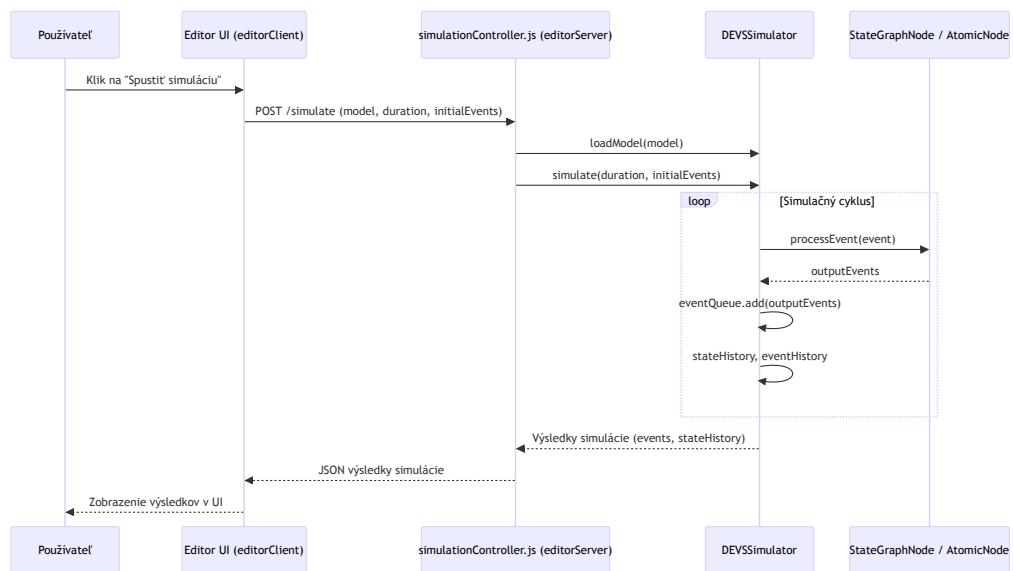


# Príloha B

## Diagramy



Obr. B.1: Diagram tried simulátora



Obr. B.2: Sekvenčný diagram simulácie