PANEURÓPSKA VYSOKÁ ŠKOLA
FAKULTA INFORMATIKY

FI-185022-16338

Návrh a implementácia frameworku pre Petriho siete v jazyku C#

Diplomová práca

2025
Bc. Michal Drobný

PANEURÓPSKA VYSOKÁ ŠKOLA
FAKULTA INFORMATIKY

Návrh a implementácia frameworku pre Petriho siete v jazyku C#

Diplomová práca

Bc. Michal Drobný

Študijný program:	Aplikovaná informatika
Študijný odbor:	Informatika
Školiace pracovisko:	Ústav aplikovanej informatiky
Školiteľ:	prof. RNDr. Gabriel Juhás, PhD.

Bratislava 2025

PANEURÓPSKA VYSOKÁ ŠKOLA

Fakulta informatiky

ZADANIE DIPLOMOVEJ PRÁCE

Meno a priezvisko študenta:	**Bc. Michal Drobný**
Evidenčné číslo diplomovej práce:	**FI-185022-16338**
Študijný odbor:	**informatika**
Študijný program:	**Aplikovaná informatika**
Forma a metóda štúdia:	**externá prezenčná**
Vedúci diplomovej práce:	**prof. RNDr. Gabriel Juhás, PhD.**
Ústav/katedra:	**Ústav aplikovanej informatiky**
Dátum zadania diplomovej práce:	**04. 03. 2025**

Názov: **Návrh a implementácia frameworku pre Petriho siete v jazyku C#**

Anotácia: Cieľom práce je navrhnúť a implementovať framework pre Petriho siete a ich vhodné rozšírenia v jazyku C#. SúČasťou frameworku bude interpreter, ktorý umožní nahratie Petriho siete vo vhodnom formáte (napr. XML alebo JSON), umožní vytváranie inštancií pre vloženú Petriho sieť a sprístupní spúšťanie prechodov inštancií Petriho sietí prostredníctvom aplikačného rozhrania a REST API. Súčasťou frameworku bude aj jednoduchá webová aplikácia, ktorá sprístupní používateľovi zoznam vytvorených inštancií a ich spustiteľných prechodov.

Úlohy:
1. Naštudujte problematiku Petriho sietí.
2. Navrhnite vzorové riešenie frameworku pre Petriho siete.
3. Implementujte vzorové riešenie takto navrhnutého frameworku v jazyku C#.

Literatúra:
HEE, Kees van; AALST, Will van der. Workflow Management Models, Methods, and Systems. Cambridge,: MIT Press, 2002. 384 s. ISBN 0-262-01189-1.

prof. RNDr. Gabriel Juhás, PhD.
vedúci práce

Ing. Juraj Štefanovič, PhD.
vedúci ústavu

Poďakovanie:

Za odborné vedenie, cenné rady a konzultácie, ktoré významne prispeli k vypracovaniu tejto práce, patrí moja vďaka pánovi prof. RNDr. Gabrielovi Juhásovi, PhD. Taktiež by som rád poďakoval RNDr. Jánovi Lackovi, PhD. za jeho neoceniteľnú pomoc pri spracovaní formálnych častí práce a za čas, ktorý mi ochotne venoval počas celého procesu. Osobitné poďakovanie patrí Edite Kyselej za morálnu podporu a povzbudenie v momentoch, keď moja motivácia klesala. Jej záujem a pochopenie boli pre mňa veľkou oporou.

Čestné prehlásenie:

Čestne vyhlasujem, že záverečnú prácu som vypracoval samostatne a že som uviedol všetku použitú literatúru.

. . . . . .

Bc. Michal Drobný

Abstrakt

Táto diplomová práca sa zaoberá návrhom a implementáciou modulárneho a rozšíriteľného frameworku pre Petriho siete v jazyku C#. Výsledný systém umožňuje načítanie Petriho sietí definovaných vo formáte Petriflow XML, ich transformáciu do objektového modelu, logické prepojenie komponentov prostredníctvom PetriNetBuilder a spúšťanie prechodov prostredníctvom rozhrania REST API. Súčasťou riešenia je aj sprievodná webová aplikácia postavená na Laraveli s využitím Livewire, ktorá slúži na vizualizáciu, interakciu a simuláciu sietí v reálnom čase. MongoDB slúži ako úložisko aktuálnych stavov a výsledkov simulácií. Architektúra systému umožňuje budúcu integráciu s protokolom OPC UA a simulačnými nástrojmi ako Factory I/O. Testovanie prebehlo na všetkých úrovniach systému so zameraním na korektnosť a škálovateľnosť. Výsledkom je plne funkčné a samostatne stojace simulačné prostredie vhodné pre akademické aj priemyselné nasadenie.

Kľúčové slová: Petriho sieť, simulácia, C#, REST API, Laravel, Priemysel 4.0

Abstract

The thesis presents the design and implementation of a modular and extensible framework for Petri nets using the C# programming language. The developed system supports the parsing of Petri nets defined in the Petriflow XML format, transformation into internal object models, logical linking of components via the PetriNetBuilder, and execution of transitions through a REST API interface. A supplementary web interface built in Laravel, powered by Livewire, enables visualization, interaction, and real-time simulation of networks. MongoDB serves as the primary data store for state management. The system architecture allows for future integration with OPC UA and industrial simulation platforms such as Factory I/O. Testing was conducted across all system layers, with a focus on correctness and scalability. The outcome is a complete, independently functioning simulation environment suitable for both academic and industrial applications.

Keywords: Petri net, simulation, C#, REST API, Laravel, Industry 4.0

Obsah

Zoznam obrázkov

Zoznam skratiek

.NET	Microsoft .NET Framework Microsoft .NET Framework
API	Application Programming Interface Aplikačné programové rozhranie
BSON	Binary JSON Binárny JSON
CPN	Coloured Petri Net Farebná Petriho sieť
CPN Tools	Coloured Petri Net Tools Nástroj na prácu s farebnými Petriho sieťami
DSL	Domain-Specific Language Doménovo špecifický jazyk
GreatSPN	Generalized Stochastic Petri Net Generalized Stochastic Petri Net
HTTP	Hypertext Transfer Protocol Hypertextový prenosový protokol
IDE	Integrated Development Environment Integrované vývojové prostredie
JSON	JavaScript Object Notation Formát pre výmenu údajov v objektovej štruktúre
JWT	JSON Web Token JSON Web Token
MVC	Model-View-Controller Model-View-Controller

MVEL	MVFLEX Expression Language MVFLEX Expression Language
OOP	Object-oriented programming Objektovo orientované programovanie
OPC UA	Open Platform Communications Unified Architecture Otvorená architektúra pre priemyselnú komunikáciu
ORM	Object-Relational Mapping Objektovo-relačné mapovanie
PIPE	Platform Independent Petri Net Editor Platform Independent Petri Net Editor
REST	Representational State Transfer Architektúra pre tvorbu webových API
SQL	Structured Query Language Štruktúrovaný vyhľadávací jazyk
XML	eXtensible Markup Language Značkovací jazyk pre štruktúrované dáta
XSD	XML Schema Definition XML Schema Definition
XSLT	eXtensible Stylesheet Language Transformations Jazyk pre transformáciu XML dokumentov

1 Úvod

Petriho siete sú široko používaným formalizmom na modelovanie a analýzu distribuovaných systémov, pretože dokážu zachytiť dynamické správanie súbežných procesov (Mejía et al., 2016). Poskytujú grafickú reprezentáciu stavov systému, prechodov a toku informácií alebo zdrojov medzi nimi. Cieľom tejto práce je navrhnúť a implementovať komplexný framework pre Petriho siete v programovacom jazyku C#, ktorý bude možné použiť na modelovanie, simuláciu a analýzu rôznych typov diskrétnych systémov. Hlavné zameranie použitia tohto frameworku bude na simuláciu automatizaťných procesov v mechatronike.

Motivácia pre vývoj robustného a univerzálneho frameworku Petriho sietí v jazyku C# vyplýva zo stupňujúcej sa zložitosti a širokého rozšírenia distribuovaných systémov v rôznych oblastiach. Vzhľadom na stále rastúci dopyt po nástrojoch, ktoré môžu pomôcť pri formálnej špecifikácii, verifikácii a validácii takýchto systémov, sa táto práca snaží odstrániť existujúce nedostatky poskytnutím softvérového frameworku, ktorý je nielen výpočtovo efektívny, ale aj užívateľsky prívetivý a ľahko rozšíriteľný. Petriho siete ponúkajú matematicky pevný prístup, ktorý možno všeobecne aplikovať na riadenie procesov súvisiacich s udalosťami, čo umožňuje simulovať, kontrolovať a poskytovať kvantitatívne a kvalitatívne pochopenie príslušných procesov (Fountas et al., 1999).

Hlavným cieľom tejto práce je vyvinúť modulárny, škálovateľný a rozšíriteľný framework, ktorý možno ľahko integrovať do aplikácií založených na jazyku C#. Tento framework bude poskytovať súbor nástrojov a Application Programming Interface (API) na vytváranie, manipuláciu a vykonávanie modelov Petriho sietí, ako aj prostriedky na serializáciu a deserializáciu reprezentácií Petriho sietí v štandardných formátoch, ako sú eXtensible Markup Language (XML) a JavaScript Object Notation (JSON). Okrem toho bude framework obsahovať RESTful API, ktoré umožní externým aplikáciám komunikovať s Petriho sieťou, spúšťať prechody a vyhľadávať stav siete.

Na prezentáciu možností frameworku bude vyvinutá jednoduchá webová aplikácia na vizualizáciu a interakciu s inštanciami Petriho sietí.

Framework bude navrhnutý s dôrazom na použiteľnosť a rozšíriteľnosť, čo umožní vývojárom ľahko začleniť možnosti modelovania a analýzy Petriho sietí do ich aplikácií založených na jazyku C#. Poskytnutím flexibilnej a na funkcie bohatej platformy má táto práca za cieľ zjednodušit simuláciu pre disktrétne mechatronické systémy s prihliadnutím na potrebu rýchlej odozvy pri simulácií.

Použiteľnosť frameworku bude demonštrovaná na príklade simulácie automatizačného procesu v mechatronike, pričom sa zameriame na modelovanie a analýzu Petriho sietí, ktoré sú schopné efektívne reprezentovať a analyzovať zložitú dynamiku týchto systémov.

Pri návrhu frameworku sa kladie dôraz na podporu rôznych rozšírení základného modelu Petriho sietí, ako sú farebné Petriho siete, hierarchické siete či časované siete. Tieto rozšírenia umožňujú modelovať komplexnejšie scenáre, kde nestačí základná binárna reprezentácia tokenov alebo kde je potrebné zachytiť aspekt času. Zavedením takýchto rozšírení sa framework stáva použiteľnejším pri simuláciách, ktoré si vyžadujú vyššiu mieru expresivity a presnosti, napríklad v oblasti výrobnej logistiky, robotiky alebo adaptívnych riadiacich systémov.

Ďalším dôležitým aspektom tejto práce je zabezpečenie zrozumiteľnosti a intuitívnosti používateľského rozhrania, ktoré bude súčasťou sprievodnej webovej aplikácie. Táto aplikácia nebude slúžiť len ako vizualizačný nástroj, ale aj ako interaktívne prostredie pre testovanie a ladenie modelov. Používatelia budú môcť dynamicky upravovať siete, sledovať tok tokenov, spúšťať prechody a exportovať výsledky simulácií. Tým sa zabezpečí nielen podpora vývojárov, ale aj možnosť výučby a prezentácie princípov Petriho sietí v pedagogickom prostredí.

Napokon, ambíciou tejto práce je pripraviť framework tak, aby bol v budúcnosti schopný integrácie s priemyselnými štandardmi ako OPC UA a nástrojmi na simuláciu ako Factory I/O. Takáto integrácia by umožnila nasadenie vytvoreného riešenia priamo v priemyselných simuláciách a laboratórnych podmienkach. Framework tak môže slúžiť nielen ako akademický nástroj, ale aj ako most medzi teóriou a reálnym nasadením v oblasti priemyselnej automatizácie a mechatronických systémov.

2 Teoretický základ

Petriho siete boli prvýkrát navrhnuté nemeckým matematikom Carlom Adamom Petrim v roku 1962. Ich primárnym cieľom bolo vytvoriť nástroj schopný modelovať komunikačné procesy medzi viacerými automatmi a analyzovať ich správanie v podmienkach paralelizmu a asynchrónnosti (Petri, 1962). Od svojho vzniku sa Petriho siete stali silným teoretickým nástrojom pre skúmanie rôznych systémov, najmä v informatike a inžinierstve.

Petriho siete sú definované ako orientované bipartitné grafy. (Das et al., 2021) uvádzajá vo svojej práci nasedujúcu definíciu orientovaného bipartitného grafu:

Definícia 1 (Orientovaný bipartitný graf)**.** *Orientovaný bipartitný graf je usporiadaná dvojica $G = (V, E)$, kde množina vrcholov V môže byť rozdelená na dve disjunktné množiny X a Y, pričom platí:*

1. *Každá orientovaná hrana (oblúk) z množiny E spája vrchol z množiny X s vrcholom z množiny Y alebo vrchol z množiny Y s vrcholom z množiny X. Neexistujú teda oblúky medzi vrcholmi v rámci tej istej množiny.*
2. *Pre každú dvojicu vrcholov $u, v \in V$ existuje najviac jeden oblúk (u, v) alebo (v, u). Neexistujú teda obojsmerné (recipročné) oblúky.*

Takýto graf je zložený z dvoch typov uzlov: miest (places) a prechodov (transitions). Miesta sú reprezentované kruhmi a môžu obsahovať tokeny, ktoré symbolizujú určitý stav alebo dostupnosť zdrojov. Prechody, reprezentované obdĺžnikmi alebo pásikmi, označujú udalosti, ktoré môžu byť vykonané, ak sú splnené určité podmienky.

(Murata, 1989) uvádza matematickú definíciu Petriho siete ako štvoricu $PN = P, T, F, M0$ kde:

- P je konečná množina miest,
- T je konečná množina prechodov
- $F \subseteq (P \times T) \cup (T \times P)$ je množina orientovaných hrán (oblúkov),
- $M_0 : P \to \mathbb{N}$ je počiatočné označenie (marking), ktoré priraďuje každému miestu počet tokenov.

Dôležitou vlastnosťou Petriho sietí je ich schopnosť intuitívne modelovať paralelizmus, synchronizáciu a konflikty, čo ich robí vhodnými pre rôzne aplikácie od paralelného výpočtu cez

riadenie výrobných procesov až po analýzu distribuovaných systémov. Petriho siete sú dodnes významnou súčasťou výskumu v oblasti automatov a teórie systémov, kde poskytujú robustný rámec na simuláciu a overovanie.

Petriho siete sú tiež dôležité v oblasti výpočtovej zložitosti, kde sa využívajú na analýzu zložitých systémov a odhalenie potenciálnych problémov. Ich grafická reprezentácia umožňuje vizuálnu kontrolu systému, čo je významné najmä pri návrhu softvéru alebo pri overovaní správnosti komplexných systémov.

V priebehu rokov sa Petriho siete významne vyvinuli a boli rozšírené rôznymi spôsobmi, aby mohli efektívne modelovať a analyzovať čoraz komplexnejšie systémy. Tieto rozšírenia zahŕňajú farebné Petriho siete, časované Petriho siete a hierarchické Petriho siete, ktoré poskytujú vyššiu flexibilitu pri modelovaní špecifických požiadaviek.

