VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Nástroj pro editaci a analýzu P/T Petriho sítí

A Tool for Editing and Analyzing P/T Petri nets

2016 Tomáš Blahut

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Tomáš Blahut**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Nástroj pro editaci a analýzu P/T Petriho sítí
A Tool for Editing and Analyzing P/T Petri Nets

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem této práce je vytvořit program, který umožní vytvořit, editovat a dále pak analyzovat P/T Petriho síť. Analýza Petriho sítí bude zaměřena na základní vlastnosti jako jsou živost, bezpečnost a reverzibilnost.

Dílčí body práce budou:
1. Seznamte se a vytvořte přehled dostupných nástrojů pro editaci a analýzu PN.
2. Implementujte editační nástroj pro tvorbu P/T PN.
3. Implementujte metody pro analýzu P/T PN.
4. Vizualizujte výsledky (např. grafu dosažitelnosti).

Seznam doporučené odborné literatury:

[1] Jaroslav Markl: Petriho sítě I.
[2] Reisig, W. (2012). Petri nets: an introduction (Vol. 4). Springer Science & Business Media.
[3] Tantau, T. (2013, January). Graph drawing in TikZ. In Graph Drawing (pp. 517-528). Springer Berlin Heidelberg.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Mgr. Pavla Dráždilová, Ph.D.**

Datum zadání: 01.09.2015
Datum odevzdání: 29.04.2016

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 29. dubna 2016

Tomáš Lelák

Rád bych na tomto místě poděkoval vedoucí své diplomové práce paní Mgr. Pavle Dráždilové, Ph.D. za udělené rady, vstřícný přístup a trpělivost. Čas, který mi vždy ochotně věnovala, mě byl velkou oporou. Velmi si také cením poskytnuté zpětné vazby a celkové kontroly mých dosažených výsledků.

Abstrakt

V této práci popisujeme vyvinutý nástroj pro editaci a analýzu P/T Petriho sítí. Systém se skládá ze dvou hlavních součástí: webového grafického editoru a analytického serveru. V úvodu práce stručně probíráme základy teorie Petriho sítí. Dále uvádíme seznam dostupných nástrojů pro práci s Petriho sítěmi společně s jejich popisem. V následujících kapitolách se věnujeme celkové architektuře vytvořeného systému a detailům jeho jednotlivých komponent. Na závěr uvádíme implementované analytické metody, mezi které patří například analýza stavového prostoru, detekce pastí a zámků nebo výpočet P a T invariantů. V rámci metody analýzy stavového prostoru na příkladech demonstrujeme vlastní způsob generování stromu pokrytí a uvádíme, jak jej lze využít při analýze neomezených Petriho sítí.

Klíčová slova: diplomová práce, Petriho sítě, grafický editor, stavový prostor, pokrývání značení

Abstract

The purpose of this master thesis is to describe a tool developed for editing and analyzing of P/T Petri nets. System consists of two main components: web graphical editor and analysis server. We briefly cover the fundamentals of Petri nets theory in the introduction. Next we mention a list of existing tools along with their description. In the following chapters the overall architecture of developed system is considered and details of its components are described. At the end we mention implemented analysis methods, which include for example state space analysis, traps and cotraps detection or also calculation of P and T invariants. In the context of the state space analysis method we demonstrate on examples our own means for constructing coverability tree and show how it can be used to decide some properties of unbounded Petri nets.

Key Words: master thesis, Petri nets, graphical editor, state space, marking covering

Obsah

Seznam použitých zkratek a symbolů

P/T	–	Place/Transition
C/E	–	Condition/Event
TGI	–	Theoretische Grundlagen der Informatik
CPN	–	Colored Petri net
PNML	–	Petri net Markup Language
AIS	–	Architecture of Information Systems
SML/NJ	–	Standard ML of New Jersey
GNU GPL	–	GNU General Public License
IRT	–	Institut für Regelungstechnik
RWTH	–	Rheinisch-Westfälische Technische Hochschule
DOS	–	Disk Operating System
MCKIT	–	Model Checking Kit
INA	–	Integrated Net Analyzer
PIPE	–	Platform Independent Petri net Editor
PNG	–	Portable Network Graphics
XML	–	Extensible Markup Language
JPG	–	Joint Photographic Graphics
DHBW	–	Duale Hochschule Baden-Württemberg
GNU LGPL	–	GNU Lesser General Public License
JRE	–	Java Runtime Enviroment
IOPT	–	Input-Output Place-Transition
CRUD	–	Create Read Update Delete
HTML	–	Hyper Text Markup Language
SVG	–	Scalable Vector Graphics
DOM	–	Document Object Model
CSS	–	Cascading Style Sheets
API	–	Application Programming Interface
BLOB	–	Binary Large Object
SASS	–	Syntactically Awesome Style Sheets
REST	–	Representational State Transfer
URI	–	Uniform Resource Identifier
HTTP	–	Hypertext Transfer Protocol
POJO	–	Plain Old Java Object
JSON	–	JavaScript Object Notation
JDK	–	Java Development Kit
CNF	–	Conjunctive Normal Form

Seznam obrázků

1 Úvod

Petriho sítě jsou od okamžiku jejich představení neocenitelným nástrojem pro zkoumaní paralelních a distribuovaných systému. Náročnost aplikace teorie Petriho sítí ovšem roste spolu se složitostí zkoumaného systému. V praxi jsou za tímto účelem proto využívány podpůrné nástroje. Jelikož je teorie Petriho sítí velice obsáhlá, jednotlivé nástroje se často liší v množině jejich dovedností a v oblasti, na kterou se zaměřují. Cílem této diplomové práce byla tvorba nástroje pro editaci a analýzu jedné ze tříd Petriho sítí, konkrétně Place/Transition Petriho sítí.

V úvodní části práce se věnujeme základům teorie P/T Petriho sítí. Zmiňujeme přesnou definici jejich struktury, popisujeme pravidla jejich chování a způsob jakým mohou být graficky znázorněny. Kapitola 2 dále uvádí základní pojmy z oblasti analýzy Petriho sítí a seznam jejich nejdůležitějších vlastností. Z této teoretické báze jsme vycházeli při implementaci zde popisovaného systému.

Nástroje pro práci s Petriho sítěmi lze rozdělit do dvou hlavních kategorií. V první skupině nalezneme grafické editory využívané pro tvorbu modelů Petriho sítí. Do kategorie druhé spadají analytické aplikace, které tyto modely zkoumají. Takovýchto nástrojů existuje veliké množství, většina z nich je ovšem zastaralá. Kapitola 3 uvádí přehled vybraných aplikací pro práci s Petriho sítěmi společně s jejich stručným popisem.

Časté jsou také nástroje obsahující jak grafický editor, tak i analytickou část. Kapitola 4 představuje celkovou architekturu našeho řešení a zdůvodňuje proč jsme se rozhodli zvolit přístup, kdy je každá součást systému implementována jako samostatná komponenta. Komponenty popisujeme tak, aby byla umožněna jejich snadná rozšířitelnost. Z tohoto důvodu jsme zvolili také vyšší úroveň detailnosti.

Způsob realizace a výčet dovedností grafického editoru jsou obsaženy v kapitole 4.2. Analytické části systému je pak věnována kapitola 4.3. Uvádíme zde jednotlivé kroky procesu analýzy Petriho sítí a základní principy použité při realizaci komunikačního rozhraní mezi oběma součástmi systému.

Pro implementované analytické metody jsme vyčlenili samostatnou kapitolu 5. Součástí popisu každé z nich je její teoretický základ, konkrétní implementace a způsob využití při rozhodování vlastností Petriho sítí. U metody zabývající se stavovým prostorem sítě dále prezentujeme naše rozšíření základní techniky pro konstrukci jeho grafové reprezentace. Na příkladech demonstrujeme přínosy této úpravy při detekci živosti a reverzibilnosti u neomezených Petriho siti.

2 Petriho sítě

Ačkoliv je pojem Petriho sítí v různé literatuře vysvětlován mírně odlišně, hlavní myšlenka zůstává stejná. Neformálně lze Petriho sítě popsat jako nástroj pro modelování a zkoumání vlastností a chování systémů, které lze charakterizovat jako souběžné, distribuované nebo také paralelní.

Teorie Petriho sítí umožňuje zkoumané systémy popsat pomocí jejich grafické a matematické reprezentace. Grafický model slouží pro vizuální komunikaci a znázornění procesů v systému, podobně jako tomu je například u diagramů aktivit, stavových nebo vývojových diagramů. Pomocí matematické reprezentace lze poté analyzovat vlastnosti modelu a simulovat jeho chování. Informace získané analýzou Petriho sítě jsou posléze interpretovány v kontextu modelovaného systému a využity k jeho vyhodnocení a případnému vylepšování.

Poprvé byly Petriho sítě popsány v dizertační práci Carla Adama Petriho v roce 1962 na Matematicko-fyzikální fakultě Technické University v Darmstadu v Německu. Od té doby jsou předmětem probíhajícího výzkumu. Jazyk Petriho sítí je rozšířením teorie konečných automatů o schopnost vyjádření souběžně probíhajících událostí. Brzy po svém představení byl považován za jeden z nejlepších jazyků pro popis a analýzu synchronizace, komunikace a sdílení prostředků mezi souběžnými procesy.

Petriho sítě mohou být aplikovány na libovolné systémy, které lze graficky reprezentovat pomocí vývojových diagramů a které potřebují vyjádřit paralelní a/nebo souběžné procesy. Jednou z oblastí, ve kterých jsou úspěšně využívány, jsou komunikační protokoly [20, 26].

2.1 P/T Petriho sítě

Postupem času byly Petriho sítě rozšiřovány tak, aby jejich modelovací možnosti vyhovovaly specifičtějším požadavkům. Definováno bylo několik základních typů sítí, které se vzájemně liší jejich expresivní silou a sadou dostupných konstrukcí pro jejich modelování. Výčet typů Petriho sítí byl převzat z [18].

- Condition/Event Petriho sítě - jsou nejjednodušším typem Petriho sítí.
- Place/Transition Petriho sítě - jedná se o rozšíření C/E sítí, které zavádí kapacitu míst a váhu hran. Mezi varianty P/T sítí patří P/T sítě s inhibičními hranami a P/T sítě s prioritami.
- Časované Petriho sítě - obohacují Petriho sítě o pojem času a uvažují změny stavu systému jako události, které nemusí proběhnout okamžitě.
- Barevné Petriho sítě - zavádí třídy (barvy) tokenů, podmínky přechodů a hranové výrazy.
- Hierarchické Petriho sítě - umožňují rozdělit modelovaný systém do znovupoužitelných komponent. Ty v sobě ukrývají jejich vnitřní strukturu, činí tak celkový systém přehlednějším a usnadňují jeho údržbu.

- Objektové Petriho sítě - v sobě kombinují vlastnosti barevných a hierarchických Petriho sítí.

Z výše uvedených skupin jsou nejlépe prozkoumanou třídou Place/Transition Petriho sítě. Tato třída je hlavním tématem práce a byl pro ni speciálně navržen zde popisovaný nástroj. V této kapitole zavedeme základní informační bázi. Aplikace z těchto definic a pravidel vychází jak při modelování systémů, tak při jejich analýze. Není-li uvedeno jinak, jsou všechny definice týkající se Petriho sítí převzaty z [18].

2.2 Struktura Petriho sítě

Podle [27] strukturu Petriho sítě tvoří několik odlišných prvků. Prvním z nich jsou místa, která představují pasivní část sítě a která v sobě nesou informaci o jejím aktuálním stavu. Druhým prvkem jsou přechody. Na rozdíl od míst reprezentují přechody aktivní část Petriho sítě. Mohou generovat, konzumovat nebo přenášet objekty (v případě P/T Petriho sítí tokeny) a měnit tak její stav. Za účelem propojení míst a přechodů jsou využívány orientované hrany. Ty nepředstavují komponentu sítě, ale reprezentují relaci propojení mezi místy a přechody.

Každé místo v P/T Petriho síti má stanovenu svou kapacitu, která udává maximální počet tokenů, které se v tomto místě mohou najednou vyskytovat. Hrany jsou ohodnoceny přirozenými čísly, jež udávají jejich násobnost. Není-li kapacita míst ani násobnost hran uvedena, předpokládáme hodnoty ∞ a 1 respektive.

Definice 1 *Struktura Petriho sítě je čtveřice* $\langle P, T, I, O \rangle$*, kde*

- P *je konečná množina míst*
- T *je konečná množina přechodů*
- I *a* O *jsou zobrazení* $T \rightarrow P_{MS}$*, kde* I *je vstupní funkce a* O *funkce výstupní.* P_{MS} *je množina všech multimnožin nad množinou* P*.*

V literatuře bývá množina hran sítě popsána namísto uvedených funkcí I a O také relací $F \subseteq (P \times T) \cup (T \times P)$ a zobrazením $W : F \rightarrow N = \{0, 1, 2, ...\}$, které udává ohodnocení hran Petriho sítě. Struktura Petriho sítě je poté dána čtveřicí ⟨P, T, F, W⟩. V rámci této práce se budeme držet výkladu dle definice 1. Násobnost hran je zde vyjádřena pomocí multimnožin.

Multimnožinu lze formálně definovat jako zobrazení $a : U \rightarrow N$, kde $a(u) \in N$ udává počet výskytů prvku u v universu U. Multimnožiny jsou tedy jakýmsi zobecněním množin, které zachycuje nejen informaci o tom, které prvky do množiny patří, ale i počet výskytů jednoho prvku v multilmnožině.

2.3 Značení Petriho sítí

Chování mnoha systémů lze popsat pomocí množiny jejich stavů a změn jejich aktuálního stavu. Stav Petriho sítě je dán rozmístěním tokenů v jejích místech. Takové rozložení se v teorii Petriho

sítí nazývá značení a je formálně definováno jako zobrazení typu $P \to N$ z množiny míst do množiny přirozených čísel. Skutečný počet tokenů v místě p ve značení M poté udává zápis $M(p)$. Značení M je možné zapsat jako vektor $M = (m_1, m_2, ..., m_n)$, kde $n = |P|$ a m_i udává počet tokenů v místě p_i pro $i = 1, ..., n$. Hodnota jednotlivých složek vektoru M je poté dána vztahem $m_i = M(p_i)$.

Na rozdíl od C/E Petriho sítí mohou místa P/T Petriho sítě obsahovat libovolný nezáporný počet tokenů. V porovnání s barevnými Petriho sítěmi nejsou u P/T Petriho sítí tokeny nijak vzájemně rozlišovány. Využívány jsou pouze abstraktní černé tokeny, které mohou reprezentovat skutečnost splnění jistých podmínek, které jsou modelovány jako místa sítě, ale také zastupovat entity ze skutečného světa. Jako příklad zde uvedeme místo v blíže neurčené Petriho sítí, které znázorňuje vlakové nástupiště. Počet tokenů v takovém místě může vyjadřovat počet lidí čekajících na vlak. U jednotlivců nás nezajímají jejich specifické informace jako je jméno, věk anebo adresa. Důležitá je pouze informace o jejich počtu.

Definice 2 *Systém Petriho sítě je pětice* $\langle P, T, I, O, M_0 \rangle$ *kde*

- $\langle P,\ T,\ I,\ O \rangle$ *je struktura Petriho sítě popsaná v definici 1*
- M_0 *je počáteční značení sítě*

Značení M_0 reprezentuje původní stav sítě. Pokud počáteční značení přiřazuje každému místu sítě nulový počet tokenů tedy $(\forall p \in P)[M_0(p) = 0]$, pak takový systém Petriho sítě reprezentuje pouze její strukturu.

2.4 Dynamika Petriho sítí

Abychom mohli simulovat chování Petriho sítě, je nezbytná existence nějakého mechanismu, který umožňuje provádět změny značení. V teorii Petriho sítí jsou změny značení uskutečňovány pomocí provádění jejích přechodů. Dynamika Petriho sítí popisuje pravidla, která rozhodují o tom, které přechody je možné v daném značení sítě provést a v jakém značení se po provedení konkrétního přechodu bude síť nacházet. Dříve než popíšeme pravidla změny značení sítě, uvedeme zde několik pojmů, které budeme nadále využívat:

- $I(t, p)$ udává násobnost hrany vedoucí z místa p do přechodu t,
- $O(t, p)$ udává násobnost hrany vedoucí z přechodu t do místa p,
- ${}^\bullet t$ je množina vstupních míst přechodu t,
- $t^\bullet$ je množina výstupních míst přechodu t.

2.4.1 Pravidlo proveditelnosti přechodu

Přechod může být proveden pouze v případě, kdy je v daném značení proveditelný. Přechod je považován za proveditelný, pokud každé z jeho vstupních míst obsahuje alespoň takový počet tokenů, jaké je ohodnocení hrany vedoucí ze vstupního místa k přechodu.

Definice 3 *Přechod t je ve značení M proveditelný, pokud platí:* $(\forall p \in {}^\bullet t)[M(p) \geq I(t,p)]$

V [20] jsou přechody, které nemají žádné vstupní místo (${}^\bullet t = \emptyset$), označovány jako zdrojové (source) a přechody bez míst výstupních ($t^\bullet = \emptyset$) jako konzumující (sink). Zdrojové přechody jsou vždy bezpodmínečně proveditelné. Definice proveditelného přechodu 3 neuvažuje případnou kapacitu výstupních míst přechodů. Některá místa sítě by po provedení přechodu mohla ve výsledném značení obsahovat počet tokenů převyšující jejich kapacitu. Síť by se tak nenacházela v korektním stavu.

V rámci této práce jsme si dovolili provést modifikaci definice Place/Transition Petriho sítí takovou, kdy je kapacita všech míst v sítí rovna ∞. Jak bylo poznamenáno v [18], kapacita míst nezvyšuje expresivní sílu P/T Petriho sítí. Stejně tak bylo ukázáno, že každou P/T Petriho sít lze převést na síť bez omezení kapacity míst, přičemž takto upravená sít modeluje stejný problém. Pravidlo proveditelnosti přechodů se dle [20] proto dělí na dvě varianty: striktní a slabé. Striktní verze pravidla přidává oproti svému slabému protějšku navíc jednu podmínku, která musí být splněna, aby byl přechod proveditelný.

Definice 4 *Přechod t je ve značení M striktně proveditelný, pokud platí:*
$(\forall p \in {}^\bullet t)[M(p) \geq I(t,p)] \wedge (\forall p \in t^\bullet)[M'(p) \leq K(p)]$*, kde $K(p)$ je funkce*
$P \to N = \{0, 1, ..., \infty\}$*, která stanovuje kapacitu místa p a $M \xrightarrow{t} M'$.*

Dodatečná podmínka striktní verze pravidla proveditelnosti přechodů říká, že je přechod proveditelný pouze tehdy, pokud po jeho provedení není počet tokenů v žádném výstupním místě přechodu vyšší než jeho kapacita.

2.4.2 Pravidlo provedení přechodu

Přechod je proveden odebráním příslušného počtu tokenů ze všech jeho vstupních míst a následným přidáním tokenů do míst výstupních. Zkonzumovaný a vygenerovaný počet tokenů je stanoven váhou hran vedoucích do a z provedeného přechodu. Jelikož mohou být provedeny vždy pouze ty přechody, které jsou v daném značení proveditelné, nikdy nemůže nastat situace, kdy by v nově dosaženém značení obsahovala některá místa sítě záporný počet tokenů.

Provedením přechodu t ve značení M je dosaženo nové značení M'. V různé literatuře se lze setkat s několika způsoby zápisu této skutečnosti například $M[t\rangle M'$ anebo $M \xrightarrow{t} M'$. Existuje-li v určitém značení více proveditelných přechodů, je prováděný přechod zvolen nedeterministicky.

Definice 5 *Značení M' je ze značení M, po provedení přechodu t, vypočteno tímto vztahem:*
$(\forall p \in P)[M'(p) = M(p) - I(t,p) + O(t,p)]$

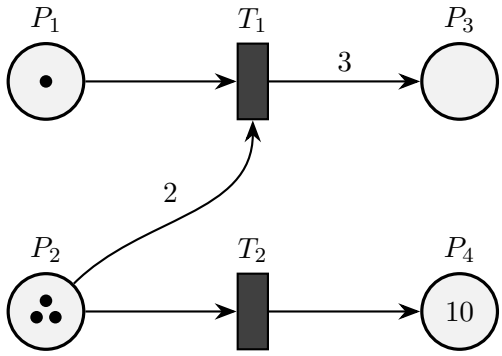
Provedením dříve zmíněných zdrojových přechodů jsou do jeho výstupních míst generovány tokeny. Stejně tak jsou provedením konzumujícího přechodu pouze čerpány tokeny z jeho vstupních míst a jsou tak navždy odebrány z celé Petriho sítě.

2.5 Modelování Petriho sítí

V mnoha výzkumných oborech není objekt zkoumán přímo, ale pomocí jeho modelu. Cílem modelu je zachytit vlastnosti zkoumaného objektu nebo systému, které jsou považovány za důležité. Jeho studium pak může přinést nové poznatky o zkoumaném objektu, aniž bychom byli vystaveni rizikům nebo nákladům spojeným s jeho přímou manipulací. Aplikace teorie Petriho sítí je v praxi prováděna pomocí modelování zkoumaného systému [26].

Model Petriho sítě je graficky realizován jako orientovaný bipartitní graf, jehož množinu vrcholů lze rozložit na dvě disjunktní skupiny. První skupina představuje místa sítě, která jsou vykreslována v podobě kruhů nebo elips. Přechody sítě spadají do druhé skupiny vrcholů a jsou zobrazovány jako obdélníky nebo čtverce. Hrany mohou vést pouze z míst do přechodů a naopak. Ohodnocení hrany je dáno její váhou. Popisek hran je v grafu Petriho sítě možné zanedbat v případech, kdy je jejich váha rovna jedné.

Tokeny jsou v grafu sítě zobrazeny pomocí černých teček situovaných uvnitř vrcholů míst. Je-li počet tokenů příliš vysoký na to, aby mohl být zachycen tímto způsobem, je jejich množství zobrazeno pomocí číslic.



Obrázek 1: Ukázka grafické reprezentace Petriho sítě

2.6 Analýza Petriho sítí

Hlavní síla Petriho sítí spočívá v možnostech analýzy jejího modelu. Samotný model je nepochybně užitečný pro vizualizaci zkoumaného systému. Sám o sobě ovšem nevypovídá mnoho o jeho chování. Cílem analýzy je zodpovědět otázky, které mohou být na síť kladeny. Otázkami zde rozumíme vlastnosti Petriho sítí, které dělíme na dvě skupiny. V první nalezneme ty, které jsou závislé na počátečním značení sítě, označovány také jako vlastnosti chování. Do druhé skupiny patří poté vlastnosti, které závisí pouze na struktuře zkoumané sítě.

Za účelem rozhodnování těchto vlastností bylo vyvinuto několik různých analytických metod. Popisu metod, které v sobě vyvinutý systém zahrnuje, a detailům jejich implementace je věnována kapitola 5. Dříve než uvedeme vybrané vlastností Petriho sítí, rozšíříme teoretickou základnu o několik pojmů, které jsou nezbytné k jejich samotnému popisu. Uvedené definice byly převzaty z [18].

2.6.1 Množina dosažitelnosti

V kapitole 2.4.2 byl popsán mechanismus provedení přechodu, který nyní zobecníme na pravidlo provedení posloupnosti přechodů.

Definice 6 *Mějme posloupnost přechodů* $\sigma = t_{(1)}, t_{(2)}, t_{(3)}, ..., t_{(n)}$. *Značení* M' *je dosažitelné ze značení* M *právě tehdy, jestliže existuje posloupnost značení* $M = M_{(1)}, M_{(2)}, M_{(3)}, ..., M_{(n+1)} = M'$ *taková, že platí* $M_{(i)} \xrightarrow{t_i} M_{(i+1)}$ *pro i = 1, 2, 3,.., n.*

Stejně jako u bezprostředně dosažitelného značení, je možné situaci, kdy je ze značení M dosažitelné značení M' pomocí sekvence přechodů σ, zapsat několika způsoby: například $M[\sigma\rangle M'$ nebo $M \xrightarrow{\sigma} M'$. Připomeneme, že stav Petriho sítě je určen distribucí tokenů v jejích místech neboli značením.

Definice 7 *Množinu všech dosažitelných značení ze značení* M *budeme značit jako* $RS(M) = \{M' : (\exists \sigma \in T^*)[M \xrightarrow{\sigma} M']\}$, *kde* T^* *představuje množinu všech slov nad abecedou* T, *jejíž množina symbolů je rovna množině přechodů dané sítě.*

Uvažujeme-li počáteční značení PN-systému M_0, pak $RS(M_0)$ reprezentuje množinu všech dosažitelných značení Petriho sítě. Pro úplnost zde uvedeme také induktivní definici množiny dosažitelnosti PN-systému, kterou lze využít při konstrukci grafické reprezentace jeho stavového prostoru.