2.1 Miesta, prechody a tokeny

Miesta v Petriho sieti reprezentujú stavy systému alebo podmienky, ktoré musia byť splnené pre vykonanie určitých udalostí. Každé miesto môže obsahovať ľubovoľný počet tokenov. Tokeny symbolizujú prítomnosť zdrojov alebo stavov. Dynamika systému je riadená presúvaním tokenov medzi miestami cez prechody. (Murata, 1989)

Prechody reprezentujú udalosti, ktoré menia stav systému. Prechod môže byť aktivovaný, ak všetky jeho vstupné miesta obsahujú dostatočný počet tokenov, definovaných váhami vstupných hrán. Po aktivácii prechod „odoberie“ tokeny zo vstupných miest a presunie ich do výstupných miest (Wolfgang Reisig, 1985)

Tok systému je definovaný orientáciou hrán v sieti. Týmto spôsobom Petriho siete umožňujú vizualizáciu dynamického správania systémov, kde tok zdrojov alebo udalostí je explicitne modelovaný.

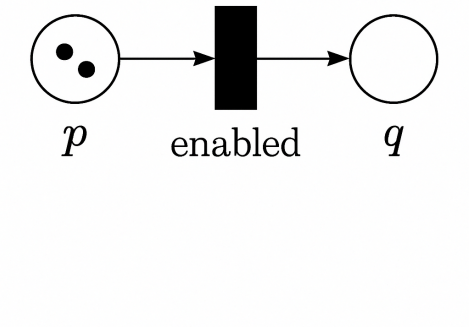
2.2 Spúšťacie pravidlá

Pravidlá spúšťania sú kritické pre správne pochopenie dynamiky Petriho sietí. Prechod môže byť aktivovaný, ak všetky vstupné miesta obsahujú aspoň toľko tokenov, koľko je vyžadovaných váhou príslušných vstupných hrán. Ak je táto podmienka splnená, prechod sa považuje za *spustiteľný* (*enabled*) (Murata, 1989).

Definícia 2 (Spustiteľnosť prechodu)**.** *Nech $PN = (P, T, F, W, M)$ je Petriho sieť s váhovou funkciou $W : F \to \mathbb{N}$, ktorá priraďuje každej hrane jej váhu. Prechod $t \in T$ je* ***spustiteľný*** *v značkovaní M, ak:*

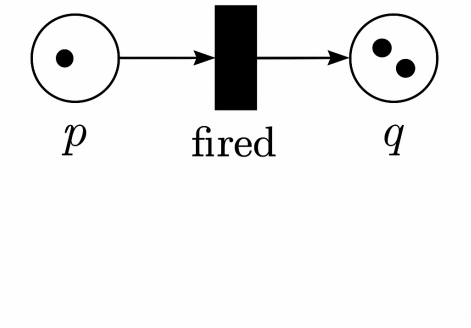
$$\forall p \in \bullet t : M(p) \geq W(p, t),$$

kde $\bullet t$ označuje množinu vstupných miest prechodu t, teda všetky miesta, z ktorých vedie hrana do t, a $W(p, t)$ je váha tejto hrany.



Obr. 1: Petriho sieť pred spustením prechodu
(prechod je spustiteľný)

Po spustení prechodu dochádza k zmene značkovania, pri ktorej sa z každého vstupného miesta odpočíta toľko tokenov, koľko určuje váha vstupnej hrany, a do každého výstupného miesta sa pridá toľko tokenov, koľko určuje váha výstupnej hrany.



Obr. 2: Petriho sieť po spustení prechodu
(zmenené značkovanie)

2.3 Matematické vlastnosti Petriho sietí

2.3.1 Dosiahnuteľnosť

Dosiahnuteľnosť (anglicky *reachability*) predstavuje základnú vlastnosť Petriho siete, ktorá umožňuje určiť, či môže byť určitý stav siete dosiahnutý z počiatočného označenia prostredníctvom postupného vykonávania prechodov.

Definícia 3 (Dosiahnuteľnosť v Petriho sieti)**.** *Nech je daná Petriho sieť $PN = (P, T, F, M_0)$, kde P je množina miest, T množina prechodov, F množina orientovaných hrán a M_0 počiatočné označenie. Označenie M' nazývame **dosiahnuteľným** z počiatočného označenia M_0, ak existuje postupnosť prechodov $\sigma = t_1, t_2, \ldots, t_n$, pričom každé prechodové označenie M_i splní podmienku aktivácie príslušného prechodu t_i. Formálne to môžeme zapísať ako:*

$$M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \xrightarrow{t_3} \ldots \xrightarrow{t_n} M_n = M'$$

pričom pre každé $i = 1, 2, \ldots, n$ platí:

$$M_{i-1}[p] \geq W(p, t_i), \quad \forall p \in P$$

kde $W(p, t_i)$ je počet tokenov potrebných v mieste p pre aktiváciu prechodu t_i.

Množinu všetkých označení, ktoré sú dosiahnuteľné z M_0, označujeme ako množinu dosiahnuteľných označení a zapisujeme ako $R(M_0)$:

$$R(M_0) = \{M \mid M \text{ je dosiahnuteľné z } M_0\}$$

Táto vlastnosť Petriho sietí je mimoriadne dôležitá pri analýze systémov, pretože umožňuje identifikovať, či systém môže dosiahnuť požadovaný stav alebo či existujú nežiaduce stavy (napr. slepé uličky). Overovanie dosiahnuteľnosti je často základom rôznych verifikačných metód a algoritmov pre Petriho siete (Murata, 1989; Wolfgang Reisig, 1985).

2.3.2 Živosť

Živosť (anglicky *liveness*) je jednou zo základných vlastností Petriho siete, ktorá zabezpečuje, že systém modelovaný touto sieťou nebude nikdy uviaznutý v stave, kde už nie je možné spustiť žiadnu udalosť (prechod).

Definícia 4 (Živosť Petriho siete)**.** *Nech je daná Petriho sieť $PN = (P, T, F, M_0)$, kde P je množina miest, T množina prechodov, F množina orientovaných hrán a M_0 počiatočné označenie. Prechod $t \in T$ nazývame* ***živý*** *(anglicky live), ak pre každé označenie M, ktoré je dosiahnuteľné z M_0, existuje také označenie M', dosiahnuteľné z M, že prechod t je v M' spustiteľný. Formálne to môžeme zapísať nasledovne:*

$$\forall M \in R(M_0), \exists M' \in R(M) : M'[p] \geq W(p,t) \quad \forall p \in P,$$

kde $R(M_0)$ označuje množinu všetkých označení dosiahnuteľných z počiatočného označenia M_0, a $W(p,t)$ označuje počet tokenov potrebných na aktiváciu prechodu t z miesta p.

Petriho sieť nazývame **živou**, ak sú všetky jej prechody živé. Inými slovami, v živej Petriho sieti je vždy možné pokračovať vo vykonávaní udalostí bez dosiahnutia slepej uličky (deadlocku).

Živosť je dôležitá pri analýze systémov, pretože poskytuje informáciu o tom, či systém dokáže kontinuálne a spoľahlivo fungovať v dlhodobom horizonte bez toho, aby uviazol v stave, kde už žiadna ďalšia aktivita nie je možná. Táto vlastnosť je zásadná napríklad pri modelovaní výrobných procesov, sieťových protokolov a distribuovaných systémov (Murata, 1989).

2.3.3 Obmedzenosť

Obmedzenosť (anglicky *boundedness*) je dôležitá vlastnosť Petriho siete, ktorá určuje, či môže v ľubovoľnom mieste siete vzniknúť neobmedzený počet tokenov. Omedzené siete sú z hľadiska implementácie bezpečné, keďže zabezpečujú, že žiadna časť systému nevyžaduje neobmedzené zdroje.

Definícia 5 (Obmedzenosť Petriho siete)**.** *Petriho sieť $PN = (P, T, F, M_0)$ nazývame **k-obmedzenou** (alebo jednoducho omedzenou), ak existuje konštanta $k \in \mathbb{N}$, taká že pre všetky označenia $M \in R(M_0)$ a pre všetky miesta $p \in P$ platí:*

$$M(p) \leq k$$

kde $M(p)$ označuje počet tokenov v mieste p pri označení M, a $R(M_0)$ je množina všetkých označení dosiahnuteľných z počiatočného označenia M_0.

Ak platí vyššie uvedená podmienka pre $k = 1$, sieť je nazývaná **bezpečná** (safe). To znamená, že v každom mieste siete sa môže nachádzať nanajvýš jeden token počas behu systému.

Omedzenosť je dôležitým kritériom pre rozhodovanie o správnosti a spoľahlivosti modelovaného systému, pretože zabezpečuje, že žiadna časť systému nebude nekontrolovateľne „preplnená“ tokenmi, čo by mohlo znamenať stratu kontroly nad jeho správaním (Murata, 1989).

2.4 Rozšírenia Petriho sietí

Petriho siete, ako formálny nástroj na modelovanie a analýzu systémov s diskrétnymi udalosťami, preukázali svoju užitočnosť v širokom spektre aplikácií. Ich základná verzia poskytuje dostatočný rámec pre reprezentáciu paralelizmu, synchronizácie a konfliktov. Avšak, s narastajúcou zložitosťou reálnych systémov vznikla potreba obohatiť túto formálnu štruktúru o ďalšie prvky, ktoré by umožnili zachytiť špecifické aspekty správania sa systémov.

Rozšírenia Petriho sietí vznikli ako reakcia na praktické požiadavky z oblastí ako priemyselné riadiace systémy, softvérové inžinierstvo či sieťové protokoly. Tieto rozšírenia umožňujú modelovať nielen komplexnejšie dátové štruktúry a časovanie, ale aj vyššiu mieru abstrakcie a modularity. Ich cieľom je zachovať formálnu analyzovateľnosť pôvodného modelu, pričom sa zvyšuje jeho vyjadrovacia schopnosť a praktická použiteľnosť.

Tieto obohatené verzie Petriho sietí vznikli s cieľom lepšie reprezentovať zložité systémy, ktoré si vyžadujú viac ako len jednoduché značkovanie miest a deterministické prepínanie prechodov. V závislosti od charakteru systému sa jednotlivé rozšírenia zameriavajú na rôzne aspekty – napríklad možnosť rozlíšiť medzi rôznymi typmi tokenov, zahrnutie časových parametrov alebo podporu hierarchickej štruktúry. Vďaka tomu sú tieto siete vhodné na modelovanie dynamických systémov s časovaním, dátovými tokmi či viacúrovňovým správaním.

Viaceré štandardné rozšírenia boli detailne formalizované a podrobené teoretickej aj praktickej analýze. Napriek svojej zvýšenej zložitosti si tieto modely zachovávajú matematickú presnosť a sú podporované analytickými technikami, ktoré umožňujú formálnu verifikáciu

vlastností systémov. Vďaka tomu sa rozšírené Petriho siete stali základným stavebným prvkom pri návrhu a validácii systémov v oblastiach ako embedded systémy, priemyselné automatizácie či bezpečnostne kritické aplikácie (Jensen et al., 2009; Murata, 1989).

2.4.1 Farebné Petriho siete

Farebné Petriho siete Coloured Petri Net (CPN) rozširujú klasickú formu tým, že tokenom priraďujú typy, tzv. "farby". Umožňujú tak reprezentovať rôzne dátové hodnoty v jednom modeli a spracovávať ich prostredníctvom prechodov. Každý prechod môže mať definované podmienky (guardy) a funkcie, ktoré tokeny transformujú. To výrazne zvyšuje vyjadrovaciu silu modelu a umožňuje simuláciu systémov s komplexnou logikou či údajmi, ako napr. počítačové siete, výrobné systémy alebo workflowy. Teoretický základ pre CPN rozpracoval Kurt Jensen (Jensen et al., 2009).

2.4.2 Časované Petriho siete

Časované Petriho siete (Timed Petri Nets) zavádzajú do modelu pojem času, a to buď ako časové oneskorenie prechodov, alebo životnosť tokenov. Časované modely sú mimoriadne dôležité pri simulácii systémov, kde závisí správanie od presného časovania udalostí – napríklad v oblasti dopravy, výrobných liniek alebo reálnych časových systémov. Zavedenie časových parametrov umožňuje lepšiu predikciu a optimalizáciu výkonnostných vlastností (Ramchandani, 1973).

2.4.3 Hierarchické Petriho siete

Hierarchické Petriho siete umožňujú rozložiť komplexný model na menšie časti – podsiete, ktoré je možné zoskupiť do vyššej štruktúry. Tým sa zlepšuje prehľadnosť modelu, jeho znovupoužiteľnosť a modularita. Využívajú sa najmä pri návrhu veľkých systémov, kde každá podsieť môže reprezentovať konkrétnu časť systému. Takýto prístup zjednodušuje návrh, analýzu a údržbu modelu (Huber et al., 1991).

2.4.4 Petriho siete s rozšírenými typmi hrán

Ďalším významným rozšírením základnej Petriho siete je možnosť definovať rôzne typy hrán medzi miestami a prechodmi. Okrem štandardných orientovaných hrán, ktoré zabezpečujú pohyb tokenov medzi miestami a prechodmi podľa váh, sa zavádzajú aj tzv. *reset hrany* a *inhibítorové hrany*. Tieto špecializované hrany poskytujú vyššiu flexibilitu a presnosť pri modelovaní systémov, kde sú potrebné špecifické logické podmienky na vykonávanie prechodov.

Reset hrana (anglicky *reset arc*) je typ hrany, ktorá spôsobí vymazanie všetkých tokenov z pripojeného miesta v momente, keď je pripojený prechod aktivovaný. Nezávisle od počtu tokenov, ktoré sa v danom mieste nachádzajú, po aktivácii prechodu bude toto miesto

vyprázdnené. Tento mechanizmus je užitočný napríklad v prípadoch, keď je potrebné po dokončení určitej akcie úplne uvoľniť zdroje alebo reštartovať časť systému. Formálne možno reset hranu chápať ako hranu s nešpecifikovanou váhou, ktorá vynuluje značkovanie miesta.

Inhibítorová hrana (anglicky *inhibitor arc*) definuje podmienku, že prechod je spustiteľný len vtedy, keď miesto, z ktorého vedie inhibítorová hrana, neobsahuje žiadne tokeny. Teda ak je miesto prázdne, prechod môže byť vykonaný; ak však obsahuje aspoň jeden token, prechod nie je aktivovateľný. Tento typ hrany je veľmi užitočný pri modelovaní zakazujúcich podmienok (napr. mutex logiky alebo ochranných stavov), ktoré sa v základnej Petriho sieti nedajú jednoducho vyjadriť.

Oba typy hrán zvyšujú vyjadrovaciu schopnosť modelu bez potreby zavádzania zložitých štruktúr alebo úplne nových formalizmov (Hee et al., 2002). Petriho siete s rozšírenými hranami sú používané najmä v analýze systémov so stavmi závislými od neprítomnosti alebo úplnej spotreby zdrojov (Esparza et al., 1998; David et al., 2010).

3 Existujúce riešenia

Na modelovanie a simuláciu Petriho sietí existuje viacero známych nástrojov a frameworkov, ktoré boli vyvinuté prevažne na akademické alebo špecializované účely. Ich hlavnými výhodami sú robustné analytické nástroje, grafické používateľské rozhrania a možnosť simulácie, avšak často im chýba flexibilita, integrácia do moderných softvérových architektúr a podpora pre konkrétne aplikačné oblasti ako sú mechatronické simulácie či prepojenie s priemyselnými štandardmi ako **Factory I/O** alebo **OPC UA**.