Definice 8 *Množina dosažitelnosti PN-systému* $\langle P, T, I, O, M_0\rangle$ *je množina* $RS(M_0)$ *kde:*

- $M_0 \in RS(M_0)$
- $M \in RS(M_0) \wedge (\exists t \in T)[M \xrightarrow{t} M'] \implies M' \in RS(M_0)$

2.7 Vlastnosti Petriho sítí

V této části práce popisujeme vlastnosti Petriho sítí, které je aplikace schopna na jejich modelech ověřovat. Do kategorie vlastností chování spadají omezenost, bezpečnost, živost a reverzibilnost. Dvouprvkovou množinu strukturálních vlastností poté tvoří konzervativnost a repetičnost. Výsledky analýzy Petriho sítě jsou vztaženy k jejímu modelu. Jelikož může stejný model Petriho sítě reprezentovat několik různých systémů, je na jejím designérovi, aby výsledky analýzy správně interpretoval na konkrétní systém, který síť modeluje.

2.7.1 Omezenost

Místo v Petriho sítí se nazývá k-omezené, jestliže v žádném dosažitelném značení jeho počet tokenů nepřesahuje číslo k. Číslo k se může pro každé místo sítě lišit. Pokud je ovšem místo k-omezené je poté také k'-omezené pro všechna $k' \geq k$. Jelikož je počet míst v Petriho sítí konečný, lze výslednou hodnotu k určit jako maximum z množiny hranic všech míst sítě.

Definice 9 *Systém Petriho sítě je k-omezený, je-li každé jeho místo k-omezené, tj. platí-li* $(\forall p \in P)(\forall M \in RS(M_0))[M(p) \leq k]$

V určitých případech, nás zajímá pouze skutečnost, zda je daná síť omezená, ale konkrétní hodnotu hranice k nepotřebujeme. Místo se poté nazývá omezené, pokud je k-omezené pro libovolné $k \neq \infty$. Stejně tak se Petriho síť nazývá omezená, je-li omezené každé její místo.

Omezenost je důležitou vlastností například v situacích, kdy Petriho síť modeluje skutečný hardwarový systém. Místa v takové sítí jsou často využívána pro reprezentaci bufferů, registrů nebo čítačů. Pokud je síť omezená, jsou vyloučeny situace, kdy by počet prvků v takových zásobnících překročil jejich kapacitu. Systém, který je popsán neomezenou Petriho sítí, by tak nemohl být realizován hardwarově [20, 26].

2.7.2 Bezpečnost

Bezpečnost je zvláštním případem omezenosti. Petriho síť je bezpečná pokud je 1-omezená. Znamená to tedy, že všechna místa v dané síti obsahují v každém dosažitelném značení pouze jeden anebo žádný token. Místa takové sítě poté mohou představovat podmínky, které jsou buďto splněny (token je v místě přítomen), nebo splněny nejsou (token v místě chybí). Přechod, který realizuje nějakou událost, lze provést pouze tehdy, jsou-li splněny všechny jeho podmínky (každé jeho vstupní místo obsahuje token). Na stejném principu fungují Condition/Event Petriho sítě. Podobně jako u omezenosti budou nároky na bezpečnost kladeny především u hardwarových systémů.

2.7.3 Reverzibilnost

Petriho síť se nazývá reverzibilní, jestliže je její počáteční značení dosažitelné z každého dosažitelného značení. V důsledku to znamená, že rozmístění tokenů v dané Petriho síti je kdykoliv možné úvést do původního stavu. Dle [20] není v mnoha případech nezbytné, aby značení, do kterého se lze kdykoliv navrátit, bylo právě to počáteční, ale postačuje pouze existence takového značení. Definujeme tedy takzvané vždy dosažitelné značení.

Definice 10 *Značení M Petriho sítě* $\langle P, T, I, O, M_0 \rangle$ *se nazývá vždy dosažitelným, platí-li:* $(\forall M' \in RS(M))[M \in RS(M')]$

S ohledem na definici 10 je vlastnost reverzibility Petriho sítě formulována následovně: $(\forall M \in RS(M_0))[M_0 \in RS(M)]$.

2.7.4 Absence uzamčení

Uzamčení PN-systému představuje situace, kdy se síť nachází ve značení, ve kterém není proveditelný žádný přechod. Jak bylo poukázáno v [18], je existence uzamčení v určitých případech žádoucí a v jiných ne. Pokud Petriho síť popisuje například nějaký algoritmus, musí svou exekuci ukončit po provedení konečného počtu kroků. Takový systém bude obsahovat nejméně jedno uzamčení.

Definice 11 *PN-systém* $\langle P, T, I, O, M_0 \rangle$ *se nazývá systémem bez uzamčení, jestliže z počátečního značení* M_0 *není dosažitelné žádné značení, ve kterém není proveditelný žádný přechod, tj. platí-li:* $(\forall M \in RS(M_0))(\exists t \in T)[t \in E(M)]$*, kde* $E(M)$ *udává množinu proveditelných přechodu ve značení* M*.*

Jako příklad systému, kde je existence uzamčení nežádoucí, zde uvedeme jeden z klasických problémů řízení souběžného přístupu ke sdíleným zdrojům a sice problém večeřících filozofů. U kulatého stolu sedí pět filozofů a každý z nich má před sebou večeři. Mezi jednotlivými dvojicemi filozofů leží na stole jeden kus příboru. Všichni filozofové buďto přemýšlí nebo večeří a obě tyto činnosti pravidelně střídají. Příbor může v danou chvíli držet pouze jeden filozof. Zvednout jej může, kdykoliv leží na stole. K tomu, aby mohl filozof začít večeřet, potřebuje držet příbor v obou svých rukách. Uzamčení takového systému nastává v okamžiku, kdy každý z filozofů zvedne příbor na své levé nebo pravé straně stolu. Všichni v tomto okamžiku čekají na druhý příbor, ale žádný z nich neuvolní ten, který už drží. Tato situace vede k takzvanému vyhladovění.

2.7.5 Živost

Neformálně lze živost petriho sítě popsat jako skutečnost, kdy bez ohledu na to, jaké značení bylo z počátečního značení již dosaženo, bude možné provést libovolný přechod sítě nezávisle na zvolené posloupnosti přechodů, kterou bude síť dále rozvíjena. Jinými slovy žádný přechod nikdy neztrácí možnost být znovu proveden.

Definice 12 *Přechod* t *je v daném PN systému živý, platí-li:*
$(\forall M \in RS(M_0))(\exists M' \in RS(M))[t \in E(M')]$

PN systém je živý, jsou-li všechny jeho přechody živé, tj. platí-li:
$(\forall t \in T)(\forall M \in RS(M_0))(\exists M' \in RS(M))[t \in E(M')]$

Jak bylo zmíněno v [20], živost je pro mnoho systémů velice dobrá vlastnost, nicméně náročnost její analýzy pro rozsáhlé systémy přesahuje praktičnost. Z toho důvodu bylo zavedeno několik různých úrovní živosti přechodu:

- L0 - přechod t nemůže být nikdy proveden,

- L1 - přechod t může být proveden alespoň jednou (je potencionálně proveditelný), tj platí $(\exists M \in RS(M_0))[t \in E(M)]$,
- L2 - přechod t patří do této úrovně, jestliže pro libovolné číslo $n \in N$ existuje sekvence přechodů taková, ve které se přechod t vyskytuje alespoň n-krát,
- L3 - přechod t spadá do této úrovně, pokud se v některé sekvenci přechodů vyskytuje nekonečně mnohokrát,
- L4 - přechod t je živý dle definice 12.

Triviálním případem neživého PN-systému je libovolná struktura s počátečním značením, které každému místu sítě přiřazuje nulový počet tokenů. Živost Petriho sítě ve své podstatě garantuje absenci uzamčení. Na základě neživosti PN systému ovšem nelze usuzovat o existenci jeho uzamčení. Stejně tak absence uzamčení nerozhoduje jeho živost.

2.7.6 Konzervativnost

Konzervativnost patří do skupiny strukturálních vlastností Petriho sítí. Síť, která je konzervativní, nezískává ani neztrácí tokeny, ale pouze je přesouvá mezi svými místy. Důležitost konzervativnosti u konkrétních systémů demonstruje příklad z [26]. V síti existuje místo, jehož tokeny reprezentují dostupné zdroje v počítačovém systému (například tiskárny). Žádost o alokaci zdroje provádí výstupní přechod z tohoto místa a uvolnění zdroje zase přechod vstupní. Je žádoucí, aby počet tokenů reprezentující tyto zdroje zůstal konstantní v libovolném dosažitelném značení.

Petriho síť by tedy měla uchovávat zdroje, které modeluje. Mezi tokeny a zdroji, které představují, nemusí být vždy vztah jedna ku jedné. Jediný token může reprezentovat několik zdrojů najednou, které budou separovány provedením přechodu s více než jedním výstupním místem. Konzervativnost je proto definována vzhledem k váhovému vektoru, který každému místu sítě přiřazuje váhu jeho tokenů.

Definice 13 *Petriho síť $\langle P, T, I, O\rangle$ s počátečním značením M_0 je konzervativní vzhledem k váhovému vektoru $w = (w_1, w_2, w_3, ..., w_n)$, kde $n = |P|$ a $w_i \geq 0$ pro $i = 1, 2, ..., n$, pokud platí* $(\forall M \in RS(M_0))[\sum_{i=1}^{n} w_i \cdot M(p_i) = \sum_{i=1}^{n} w_i \cdot M_0(p_i)]$

Petriho síť se nazývá striktně konzervativní, má-li vektor w podobu $(1, 1, ..., 1)$. Z definice 13 vyplývá, že každá Petriho síť je konzervativní vzhledem k triviálnímu nulovému vektoru. Díky tomu je v praxi je konzervativnost uvažována pouze vzhledem k vektorům, pro jejichž složky platí $w_i > 0$. Váhové vektory w se rovněž nazývají P-invarianty. Jejich významem se budeme podrobněji zabývat v kapitole **??**.

2.7.7 Repetičnost

Další ze strukturálních vlastností Petriho síťí je repetičnost. Petriho síť je repetiční, existuje-li počáteční značení M_0 a sekvence přechodů σ spustitelná z a vedoucí zpět do M_0 taková, že se v ní každý přechod sítě vyskytuje alespoň jednou. Zmíněná sekvence přechodů bývá reprezentována charakteristickým (v literatuře také označováno jako Parikhovým) vektorem, který mapuje jednotlivé přechody sítě na počet jejich výskytů v sekvenci. Hodnotu pro konkrétní přechod určuje složka vektoru na stejném indexu, jaký má přechod přiřazen v seřazeném seznamu všech přechodů sítě.

Definice 14 *Systém Petriho sítě* $\langle P, T, I, O, M_0 \rangle$ *je repetiční, pokud platí* $(\exists \sigma \in T^*)[M_0 \xrightarrow{\sigma} M_0 \wedge \forall t \in T[w(t, \sigma) > 1]]$*, kde* w *je funkce udávající počet výskytů přechodu* t *v sekvenci* σ*.*

Podobně jako u konzervativnosti je síť striktně repetiční, pakliže je každá složka charakteristického vektoru rovna jedné. Repetičnost bývá v literatuře také někdy označována jako konzistence. Například Murata v [20] tyto pojmy vysvětluje dokonce odděleně a uvádí konzistenci jako zvláštní případ repetičnosti. Provedním sekvence přechodů σ se nezmění značení sítě, díky čemuž může být tato sekvence prováděna donekonečna. Vektoru, který takovou sekvenci popisuje se také říká T-invariant.

3 Přehled dostupných nástrojů

Nezbytnou součástí praktického užití teorie Petriho sítí jsou softwarové nástroje. Není neobvyklým úkazem, že si jednotlivé výzkumné skupiny samy vyvíjejí aplikace, které jim usnadňují modelování, simulace a analýzu oblastí, na které se zaměřují. V této části práce uvádíme přehled vybraných nástrojů, využívaných v oboru Petriho sítí spolu s jejich stručným popisem.

Jako nejlepší zdroj informací se při průzkumu existujících nástrojů ukázal webový projekt s názvem Petri Nets World. Jeho účelem je poskytnout širokou škálu online služeb pro mezinárodní komunitu zabývající se Petriho sítěmi. Mimo jiného v sobě tyto služby nesou informace o publikacích, vědeckých pracích a článcích, databázi softwarových balíků nebo například kalendář konferencí, sympózií a seminářů. Projekt je spravován skupinou TGI na německé Hamburgské univerzitě [12].

Databáze nástrojů pro práci s Petriho sítěmi spravovaná v rámci projektu Petri Nets World je poměrně obsáhlá. Přehledně uvádí informace o dovednostech nástroje, podporovaných typech Petriho sítí a platformách, na kterých jej lze provozovat. Většina záznamů je nicméně zastaralá a pouze malá část z nich je v praxi využitelná i v současnosti. Na seznam lze proto nahlížet spíše jako na historický archív. I přes tento fakt je v databázi možné nalézt novější a udržované aplikace. Hlavním kritériem pro výběr nástrojů uvedených v následujícím přehledu byla primárně jeho aktuálnost, dostupnost a použitelnost na moderních platformách. Aplikace jsou popisovány z pohledu jejich dovedností v oblastech modelování, simulace a analýzy Petriho sítí.

3.1 CPN Tools

CPN Tools je nástroj pro editaci, simulování a analýzu vysokoúrovňových Petriho sítí. Primárně tato aplikace slouží pro práci s barevnými Petriho sítěmi, ale dokáže pracovat také se základními třídami. Implementováno je rozhraní pro zakomponování modulů třetích stran a také například funkce exportu vytvořených modelů do souborů PNML formátu. Aplikace se skládá se ze dvou hlavních komponent: grafického editoru a simulátoru.

Grafický editor umožňuje manuální modelování a editaci Petriho sítí pomocí operací jako je vkládání, mazání, klonování nebo přesouvání jejich prvků. Tyto komponenty je možné logicky seskupovat, upravovat jejich popis a barvu a také jim přiřazovat funkční výrazy. Práce s editorem je realizována pomocí plovoucích palet a kruhového kontextového menu, jehož dostupné operace jsou závislé na oblasti, ze které byla tato nabídka vyvolána. Aplikace dokáže pracovat s více Petriho sítěmi najednou.

Simulováno může být chování časovaných i časově nezávislých modelů. V průběhu simulace nástroj poskytuje zpětnou vazbu například o aktuálním značení sítě, které může uživatel také upravovat. Samozřejmostí je podsvícení proveditelných přechodů. V případech, kdy během simulace došlo k chybě (například se síť provedením daného přechodu dostává do nekorektního stavu), je editor schopen zobrazit notifikace v problémovém místě modelu. Za účelem pozorování a řízení simulací existuje v CPN Tools mechanismus nazývaný monitor. Těch může být v jedné

síti přítomno více najednou a s jejich pomocí lze sbírat data, zastavit simulaci nebo definovat vlastní akce prováděné při splnění určitých podmínek. Simulaci je kromě monitorů možné řídit skrze příslušnou paletu nástrojů.

Aplikace dokáže generovat, vizualizovat a posléze analyzovat stavový prostor Petriho sítě. Vizualizaci stavového prostoru realizuje v podobě grafu dosažitelnosti. Výstupem analýzy jsou zprávy, které obsahují informace například o živosti nebo omezenosti dané sítě. Z dat sesbíraných pomocí výše zmíněných monitorů lze vytvořit širokou škálu statistik.

Původně byl tento nástroj vyvíjen skupinou CPN Group na Aarhuské univerzitě v letech 2000 až 2010. Hlavními architekty tohoto projektu jsou Kurt Jensen, Søren Christensen, Lars M. Kristensen, and Michael Westergaard. Od podzimu 2010 byl CPN Tools přesunut pod správu skupiny AIS na Eindhoven University of Technology v Nizozemí. Nejnovější stabilní verze vydaná v únoru 2015 nese číslo 4.0.1. Grafický editor je napsán v akademickém jazyce BETA a simulátor byl vytvořen pomocí SML/NJ.

CPN Tools funguje primárně pod operačním systémem Windows. Pro Linux je k dispozici jeho starší verze, samotní autoři zde ovšem doporučují spouštět verzi pro Windows ve virtuálním prostředí. Nástroj je veřejně dostupný pod licencí GNU GPL v.2 na jeho domovské adrese `http://cpntools.org`, kde lze nalézt také jeho dokumentaci, video návody a odkazy ke stažení [6].

3.2 GreatSPN

GreatSPN je softwarový balík pro modelování, validaci a vyhodnocování výkonu distribuovaných systémů pomocí generalizovaných stochastických Petriho sítí. Framework se skládá z mnoha oddělených podprogramů, které spolupracují na konstrukci a analýze modelů Petriho sítí. Jednotlivé analytické moduly mohou být provozovány na různých zařízeních v distribuovaném výpočetním prostředí. Architektura tohoto nástroje umožňuje jeho snadné rozšiřování o zásuvné moduly [35].

Jednou ze součástí GreatSPN frameworku je nástroj, jehož primární účel je umožnit grafické modelování Petriho sítí. Kromě standardních operací, kterými je model utvářen, lze jeho prvky například rotovat, měnit jejich rozměry anebo upravovat úroveň pohledu na model. Elementy sítě mohou být do modelu vkládány jednorázově nebo kontinuálně v závislosti na zvoleném režimu příslušného nástroje. Nechybí ani vodící čáry při umísťování prvků nebo indikace neplatných operací, jako je pokus o vzájemné propojení dvou míst nebo přechodů.

V aplikace je rovněž zakomponován modul pro interaktivní událostmi řízené simulace v podobě hry tokenů. Simulace znázorňuje převod modelovaného systému mezi jeho značeními. V aktuálním značení sítě je k dispozici seznam proveditelných přechodů, které jsou v modelu barevně odlišeny od těch ostatních. Další prováděný přechod v pořadí může být zvolen uživatelem, nebo určen náhodně. Přesun tokenů je při provedení přechodu animován. Simulaci lze řídit krokováním, spouštět automaticky, nebo se navracet k předchozím stavům. Jelikož je simulace animována, je samozřejmostí také úprava její rychlosti.

Analytické dovednosti samotného nástroje jsou poněkud omezené. Dokáže vypočítat P a T invarianty modelované struktury a graficky znázornit komponenty v sítí, které jsou indukovány jejich nosičemi. GreatSPN framework jakožto celek poskytuje mnohem obsáhlejší možnosti analýzy. Kromě P a T invariantů je ze struktury sítě schopen detekovat například pasti a zámky, strukturální konflikty a vzájemně se vylučující přechody. Další z implementovaných analytických technik je generování a rozbor stavového prostoru. Nad modelovaným systémem lze zkonstruovat graf dosažitelnosti a detekovat uzamčení systému nebo určit stupeň živosti jeho přechodů.

GreatSPN framework je vyvíjen katedrou informatiky na univerzitě Universitá degli Studi di Torino v Itálii. Veškeré moduly jsou napsány v programovacím jazyce C za účelem snadné přenositelnosti mezi Unixovými systémy. Grafický editor je napsán v Javě. Dle [36] je v současné době tento nástroj spravován Elvio G. Amparorem, pod dohledem Susanny Donatelli. Balík je dostupný zdarma pro univerzity a neziskové organizace na své domovské stránce `http://www.di.unito.it/~greatspn/index.html`. Kromě postupu jak tento framework získat tam lze nalézt také jeho dokumentaci. Nejnovější grafický editor je dostupný na adrese `http://www.di.unito.it/~amparore/mc4cslta/editor.html`.

3.3 Netlab

Netlab je nástroj pro modelování a analýzu P/T Petriho sítí. Je zamýšlen jako rozhraní mezi uživatelem a strukturou Petriho sítě. Vytvořené modely uchovává v PNML formátu, což umožňuje jejich sdílení s ostatními editory kompatibilními s tímto typem souborů. Zvládá také obsluhu několika samostatných projektů najednou [30, 31].

Editor disponuje čtyřmi základními nástroji, které slouží pro práci s modelem sítě. Pomocí prvního z nich lze manipulovat s existujícími objekty sítě. Přináší s sebou funkce jako jsou výběr, přesouvání, kopírování nebo mazání prvků modelu. Ostatní nástroje umožňují vícenásobné vkládání jednotlivých elementů sítě jmenovitě míst, přechodů a hran. Jejich vlastnosti, které jsou převážně grafického charakteru, mohou být upravovány pomocí dialogů, vyvolaných po dvojitém kliknutí na požadovaný prvek sítě. Za účelem usnadnění umisťování míst a přechodů editor poskytuje tečkovanou mřížku na pozadí kreslící plochy s nastavitelnou velikosti mezer.

Netlab je možné přepnout do režimu, který slouží pro zobrazení krokované simulace chování Petriho sítě. Aktivní přechody jsou v průběhu simulace orámovány tlustou černou linií. Následující krok simulace lze vyvolat stisknutím konkrétního přechodu a převést tak Petriho síť do nového značení. Model lze také uvést do původního značení a to buď stiskem příslušného tlačítka v panelu nástrojů anebo zastavením a opětovným spuštěním celé simulace. Množina proveditelných přechodů je závislá na použitém pravidlu proveditelnosti přechodů. K dispozici jsou jeho dvě varianty: slabé a striktní.

Nástroj má četné analytické dovednosti, které lze rozdělit do dvou kategorií. První z nich je algebraická analýza struktury sítě. Do této kategorie spadají výpočty pozitivní a negativní incidenční matice a invariantů. Invarianty mohou být zobrazeny jako vektory nebo vykresleny přímo v modelu sítě jako barevné rámečky v okolí míst a přechodů. Uživatel má možnost zapnout

automatické přepočítávání invariantů v průběhu modelování systému. Druhá z analytických kategorií je analýza stavového prostoru pomocí teorie grafů. Aplikace je schopna generovat grafy dosažitelnosti a pokrytí. Ty mohou být reprezentovány jak graficky tak i textově. Spojením výsledků zde zmíněných kategorií vzniká celková analytická zpráva pokrývající vlastnosti jako jsou živost, reverzibilnost, omezenost a uzamčení sítě.

Netlab byl vyvinut institutem Automatického řízení (IRT) na univerzitě RWTH Aachen, za účelem podpory vnitřních projektů a výuky. Není optimalizován pro podnikové využití. Založen je na bývalém nástroji určeném pro operační systém DOS. Posléze byl adaptován pro standardní windows 32-bitovou platformu. Pro vědecké a výukové účely je dostupný zdarma na své domovské adrese `http://www.irt.rwth-aachen.de/en/fuer-studierende/downloads/petri-net-tool-netlab/`. Detailnější popis všech jeho funkcí spolu se stručným uživatelským manuálem lze nalézt na adrese `http://www.irt.rwth-aachen.de/fileadmin/IRT/Download/NetLab/HilfeHTML/index.html`

3.4 Petruchio

Petruchio je nástroj pro modelování Petriho sítí založených na bázi π-kalkulu. Grafické uživatelské rozhraní editoru je postaveno na vývojovém prostředí Eclipse. Jednotlivé projekty jsou uchovávány v přenositelných pracovních prostorech. Jejich správa proto bude intuitivní pro uživatele, kteří se již s tímto prostředím setkali.

Aplikaci tvoří několik různých součástí. Editor π-kalkulu s podporou syntaktického zvýrazňování kódu umožňuje specifikovat modelovaný systém pomocí zdrojového kódu. Samotná Petriho síť je generována kompilátorem pro převod zdrojového kódu π-kalkulu do Petriho sítě založeným na využití strukturální nebo sekvenční sémantiky [4]. Výstupem kompilace jsou dva soubory popisující síť v podobě zdrojového kódu a grafického modelu. Model sítě lze upravovat pouze ve smyslu přesouvání jejích elementů. Rozmístění těchto prvků může být automaticky organizováno pomocí vybraných algoritmů knihovny Graphviz, kterou je nutné do aplikace integrovat manuálně.

Pro vygenerovanou síť je možné simulovat její chování. Aplikace za tímto účelem nabízí panel nástrojů podobný uživatelskému rozhraní hudebního přehrávače. Dříve než lze simulaci provést, je nutné editor přepnout do příslušného režimu. Množinu funkcí, jimiž je simulace řízena, tvoří akce pro návrat do počátečního značení, exekuce náhodného přechodu a automatické provádění simulačního cyklu. V každém kroku simulace je vypočteno nové dosažené značení a následně zobrazeno v grafickém editoru. Aktuálně proveditelné přechody jsou barevně orámovány.

Petruchio umožňuje analyzovat vlastnosti modelovaných sítí pomocí nástroje MCKIT. Analýza je stejně jako kompilace dostupná skrze položku v kontextové nabídce příslušného projektového souboru. Výsledky analýzy jsou zobrazovány ve speciálním editoru. Ten mimo jiné podává informace o jejím průběhu a rovněž umožňuje zkoumanou Petriho síť zobrazit graficky.

Aplikace byla vytvořena skupinou pracovníků Univerzity v Olderburgu. Největšími přispěvateli jsou Tim Strazny, Roland Meyer a Sven Linker. V současné době neprobíhá její

další vývoj. Dostupná je pro operační systémy Windows, Linux a MacOS. Ke svému plnohodnotnému fungování vyžaduje knihovnu Graphviz spolu s nástroji INA a MCKIT, které nejsou v aplikaci zahrnuty a musí být instalovány dodatečně. Veškeré informace o tomto projektu spolu s odkazem ke stažení nejnovější verze a stručným tutoriálem lze nalézt na adrese `http://petruchio.informatik.uni-oldenburg.de/index.html`. Součástí instalace je také několik vzorových projektů.