Navrhovaný framework sa preto odlišuje svojím zameraním – poskytovať vývojárom z oblasti **mechatroniky** a **automatizácie** nástroj, ktorý im umožní modelovať, simulovať a integrovať Petriho siete priamo v aplikáciách postavených na platforme Microsoft .NET Framework (.NET). Budúce rozšírenia budú orientované aj na prepojenie so simulačnými a riadiacimi nástrojmi, čím framework prekročí rámec teoretického modelovania a stane sa praktickým nástrojom v oblasti **digitálnych dvojčiat** a **priemyslu 4.0**.

3.1 PIPE – Platform Independent Petri Net Editor

Platform Independent Petri Net Editor (PIPE) je open-source nástroj na vizuálne modelovanie a simuláciu základných Petriho sietí. Umožňuje vytvárať grafické modely a testovať ich prostredníctvom simulačných mechanizmov (Bonet et al., 2003).

Je vhodný na rýchle prototypovanie a výučbu, ale absentuje programové API a podporuje iba základné typy sietí. Nie je preto vhodný na integráciu do komplexných mechatronických systémov alebo na prepojenie s .NET prostredím.

3.2 CPN Tools – Nástroj pre farebné Petriho siete

Coloured Petri Net Tools (CPN Tools) je špecializovaný akademický nástroj na prácu s farebnými Petriho sieťami CPN. Poskytuje bohaté možnosti simulácie a analýzy správania systémov (Jensen et al., 2009).

Aj keď je výpočtovo výkonný, jeho použitie je orientované na výskum a chýba mu integrácia do moderných vývojových prostredí, ako sú aplikácie založené na jazyku C# alebo .NET. Zložitosť nástroja a technológie, ktoré používa (napr. SML), sú pre vývojárov v oblasti priemyslu prekážkou.

3.3 GreatSPN – Analýza výkonu a verifikácia

Generalized Stochastic Petri Net (GreatSPN) je nástroj zameraný na verifikáciu a výkonovú analýzu časovaných a stochastických Petriho sietí. Bol vyvinutý na Politecnico di Torino a

umožňuje podrobnú analýzu vlastností systémov (Donatelli et al., 1995).

Aj keď ponúka analytickú silu, jeho architektúra nie je vhodná na priame zapojenie do priemyselných procesov alebo aplikácií. Nepodporuje štandardy ako Open Platform Communications Unified Architecture (OPC UA) (Foundation, 2022), ani sa nedá prepojiť s nástrojmi ako Factory I/O (Games, 2023).

3.4 Porovnanie existujúcich riešení

Nástroj	**Typy sietí**	**API**	**.NET**	**Zameranie**	**Pre mechatroniku**
PIPE	Základné	Nie	Nie	Vzdelávanie	Nízka
CPN Tools	Farebné	Čiastočne	Nie	Výskum	Nízka
GreatSPN	Časované, stochastické	Nie	Nie	Verifikácia	Nízka

Tabuľka 1: Porovnanie existujúcich nástrojov na prácu s Petriho sieťami

3.5 Motivácia pre vývoj nového frameworku

Na základe uvedeného porovnania je zrejmé, že aktuálne nástroje nie sú optimalizované pre potreby vývojárov v oblasti automatizácie, riadenia a simulácií. Preto navrhovaný framework kladie dôraz na:

- Modulárnu a rozšíriteľnú architektúru
- Priamu podporu pre .NET a jazyk C#
- RESTful API pre webové a cloudové aplikácie
- Serializáciu a výmenu modelov vo formátoch XML a JSON
- Webovú aplikáciu pre vizualizáciu modelov
- Budúcu integráciu s Factory I/O
- Možnosti komunikácie cez OPC UA

Framework je koncipovaný ako otvorená platforma s dôrazom na praktickosť, kompatibilitu s existujúcimi priemyselnými nástrojmi a možnosť nasadenia v simuláciách mechatronických systémov, čím ponúka most medzi teoretickým modelovaním a reálnym nasadením v prostredí digitálnej výroby.

4 Petriho sieť ako dátova štruktúra

Ako bolo uvedené v kapitole *Teoretický základ*, Petriho siete predstavujú formálny model založený na orientovanom bipartitnom grafe, kde interakcia medzi stavmi a udalosťami je sprostredkovaná prostredníctvom miest, prechodov a tokov (hrán) (Murata, 1989; W. Reisig, 2013). Tieto siete sa vyznačujú schopnosťou presne a názorne modelovať súbežnosť, synchronizáciu, podmienky a cykly, čo z nich robí ideálny nástroj pre simuláciu procesov v oblastiach ako sú mechatronické systémy, automatizácia či riadiace systémy (Blaga et al., 2021).

Vzhľadom na tieto vlastnosti boli Petriho siete identifikované ako základný formálny model, ktorý môže slúžiť ako nosná dátová štruktúra pre navrhovaný framework. Naším cieľom bolo nájsť alebo navrhnúť takú dátovú štruktúru, ktorá by bola schopná komplexne vystihnúť všetky možnosti a rozšírenia definícií Petriho sietí vrátane interaktívnych prvkov, dátových typov, rolí používateľov a podmienenej logiky.

V tejto kapitole sa preto podrobnejšie zaoberáme výberom konkrétneho formátu ich reprezentácie, s dôrazom na jazyk Petriflow, ktorý je založený na XML a ktorý sa ukázal ako vhodný kandidát. Vysvetlíme tiež dôvody, prečo bola uprednostnená reprezentácia v XML pred formátom JSON a analyzujeme jednotlivé časti špecifikácie jazyka Petriflow ako perspektívneho vstupného formátu pre modely v tomto projekte.

4.1 JSON vs. XML

Pri rozhodovaní medzi formátom XML a JSON sa zohľadňovali nielen technické parametre, ale aj praktická použiteľnosť vo formálnom modelovaní (Taylor et al., 2009).

XML

- Schémová validácia: podpora XML Schema Definition (XSD) schém pre kontrolu typu, povinných elementov, referencií medzi ID (Walmsley, 2001).
- Nástrojová podpora: silná integrácia do Integrated Development Environment (IDE) (napr. Visual Studio, Rider, PyCharm atď.), podporované aj v transformáciách (eXtensible Stylesheet Language Transformations (XSLT)).
- Deklaratívna povaha: vhodné na popis procesov, kde sú dôležité presné významy a podmienky.
- Hierarchická štruktúra: prirodzené vnárenie objektov a prehľadná reprezentácia komplexných stavov.

JSON

- Ľahšie spracovanie v JavaScript aplikáciách, no väčšina logiky frameworku je mimo webových prehliadačov.
- Bez schémových garancií: JSON Schema je slabšie podporovaný a menej prísny formát (Pezoa et al., 2016).
- Horšia čitateľnosť pri komplexných modeloch.

Z týchto dôvodov bol XML zvolený ako hlavný formát. Okrem porovnania existujúcich formátov sme zvažovali aj vytvorenie vlastného dátového formátu, čo by však prinieslo vysokú záťaž na dokumentáciu, testovanie a kompatibilitu (Fielding, 2000).

4.2 Výber vhodnej dátovej štruktúry a rozhodovací proces

Pri návrhu frameworku pre prácu s Petriho sieťami bolo kľúčovým rozhodnutím určiť, akú dátovú štruktúru a formát zvoliť pre reprezentáciu modelov. Táto voľba zásadne ovplyvňuje možnosti rozšíriteľnosti, integrácie, validácie modelov a celkovej udržateľnosti riešenia. V tejto časti podrobne opisujeme proces rozhodovania a dôvody, ktoré viedli k výberu jazyka Petriflow.

4.2.1 Požiadavky na dátovú štruktúru

Pri návrhu riešenia sme sa zamerali na vytvorenie modelu, ktorý dokáže reprezentovať základnú štruktúru klasických Petriho sietí, teda miesta, prechody a ich vzájomné prepojenie prostredníctvom hrán. Zároveň bolo dôležité, aby tento model neostal len na úrovni abstraktných tokenov a prechodov, ale aby bolo možné rozšíriť ho o dátové polia a typy údajov, ktoré reflektujú parametre procesov. Takto obohatený model nám poskytuje väčšiu výpovednú hodnotu a možnosť presnejšej simulácie alebo validácie.

Ďalším dôležitým aspektom bola potreba integrovať prvky používateľskej interakcie. To zahŕňalo podporu rolí a oprávnení, ako aj možnosť definovať formuláre a vstupné dáta. Tieto sú viazané na jednotlivé prechody alebo miesta. Zároveň sme kládli dôraz na možnosť definovania logiky – podmienok a akcií, ktoré určujú, kedy môže dôjsť k aktivácii prechodu. Tieto mechanizmy umožňujú riadiť správanie systému a zabezpečujú, že model bude možné efektívne nasadiť v reálnom prostredí s viacerými aktérmi.

Okrem samotného modelovania sme kládli dôraz aj na praktické požiadavky integrácie a spracovania. Hľadali sme riešenie, ktoré bude možné jednoducho validovať a vyhodnocovať v prostredí nášho frameworku. Zároveň bolo kľúčové, aby model podporoval interoperabilitu

s externými systémami ako sú REST API alebo priemyselný štandard OPC UA, čo umožňuje jeho nasadenie v prostredí Industry 4.0. V neposlednom rade sme zohľadnili aj potrebu automatizácie a simulácie, ktoré sú nevyhnutné pre testovanie a overenie správania systému ešte pred jeho reálnym nasadením. Na základe týchto kritérií sme následne prehodnotili viaceré existujúce štandardy a riešenia.

4.2.2 Prehľad analyzovaných formátov

PNML (Petri Net Markup Language)

PNML (Petri Net Markup Language) (ISO/IEC, 2004) predstavuje oficiálny štandard podľa ISO/IEC 15909, ktorý slúži ako robustný základ na výmenu modelov medzi rôznymi nástrojmi. Jeho silnou stránkou je podpora základných aj pokročilých typov Petriho sietí, vrátane farebných a časovaných variantov. Napriek tomu však nie je vhodný pre workflow aplikácie, keďže mu chýba koncept používateľských rolí, interakcie, formulárov a stavovej logiky, ktoré by presahovali klasickú štruktúru siete. Navyše, jeho rozšírenie o tieto prvky je technicky náročné a často vedie k porušeniu štandardu.

CPN Tools XML

CPN Tools XML (Jensen et al., 2009) je formát špecializovaný na farebné Petriho siete a je veľmi obľúbený v akademickej sfére najmä vďaka svojej schopnosti vizualizovať a simulovať komplexné dátové toky. Tento formát však neobsahuje podporu pre aplikačnú logiku, ako je používateľské rozhranie, roly, akcie či integrácia prostredníctvom Representational State Transfer (REST) API. Z toho dôvodu nie je vhodný na prepojenie s reálnymi systémami a procesnými enginmi, čo výrazne obmedzuje jeho využitie mimo akademického prostredia.

YAWL XML

YAWL XML (Der et al., 2003) je workflow jazyk, ktorý rozširuje Petriho siete o bohatú sémantiku zahŕňajúcu paralelizmus, výnimky a podmienky. Je navrhnutý s cieľom modelovať komplexné procesy, čomu zodpovedá aj jeho rozsiahla funkcionalita. Pre jednoduchšie prípady je však jeho použitie často zbytočne ťažkopádne. Okrem toho je veľmi úzko spätý s vlastným runtime prostredím YAWL, čo komplikuje snahy o implementáciu alternatívnych enginov a zároveň ho diskvalifikuje pre simulácie na nižšej úrovni, napríklad v oblasti mechatroniky.

Petriflow

Petriflow (Petriflow Project, 2024; Juhás et al., 2021) je jazyk špeciálne navrhnutý na popis workflow procesov na báze Petriho sietí. Je postavený na štandarde XML a podporuje validáciu pomocou XSD schém. Jeho najväčšou výhodou je natívna podpora pre dátové polia,

používateľské formuláre, roly, triggery, REST API a prepojenie s runtime enginom. Vďaka otvorenému formátu a slobodnej licencii umožňuje bezproblémovú integráciu s externými systémami. Navyše, projekt Netgrif poskytuje stabilnú dokumentáciu a aktívnu komunitu, čo z neho robí silného kandidáta pre moderné workflow a simulačné riešenia.

4.2.3 Rozhodovací proces

Po analýze vyššie uvedených možností sme sa rozhodli nevyvíjať vlastný formát, keďže:

- vývoj a údržba vlastného štandardu je náročný na čas a zdroje,
- absentovala by kompatibilita s existujúcimi nástrojmi,
- vznikol by technický dlh pri dokumentácii, validácii a spätnom rozširovaní.

Na druhej strane, existujúce formáty ako PNML, CPN XML či YAWL XML nedokázali uspokojiť všetky naše požiadavky, najmä čo sa týka aplikačnej vrstvy, používateľskej interakcie a REST integrácie.

Preto bol zvolený jazyk Petriflow ako najvhodnejšia dátová štruktúra pre reprezentáciu Petriho sietí v rámci navrhovaného frameworku. Jeho modularita, praktická použiteľnosť a aktívny vývoj umožňuje rýchlu integráciu, testovanie aj ďalšie rozširovanie v budúcnosti.

5 Analýza jazyka Petriflow a návrh dátovej štruktúry pre reprezentáciu Petriho sietí v C#

Petriflow je XML založený formálny jazyk navrhnutý tímom Netgrif na modelovanie Petriho sietí s rozšíreniami, ktoré sú vhodné najmä na popis procesov vo webových a informačných systémoch. Jeho cieľom je poskytnúť dostatočne expresívny a ľahko čitateľný jazyk na opis správania systému pomocou miest, tranzícií, väzieb a akcií.

Jazyk Petriflow nie je len syntaktickým popisom štruktúry Petriho sietí, ale integruje aj koncepty ako dátové polia, udalosťami spúšťané akcie, podmienky, a dokonca aj logiku riadenia.

5.1 Komponenty jazyka Petriflow

5.1.1 Proces

Element `<process>` je koreňovým uzlom každého Petriflow XML súboru. Obsahuje všetky ostatné komponenty a predstavuje jeden samostatný Petriho model (proces).

Atribúty:

- identifier: unikátny identifikátor procesu
- title: čitateľný názov procesu
- version: verzia procesu

Obsahuje:

- miesta (place)
- tranzície (transition)
- hrany (arc)
- roly (role)
- údaje (dataField)
- akcie a udalosťové správy

5.1.2 Miesto

Miesto reprezentuje stav v procese. V klasickej Petriho sieti predstavuje držiteľa tokenov. V Petriflow môže obsahovať aj dátové polia a mať špeciálny typ.

Atribúty:

- id: identifikátor miesta
- type: typ miesta – start, end, immediate, cancel, atď.
- initialMarking: počet počiatočných tokenov (voliteľné)

Obsahuje:

- dataField – dátové polia dostupné na tomto mieste
- visibility – definovanie podmienok viditeľnosti
- authorization – priradenie rolí pre zobrazenie alebo manipuláciu

Možnosti:

- Miesto môže uchovávať dáta, byť podmienkou pre tranzície alebo byť začiatkom/koncom procesu.

5.1.3 Tranzícia

Tranzícia reprezentuje udalosť, ktorá môže nastať, ak sú splnené všetky podmienky a aktívne miesta sú pripravené. Spúšťa zmeny v sieti a často má priradené akcie.

Atribúty:

- id: identifikátor tranzície
- type: typ (napr. manual, automatic)
- title: názov tranzície

Obsahuje:

- precondition: podmienky pre spustenie
- postcondition: čo sa stane po vykonaní
- action: samotná operácia (napr. zmena hodnoty dátového poľa)

- authorization: zoznam rolí oprávnených spustiť tranzíciu
- immediate: či sa má tranzícia spustiť okamžite bez zásahu používateľa

Možnosti:

- Podpora komplexnej logiky (IF-THEN, výrazy)
- Dynamická manipulácia s dátami
- Automatizované aj manuálne tranzície

5.1.4 Hrana

Hrany spájajú miesta a tranzície. Určujú tok medzi stavmi a udalosťami. Typ hrany ovplyvňuje, ako funguje logika siete.

Atribúty:

- id: identifikátor hrany
- sourceId: zdroj (miesto alebo tranzícia)
- targetId: cieľ (miesto alebo tranzícia)
- type: input, output, reset, inhibitor, read

Typy hrán:

- input – štandardný vstup
- output – výstup po vykonaní
- inhibitor – zabraňuje spusteniu, ak je miesto označené
- reset – odstráni všetky tokeny z miesta
- read – číta stav bez odobratia tokenov

Možnosti:

- Komplexné podmienky pre aktiváciu
- Práca s rôznymi typmi pripojení

5.1.5 Dátové pole

Dátové polia umožňujú pracovať s dátami v miestach aj tranzíciách. Môžu byť viazané na používateľa, na automatické výpočty alebo ako podmienky pre tranzície.

Atribúty:

- id: identifikátor poľa
- type: typ poľa (text, number, boolean, date, enumeration, atď.)
- defaultValue: predvolená hodnota
- required: či je pole povinné
- editable: či môže používateľ meniť hodnotu

Možnosti:

- Dynamické formuláre
- Zber údajov od používateľov
- Výpočty v rámci precondition/postcondition

5.1.6 Rola

Roly sú základným autorizačným mechanizmom. Definujú, kto má prístup k určitým akciám alebo údajom.

Atribúty:

- id: interný identifikátor
- name: čitateľný názov roly

Použitie:

- Obmedzenie viditeľnosti dát
- Priradenie práv pre spúšťanie tranzícií
- Kontrola prístupu k miestam a hodnotám

5.1.7 Akcie a výrazy

Tieto prvky umožňujú dynamické správanie siete. Výrazy sa zapisujú v MVFLEX Expression Language (MVEL), čo je Domain-Specific Language (DSL) podobný JavaScriptu.

Príklady výrazov:

- `data['age'] > 18`
- `role == 'admin'`
- `set('approved', true)`

Použitie:

- Vyhodnotenie podmienok pred spustením tranzície
- Výpočty alebo validácie
- Nastavovanie hodnôt dát

5.1.8 Udalosť

Nie všetky modely ich obsahujú, ale môžu byť súčasťou pokročilejších scénarov. Udalosti môžu byť signály, časovače, správy atď.

Možnosti:

- Časované tranzície (napr. „spusti po 5 dňoch“)
- Odozva na externú správu alebo REST volanie

5.1.9 Podproces

Petriflow umožňuje vkladať jeden proces do druhého, čo zjednodušuje modelovanie zložitých systémov.

Výhody:

- Opätovná použiteľnosť
- Oddelenie zodpovedností
- Prehľadnejší hlavný model

6 Architektúra systému

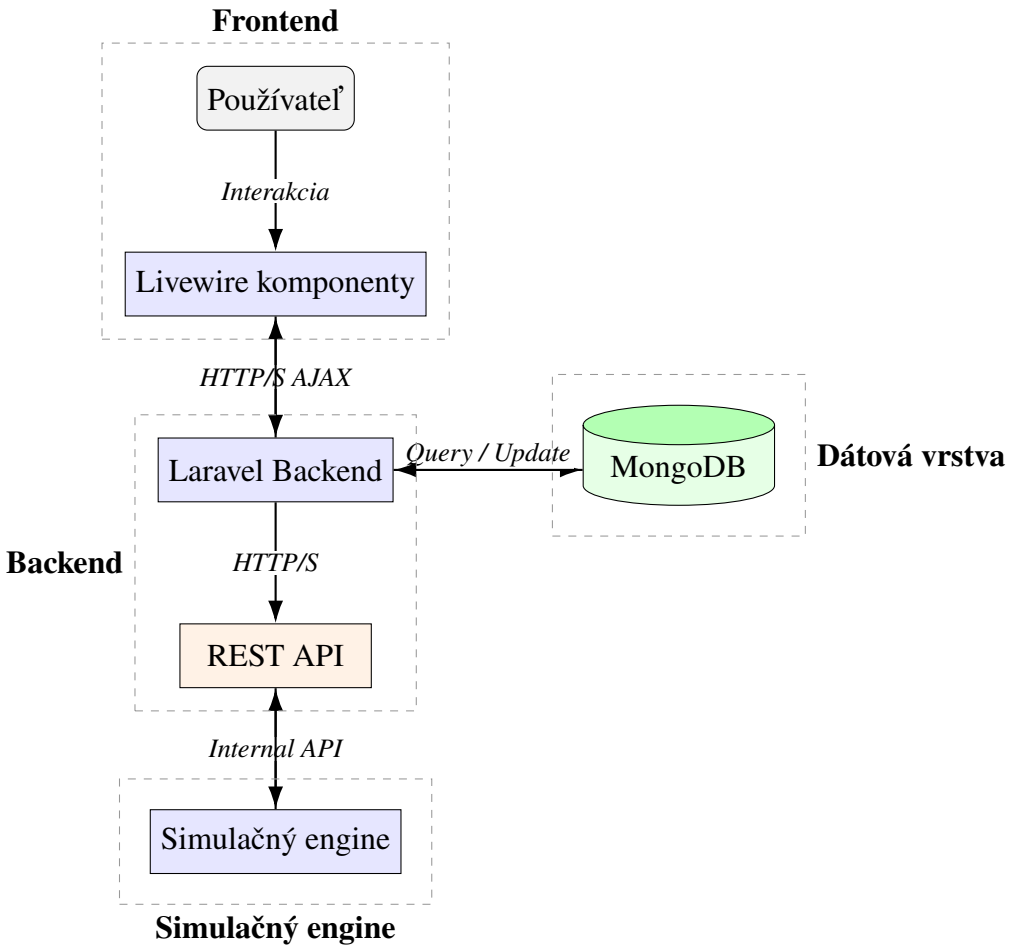
Architektúra riešenia je navrhnutá ako viacvrstvový systém, ktorý spája serverovú logiku, interaktívne používateľské rozhranie a externé simulačné jadro do jedného celku. Základom celej aplikácie je Laravel framework, ktorý plní úlohu aplikačného backendu. V rámci neho sú implementované REST API rozhrania, autentifikačné a autorizačné mechanizmy, logika spracovania požiadaviek a komunikácia s databázou. Laravel zároveň slúži ako sprostredkovateľ medzi frontend vrstvou a samotným simulačným jadrom systému.

Používateľské rozhranie je postavené pomocou technológie Laravel Livewire. Tá umožňuje tvorbu reaktívnych komponentov, ktoré komunikujú so serverom bez potreby manuálneho písania JavaScriptu. Komponenty Livewire reagujú na používateľské akcie ako kliknutia, zadávanie údajov či prepínanie stavov, pričom ich zmeny sa okamžite odosielajú na server a späť. Vďaka tejto technológii je možné vyvíjať moderné interaktívne rozhrania s vysokou mierou dynamiky a interakcie, pričom všetka logika ostáva centralizovaná na strane PHP backendu.

Dátová vrstva systému je zabezpečená pomocou dokumentovo orientovanej databázy MongoDB. V tejto databáze sú uchovávané štruktúrované dáta reprezentujúce definície Petriho sietí, inštancie sietí, aktuálne stavy miest a prechodov, ako aj výstupy jednotlivých simulácií. MongoDB bola zvolená najmä kvôli schopnosti efektívne pracovať s polostrukturovanými dátami vo formáte JSON, čím sa zjednodušuje proces ukladania a načítavania modelov.

Komunikácia medzi Laravel backendom a simulačným jadrom prebieha prostredníctvom REST API rozhrania. Simulačný engine, implementovaný ako samostatná aplikácia, prijíma požiadavky na vykonanie prechodov, vytvorenie novej inštancie siete alebo načítanie aktuálneho stavu. Výstupy zo simulácie sú vo forme JSON, ktoré sú následne uchované v databáze alebo priamo vrátené späť klientovi. REST API vrstva zároveň poskytuje možnosť budúcej integrácie s externými systémami.

Zvolený architektonický prístup tak podporuje jasné oddelenie zodpovedností medzi vrstvami systému a zároveň umožňuje vysokú mieru rozšíriteľnosti. Frontend komunikuje výlučne s backendovou vrstvou, čím sa zabezpečuje jednotné miesto pre kontrolu prístupu, logiky a spracovania dát. Backend sa stará o orchestráciu medzi používateľským rozhraním, databázou a simulačným enginom ako je zobrazené na Obrázku 3. Celý systém je vďaka tomu flexibilný, dobre testovateľný a pripravený na nasadenie v prostredí, kde je požadovaná vysoká dynamika, presnosť a možnosť ďalšieho vývoja.



Obr. 3: Architektonický model systému s vyznačenými logickými vrstvami a komunikačnými protokolmi

V nasledujúcich podkapitolách sa podrobnejšie pozrieme na jednotlivé technológie, ktoré boli použité pri vývoji simulačného jadra a aplikačného backendu. Zameriame sa na jazyk C#, platformu .NET a framework Laravel, pričom zdôrazníme ich výhody a nevýhody v kontexte našej aplikácie.

6.1 Programovací jazyk C#

Programovací jazyk C# je vyvíjaný spoločnosťou Microsoft. Bol predstavený v roku 2000 ako súčasť iniciatívy .NET a odvtedy sa stal jedným z najpoužívanejších jazykov pre vývoj desktopových, webových, mobilných a cloudových aplikácií. Navrhnutý bol ako jazyk typu C, no s modernejšími konštrukciami, automatickým spravovaním pamäte a silnou typovou kontrolou. C# kombinuje výhody jazykov ako Java a C a zároveň eliminuje mnohé ich nedostatky.

Jednou z kľúčových predností C# je jeho syntax, ktorá je zrozumiteľná, striktne typovaná a umožňuje vývoj v rôznych štýloch od objektovo-orientovaného programovania (OOP), cez

funkcionálne konštrukcie, až po asynchrónne operácie s využitím async/await. To výrazne prispieva k čitateľnosti kódu, jeho udržiavateľnosti a zároveň zvyšuje produktivitu vývojára. V oblasti vývoja simulačných nástrojov je práve čitateľnosť a bezpečnosť kódu rozhodujúca, pretože simulované systémy môžu byť komplexné a rozsiahle.

Ďalšou dôležitou vlastnosťou jazyka C# je jeho výkonnosť. Aj keď nejde o nízkoúrovňový jazyk ako C alebo C, dokáže dosahovať veľmi dobré výsledky v oblasti výpočtovej efektivity, najmä vďaka optimalizáciám v .NET a možnosti efektívne využívať paralelizmus prostredníctvom knižníc ako System.Threading alebo TPL (Task Parallel Library). To je obzvlášť dôležité v prípade simulácií diskrétnych stavových systémov, ktoré často vyžadujú paralelné spracovanie veľkého množstva prechodov alebo stavov.

C# je ideálny kandidát pre vývoj aplikácií v prostredí Industry 4.0, pretože umožňuje jednoduchú integráciu s priemyselnými štandardmi ako OPC UA, podporuje bezpečnostné modely a umožňuje jednoduché nasadenie na rôzne platformy vďaka frameworku .NET. Aj preto je C# bežne používaný vo výrobe, energetike, doprave a ďalších oblastiach, kde sa vyžaduje vysoká spoľahlivosť a interakcia s reálnym svetom. V tejto práci práve tieto požiadavky zohrávali kľúčovú úlohu.

Rozhodnutie použiť C# pri vývoji simulačného jadra frameworku bolo teda výsledkom kombinácie viacerých faktorov: dlhodobej podpory zo strany Microsoftu, bohatého ekosystému knižníc, vysokej čitateľnosti a bezpečnosti kódu, podpory paralelizmu a schopnosti efektívne komunikovať s externými systémami. C# nám umožnil vyvíjať robustné a rozšíriteľné riešenie s jasnou architektúrou a potenciálom na ďalší vývoj.

6.2 .NET platforma

Platforma .NET je vývojové prostredie a runtime systém od spoločnosti Microsoft, ktoré slúži na vytváranie a spúšťanie aplikácií naprieč rôznymi operačnými systémami a zariadeniami. Od svojho uvedenia v roku 2002 prešla výrazným vývojom – najmä v smere otvorenosti a multiplatformovosti. Od verzie .NET Core (2016) sa .NET stal open-source technológiou pod záštitou .NET Foundation, čo výrazne zvýšilo jeho adopciu a komunitnú podporu (Microsoft, 2023). V súčasnosti sa používa najmä verzia .NET 6/7/8/9, ktorá zjednocuje pôvodné vetvy (Framework, Core, Xamarin) do jedného výkonného a modulárneho ekosystému.

Jednou z hlavných výhod .NET je jeho silná štandardná knižnica (Base Class Library), ktorá obsahuje tisíce funkcií na prácu s reťazcami, súbormi, sieťami, dátumami, kolekciami, databázami či webovými službami. Vďaka tejto knižnici vývojári nemusia implementovať bežné funkcie od nuly, čím sa výrazne skracuje čas vývoja a zvyšuje spoľahlivosť aplikácie. Táto knižnica je tiež pravidelne aktualizovaná a optimalizovaná s dôrazom na výkon, bezpečnosť

a moderné vývojové praktiky.

Z pohľadu architektúry ponúka .NET platforma pokročilé nástroje pre vývoj výkonných aplikácií: garbage collection, multithreading, paralelné výpočty, deployment v kontajneroch cez Docker a podpora pre mikroservisy pomocou ASP.NET Web API. Tieto vlastnosti sú mimoriadne dôležité pre aplikácie v oblasti simulácie, kde je potrebné zabezpečiť rýchle a spoľahlivé spracovanie veľkého množstva údajov v reálnom čase. Navyše, vďaka podpore pre tzv. self-contained aplikácie je možné dodať celú simulačnú logiku ako jeden spustiteľný balík bez závislostí na cieľovom systéme.

V kontexte tejto práce tvorí .NET technologický základ pre vývoj serverovej logiky – teda jadra simulačného systému pre petriho siete. Výber tejto platformy nám umožnil efektívne využiť jazyk C# a zároveň nasadiť riešenie na rôzne typy systémov – od vývojového prostredia v macOS, cez testovacie servery s Linuxom, až po produkčné riešenia na Windows Serveri. Táto flexibilita bola kľúčová pri testovaní výkonu a škálovania, ako aj pri budúcom nasadení v rámci Industry 4.0 scenárov.

.NET nám taktiež poskytol bohaté možnosti testovania pomocou nástrojov ako xUnit či MSTest, čím sa zvýšila robustnosť a overiteľnosť celého riešenia. V kombinácii s vývojovým prostredím Visual Studio a vstavanými nástrojmi na profilovanie aplikácií bolo možné optimalizovať výkon a odstraňovať problémy už v počiatočných fázach vývoja. Aj preto bola voľba .NET strategickým rozhodnutím, ktoré poskytlo rovnováhu medzi výkonom, bezpečnosťou, škálovateľnosťou a komfortom vývoja.