3.5 PIPE

Plarform independent Petri net editor (PIPE) je Java nástroj pro konstrukci a analýzu generalizovaných stochastických Petriho sítí. Kromě standardních funkcí, jako je grafická tvorba a editace modelu nebo jeho simulování jeho chování, poskytuje také mechanismus pro integraci nových analytických funkcí v podobě zásuvných modulů. Tím se značně odlišuje od podobných aplikací, jejichž analytické dovednosti jsou pevně stanoveny a neměnné. Nástroj využívá PNML pro specifikaci souborů generovaných sítí, což mu umožňuje spolupracovat s ostatními aplikacemi, které využívají tento formát [22].

Sítě lze kreslit na připravené plátno pomocí komponent z panelu nástrojů. K libovolné části modelu může uživatel doplnit popisky. Zakomponováno je i dynamické zvýrazňování označených komponent a prvků k nim přidruženým. Kromě těchto základních funkcí nabízí editor také možnost dynamického přiblížení pohledu na model, export do PNG nebo PostScript formátu a animované simulace chování sítě.

Simulaci lze kromě klasických způsobů, jako jsou krokování nebo provedení náhodného přechodu, spustit kontinuálně definováním počtu provedených přechodů a časovou prodlevou mezi jednotlivými kroky. Přechody, které jsou proveditelné v daném značení systému, editor graficky zvýrazňuje. Podpora zásuvných modulů je centrálním prvkem architektury aplikace. Všechny vestavěné, ale i externí, moduly jsou dostupné skrze speciální panel v uživatelském rozhraní. PIPE v základní instalaci obsahuje širokou škálu analytických modulů. Patří mezi ně moduly pro:

- klasifikaci typu sítě,
- výpočet strukturálních P a T invariantů a systémových P invariantů,
- výpočet incidenčních matic, vektorů značení a množiny proveditelných přechodů,
- detekci minimálních pastí a zámků,
- generování grafů dosažitelnosti a pokrytí spolu s analýzou stavového prostoru,
- sledování výkonnostních vlastností sítě. Modul využívá Monte Carlo simulaci, ke zjištění průměrného počtu tokenů v místech sítě a propustnosti jejich přechodů.

Vývoj tohoto nástroje započal v roce 2002 na oddělení výpočetní techniky Imperial College v Londýně. Jedná se o open source projekt, jehož současná verze nese pořadové číslo pět. Tato verze prozatím neimplementuje podporu pro zde zmiňované analytické moduly a může sloužit pouze jako grafický editor. Aplikace byla vytvořena v programovacím jazyce Java a k jejímu provozu stačí mít na cílovém počítači nainstalováno příslušné JRE. Na adrese `http://sarahtattersall.github.io/PIPE/index.html` je možné stáhnout její nejnovější verzi. Starší verzi se zabudovanou podporou analytických modulů lze spolu s dokumenty, popisujícími průběh vývoje a schopnosti aplikace, nalézt na adrese `http://pipe2.sourceforge.net`.

3.6 TimeNET

TimeNet je grafický interaktivní nástroj pro práci se stochastickými Petriho sítěmi a jejich barevnými variacemi. Třídy podporovaných sítí jsou specifikovány pomocí rozšiřitelných XML schémat. Ty obsahují množiny dostupných objektů, ze kterých lze daný typ sítě sestavit a pro každý element specifikují jeho dostupné vlastnosti a vzhled. Modely vytvořené za pomocí této aplikace jsou poté reprezentovány XML soubory, které odpovídají schématu dané třídy Petriho sítě. Pro vybrané třídy lze implementovat specifické algoritmy v podobě zásuvného modulu a rozšířit tak možnosti aplikace [41].

Uživatelské rozhraní editoru se skládá ze čtyř hlavních komponent: menu nabídky, kreslicího plátna, panelu atributů a palety komponent. Obsah této palety se spolu s množinou analytických metod dostupných v menu nabídce liší v závislosti na třídě editovaného modelu. Tvorba sítě je uskutečněna výběrem požadovaného prvku z palety a jeho opakovaným umisťováním na kreslicí plátno. Hrany jsou mezi komponentami sítě kresleny tažením myši od zdrojového po cílový element. Jejich zakřivení lze uskutečnit přidáním zlomového bodu po dvojitém kliknutím na požadovaný úsek hrany. Po výběru konkrétního objektu sítě jsou jeho atributy promítány do postranního panelu, skrze který je může uživatel editovat. Atributy jsou pro každý typ objektu sítě individuálně specifikovány ve schématu její třídy. Detaily zanořených částí sítě je možné zobrazit dvojitým kliknutím na objekt, který tuto část substituuje.

Aplikace umožňuje spouštět simulace na aktuálním zařízení anebo vzdáleném serveru. Při jejich inicializaci je poté potřeba zadat IP adresu a port serveru, na kterém běží simulační kernel. Simulace může být realizována v podobě hry tokenů, kdy jsou v jednotlivých krocích zvýrazněny proveditelné přechody a zobrazován počet tokenů v místech sítě. Do dalšího stavu je uvedena manuálním spuštěním některého z proveditelných přechodů. Při ukončení simulace uživatel specifikuje, zda chce značení sítě navrátit do původního stavu anebo ponechat to, ve kterém se model nacházel v okamžiku jejího přerušení. Výsledky simulace jsou zobrazovány v samostatném okně, které je automaticky otevřeno při jejím spuštění. Vizualizovány jsou v podobě grafů, které zobrazují změny měřených veličin v závislosti na uplynulém čase.

Mezi analytické dovednosti nástroje, založené na struktuře modelu, patří kalkulace odhadovaného počtu dosažitelných stavů, stanovení množiny minimálních pastí a zámků, výpočet minimálních P-invariantů nebo určení množiny přechodů ve vzájemném konfliktu. Pro jednu

z podporovaných tříd Petriho sítí jsou k dispozici analytické metody, které vyhodnocují chování modelu od počátečního značení až po specifikovaný koncový čas. Tyto analýzy mohou být využity například ke stanovení pravděpodobnosti, že zkoumaný systém bude provozuschopný i po nepřetržitém týdenním provozu. Kompletní popis analytických dovedností je uveden v uživatelském manuálu nástroje [41].

TimeNET je vyvíjen a spravován skupinou System and Software Engineering group na Technische Universität Ilmenau v Německu. Mezi podporované platformy patří Windows a Linux. Pro spuštění aplikace je nezbytné mít nainstalováno JRE verze sedm nebo novější a speciálně pro operační systém Windows ještě MinGW prostředí. Na domovské stránce aplikace `http://www.tu-ilmenau.de/sse/timenet/` lze nalézt informace ohledně její instalace a také uživatelský manuál. Nejnovější verze nástroje z dubna 2015 je dostupná pro nekomerční využití po vyplnění jednoduché registrace a následném zpřístupnění v informačním systému univerzity.

3.7 Woped

Woped (Workflow Petri Net Designer) je softwarový nástroj pro editaci, simulování a analýzu workflow a P/T Petriho sítí. Aplikace podporuje hierarchické zanořování v podobě substitucí přechodů a poskytuje základní funkce automatického uspořádání prvků modelu za účelem jeho lepší čitelnosti. Jako souborový formát používá PNML standard. Navržené sítě dokáže exportovat do podoby PNG, JPG a BMP obrázků [1].

Grafické rozhraní aplikace je založeno na principu přepínatelných panelů, které v sobě agregují veškeré dostupné operace nad modelovanou sítí. Panely jsou logicky členěny do oblastí editace modelu sítě, řízení simulací, změny pohledu na model anebo provádění analytických operací. Práce s okny všech aktuálně otevřených modelů je realizována podobným způsobem. Editor kromě tradičních P/T Petriho sítí podporuje modelování takzvaných workflow sítí dle van der Aalstovy notace. Za tímto účelem implementuje speciální přechodové operátory, různé typy spouštěčů přechodů a třídy zdrojů. Jednotlivé třídy zdrojů lze v aplikaci definovat pomocí speciální grafické komponenty. Ta dále umožňuje vytvářet instance zdrojů a přiřazovat je do dříve specifikovaných tříd. Přechodům v modelu je poté možné nastavit roli a skupinu, do které patří, z množiny definovaných tříd.

Simulace jsou interpretovány jako hra tokenů řízená pomocí jednoho z aplikačních panelů. Dostupné je dopředné a zpětné krokování spolu s možností návratu k původnímu značení modelu a ruční výběr vykonávaných přechodů. Proveditelné přechody jsou v každém kroku simulace graficky zvýrazněny. Pokud je během simulace zpřístupněn některý ze substitučních přechodů, lze k němu přistupovat stejně jako k běžnému přechodu anebo se zanořit do částí podsítě, kterou zastupuje. V druhém případě simulace pokračuje na úrovni substituované části modelu. Tento postup se dá přirovnat k ladění zdrojového kódu běžných aplikací.

Analýzy, které Woped dokáže nad modelem provádět, lze rozdělit na kvalitativní a kvantitativní. Do kvalitativní analýzy spadá detekce špatně užitých operátorů, porušení pravidla volného výběru a kontrola, zda je síť správně utvořená. V poslední řadě zde spadá ověřování správnosti

workflow sítí. Mezi schopnosti nástroje dále patří generování a vizualizace grafů dosažitelnosti a pokrytí. Uzly těchto grafů lze uspořádat do hierarchické nebo kruhové struktury a exportovat stejně jako samotný model v podobě obrázků.

Aplikace je od května roku 2003 vyvíjena pracovníky a studenty Karlsruhe DHBW univerzity v Německu. Dodnes se na tomto nástroji, který je distribuován pod GNU LGPL licencí podílelo více než sto přispěvatelů. Vytvořen byl v programovacím jazyce Java, a tudíž ke svému fungování vyžaduje příslušné virtuální prostředí. Podporovány jsou všechny operační systémy, pro které je dostupné JRE 5.0 a vyšší. Spolu s dokumentací je zdarma ke stažení na internetové adrese `http://www.woped.org/`.

3.8 Další nástroje

Nástroje uvedené v předešlých částech této kapitoly nepředstavují celý soubor aplikací, které jsou v současné době použitelné za účelem práce v oblasti Petriho sítí. Jelikož není tento výčet hlavním tématem práce, budou zde ostatní nástroje uvedeny pouze v podobě seznamu s jejich stručným popisem. Aplikace jsou v tomto seznamu uvedeny, protože nepřináší oproti jiným aplikacím žádné zvláštní funkce navíc, nebo jsou z pohledu kritérií pro výběr uvedených nástrojů méně relevantní.

- **QPME**
  Open-source program založený na formalismu modelování frontových Petriho sítí. Kromě běžné funkcionality podporuje aplikace také hierarchické strukturování a tvorbu víceúrovňových modelů. K dispozici jsou takzvané sondy, které vymezují oblast, pro kterou by měla být shromažďována data v průběhu simulace. Sonda je specifikována pomocí počátečního a koncového místa sítě. Jejím cílem je vyhodnotit čas, který tokeny stráví v oblasti vymezené sondou, při jejich přesunu od počátečního po koncové místo. Každá sonda může být nastavena tak, aby reagovala na tokeny konkrétních barev. Výsledky simulace lze také zpracovávat a vizualizovat pomocí pokročilého dotazovacího enginu [23].

- **Renew**
  Editor a simulátor, který poskytuje flexibilní přístup k modelování založený na formalismu referenčních sítí [33]. Aplikace je schopná modely exportovat do několika různých formátů včetně PNML, z něhož je zvládá také importovat. Kromě standardních elementů nástroj nabízí takzvaná virtuální místa, která slouží pro definování kopií existujících míst. Účelem virtuálních míst je zvýšit čitelnost a grafickou přehlednost sítě v místech, která jsou využívána velkým počtem přechodů.

  Jako jeden z mála, nabízí tento editor funkci automatického rozmisťování komponent sítě na kreslícím plátně. Po spuštění této funkce je pozice prvků sítě kontinuálně upravována dle pravidel nastavených v nově zobrazeném panelu. Na hrany se v daný okamžik nahlíží jako na pružiny, u kterých lze měnit vlastnosti jako je délka a síla. Pozici kteréhokoliv místa a přechodu lze uzamknout. Uzamčené prvky však nepřestávají ovlivňovat své okolí.

- **Tapaal**

  Open source simulátor, verifikační nástroj a grafický editor pro Petriho sítě s časovanými hranami, jejichž třídu rozšiřuje o věkové invarianty, urgentní přechody a transportní a inhibiční hrany [8]. Uživatelské rozhraní aplikace je založeno na dříve popisovaném nástroji PIPE. Nástroj umožňuje vytvářet dotazy nad modelem sítě pomocí speciálního dialogu. Dotazy mohou v závislosti na svém typu ověřovat několik skutečností například:

  - existenci dosažitelného značení, které splňuje zadané podmínky,
  - existenci sekvence, kde každé značení vyhovuje stanoveným podmínkám,
  - skutečnost, že všechna dosažitelná značení vyhovují daným podmínkám.

- **Yasper**

  Navržen pro snadné modelování a simulaci pracovních procesů. Vytvořené modely umožňuje vytisknout nebo exportovat jako obrázky různých formátů. Před započetím automatické simulace chování sítě musí být její počáteční a koncové body reprezentovány takzvanými emitory a kolektory. Emitory slouží pro generování tokenů, zatímco kolektory pro jejich konzumaci. Dobu trvání simulace lze omezit stanovením maximálního času, po který může simulace běžet, nebo maximálním množstvím tokenů, které emitory vytvoří nebo kolektory zkonzumují. V průběhu automatické simulace jsou všechny proveditelné přechody voleny náhodně v závislosti na nastavení sítě.

  V okamžiku zastavení simulace je vygenerována zpráva o jejím průběhu. Ta poskytuje přehled o počtu vygenerovaných a zkonzumovaných tokenů pro každou dvojici emitor – kolektor. Pro jednotlivé úspěšné průchody sítí (vygenerovaný token v emitoru byl zkonzumován kolektorem) jsou počítány statistiky jako průměrný čas zpracování tokenu v jednotlivých přechodech, průměrný čas strávený čekáním na zpřístupnění přechodů nebo průměrný čas, který token strávil v systému [9].

- **IOPT Tools**

  Sada nástrojů určená pro modelování a testování aplikací průmyslové automatizace a jiných digitálních systémů. Pro jejich specifikaci využívá takzvané IOPT sítě. Jedná se o třídu Petriho sítí rozšířenou o vstupní a výstupní dovednosti, nezbytné pro komunikaci s externím prostředím (čtení senzorických dat, komunikace s jinými systémy) [10]. Přechodům sítě lze přiřadit takzvané záchytné body (breakpointy). Simulace je pozastavena, pokud je při jejím vykonávání proveden některý z takto označených přechodů.

  Pro vytvořené modely dokáže nástroj generovat graf dosažitelnosti. Analytický význam jednotlivých vrcholů je určen jeho barvou. Pozorovat tak lze například uváznutí systému anebo přechody ve vzájemných konfliktech. Aplikace také podporuje automatické generování zdrojového kódu z vytvořeného modelu do programovacích jazyků C a JavaScript. Jako jediný z nástrojů uvedených v této práci je IOPT dostupný online a nevyžaduje instalaci na cílové zařízení.

4 Vyvinutý systém

V nadcházejících kapitolách se budeme věnovat popisu systému, který byl vytvořen v rámci této práce. Vznik nových analytických nástrojů je často motivován nedostatky existujících řešení. Problémy mohou spočívat například v dostupnosti aplikace anebo její množině dovedností, která není pro potřeby uživatele dostatečná. Neobvyklým jevem také není to, že je vznik takových nástrojů podnícen potřebou ověřit nové teoretické poznatky v praxi, nebo jen podpořit výuku v dané oblasti. Hodnota času investovaného do vývoje je poté zúročena v jednoduchosti nového řešení. Neopomenutelnou výhodou je vlastnictví zdrojových kódů systému a jeho snadná rozšiřitelnost a upravitelnost, což není u externích řešení možné.

Jako motivace pro tvorbu zde popisovaného nástroje nepochybně sloužilo několik výše uvedených kritérií. Aplikace je zaměřena výhradně na třídu Place/Transition Petriho sítí. Její modelovací a analytické dovednosti byly cíleny tak, aby co nejobsáhleji pokryly oblast základní výuky Petriho sítí na zdejší univerzitě. Při vývoji aplikace byl kladen důraz na jednoduchost jejího ovládání a srozumitelnost jejích výstupů.

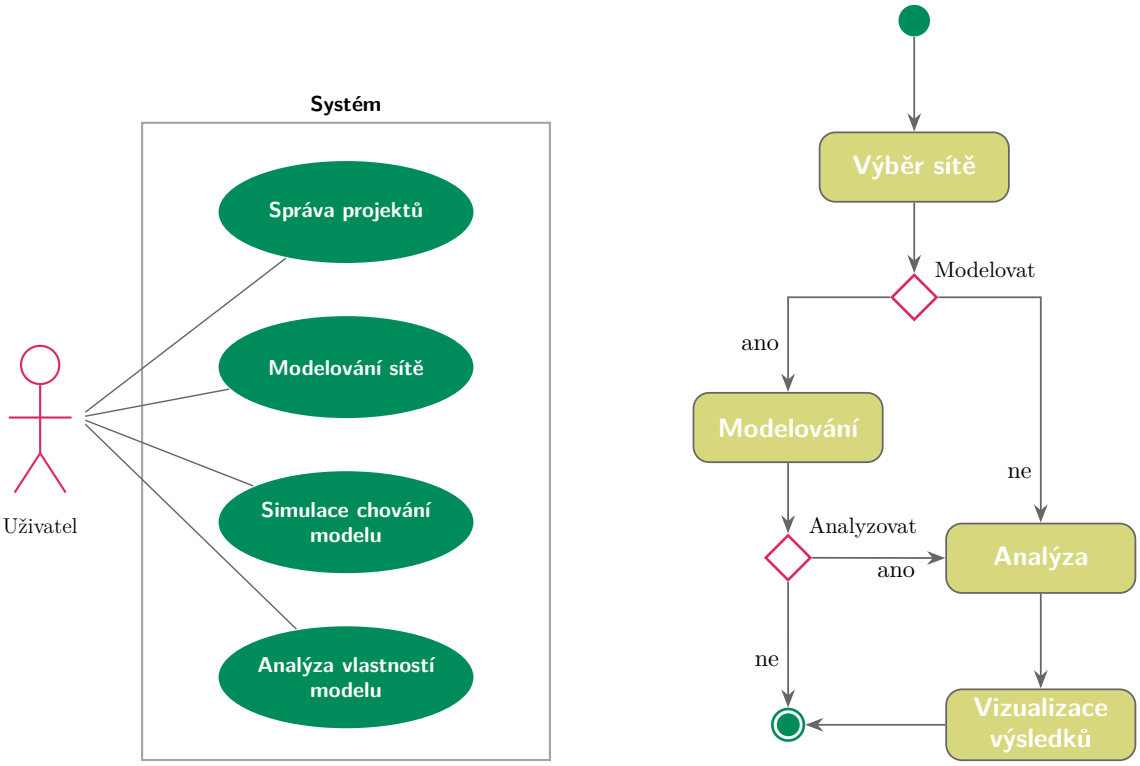
Několikrát jsme se inspirovali v nástrojích, uvedených v kapitole 3. Jednalo se jak o pozitivní zpětnou vazbu, tak o identifikaci ne příliš vhodně zvolených řešení, což nám umožnilo vyhnout se stejným chybám. Jedním z nástrojů s uživatelským rozhraním na vysoké úrovni a intuitivním ovládáním je GreatSPN. Kromě principů a funkcionality využívané při tvorbě Petriho sítí jsme z tohoto editoru převzali také ideu o promítání výsledků analýzy přímo do jejího modelu. Některé nástroje se naopak vyznačují nepohodlností jejich ovládání, nepřehledným uživatelským rozhraním nebo nejasnou reprezentací modelů Petriho sítí.

4.1 Architektura systému

Systém se skládá ze dvou hlavních součástí. První z nich je analytický server, který je schopen nad modelem sítě spustit různé analytické metody a zodpovědět otázky, které jsou pro Petriho sítě relevantní. Za účelem tvorby modelů sítí a vizualizace výsledků jejich analýzy byla vytvořena druhá část systému. Jedná se o grafický editor implementovaný ve formě tenkého klienta.

Toto rozdělení umožňuje snadnou údržbu, rozšiřování a úpravy systému. Například k obohacení analytických dovedností serveru, není nutné upravovat klienty, bude-li zachována kompatibilita s jejich komunikačním rozhraním. Jako nevýhoda se může jevit závislost klienta na serveru v situacích, kdy uživatel potřebuje navržený model analyzovat. Vzhledem k dnešním technologickým možnostem a řešením v oblasti počítačových sítí tuto skutečnost považujeme za nepodstatnou. Popisu obou částí systému jsou věnovány samostatné části této kapitoly.

Základní případy užití systému popisuje diagram na obrázku 2a. V systému figuruje jediný aktér, který reprezentuje běžného uživatele aplikace. Účel uvedeného diagramu případů užití je vytvořit hrubou představu o tom, jaké operace jsou v systému pro jeho uživatele dostupné. Jednotlivé případy užití představují abstraktní oblasti, které mohou být dále podrobněji rozvíjeny samostatnými diagramy.



(a) Diagram případů užití (b) Diagram aktivit

Obrázek 2: Diagramy popisující vyvinutý systém

1. Případ užití *Správa projektů* zahrnuje operace, nezbytné pro manipulaci s projekty Petriho sítí. Spadá zde tvorba nové sítě, načítání existujících sítí a ukládání provedených změn. Poslední prvek množiny CRUD operací je reprezentován případem užití smazání projektu.

2. *Modelování sítě* obsahuje veškeré případy užití spojené s tvorbou modelu Petriho sítě.

3. U existujících modelů aplikace umožňuje simulovat její chování. Tuto funkcionalitu reprezentuje případ užití *Simulace chování modelu*.

4. V oblasti funkcí systému, které pokrývá případ užití *Analýza vlastností modelu*, lze kromě operace provedení analýzy nalézt také nastavování jejích parametrů a vizualizace jejích výsledků.

Diagram aktivit 2b znázorňuje jeden ze standardních postupů uživatele při práci s tímto nástrojem. Jedná se o zřetězení vybraných případů užití systému. Nezbytnou podmínkou pro provedení jakékoliv operace nad modelem Petriho sítě je jeho existence. Proces proto začíná výběrem sítě a pokračuje dále dle preferencí uživatele. Před analýzou sítě systém umožňuje provést změny v jejím modelu. Stejně tak je přípustný průběh, kdy je cílem aktéra model pouze editovat. S analýzou bývá typicky asociována také vizualizace jejich výsledků.

4.2 Webový klient

Jak jsme již zmínili v úvodním popisu architektury systému, jedna z jeho součástí je grafický editor, který slouží primárně pro editaci modelu Petriho sítí a vizualizaci výsledků jejich analýzy. Klienta jsme implementovali v podobě webové aplikace, což s sebou oproti klasickému desktopovému řešení přináší několik výhod:

- odpadá potřeba instalace na cílová zařízení,
- systém je dostupný odkudkoliv, kde je realizováno připojení k internetu,
- aplikace není závislá na platformě, k jejímu spuštění postačuje webový prohlížeč,
- údržba a aktualizace systému jsou snadnější.

Hlavním argumentem proti použití webové aplikace je srovnání výkonu s jejím desktopovým protějškem. Tento problém je našem systému řešen existencí analytického serveru. Grafický editor vystupuje pouze jako tenký klient a veškeré složité výpočetní operace jsou ponechány k realizaci na serveru.

4.2.1 Použité technologie

Javascript zůstává v současné době dominantním programovacím jazykem, využívaným pro tvorbu nejen webových aplikací. K dispozici je široká škála frameworků (a vznikají stále nové), jejichž cílem je usnadnit a urychlit vývoj webových aplikací. Za zmínku stojí například React, Ember, Backbone nebo také Angular. Poslední ze zmíněných frameworků jsme využili při vývoji zde popisovaného webového klienta systému.

4.2.1.1 Angular

Angular byl poprvé představen společností Google v roce 2009. Od té doby kolem něj byla vytvořena mohutná komunita a je v současnosti nejpoužívanějším Javascriptovým frameworkem pro tvorbu webových aplikací. Hlavním přínosem Angularu je jeho dovednost rozšířit HTML atributy pomocí takzvaných direktiv. Programátor má k dispozici sadu základních vestavěných direktiv a možnost vytvořit si své vlastní.

Angular kompilátor poté při kompilaci a vykreslování HTML uživatelského rozhraní manipuluje s objektovým modelem webové stránky a přidává dovednosti, které s sebou přinášejí použité direktivy. Jádrem frameworku je takzvané obousměrné datové provázání (Two-way data binding). Tento mechanismus slouží pro automatickou synchronizaci uživatelského rozhraní a jeho datového modelu a to v obou případech, kdy je změna provedena na straně modelu nebo na straně uživatelského rozhraní [2].

V rámci webové aplikace jsou uživatelské rozhraní a aplikační logika oddělené oblasti. Stejně jako pro vývoj aplikační logiky existují za účelem tvorby jejího uživatelského rozhraní různé

knihovny, mezi nimiž jednoznačně v popularitě vítězí Bootstrap. Vzhledem ke skutečnosti, že je naše aplikace postavena na Angular frameworku, jsme se za účelem tvorby jejího uživatelského rozhraní rozhodli použít knihovnu Angular material. Jedná se o sadu komponent uživatelského rozhraní založených na Material Design specifikaci od firmy Google. Samotné komponenty jsou implementovány jako direktivy. Součástí knihovny je také několik pomocných služeb například pro zobrazování dialogů. Nechybí ani grafická témata s možností jejich přizpůsobení vlastním představám.