6.3 Laravel Framework

Laravel je moderný open-source webový framework pre jazyk PHP, ktorý si od svojho vzniku v roku 2011 získal veľkú popularitu vďaka svojej čistej architektúre, elegantnej syntaxe a rozsiahlemu ekosystému balíkov. Autorom Laravelu je Taylor Otwell, ktorý ho navrhol ako reakciu na nedostatky vtedajších PHP frameworkov a s cieľom zjednodušiť a spríjemniť vývoj webových aplikácií (Otwell, 2013). Laravel využíva architektúru Model-View-Controller (MVC) (Model–View–Controller), čo napomáha k čistej separácii zodpovedností v aplikácii a uľahčuje testovanie a údržbu.

Jednou z hlavných predností Laravelu je rozsiahla podpora pre bežné webové úlohy – autentifikácia, autorizácia, routovanie, kešovanie, validácia formulárov, práca s API a oveľa viac. Všetky tieto komponenty sú predpripravené a integrované v rámci frameworku tak, aby vývojár mohol rýchlo vytvoriť plne funkčnú aplikáciu bez nutnosti opakovane riešiť základné technické problémy. Laravel tiež podporuje tzv. „service container“, čím uľahčuje správu závislostí a umožňuje jednoduché injektovanie služieb do rôznych častí aplikácie.

Vďaka systému migrácií a Eloquent Object-Relational Mapping (ORM) je Laravel ideálnym riešením pre aplikácie, ktoré pracujú s databázami a to klasickými relačným ako aj NoSQL databázami. ORM vrstva umožňuje jednoduché definovanie modelov, ich vzťahov a dotazov bez nutnosti písať zložitý Structured Query Language (SQL) kód. To výrazne zjednodušuje vývoj REST API, ktoré bolo v tejto práci použité ako rozhranie medzi simulačným jadrom v C# a používateľským webovým rozhraním. Vďaka Laravel Sanctum je možné jednoducho implementovať autentifikáciu a zabezpečenie prístupov k rôznym API endpointom, čo sme využili pri napojení na simulačné jadro.

Laravel je tiež známy svojou výbornou podporou pre testovanie – obsahuje nástroje na testovanie HTTP requestov, databázových operácií, komponentov aj frontendu. Pre vývojára to znamená možnosť pokryť kľúčové časti systému automatizovanými testami, čo znižuje počet chýb pri nasadzovaní a zvyšuje spoľahlivosť celej aplikácie. Navyše Laravel je aktívne vyvíjaný a má veľmi silnú komunitu, čo znamená, že akýkoľvek problém je možné rýchlo konzultovať alebo vyriešiť prostredníctvom dostupných balíkov.

V kontexte tejto práce bol Laravel použitý ako „frontend backend“ – teda aplikácia, ktorá slúži používateľovi, no zároveň sprostredkuje komunikáciu so simulačným backendom. Vytvára napojenie na REST API, ktoré sprístupňuje funkcionalitu C# jadra, umožňuje správu sietí, zobrazenie simulačných výsledkov a zabezpečuje interakciu cez používateľské rozhranie. Laravel sme zvolili kvôli jeho rýchlej použiteľnosti, čistej architektúre, podpore testovania a osobnej skúsenosti s týmto frameworkom pri budovaní iných webových systémov.

6.4 Livewire

Livewire je moderná knižnica pre vývoj interaktívnych používateľských rozhraní v rámci Laravel ekosystému. Ide o open-source balík, ktorý umožňuje vytvárať dynamické komponenty v rámci Blade šablón bez potreby písať JavaScript. Toto neznamená, že frontend napoužíva JavaScript. Použitie JavaScript si riadi knižnica samotná a ak je potrebné vykonávať akcie na strane klienta, je použitý AplineJS Framework. Aj preto možme povedať veľkou výhodou Livewire je, že sa snaží čo najviac zjednodušiť vývoj reaktívnych aplikácií tak, aby sa vývojár mohol sústrediť na PHP logiku na serveri a nemusel riešiť zložitosti klientského JavaScriptového frameworku ako je Vue.js či React.

Livewire vznikol ako reakcia na potrebu mnohých Laravel vývojárov po jednoduchom, serverovo orientovanom prístupe k tvorbe dynamických rozhraní. Na rozdiel od klasických JavaScriptových riešení, kde je stav aplikácie držaný v prehliadači a vyžaduje synchronizáciu s backendom, Livewire umožňuje stav uchovávať a spravovať na strane servera. Pri každej interakcii komponentu (napr. kliknutí na tlačidlo alebo zmene hodnoty vstupu) sa vykoná požiadavka na server pomocou AJAX-u a komponent sa znovu vyrenderuje.

Jedným z kľúčových benefitov Livewire je jeho prirodzená integrácia s Laravelom. Keďže ide o oficiálne podporovaný balík, vyvíjaný a udržiavaný komunitou okolo Laravelu (vrátane hlavného vývojára Caleba Porzia a podporený Laravel tímom), jeho aktualizácie, dokumentácia a podpora spĺňajú vysoké štandardy Laravel ekosystému. Tento fakt zaručuje dlhodobú kompatibilitu, stabilitu a bezpečnostné aktualizácie.

Vývojárske prostredie s Livewire si zachováva známe postupy Laravel frameworku, vrátane využívania Blade šablón (View časť MVC), komponentov, dátovej väzby (data binding) a validácie. Z pohľadu architektúry ponúka jednoduchší model ako tradičné JavaScript SPA aplikácie, čo uľahčuje vývoj aj menej skúseným programátorom alebo v menších tímoch. Navyše, Livewire je výborne kombinovateľný s ďalšími Laravel technológiami ako je Laravel Echo a iné.

Livewire taktiež uľahčuje prácu s formulármi, spracovávaním udalostí, dynamickým obsahom a stavovým manažmentom. To všetko bez nutnosti prepínať medzi PHP a JavaScriptom. V prípade potreby si však vývojár môže ručne definovať vlastné udalosti a priamo pracovať s JavaScriptom, čím získava flexibilitu porovnateľnú s klasickými frontend frameworkami.

Použitie Livewire je obzvlášť výhodné pri vývoji administračných rozhraní, CRUD formulárov, reaktívnych tabuliek, dashboardov alebo jednoducho všade tam, kde je potrebné dynamické používateľské rozhranie s dôrazom na server-side logiku. Zároveň poskytuje vysokú úroveň testovateľnosti a konzistencie, keďže väčšina logiky ostáva v PHP, čím sa znižuje riziko vzniku chýb spôsobených rozdielnym správaním klienta a servera.

6.5 MongoDB

MongoDB je moderná, dokumentovo orientovaná databáza typu NoSQL, ktorá umožňuje ukladanie a spracovanie dát vo formáte Binary JSON (BSON). Vznikla v roku 2009 ako alternatíva k relačným databázovým systémom, ktoré boli často príliš rigidné pre dynamicky meniace sa dáta moderných webových a cloudových aplikácií (Chodorow, 2013). MongoDB sa odvtedy etablovala ako jeden z najpopulárnejších NoSQL systémov a používa sa vo firmách ako Adobe, Bosch, eBay, Forbes či Cisco.

Jednou z kľúčových výhod MongoDB je jej flexibilita. Na rozdiel od tradičných SQL databáz nevyžaduje pevne definovanú schému, čo znamená, že každý dokument (záznam) môže mať inú štruktúru. Táto vlastnosť je veľmi užitočná pri ukladaní dát, ktoré sa vyvíjajú počas životného cyklu aplikácie – napríklad konfiguračné údaje o Petriho sieťach, stavy simulácií alebo dynamicky generované štatistiky. Pre túto implementáciu bolo práve toto kľúčové, pretože štruktúra stavov a prechodov môže byť v rôznych modeloch odlišná a meniť sa v čase.

Poskytuje tiež rýchle operácie čítania a zápisu, podporuje horizontálne škálovanie cez tzv. sharding a ponúka integrovanú replikáciu pre zabezpečenie vysokodostupných systémov. V praxi to znamená, že MongoDB je pripravená na nasadenie v produkčných systémoch s veľkým objemom dát a vysokými nárokmi na dostupnosť. Vývojár má zároveň možnosť pracovať s dátami prirodzene – ako s objektmi v jazykoch ako PHP či C#, čo znižuje potrebu konverzií a zjednodušuje vývoj.

Disponuje silným ekosystémom nástrojov, vrátane Mongo Compass (grafické UI pre prácu s databázou), Atlas (cloudová platforma pre MongoDB), ako aj rôznych ovládačov a knižníc pre integráciu s populárnymi frameworkmi ako Laravel či .NET. V rámci nášho projektu sme MongoDB použili ako úložisko pre stavové dáta – teda miesta, tokeny, prechody, záznamy o výpočtoch a výsledky simulácií. Výhodou bola najmä jednoduchosť serializácie a deserializácie týchto dátových štruktúr do formátu JSON/ BSON.

Dôležitým aspektom použitia MongoDB bola aj možnosť rýchleho vývoja bez nutnosti navrhovať striktne normalizovanú schému vopred. To nám umožnilo sústrediť sa najprv na funkcionalitu a až následne optimalizovať dátové modely. MongoDB zároveň poskytuje aj nástroje pre agregáciu dát a pokročilé dotazy, ktoré boli využité pri generovaní prehľadov a exportov simulačných výstupov. V kombinácii s výkonným backendom v C# a používateľským rozhraním v Laraveli tvorí MongoDB spoľahlivý základ pre perzistenciu dát v celom systéme.

7 Implementácia

Pri návrhu a implementácii frameworku a jeho následnej integrácii s webovým používateľským rozhraním sme sa rozhodoli použiť kombináciu technológií, ktoré reflektujú potreby výkonnosti, škálovateľnosti, modularity a komfortu pri vývoji (Hejlsberg et al., 2003). Konkrétne ide o jazyk C# a platformu .NET na serverovú časť simulačného jadra, Laravel framework pre napojenie na .NET API a používateľské rozhranie a databázu MongoDB na ukladanie stavov a metaúdajov o simuláciách.

7.1 Transformácia Petriflow XML do objektového modelu v C#

Po analýze štruktúry jazyka Petriflow a jeho jednotlivých komponentov sme pristúpili k praktickej implementácii objektového modelu v jazyku C#. Cieľom bolo vytvoriť takú architektúru tried, ktorá by nielen verne zrkadlila štruktúru XML dokumentu, ale zároveň bola rozšíriteľná a vhodná na ďalšiu prácu – napríklad simuláciu, vizualizáciu alebo validáciu Petriho siete. Keďže Petriflow je postavený na štandardnom XML formáte, ideálnym prístupom bolo využiť .NET-ový mechanizmus `XmlSerializer`, ktorý umožňuje jednoducho mapovať XML elementy na triedy s minimálnym množstvom kódu navyše.

Na začiatku transformácie sme navrhli koreňovú triedu `PetriflowProcess`, ktorá reprezentuje samotný proces. Pri využitý paradigiem Object-oriented programming (OOP), by bolo možné túto triedu abstrahovať do všeobecnejšej definicie, v našom prípade nebola takáto úroveň abstrakcie potrebná. Táto trieda zahŕňa všetky dôležité komponenty procesu – miesta, tranzície, hrany a roly.

Každý z týchto komponentov je reprezentovaný ako vlastná trieda, pričom medzi nimi existujú nepriame väzby cez identifikátory. Namiesto priameho referencovania objektov pomocou inštancií triedy (napr. Place v triede Arc) sú komponenty prepojené prostredníctvom identifikátorov id, sourceId, targetId atď. Táto voľba bola vedomá a umožňuje jednoduchšie parsovanie XML súborov bez nutnosti komplikovaného rozbaľovania závislostí už počas deserializácie.

```
[XmlRoot("process")]
public class PetriflowProcess
{
        [XmlAttribute("identifier")]
        public string Identifier { get; set; }

        [XmlAttribute("title")]
```

```
        public string Title { get; set; }

        [XmlElement("place")]
        public List<Place> Places { get; set; }

        [XmlElement("transition")]
        public List<Transition> Transitions { get; set; }

        [XmlElement("arc")]
        public List<Arc> Arcs { get; set; }

        [XmlElement("role")]
        public List<Role> Roles { get; set; }
}
```

Trieda `Place` slúži ako reprezentácia miesta v Petriho sieti. Každé miesto má svoj unikátny identifikátor a typ, ktorý definuje jeho správanie (napr. „start“, „end“, „immediate“). Miesto môže obsahovať aj viacero dátových polí, ktoré sú uložené v zozname DataFields. Tieto polia sú dôležité najmä pre rozšírenú funkcionalitu, kde Petriho sieť nie je len o tokenoch, ale aj o uchovávaní a manipulácii s údajmi.

```
public class Place
{
        [XmlAttribute("id")]
        public string Id { get; set; }

        [XmlAttribute("type")]
        public string Type { get; set; }

        [XmlElement("dataField")]
        public List<DataField> DataFields { get; set; }
}
```

Tranzície predstavujú dynamické prvky siete – udalosti alebo činnosti, ktoré môžu nastať za určitých podmienok. Trieda `Transition` preto obsahuje okrem identifikátora aj tri dôležité bloky: `precondition`, `action` a `postcondition`. Tieto sú reprezentované ako reťazce a obsahujú výrazy napísané v MVEL, ktoré sa neskôr počas vykonávania vyhodnocujú. Týmto spôsobom je možné podmieniť vykonanie tranzície určitým stavom dát alebo tokenov v sieti.

```
public class Transition
{
        [XmlAttribute("id")]
        public string Id { get; set; }

        [XmlElement("precondition")]
        public string Precondition { get; set; }

        [XmlElement("action")]
        public string Action { get; set; }

        [XmlElement("postcondition")]
        public string Postcondition { get; set; }
}
```

Spojnice medzi miestami a tranzíciami sú reprezentované triedou `Arc`. Táto trieda je navrhnutá tak, aby uchovávala základné informácie o smere spojenia (pomocou `sourceId` a `targetId`) a o type samotného spojenia. Typ môže byť napríklad `input`, `output`, `inhibitor`, `reset`, alebo `read`. Tieto typy sú nevyhnutné pre správne vykonávanie logiky Petriho siete, nakoľko určujú spôsob práce s tokenmi – či sa odoberajú, vkladajú, čítajú alebo blokujú prechod.

```
public class Arc
{
    [XmlAttribute("id")]
    public string Id { get; set; }

    [XmlAttribute("sourceId")]
    public string SourceId { get; set; }

    [XmlAttribute("targetId")]
    public string TargetId { get; set; }

    [XmlAttribute("type")]
    public string Type { get; set; }
}
```

Na úrovni práce s údajmi je kľúčovou triedou `DataField`. Táto trieda uchováva základné informácie o dátovom poli: jeho identifikátor, typ (napr. `string`, `integer`, `boolean`), predvolenú hodnotu, a či je pole editovateľné alebo povinné. Vďaka tomu je možné v procese

nielen sledovať stav tokenov, ale aj zaznamenávať kontextové údaje potrebné pre rozhodovanie. Dátové polia sú deklarované ako súčasť miest a tranzícií.

```
public class DataField
{
    [XmlAttribute("id")]
    public string Id { get; set; }

    [XmlAttribute("type")]
    public string Type { get; set; }

    [XmlAttribute("defaultValue")]
    public string DefaultValue { get; set; }

    [XmlAttribute("required")]
    public bool Required { get; set; }

    [XmlAttribute("editable")]
    public bool Editable { get; set; }
}
```

Pre implementáciu prístupových práv sme vytvorili triedu `Role`, ktorá reprezentuje roly používateľov, resp. systémových entít v procese. Každá rola má identifikátor a meno, ktoré sa používajú na spresnenie oprávnení pre spúšťanie tranzícií alebo prístup k údajom. Autorizácia nie je implementovaná ako samostatná trieda, ale ako atribúty alebo vnorené elementy v miestach a tranzíciách, kde sa roly používajú.

```
public class Role
{
    [XmlAttribute("id")]
    public string Id { get; set; }

    [XmlAttribute("name")]
    public string Name { get; set; }
}
```

Po návrhu tried sme vytvorili základný mechanizmus pre deserializáciu XML súboru do inštancie objektového modelu. Využili sme štandardný .NET `XmlSerializer`, ktorý je ideálny pre spracovanie dátových štruktúr z XML. Vďaka preddefinovaným atribútom `[XmlElement]`, `[XmlAttribute]` a `[XmlRoot]` nebolo potrebné implementovať žiadnu vlastnú logiku parsovania.

```
var serializer = new XmlSerializer(typeof(PetriflowProcess));
using (var reader = new StreamReader("model.xml"))
{
    PetriflowProcess process = (PetriflowProcess)serializer
                                        .Deserialize(reader);
}
```

Všetky komponenty sú navrhnuté tak, aby zodpovedali hierarchickej štruktúre XML súboru, no zároveň umožnili ich flexibilné využitie v ďalších častiach aplikácie. Pre vytváranie logických väzieb medzi komponentmi (napr. zistenie, ktoré miesta vstupujú do tranzície) bola vytvorená pomocná vrstva `PetriNetBuilder`, ktorá na základe identifikátorov prepojí objekty medzi sebou a vytvorí logickú štruktúru siete, pripravenú na simuláciu alebo vykonávanie.