4.2.1.2 Javascript Prototypy

Javascript jakožto dynamický skriptovací jazyk nezná klasickou konstrukci třídy, tak jak je definována v jiných objektově orientovaných jazycích (Java, C++ a další). Každý objekt v Javascriptu má vnitřní referenci na svůj takzvaný prototyp. Prototyp je objekt, ze kterého jeho potomci dědí všechny atributy a metody. Jelikož je samotný prototyp také objekt, má i on svůj vlastní prototyp. Tato kompozice se nazývá protypový řetězec. Na vrcholu prototypového řetězce se nachází *Object.prototype*, ze kterého jsou všechny ostatní prototypy odvozeny. Při pokusu o přístup ke konkrétnímu atributu objektu je prohledáván jeho prototypový řetězec směrem ke kořeni a navrácen první výskyt atributu s odpovídajícím názvem. Stejný postup je uplatněn při volání metod [21, 38]. Z důvodu jednotné terminologie budeme v rámci této práce dále používat pojem třída při popisu objektových prototypů. Tento přístup nám umožní srozumitelně popsat konstrukce, které nejsou v Javascriptu přítomny, ale lze je napodobit.

4.2.1.3 HTML5 grafické technologie

V oblasti vykreslování grafického obsahu ve webovém prostředí v současné době vévodí dva přístupy: SVG a canvas. SVG je jazyk pro popis vektorové grafiky pomocí XML. Stejně jako HTML má i SVG svůj vlastní DOM. Každý vykreslený element je trvale součástí tohoto objektového modelu. Kromě standardních atributů disponují SVG elementy také takzvanými prezentačními atributy, které mohou být stylovány pomocí CSS pravidel. K aktualizaci stavu scény postačuje znovu vykreslit pouze pozměněný element. Jelikož jsou SVG elementy součástí objektového modelu, lze k nim také přiřadit posluchače na události, jako je klikání myší.

Canvas oproti SVG přináší více programátorský přístup pro tvorbu webové grafiky. Je vykreslován pixelově a jeho kontext je zapomenut v okamžiku, kdy je dokončeno vykreslování celé scény. V praxi to znamená, že pokaždé, kdy je požadován nový snímek, musí být znovu provedeny všechny grafické příkazy, kterými je scéna vytvořena. Jakákoliv manipulace s grafikou je v kontextu canvasu prováděna pomocí Javascriptu. Možnosti reakce na události jsou oproti SVG limitovány. Jelikož nejsou vykreslené objekty součástí objektového modelu dokumentu, jsou události odchytávány na úrovni canvas elementu. Programátor tak musí manuálně překládat koordináty kurzoru myši vzhledem k objektům vykresleným na plátně [19].

Vhodnost obou technologií je pro účely zde popisované aplikace srovnatelná. Jejich detailnější porovnání lze nalézt v [19]. Při vývoji webového klienta systému jsme se přiklonili ke druhé z uvedených variant. K rozhodnutí přispělo několik faktorů:

- vyšší výkon pří práci s vysokým počtem objektů a menším obsahem plochy plátna,
- existence frameworků, které usnadňují práci s canvasem a řeší jeho zmíněné nedostatky,
- naše obeznámenost s canvas API.

4.2.1.4 EaselJS

Množina frameworků pro práci s canvasem je přinejmenším stejně početná jako ta pro tvorbu webových aplikací samotných. Při výběru řešení nás nejvíce oslovil EaselJS framework. EaselJS je Javascriptová knihovna, která pro canvas přináší režim uchovávané grafiky (retained graphics mode). Všechny zobrazené elementy v plátně jsou podobně jako u SVG reprezentovány pomocí objektů a interně uchovávány.

Mezi základní dovednosti frameworku patří tvorba hierarchické struktury zobrazených objektů pomocí kontejnerů. Vrcholem této hierarchie je objekt typu *Stage*, který obaluje canvas element. Skrze něj jsou poté přidávány ostatní objekty a prováděny základní operace jako je volání instrukce k aktualizaci plátna.

Objekty, které jsou součástí zobrazené hierarchie, vyvolávají události při jejich interakci s myší. EaselJS podporuje všechny základní události jako jsou kliknutí, stisk, tažení a a uvolnění. Nechybí ani detekce elementu, nad kterým se kurzor právě nachází.

Kromě již zmíněných vlastností stojí za zmínku ještě vestavěná podpora pro kreslení obrázků užitím bitmap, vektorovou grafiku nebo renderování textových řetězců. EaselJS funguje ve všech webových prohlížečích, které podporují canvas element. Jeho výkon se ovšem odvíjí od vnitřní implementace canvas elementu v prohlížeči [11].

4.2.2 Objektová reprezentace Petriho sítě

Každá Petriho síť je v aplikaci reprezentována jako instance třídy *PetriNet*. Místa, přechody a hrany jsou poté realizovány jako instance tříd *PNPlace*, *PNTransition* a *PNArc* respektive. Jejich vzájemnou kompozici zachycuje třídní diagram na obrázku 3. Každý objekt sítě je jednoznačně identifikován pomocí svého ID. Kromě atributů uvedených v diagramu uchovávají ještě informace o pozici svého popisku a volitelně také data, která představují změny oproti výchozím hodnotám využívaných pří vykreslování těchto objektů.

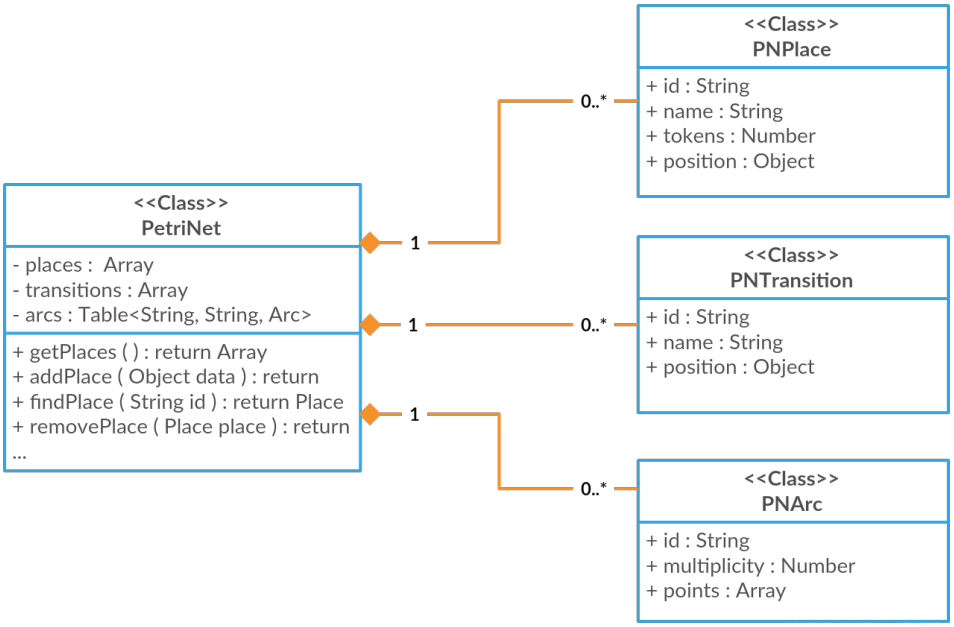
Místa a přechody jsou v sítí obsaženy v polích. Informace o hranách uchovává speciální kolekce typu *Table*. Ta umožňuje mapovat hodnoty pod unikátními dvojicemi klíčů (jedním pro řádek a druhým pro sloupec tabulky). Jako klíče jsou v kontextu třídy *PetriNet* použity ID objektů, které hrana propojuje a to ve směru od řádku ke sloupci. Existuje-li v Petriho sítí

hrana, která propojuje místo p s ID = 1 a přechod t s ID = 2, je objekt takové hrany v tabulce uchováván pod řádkovým klíčem 1 a sloupcových klíčem 2.

Při implementaci této kolekce jsme čerpali inspiraci v jejím protějšku, využívaném na serveru. Jedná se o Java rozhraní *com.guava.collection.Table* z knihovny Guava od společností Google. Naše implementace napodobuje její veřejné rozhraní a definuje několik metod:

- metody *put*, *get* a *remove* jsou určeny pro vkládání, vyhledání a smazání hodnoty, uložené pod konkrétním řádkovým a sloupcovým klíčem,
- metoda *row* vrací pomyslný řádek tabulky, kdy jsou konkrétní hodnoty mapovány pouze pod sloupcovým klíčem,
- metoda *column* vrací pomyslný sloupec tabulky, kdy jsou konkrétní hodnoty mapovány pouze pod řádkovým klíčem.

V kontextu Petriho sítě umožňují metody *row* a *column* efektivní vyhledávání například vstupních a výstupních míst daného přechodu.



Obrázek 3: Třídní diagram objektové struktury Petriho sítě

4.2.3 Tvorba modelu Petriho sítě

Je zřejmé, že práce s canvas elementem se liší v závislosti na typu vyvíjené aplikace. Webového klienta, který vznikl v rámci této práce, zařadíme do kategorie nástrojů pro tvorbu a editaci diagramů. U této skupiny aplikací je možné identifikovat množinu vlastností a dovedností, které nezbytně musí mít každý nástroj zde spadající.

Za účelem realizace těchto dovedností jsme vytvořili vlastní nadstavbu pro práci s canvasem, která je založena na již zmíněném EaselJS frameworku. Primární element našeho API je abstraktní třída *EaselCanvasManager*. Samotnou logiku práce s canvasem vykonávají její interní komponenty, kterým se budeme věnovat v následující části práce. *EaselCanvasManager* vystupuje především v roli prostředníka pro vzájemnou komunikaci mezi svými vnitřními komponentami. Pořadí inicializace těchto podsystémů je určeno jejich vzájemnými závislostmi. Při návrhu byl kladen důraz na znovupoužitelnost a vysokou úroveň abstrakce. Tyto principy nám umožnily nadstavbu využít jak pro práci s modelem Petriho sítí, tak pro vizualizaci výsledků jejich analýzy.

4.2.3.1 Grafické objekty

Scéna je v kontextu EaselJS frameworku tvořena objekty, které jsou instancí jedné ze základních vestavěných tříd:

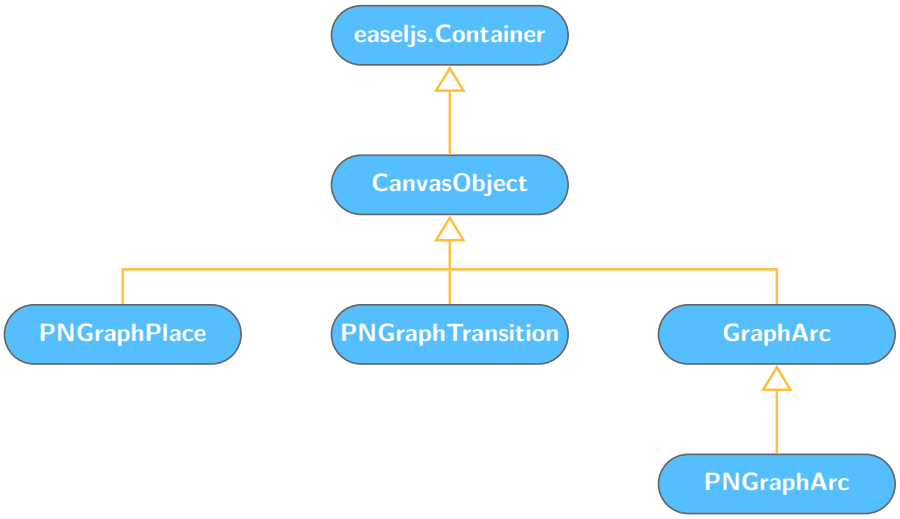
- *Shape* slouží pro vykreslování jednoduchých obrazců jako jsou obdélníky, čáry nebo kruhy stejně jako jejich kompozic,
- *Text* umožňuje na plátno vkládat textové řetězce, jejichž vzhled je ovlivňován podobným způsobem jako u stylování v CSS,
- *Container* má za úkol agregovat ostatní zobrazené objekty.

Nové typy objektů lze vytvářet rozšířením některé z výše uvedených tříd. Tento přístup jsme aplikovali pro definici všech vlastních grafických objektů. Volba rodičovské třídy závisí na struktuře nově vytvářeného potomka. Je-li například definován objekt, jehož součástí je textový popis, je nezbytné jako předka využít *Container*. Početná část objektů, které jsou v aplikaci definovány, má několik společných atributů. Jejich abstrakce vedla ke vzniku třídy *CanvasObject*, která figuruje jako primární předek všech grafických elementů Petriho sítě. Hierarchii dědičnosti znázorňuje diagram na obrázku 4.