7.2 Konštrukcia logickej štruktúry siete pomocou PetriNetBuilder

Po načítaní XML do objektového modelu pomocou `XmlSerializer` sme získali samostatné inštancie tried ako `Place`, `Transition`, `Arc` či `DataField`. Tieto objekty však zatiaľ existujú len ako dátové kontajnery bez explicitných prepojení. Miesto síce pozná svoje id, ale nevie, aké tranzície naň nadväzujú. Tranzícia vie, že má identifikátor, ale netuší, z akých miest prijíma tokeny. A práve tento problém rieši pomocná vrstva s názvom `PetriNetBuilder`.

Trieda `PetriNetBuilder` slúži na to, aby prepojené entity medzi sebou nadviazali logické vzťahy na základe identifikátorov. Cieľom je premeniť jednoduchý dátový model na plne prepojenú sieť, nad ktorou je možné vykonávať operácie ako simulácia, validácia, vykonanie tranzície či vizualizácia.

Interná štruktúra triedy PetriNetBuilder

Trieda obsahuje referenciu na `PetriflowProcess`, ako aj interné slovníky `(Dictionary)` pre rýchle vyhľadávanie komponentov podľa ID. Tým sa zabezpečí efektívnosť pri mapovaní väzieb.

```
public class PetriNetBuilder
{
    private readonly PetriflowProcess _process;
    private Dictionary<string, Place> _placesById;
    private Dictionary<string, Transition> _transitionsById;

    public PetriNetBuilder(PetriflowProcess process)
    {
        _process = process;
```

```
        InitializeLookups();
    }

    private void InitializeLookups()
    {
        _placesById = _process.Places.ToDictionary(p => p.Id);
        _transitionsById = _process.Transitions.ToDictionary(t => t.Id);
    }
}
```

Prepojenie miest a tranzícií cez hrany

Najdôležitejším krokom je iterácia cez zoznam hrán (Arc) a identifikácia ich typu. Podľa toho sa vykoná vytvorenie prepojenia medzi príslušnými objektmi. Tento krok je rozhodujúci pre vytvorenie logickej štruktúry siete, ktorá umožňuje simulovať tok tokenov.

```
public void LinkComponents()
{
    foreach (var arc in _process.Arcs)
    {
        if (arc.Type == "input")
        {
            if (_placesById.TryGetValue(arc.SourceId, out var place) &&
            _transitionsById.TryGetValue(arc.TargetId, out var transition))
            {
                transition.InputPlaces.Add(place);
                place.OutputTransitions.Add(transition);
            }
        }
        else if (arc.Type == "output")
        {
            if (_transitionsById.TryGetValue(arc.SourceId,
                    out var transition) &&
                _placesById.TryGetValue(arc.TargetId, out var place))
            {
                transition.OutputPlaces.Add(place);
                place.InputTransitions.Add(transition);
            }
        }
    }
}
```

Úprava tried Place a Transition pre navigáciu

Aby vyššie uvedené prepojenia vôbec bolo kam uložiť, bolo potrebné rozšíriť triedy `Place` a `Transition` o referencie na naviazané objekty. Tieto kolekcie slúžia na logickú navigáciu v sieti – napríklad z miesta môžeme zistiť, aké tranzície sú od neho spustiteľné.

```
public class Place
{
    [XmlAttribute("id")]
    public string Id { get; set; }

    [XmlAttribute("type")]
    public string Type { get; set; }

    [XmlElement("dataField")]
    public List<DataField> DataFields { get; set; }

    // Pridané navigačné vlastnosti
    [XmlIgnore]
    public List<Transition> InputTransitions { get; } = new();

    [XmlIgnore]
    public List<Transition> OutputTransitions { get; } = new();
}

public class Transition
{
    [XmlAttribute("id")]
    public string Id { get; set; }

    [XmlElement("precondition")]
    public string Precondition { get; set; }

    [XmlElement("action")]
    public string Action { get; set; }

    [XmlElement("postcondition")]
    public string Postcondition { get; set; }

    // Navigačné väzby
    [XmlIgnore]
    public List<Place> InputPlaces { get; } = new();
```

```
    [XmlIgnore]
    public List<Place> OutputPlaces { get; } = new();
}
```

Použitie buildera v praxi

Po načítaní XML a vytvorení modelu je možné jednoduchým spôsobom zostaviť sieť tak, že všetky komponenty budú logicky previazané.

```
    var serializer = new XmlSerializer(typeof(PetriflowProcess));
    using var reader = new StreamReader("model.xml");
    var process = (PetriflowProcess)serializer.Deserialize(reader);

    var builder = new PetriNetBuilder(process);
    builder.LinkComponents();

    foreach (var place in process.Places)
    {
        Console.WriteLine($"Miesto {place.Id} vedie do tranzícií:");
        foreach (var transition in place.OutputTransitions)
        {
            Console.WriteLine($" - {transition.Id}");
        }
    }
```

Vrstva `PetriNetBuilder` predstavuje most medzi čistou dátovou reprezentáciou získanou zo vstupného XML a logickou štruktúrou potrebnou pre simuláciu a interpretáciu procesu. Vďaka nej sa z „neprepojených“ dát stáva plnohodnotný graf, kde sú jasne definované vzťahy medzi jednotlivými miestami a tranzíciami. Tento krok je nevyhnutný pre akékoľvek ďalšie spracovanie, ako sú testovanie korektnosti modelu, vizualizácia grafu, simulácia správania alebo automatické generovanie REST rozhraní pre interakciu s procesom.

7.3 Implementácia REST API rozhrania pre prácu s PetriNetBuilderom

REST API navrhnuté v rámci tejto práce slúži na komunikáciu s komponentom PetriNetBuilder, konkrétne na jeho integráciu s webovým používateľským rozhraním. Použitie REST architektúry bolo zvolené z dôvodu jednoduchosti, štandardizácie a širokej podpory v rôznych programovacích jazykoch a technológiách. Podľa Fieldinga (Fielding, 2000) REST API definuje jasne oddelené zdroje dostupné pomocou štandardných Hypertext Transfer Protocol

(HTTP) metód (GET, POST, DELETE, atď.) s jednoznačnou identifikáciou URI adries. Dáta sú prenášané vo formáte JSON, ktorý je bežnou praxou vo webových aplikáciách (Blaga et al., 2021).

7.3.1 Autorizácia cez JWT

Bezpečnosť REST API je zabezpečená prostredníctvom autorizácie pomocou JSON Web Token (JWT). JWT je otvorený štandard (RFC 7519) definovaný spoločnosťou IETF, ktorý špecifikuje bezpečný prenos informácií medzi entitami ako JSON objekt (Jones et al., 2015). Tento okjekt alebo tiež nazývaný token sa generuje po úspešnom prihlásení používateľa a je následne priložený k HTTP požiadavkám v hlavičke `Authorization`.

JWT sa skladá z troch častí: hlavičky, payloadu a podpisu. Hlavička špecifikuje typ tokenu (typicky JWT) a použitý algoritmus podpisu (napríklad HS256 alebo RS256). Payload obsahuje tvrdenia (claims) o používateľovi, napríklad identifikátor používateľa, jeho roly, oprávnenia alebo platnosť tokenu. Podpis je použitý na overenie integrity a autenticity tokenu, čím zabezpečuje, že token nebol zmenený počas prenosu.

JWT je vhodný na použitie v REST API z viacerých dôvodov. Predovšetkým preto, že je nezávislý od stavu (stateless), server nemusí uchovávať relácie ani stavové informácie o klientoch, čo značne zjednodušuje škálovateľnosť aplikácie. Každá požiadavka obsahuje všetky potrebné informácie na overenie a autorizáciu, čo eliminuje potrebu opakovaného dotazovania databázy alebo iných úložísk údajov pre každú požiadavku.

Tokeny majú nastavenú dobu expirácie, ktorá môže byť krátkodobá (napríklad niekoľko minút alebo hodín), čo výrazne zvyšuje bezpečnosť aplikácie. Po skončení platnosti tokenu musí používateľ získať nový token prostredníctvom opätovného prihlásenia alebo obnovy tokenu cez refresh token. Táto stratégia zabezpečuje minimalizáciu rizika v prípade kompromitácie tokenu.

Implementácia JWT autorizácie je relatívne jednoduchá vďaka existencii množstva knižníc a frameworkov, ktoré podporujú JWT štandard vo väčšine moderných programovacích jazykov vrátane C#. ASP.NET Core napríklad poskytuje zabudované nástroje na rýchlu a efektívnu implementáciu JWT autentifikácie, vrátane generovania a overovania tokenov priamo v middleware aplikácie.

Na strane klienta sa JWT token odosiela spolu s každou požiadavkou v hlavičke HTTP. V našej aplikácii bude požiadavka obsahovať hlavičku v tvare:

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2...
```

Táto hodnota predstavuje token, ktorý obsahuje zakódované údaje o používateľovi. Server pri spracovaní požiadavky overí podpis a platnosť tokenu a následne rozhodne, či používateľ má oprávnenie na požadovanú operáciu.

Tento druh autorizácie nám umožnil efektívnu a rýchlu implementáciu. Technológie, ktoré sme použili, majú existujúcu podporu pre JWT a jeho generovanie. Vďaka tomu sme sa mohli sústrediť na implementáciu logiky aplikácie a nie na detaily autentifikácie a autorizácie.

7.3.2 Rozdelenia REST API na funkčné metódy

Pre potreby implmentácie sme sa rozhodli rozdeliť REST API na niekoľko základných funkčných metód, ktoré pokrývajú najčastejšie operácie potrebné pre prácu s PetriNetBuilderom.

Rozdelenie REST API na jednotlivé funkčné metódy, ktoré vykonávajú špecifické akcie, prináša niekoľko zásadných výhod. Prvou výhodou je jasnosť a prehľadnosť API rozhrania, ktorá umožňuje klientom ľahko pochopiť funkcionalitu dostupnú cez jednotlivé endpointy. Každá metóda má jasne definovaný účel a očakávané vstupné a výstupné parametre, čo zjednodušuje vývoj a integráciu na strane klientov.

Ďalšou výhodou je modularita a rozšíriteľnosť API, pretože pridávanie nových funkcií alebo modifikácia existujúcich funkcií sa realizuje izolovane bez ovplyvnenia zvyšku systému. V prípade potreby je možné samostatne optimalizovať výkon, bezpečnosť a škálovateľnosť jednotlivých metód.

Taktiež takéto rozdelenie zlepšuje testovateľnosť systému, nakoľko každá metóda môže byť testovaná samostatne, čo umožňuje jednoduchšiu diagnostiku a opravu chýb. V konečnom dôsledku to vedie k stabilnejšiemu a robustnejšiemu systému.

7.3.3 Funkčné metódy REST API

Metóda StoreNet

Metóda StoreNet umožňuje uloženie XML definície Petriho siete vo formáte Petriflow. Klient odosiela XML súbor cez HTTP POST požiadavku, ktorý je následne parsovaný a uložený pomocou triedy PetriNetBuilder (Jenrich, 2013). Výsledkom úspešného spracovania je vygenerovanie unikátneho identifikátora siete (netId), ktorý sa používa pre ďalšiu komunikáciu.

Príklad volania:

```
POST /api/nets
Content-Type: application/xml

<!-- XML Petriflow definícia -->
```

Úspešná odpoveď vracia JSON s vygenerovaným identifikátorom a HTTP statusom 200 OK. V prípade chyby sa vráti chybový kód a správa s podrobnosťami o chybe.

```
{
    "netId": "a859rf",
    "status": "success",
    "message": "Network was successfully stored."
}
```

Metóda GetNetGraph

Táto metóda poskytuje grafickú reprezentáciu Petriho siete v podobe SVG obrázka. Graf je generovaný pomocou knižnice GraphViz (Gansner et al., 2000), ktorá na základe jednoduchého definovaného jazyka DOT vytvorí obrázok siete. Tento obrázok sa priamo integruje do webového rozhrania.

Príklad požiadavky:

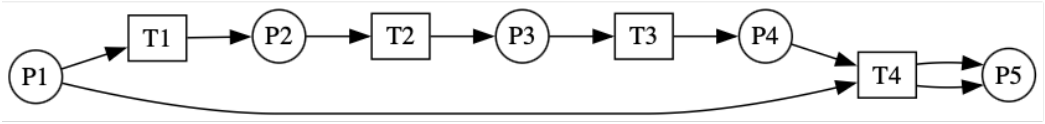
```
GET /api/nets/{netId}/graph
```

JSON odpoveď obsahuje SVG obrázok aj samotnú DOT definíciu Petriho siete so statusom 200 OK. V prípade, že sieť neexistuje, vráti sa chybový kód 404 Not Found.

```
{
    "netId": "a859rf",
    "svgImage": "<svg><!-- SVG obsah --></svg>",
    "dotDefinition": "digraph G { ... }",
}
```

Ukážka GraphViz DOT definície pre jednoduchú sieť ako aj jej vizuálnu prezentáciu je zobrazená na obrázku 4. Tento formát je veľmi efektívny pre vizualizáciu grafov a umožňuje jednoduché prispôsobenie vzhľadu a rozloženia siete.

```
digraph G {
    rankdir=LR;
    center=true;margin=1;
    subgraph place {
        node [shape=circle,fixedsize=true,height=.4,width=.4];
        P1,P2,P3,P4,P5;
    }
    subgraph transitions {
        node [shape=rect,height=0.2,width=.2];
        T1,T2,T3,T4;
    }
    P1->T1->P2;
    P2->T2->P3;
    P3->T3->P4;
    P4->T4->P5;
    P1->T4->P5;
}
```



Obr. 4: Ukážka DOT vizualizácie Petriho siete

Metóda GetNetStatus

Aby sme získali aktuálny stav siete vrátane rozmiestnenia tokenov v jednotlivých miestach a informácie o dostupných tranzíciách, ktoré môžu byť spustené v aktuálnom stave použijeme metódu GetNetStatus.

Príklad požiadavky:

```
GET /api/nets/{netId}/status
```

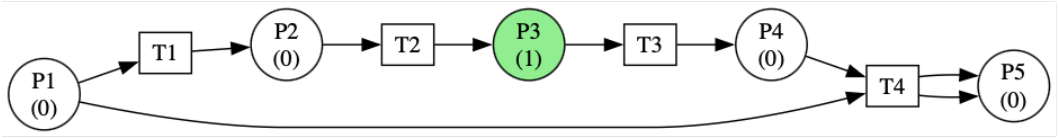
Odpoveď obsahuje aktuálny stav siete vo formáte JSON s informáciami o miestach, tranzíciách a tokenoch. V prípade, že sieť neexistuje, vráti sa chybový kód 404 Not Found.

```
{
    "netId": "a859rf",
    "tokens": { "p1": 1, "p2": 0 },
    "activeTransitions": ["t1"],
    "completed": false
}
```

Ak pridáme do volania aj prepínač `?image=true`, vráti sa aj SVG obrázok aktuálneho stavu siete. Tento obrázok je generovaný na základe aktuálneho rozmiestnenia tokenov a zobrazuje ich v miestach siete. Chybové správanie je rovnaké pre oba prípady.

Metóda RunNet

Metóda umožňuje spustenie konkrétnej tranzície. Po odoslaní požiadavky server overí, či sú splnené podmienky na spustenie tranzície a aktualizuje stav siete.



Obr. 5: Ukážka zobrazenia stavu siete s tokenmi

Príklad požiadavky:

```
POST /api/nets/{netId}/run
Content-Type: application/json

{
    "transitionId": "t1"
}
```

Server následne odpovie aktualizovaným stavom siete:

```
{
    "netId": "a859rf",
    "executedTransition": "t1",
    "status": "success",
    "tokens": { "Place1": 0, "Place2": 1 },
    "activeTransitions": [],
    "completed": true
}
```

7.4 Implementácia používateľského rozhrania a napojenie REST API v Laraveli

Pre zjednodušenie interakcie používateľa so simulačným jadrom implementovaným v jazyku C# bolo navrhnuté samostatné webové rozhranie, ktoré komunikuje s REST API. Webové rozhranie je postavené na PHP frameworku Laravel, pričom pri návrhu sa kládol dôraz na prehľadnosť, rychlosť odozvy a minimálne technické nároky na klientskom zariadení.

Zvolený prístup rešpektuje princípy klient-server architektúry. Laravel aplikácia vystupuje ako frontend-backend server, ktorý zabezpečuje nielen spracovanie požiadaviek používateľa, ale aj zabezpečenie prístupu, správu session a autorizáciu prostredníctvom JWT. Zároveň umožňuje dynamickú prácu s údajmi a ich reprezentáciu v reálnom čase.

Používateľské rozhranie je navrhnuté ako samostatný Laravel projekt. V rámci tohto projektu sú definované komponenty, ktoré pokrývajú najdôležitejšie operácie: výber siete, zobrazenie aktuálneho stavu, zoznam dostupných tranzícií a možnosť ich spúšťania. Údaje sú získavané volaním REST API metód na strane C# simulátora.

Jadrom dynamiky aplikácie je použitie technológie Laravel Livewire, ktorá umožňuje vytváranie interaktívnych komponentov bez potreby písania JavaScript kódu. Livewire zabezpečuje, že komponenty na stránke môžu komunikovať so serverom, vykonávať logiku a aktualizovať svoje zobrazenie bez toho, aby došlo k obnoveniu celej stránky.

Použitie Livewire sa ukázalo ako mimoriadne výhodné z viacerých dôvodov. Po prvé, umožňuje veľmi rýchly vývoj – namiesto písania oddelených JS modulov alebo SPA riešení v Reacte či Vue, je možné celú logiku spravovať v PHP triedach. Po druhé, Livewire komponenty integrujú serverové aj klientské spracovanie do jedného celku, čo zjednodušuje údržbu a čitateľnosť kódu.

Dôležité je však spomenúť, že Livewire do istej miery narúša klasický MVC model, na ktorom je Laravel postavený. Komponenty Livewire totiž kombinujú View aj Controller do jednej triedy, čím sa logika a vykresľovanie stierajú. Aj keď to znižuje abstrakciu a prísnu separáciu zodpovedností, v menších a stredne veľkých aplikáciách to zvyšuje produktivitu a zjednodušuje orientáciu v kóde.

Na ilustráciu toho, ako prebieha komunikácia medzi frontend-backend aplikáciou a simulátorom, uvádzame ukážku Livewire komponentu `GetNetStatusComponent`, ktorý zabezpečuje načítanie aktuálneho stavu siete a spúšťanie tranzícií:

```php
use Illuminate\Support\Facades\Http;
use Livewire\Component;

class GetNetStatusComponent extends Component
{
    public string $netId;
    public string $svg = '';
    public array $activeTransitions = [];

    public function mount(string $netId)
    {
        $this->netId = $netId;
        $this->refreshNetStatus();
    }

    public function refreshNetStatus(): void
    {
        $response = Http::withToken(auth()->user()->jwt_token)
            ->get(
                    config('core.net-simulation.url') .
                    "/api/nets/{$this->netId}/status?image=true"
                );

        if ($response->successful()) {
            $this->svg = $response['svgImage'];
            $this->activeTransitions = $response['activeTransitions'];
        } else {
            session()->flash('error', 'Nepodarilo sa načítať stav siete.');
        }
    }

    public function runTransition(string $transitionId): void
    {
        $response = Http::withToken(auth()->user()->jwt_token)
            ->post(
                config('core.net-simulation.url') .
                "/api/nets/{$this->netId}/run",
                [
                    'transitionId' => $transitionId,
                ]
            );
```

```
        if ($response->successful()) {
            $this->refreshNetStatus();
        } else {
            session()->flash('error', 'Nepodarilo sa spustiť tranzíciu.');
        }
    }

    public function render()
    {
        return view('app.net-simulation.livewire.get-net-status');
    }
}
```

Metóda `refreshNetStatus` predstavuje volanie REST API endpoint `GET /api/nets/{netId}/status?image=true`, ktorý vracia aktuálny stav siete vrátane vizualizácie vo formáte SVG. Dáta sú následne uložené do verejných premenných komponentu a automaticky zobrazené používateľovi.

REST API odpoveď má typickú štruktúru:

```
    {
        "netId": "a859rf",
        "svgImage": "<svg>...</svg>",
        "activeTransitions": ["T1", "T3"],
        "tokens": {
            "P1": 1, "P2": 0
        },
        "completed": false
    }
```

Na strane servera (aplikačnej vrstvy aplikácie používateľské rozhrania) sú informácie o aktuálnych stavoch Petriho sietí priebežne ukladané v databáze MongoDB. Typický dokument v kolekcii `net_instances` vyzerá nasledovne:

```
{
  "_id": ObjectId("66514e839dd5a4a6b9b0aa91"),
  "netId": "a859rf",
  "createdAt": ISODate("2025-05-17T10:15:00Z"),
  "places": {
    "P1": { "tokens": 1 },
```

```
    "P2": { "tokens": 0 },
    "P3": { "tokens": 0 }
  },
  "activeTransitions": ["T1"],
  "lastExecuted": "T0",
  "status": "running"
}
```

Pri každom spustení tranzície sa tento dokument aktualizuje pomocou `updateOne()`, pričom MongoDB sa využíva ako persistované, ale schemaless úložisko. Týmto spôsobom je možné kedykoľvek získať aktuálny stav siete alebo vykonať analýzu histórie spúšťania prechodov.

Tak ako je architektúra navrhnutá, v kombinácii s MongoDB mi umožnila vytvoriť plnohodnotný, samostatne stojaci FrontEnd–BackEnd, ktorý môže fungovať nezávisle od simulačného jadra napísaného v jazyku C#. Týmto spôsobom som dosiahol vysokú modularitu, flexibilitu a možnosť budúcej integrácie s inými systémami bez zásahu do samotného frontendového riešenia.

8 Testovanie

Testovanie systému prebiehalo v troch samostatných vrstvách – na úrovni simulačného enginu, na úrovni REST API a priamo v rámci Laravel aplikácie. Každá z týchto častí bola testovaná nezávisle, pričom dôraz bol kladený na konzistenciu výstupov, korektné spracovanie chybných vstupov a integritu dát. Použitý bol kombinovaný prístup, zahŕňajúci jednotkové, funkcionálne a integračné testy.

Testovanie simulačného enginu sa realizovalo priamo nad internými reprezentáciami Petriho sietí, ktoré boli načítavané z testovacích prípadov spoločnosti Netgrif. Tie sú verejne dostupné na GitHub repozitári a predstavujú prakticky overené a stabilné modely rôznych scenárov workflow procesov.

Každý scenár bol po načítaní do pamäte prevedený na inštanciu simulačného modelu, následne sa spúšťali jednotlivé prechody a testoval sa výsledný marking siete. V prípade úspešného prechodu sa test overoval pomocou porovnania aktívnych miest so známou očakávanou konfiguráciou. Testy boli implementované ako `Fact` metódy v rámci `xUnit`, pričom ich výstupy boli automaticky validované voči referenčným hodnotám.

```
[Fact]
public void ShouldFireTransitionAndUpdateMarking()
{
    var net = PetriNetLoader.LoadFromResources("netgrif/testnet.xml");
    var instance = net.CreateInstance();

    Assert.True(instance.CanFire("start"));
    instance.TriggerTransition("start");

    var marking = instance.GetMarking();
    Assert.True(marking.Contains("p2"));
}
```

REST API vrstva bola testovaná pomocou nástroja `WebApplicationFactory` v kombinácii s `HttpClient`, čím bolo možné simulovať reálnu komunikáciu medzi Laravel backendom a enginom bez potreby spúšťania aplikácie cez sieť. Testovanie pokrývalo všetky verejné endpointy – vytváranie novej inštancie siete, zisťovanie aktuálneho stavu a spúšťanie konkrétnych prechodov. Dôležitým aspektom bolo overenie správania sa pri chybných vstupoch, ako sú neexistujúce prechody alebo neplatné ID siete. Testy pracovali s vopred známymi

dátovými sadami a simulovali viacero scenárov správania, vrátane asynchrónneho spracovania požiadaviek. V prípade potreby sa využilo mockovanie logiky pomocou knižnice `Moq`.

```
[Fact]
public async Task TriggerTransition_ReturnsOk_WhenTransitionFires()
{
    var client = _factory.CreateClient();

    var response = await client.PostAsync("/api/nets/123/run", [
        new StringContent(
            "{\"transitionId\":\"t2\"}",
            Encoding.UTF8,
            "application/json"
        )
    ]);

    Assert.Equal(HttpStatusCode.OK, response.StatusCode);
    var json = await response.Content.ReadAsStringAsync();
    Assert.Contains("\"status\":\"success\"", json);
}
```

Laravel aplikácia bola testovaná pomocou frameworku Pest, ktorý umožňuje rýchle a efektívne písanie čitateľných testov. V rámci tejto vrstvy sa testovali najmä tri typy komponentov: Livewire komponenty, HTTP endpointy a interakcie s databázou MongoDB. Livewire komponenty boli testované pomocou metódy `Livewire::test()`, ktorá umožňuje simulovať celú životnosť komponentu – vrátane jeho inicializácie, vykonania metód a reakcie na zmeny vstupov. V týchto testoch sa overovala napríklad správna aktivácia prechodu v simulácii, vizuálne zobrazenie výsledkov a validácia formulárov.

```
it('can trigger transition from Livewire component', function () {
    Livewire::test('transition-runner', ['netId' => 'abc123'])
        ->call('triggerTransition', 'start')
        ->assertSee('Transition executed successfully');
});
```

API endpointy v Laravel aplikácii boli testované pomocou metód `getJson`, `postJson` a `assertJsonFragment`, ktoré overovali stavové kódy HTTP odpovedí a ich obsah. Pre každý

endpoint boli definované minimálne tri prípady: úspešná požiadavka, chyba so statusom 4xx a neočakávaný vstup vedúci k chybe na strane servera. Tieto testy umožnili včas zachytiť zmeny v API špecifikácii alebo regresie vzniknuté pri úpravách business logiky.

```
it('returns net instance from API', function () {
    $response = $this->getJson('/api/net/abc123/status');

    $response->assertOk()
            ->assertJsonFragment(['place' => 'p2']);
});
```

Interakcia s MongoDB bola testovaná s použitím testovacej databázy, pričom pred každým testom sa kolekcie explicitne čistili. Prístup ku kolekciám bol overovaný pomocou helperov, ktoré zisťovali prítomnosť konkrétnych dokumentov po vykonaní simulácie, vytvorení inštancie alebo aktualizácii stavu. Napríklad po úspešnom spustení prechodu bol test vykonaný nad kolekciou `net_states` s cieľom overiť, že nové markingy boli správne zapísané.

```
it('writes updated state to MongoDB after transition', function () {
    $netId = 'abc123';
    $this->postJson("/api/net/$netId/transitions/start");

    $doc = DB::connection('mongodb')
        ->collection('net_states')
        ->where('net_id', $netId)->first();
    expect($doc)->not->toBeNull();
    expect($doc['places'])->toContain('p2');
});
```

Na konci každého testovacieho cyklu bola vygenerovaná sumarizačná správa s počtom spustených, úspešných a zlyhaných testov. Táto správa bola využívaná ako súčasť CI procesu a zároveň poslúžila ako základný dôkaz o funkčnosti jednotlivých častí systému. Vďaka takto nastavenej testovacej infraštruktúre je možné nové funkcie nasadzovať s vysokou mierou dôvery v ich správne fungovanie.

9 Budúce rozšírenie aplikácie

Systém bol navrhnutý modulárne a jeho architektúra umožňuje ďalšie rozširovanie bez nutnosti zásadných zmien v jadre aplikácie. V budúcnosti je možné aplikáciu posunúť smerom k praktickému nasadeniu v priemyselných prostrediach, najmä v kontexte Industry 4.0, kde sa kladie dôraz na digitálnu automatizáciu, IoT, decentralizáciu rozhodovania a simuláciu procesov v reálnom čase. V tejto kapitole navrhujeme konkrétne oblasti, ktorým by sa vývoj aplikácie mohol v budúcnosti venovať.

Jednou z hlavných oblastí budúceho rozšírenia je napojenie simulačného enginu na reálne priemyselné prostredie pomocou protokolu OPC UA. Tento protokol sa stal štandardom pre komunikáciu medzi systémami v rámci automatizácie a umožňuje výmenu údajov medzi PLC riadiacimi jednotkami, senzormi a softvérovými nástrojmi. Rozšírením REST API o OPC UA klienta by bolo možné získať aktuálny stav fyzických zariadení (napríklad stav snímača, rýchlosť pásu, počet výrobkov na linke) a na základe týchto údajov dynamicky aktualizovať stav simulačnej siete. Naopak, výsledky simulácie môžu byť použité ako vstupy pre riadiace logiky, čím by sa dosiahla obojsmerná integrácia medzi softvérom a fyzickým svetom.

Ďalším krokom je rozšírenie simulačného enginu o podporu časovaných a farebných Petriho sietí, ktoré sú nevyhnutné pri modelovaní zložitých systémov s obmedzenými časovými rámcami, prioritami alebo atribútmi spojenými s tokenmi (napr. typ produktu, dátum expirácie). V rámci enginu je možné implementovať deterministické aj stochastické časovanie prechodov, čo umožní simuláciu rôznych typov procesov vrátane výrobných cyklov, prestojov alebo skladovania. Farebné siete by zase umožnili pracovať s rôznymi typmi objektov paralelne, čím by sa zvýšila flexibilita modelovania zložitých výrobných liniek.

Z pohľadu užívateľského rozhrania je možné rozšíriť Livewire frontend o vizuálny editor Petriho sietí, ktorý by umožňoval grafickú manipuláciu s miestami, prechodmi a väzbami priamo v prehliadači. Na tento účel je možné integrovať knižnice ako `GoJS`, `JointJS` alebo `Draw2D`, ktoré umožňujú prácu s grafovými modelmi. Tým by sa eliminovala potreba manuálneho zápisu alebo importovania modelu a aplikácia by bola prístupnejšia pre menej technicky zdatných používateľov.

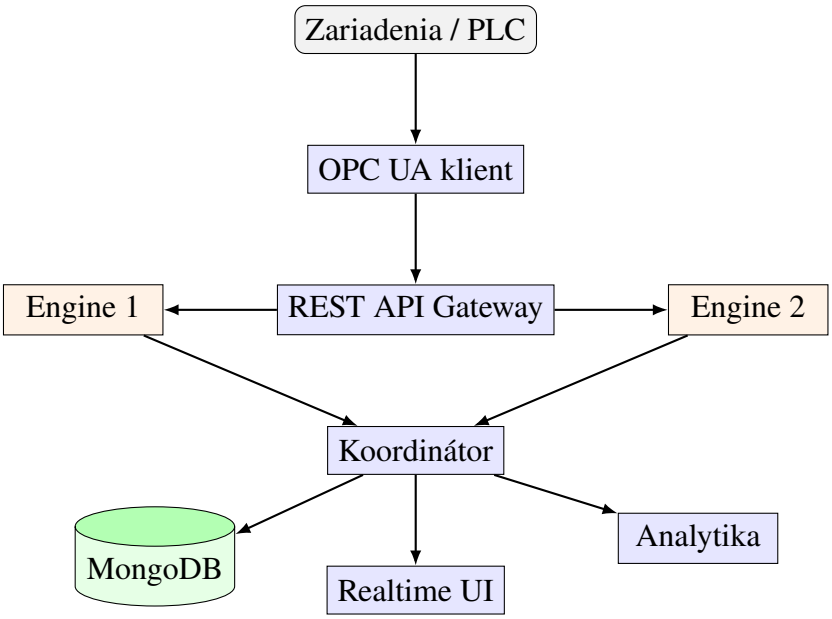
Dôležitým aspektom je aj škálovateľnosť systému. V súčasnosti je celý engine navrhnutý ako kvázi-monolitická aplikácia, ale v prostredí Industry 4.0 je bežné spracovávať veľké množstvo paralelných procesov. Preto je vhodné zvážiť migráciu enginu do podoby microservices architektúry a nasadiť ho v kontajnerizovanom prostredí (napríklad pomocou Docker a Kubernetes). To umožní horizontálne škálovanie podľa aktuálnej záťaže, čím sa zabezpečí rýchla odozva systému aj pri tisíckach súčasných simulácií.

Zároveň možno do budúcna implementovať modul na štatistické vyhodnocovanie simulácií. Takýto modul by dokázal zbierať výstupné dáta zo simulácií, analyzovať ich pomocou metrík ako priemerná doba priechodu sieťou, využitie zdrojov, detekcia zablokovania alebo odhad úzkych miest. Výsledky by mohli byť vizualizované formou grafov alebo exportované do CSV či Excel formátu pre ďalšiu analytiku.

Z technického hľadiska je možné zvážiť aj zrýchlenie výpočtov enginu prostredníctvom paralelizácie a GPU akcelerácie. Napríklad výpočet možných prechodov, simulácia ich aktivácie alebo validácia logických podmienok by mohli byť vykonávané v samostatných vláknach alebo na grafickej karte, čím by sa dramaticky zvýšil výkon pri masívnych modeloch.

Na záver, aplikácia by mohla byť rozšírená o podporu modelovania v reálnom čase, kde sa sieť neustále aktualizuje na základe dát zo senzorov alebo PLC, a používatelia môžu do simulácie zasahovať počas jej behu. Takáto funkcionalita by výrazne priblížila systém k skutočnému digitálnemu dvojčaťu výrobného procesu, čo je jeden z kľúčových konceptov Industry 4.0.

Nasledujúca ilustrácia obr. 6 zobrazuje prehľadnú architektúru systému, ktorá by mohla byť základom pre budúce rozširenie aplikácie na základe hore uvedených myšlienok.



Obr. 6: Prehľadná architektúra systému pripravená pre rozšírenie do prostredia Industry 4.0

10 Záver

V rámci tejto diplomovej práce sa nám podarilo navrhnúť a implementovať komplexný framework pre prácu s Petriho sieťami, pričom primárnym cieľom bolo vytvoriť nástroj, ktorý bude zároveň moderný, rozšíriteľný a pripravený na nasadenie v technológiách blízkych paradigme Industry 4.0. Použitím jazyka C# na strane simulačného enginu a PHP frameworku Laravel pre používateľské rozhranie sa nám podarilo spojiť silu výkonného backendového prostredia s flexibilitou webového rozhrania.

Kľúčovým prvkom navrhnutej architektúry bolo oddelenie samotného simulačného jadra od prezentačnej vrstvy. Tento prístup umožňuje jeho nezávislé nasadenie a využitie v rôznych prostrediach, kde je požiadavka na simulačné služby. Naše riešenie tak nie je limitované konkrétnym frontendom a je schopné pracovať v rámci ľubovoľnej aplikácie, ktorá podporuje volanie REST API.

Pri implementácii sme narazili na viacero praktických problémov. Významnú výzvu predstavovala integrácia medzi dvoma odlišnými technologickými svetmi – C# a PHP. Bolo potrebné zabezpečiť konzistentnú komunikáciu medzi týmito dvoma vrstvami, čo si vyžadovalo správne navrhnuté rozhrania, formátovanie dát a ošetrenie výnimiek v rámci obojsmernej výmeny informácií. Zároveň bolo potrebné riešiť otázky bezpečnosti prístupu pomocou tokenovej autorizácie a zabezpečiť konzistentné správanie systému aj v prípade výpadkov alebo chybových stavov.

Framework umožňuje načítanie modelov vo formáte Petriflow XML, ich transformáciu do interných objektových štruktúr a následnú simuláciu tokov v sieti. Vytvorená webová aplikácia zabezpečuje prehľadnú vizualizáciu siete, jej aktuálneho stavu a poskytuje používateľovi možnosť interaktívne spúšťať prechody. Všetky dáta sú udržiavané v MongoDB databáze, ktorá umožňuje efektívnu prácu s polostruktúrovanými údajmi.

Z hľadiska testovania sa nám podarilo pokryť všetky vrstvy systému – od jednotkových testov simulátora v jazyku C#, cez testovanie REST API až po funkčné testy webového frontendového rozhrania. Tento viacvrstvový testovací prístup nám umožnil identifikovať a odstrániť množstvo možných nekonzistencií medzi jednotlivými komponentmi systému, čím sa zvýšila jeho robustnosť a spoľahlivosť.

Možnosti budúceho rozšírenia boli detailne opísané v samostatnej kapitole. Už teraz je však jasné, že tento framework má potenciál pre výrazný rozvoj – najmä v smere podpory farebných a časovaných Petriho sietí, reálnočasovej integrácie so senzorickými systémami, ako aj napojenia na priemyselné štandardy ako OPC UA.

To výrazne rozširuje oblasť jeho použitia z čisto akademického prostredia do sféry praktických aplikácií v oblasti digitálnych dvojčiat a simulačných modelov priemyselnej výroby.

Na základe realizácie tejto práce sme si hlbšie uvedomili význam Petriho sietí v kontexte moderných informačných technológií. Ukazuje sa, že tento formálny model, ak je správne implementovaný a doplnený o moderné softvérové praktiky, dokáže slúžiť ako základ pre tvorbu simulačných platforiem novej generácie. V čase, keď sa kladie čoraz väčší dôraz na adaptívne riadiace systémy, prediktívne modelovanie a reálny monitoring procesov, predstavuje táto práca príspevok k vytvoreniu nástroja, ktorý dokáže tieto požiadavky efektívne naplniť.

Výsledný systém tak nie je len technologickým testom, ale plnohodnotným základom, na ktorom je možné ďalej budovať.

Literatúra

BLAGA, F S; POP, A; HULE, V; INDRE, C I, 2021. The efficiency of modeling and simulation of manufacturing systems using Petri nets. Roč. 1169, č. 1, s. 012005. Dostupné z DOI: 10.1088/1757-899X/1169/1/012005.

BONET, Pedro; LLUCH-LAFUENTE, Alberto, 2003. PIPE v1.0: A Platform Independent Petri Net Editor. In: *Proc. 8th Int. Conf. on Application and Theory of Petri Nets*.

DAS, Sandip; GHOSH, Prantar; GHOSH, Shamik; SEN, Sagnik, 2021. Oriented bipartite graphs and the Goldbach graph. *Discrete Mathematics*. Roč. 344, č. 9, s. 112497. ISSN 0012-365X. Dostupné z DOI: https://doi.org/10.1016/j.disc.2021.112497.

DAVID, Ren; ALLA, Hassane, 2010. *Discrete, Continuous, and Hybrid Petri Nets*. 2nd. Springer Publishing Company, Incorporated. ISBN 3642106684.

DER, W; AALST, Wil; TER, Arthur, 2003. YAWL: Yet another workflow language (revised version).

DONATELLI, Susanna; FRANCESCHINIS, Giuliana; CHIOLA, Giovanni, 1995. The GreatSPN Tool: Recent Enhancements. *Performance Evaluation*. Roč. 24, č. 1–2, s. 47–68.

ESPARZA, Javier; NIELSEN, Mogens, 1998. Decidability issues for Petri nets – a survey. *Bulletin of the EATCS*. Č. 55, s. 244–262.

FIELDING, Roy Thomas, 2000. *Architectural styles and the design of network-based software architectures*. Doctoral dissertation. University of California, Irvine.

FOUNDATION, OPC, 2022. *OPC Unified Architecture Specification – Part 1: Overview and Concepts*. Dostupné tiež z: https://opcfoundation.org. Accessed: 2025-05-17.

FOUNTAS, N.A; HATZIARGYRIOU, N.D; VALAVANIS, K.P, 1999. A novel framework for the process control of the restoration of electrical industrial systems. *Electric Power Systems Research*. Roč. 50, č. 3, s. 163–167. ISSN 0378-7796. Dostupné z DOI: https://doi.org/10.1016/S0378-7796(98)00135-7.

GAMES, Real, 2023. *Factory I/O Documentation*. Dostupné tiež z: https://docs.factoryio.com/. Accessed: 2025-05-17.

GANSNER, Emden R.; NORTH, Stephen C., 2000. An open graph visualization system and its applications to software engineering. *Software: Practice and Experience*. Roč. 30, č. 11, s. 1203–1233. Dostupné z DOI: https://doi.org/10.1002/1097-024X(200009)30:11<1203::AID-SPE338>3.0.CO;2-N.

HEE, Kees van; AALST, Wil M. P. van der, 2002. *Workflow Management: Models, Methods, and Systems*. Cambridge, MA: MIT Press. ISBN 0-262-01189-1.

HEJLSBERG, Anders; WILTAMUTH, Scott; GOLDE, Peter, 2003. *The CSharp Programming Language*. Addison-Wesley.

HUBER, Peter; JENSEN, Kurt; SHAPIRO, Ronald M., 1991. Hierarchies in Coloured Petri Nets. In: ROZENBERG, Grzegorz (ed.). *Advances in Petri Nets 1991*. Springer. Zv. 483, s. 313–341. Lecture Notes in Computer Science. Dostupné z DOI: 10.1007/BFb0055866.

CHODOROW, Kristina, 2013. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc.

ISO/IEC, 2004. *Petri Net Markup Language (PNML)*. Tech. spr., ISO/IEC 15909. International Organization for Standardization.

JENSEN, Kurt; KRISTENSEN, Lars, 2009. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. ISBN 978-3-642-00283-0. Dostupné z DOI: 10.1007/b95112.

JONES, M. B.; BRADLEY, J.; SAKIMURA, N., 2015. *JSON Web Token (JWT)*. RFC, 7519. Internet Engineering Task Force (IETF). Dostupné tiež z: https://datatracker.ietf.org/doc/html/rfc7519. Accessed: 2025-05-17.

JUHÁS, Gabriel; KOVÁČIK, Tomáš; KOVÁŘ, Jan; KRANEC, Martin; PETROVIČ, Ľubomír, 2021. Netgrif Application Engine. Roč. 2973. Dostupné tiež z: https://netgrif.com/wp-content/uploads/2021/09/BPM21___NETGRIF_application_engine___Camera_ready.pdf.

MEJÍA, Gonzalo; NIÑO, Karen; MONTOYA, Carlos; SÁNCHEZ, María Angélica; PALACIOS, Jorge; AMODEO, Lionel, 2016. A Petri Net-based framework for realistic project management and scheduling: An application in animation and videogames. *Computers & Operations Research*. Roč. 66, s. 190–198. ISSN 0305-0548. Dostupné z DOI: https://doi.org/10.1016/j.cor.2015.08.011.

MICROSOFT, 2023. *Introduction to .NET*. Dostupné tiež z: https://learn.microsoft.com/en-us/dotnet/fundamentals/. Accessed: 2025-05-17.

MURATA, T., 1989. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*. Roč. 77, č. 4, s. 541–580. Dostupné z DOI: 10.1109/5.24143.

OTWELL, Taylor, 2013. *Laravel: From Apprentice To Artisan*. Leanpub.

PETRI, C. A., 1962. *Kommunikation mit Automaten*. Bonn: Institut für Instrumentelle Mathematik. Dostupné tiež z: http://edoc.sub.uni-hamburg.de/informatik/volltexte/2011/160/. Schriften des IIM Nr. 2.

PETRIFLOW PROJECT, 2024. *Netgrif*. Dostupné tiež z: ttps://github.com/netgrif/petriflow. Accessed: 2025-05-17.

PEZOA, Felipe; REUTTER, Juan L.; SUAREZ, Fernando; UGARTE, Martín; VRGOČ, Domagoj, 2016. Foundations of JSON Schema. In: *Proceedings of the 25th International Conference on World Wide Web*. Montréal, Québec, Canada: International World Wide Web Conferences Steering Committee, s. 263–273. WWW '16. ISBN 9781450341431. Dostupné z DOI: 10.1145/2872427.2883029.

RAMCHANDANI, Chander, 1973. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets*. Ph.D. thesis. Massachusetts Institute of Technology.

REISIG, W., 2013. *Understanding Petri nets. Modeling techniques, analysis methods, case studies. Translated from the German by the author*. ISBN 978-3-642-33277-7. Dostupné z DOI: 10.1007/978-3-642-33278-4.

REISIG, Wolfgang, 1985. *Petri Nets: An Introduction*. Berlin, Heidelberg: Springer-Verlag.

TAYLOR, Richard N.; MEDVIDOVIC, Nenad; DASHOFY, Eric M., 2009. *Software Architecture: Foundations, Theory, and Practice*. Wiley. ISBN 978-0-470-16774-8.

WALMSLEY, Priscilla, 2001. *Definitive XML Schema*. USA: Prentice Hall PTR. ISBN 0130655678.