První z interních komponent *EaselCanvasManagera* je třída *CanvasObjectFactory*, jejímž cílem je poskytnout jednoduchý mechanismus pro tvorbu grafických objektů, bez znalosti jejich konkrétních tříd. Instance jsou vytvářeny pomocí metody *create*, jejíž argumenty jsou typ vytvářeného objektu a volitelně také atributy, které budou použity při jeho konstrukci. Množinu dostupných objektů specifikuje programátor v potomcích třídy *ObjectFactory*, kde je za tímto účelem nutné implementovat abstraktní metodu *getRegisteredObjects*. Ta vrací pole záznamů se dvěma atributy: typ objektu a jeho konstruktorová funkce.

4.2.3.2 Vrstvy grafické scény

Standardním nástrojem při práci s jakoukoliv grafikou je její rozdělení do vrstev. Hlavním přínosem grafických vrstev je fakt, že umožňují izolovat individuální prvky scény, se kterými lze poté manipulovat odděleně, aniž by byly jakkoliv ovlivněny zbylé části celku. Mimo jiného také



Obrázek 4: Hierarchie vybraných grafických objektů aplikace

jednoznačně určují, které elementy jsou zobrazeny na jednotlivých úrovních scény a usnadňují tak tvorbu výsledné kompozice.

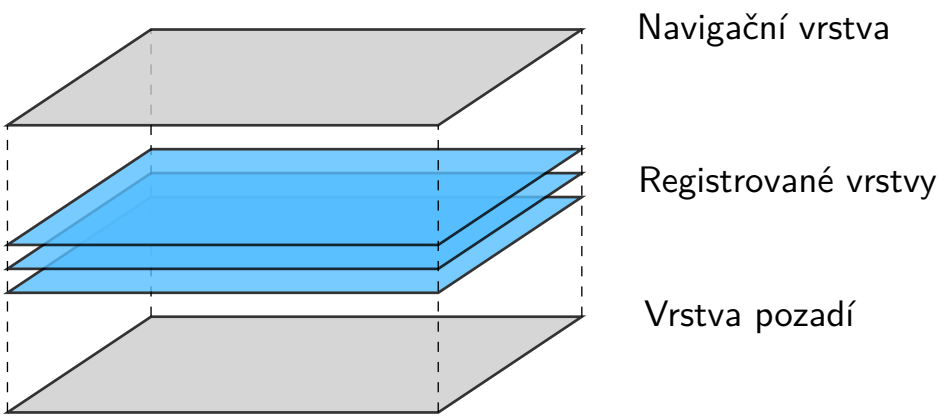
Funkcionalitu vrstev, jak zde byla popsána, je možné v prostředí EaselJS frameworku napodobit pomocí kontejnerů. Objekty jsou na scénu vykreslovány v pořadí, v jakém byly do kontejneru přidány. Jelikož je také *Stage* objekt potomek třídy *Container*, je tento princip při renderování scény uplatněn globálně. Vrstvy scény si lze tedy představit jako kontejnery, které jsou umístěny přímo do *Stage* objektu.

V roli správce vrstev vystupuje v *EaselCanvasManageru* instance třídy *CanvasLayerBus*. Mezi její kompetence spadá přidávání a odebírání vrstev ze scény, jejich vyhledávání nebo transformace globálních koordinátů do lokálního souřadnicového systému konkrétního kontejneru. Dalším důležitým prvkem je třída *CanvasLayerLoader*, pomocí níž je prováděna samotná konstrukce vrstev scény. Jejich registrace probíhá obdobným způsobem jako u grafických objektů s tím rozdílem, že je navíc nezbytné specifikovat pořadí, ve kterém mají být do scény přidány.

Kromě vrstev, které byly vytvořeny pro danou implementaci *EaselCanvasManagera*, jsou do scény přidány také dvě výchozí. První z nich se nachází na nejnižší úrovni hierarchie a vystupuje jako pozadí kreslící plochy. Druhá z výchozích vrstev je umístěna nade všemi ostatními a je využívána pro účely navigace v editoru. Celkovou kompozici znázorňuje obrázek 5.

4.2.3.3 Události EaselJS objektů

Jelikož je canvas uvnitř HTML stránky reprezentován jediným elementem, jsou všechny události spojené s jeho interakcí s myší vyvolávány přímo na něm. EaselJS tyto eventy vnitřně interpretuje a propaguje dále svou interní objektovou hierarchií. V důsledku je pak možné připojit posluchače události na konkrétní *DisplayObject*, přestože žádná taková událost v objektovém modelu stránky nikdy nenastala.



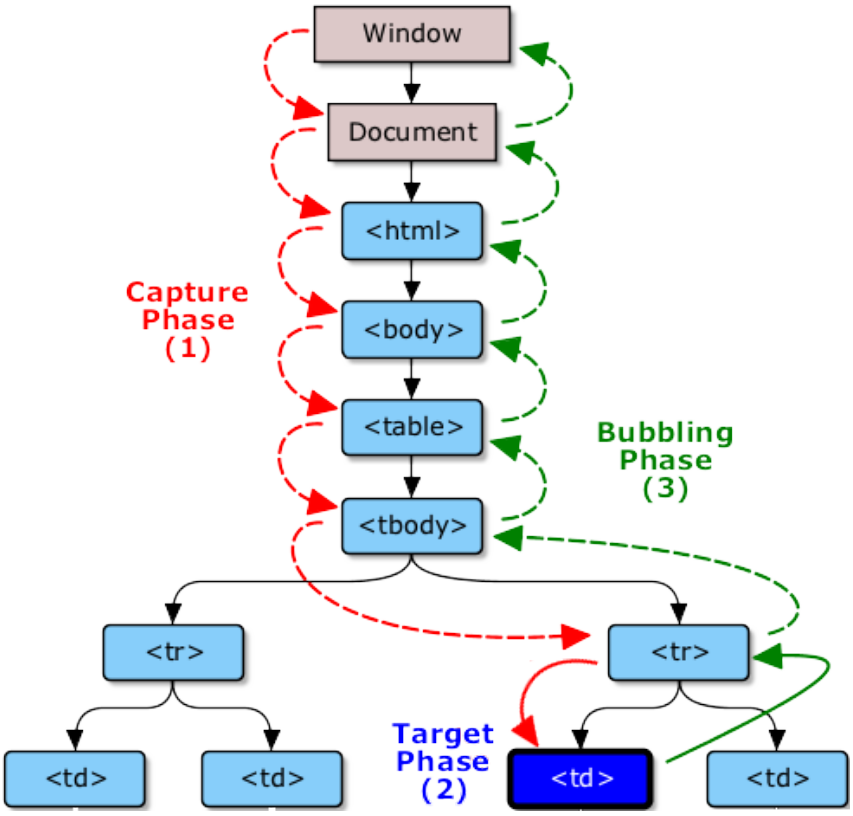
Obrázek 5: Uspořádání vrstev v EaselCanvasManageru

Interní systém událostí v EaselJS frameworku je založen na specifikaci Document Object Model Level 2 Events Specification. Vyvolávané události tak nesou nízkoúrovňové informace, které pro části systému zainteresované v odposlouchávání těchto událostí nemusí být relevantní a srozumitelné. Objekt, kde událost vzniká, musí být navíc globálně přístupný, aby na něm byla umožněna registrace příslušných posluchačů. Tato skutečnost může být v systému nežádoucí.

Zmíněné problémy řeší další z komponent *EaselCanvasManagera* a to *CanvasEventManager*. Jeho primárním úkolem je překlad nativních událostí grafických komponent do množiny programátorem definovaných událostí vyšší úrovně a jejich následná distribuce do zbylých částí systému. Reprezentuje tak globální přístupový bod pro registraci posluchačů především pro komponenty aplikace, kde nejsou grafické objekty přímo dostupné.

Množinu událostí, které je *CanvasEventManager* schopen vyvolávat, specifikuje konkrétní implementace abstraktní třídy *CanvasEventFactory*. Eventy jsou v ní registrovány stejným způsobem jako ve třídě *CanvasObjectFactory*. Při konstrukci události vyšší úrovně je jako argument, kromě jejího typu, očekávána také původní událost, ze které jsou extrahovány relevantní informace. *CanvasEventManager* navíc spravuje také výchozí události, společné pro všechny implementace třídy *EaselCanvasManager*. Jedná se o eventy vyvolávané přímo na *Stage* objektu anebo canvas elementu. Pro lepší představu zde uvedeme příklad popisované transformace událostí. Dříve než tak učiníme se ještě ovšem zmíníme o principu propagace událostí v objektovém modelu HTML stránky. Stejný princip je aplikován na hierarchii grafických objektů v EaselJS frameworku.

Postup propagace události velice přehledně zachycuje obrázek 6. Každá událost sestupuje stromem objektového modelu až k elementu, na kterém byla vyvolána. Tato fáze postupu se nazývá *Capture Phase* a bývá také označována číslem jedna. Proces se nachází v *Target* fázi (neboli fázi číslo dvě) v okamžiku, kdy event dorazí k cílovému prvku. Následně událost postupuje zpět ke kořeni objektového modelu, přičemž se propagace události nachází v takzvané *Bubbling* fázi označované číslem tři. Událost bude zachycena každým posluchačem, který je připojen na některém z objektů na cestě od kořene DOM stromu až po cílový prvek.



Obrázek 6: Model propagace události skrze objektový model stránky. Obrázek byl převzat z [37]

Výše zmíněných mechanismů jsme využili při odposlouchávání událostí na grafických objektech Petriho sítě. Ty jsou uchovávány ve speciální vrstvě a samostatných kontejnerech, na které jsme tak mohli připojit jediného posluchače pro každou sledovanou událost namísto tvorby samostatného posluchače pro každý objekt sítě. Posluchači připojení na kontejnerech byli nastaveni tak, aby na události reagovali v jejich třetí fázi propagace. V okamžiku odchycení události byl proveden její překlad na událost vyšší úrovně a propagace původní události zastavena. Pokud by posluchač reagoval na události hned v první fázi, znemožnil by tak reakci na událost posluchačům připojeným přímo k cílovému objektu.

4.2.3.4 Funkční zásuvné moduly

Doposud představené komponenty *EaselCanvasManagera* se zabývaly problémy spojenými s vykreslováním objektů na plátno. Aplikační logika editoru je realizována pomocí sady zásuvných modulů. Moduly jsou zamýšleny jako samostatná komponenta, jejímž přidáním nebo odebráním je zpřístupněna nebo vypnuta konkrétní funkce systému. Představují tak hlavní nástroj programátora při definování dovedností jednotlivých implementací třídy *EaselCanvasManager*. Každý z modulů má přístup ke všem esenciálním komponentám manažera a mohou spolu také spolupracovat.

Interní správce modulů se nazývá *CanvasModuleBus*. Jeho rozhraní a dovednosti jsou takřka totožné se sběrnicí vrstev. Za účelem načítání konkrétních modulů v aplikaci existuje abstraktní třída *CanvasModuleLoader*. Moduly jsou v potomcích této třídy registrovány standardním způsobem, tedy v podobě dvojic atributů: typ a konstruktorová funkce. Dále se zde zmíníme o nejdůležitějších modulech využívaných v implementaci *EaselCanvasManagera* pro modelování Petriho sítí a stručně popíšeme jejich význam.

1. **PNConstructor**
   Tento modul představuje most mezi doménovou reprezentací Petriho sítě a jejím grafickým protějškem. Stará se o vykreslování elementů Petriho sítě, vyhledávání jejich elementů a především pak synchronizaci změn provedených v editoru zpět do doménového modelu sítě.

2. **ObjectInserter**
   Pomocí tohoto modulu jsou do Petriho sítě přidávány nová místa a přechody. Modul také zajišťuje vykreslování vzorového objektu sítě, takže má uživatel představu o tom, jak bude síť vypadat po jeho přidání.

3. **ObjectConnector**
   ObjectConnector v sobě obsahuje veškerou logiku, spojenou s propojováním míst a přechodů Petriho sítě. Podobně jako *ObjectInserter* na scénu přidává vzorovou hranu, která zobrazuje její aktuální podobu před samotným přidáním. Elementy sítě navíc rozšiřuje o takzvaná ukotvení. Ukotvení jsou body umístěné po obvodu místa nebo přechodu, ke kterým mohou být hrany připojovány. Je-li hrana takto ukotvena, nejsou skutečné souřadnice jejího zakončení počítány vzhledem ke středu propojeného objektu, ale jsou použity přímo koordináty ukotvení.

4. **ObjectSelector**
   Aby mohly být s konkrétním grafickým objektem prováděny vybrané operace, musí být nejdříve vybrán. Za tímto účelem jsme vytvořili modul *ObjectSelector*. Implementována je funkce jak jednoprvkového výběru, tak i výběr více objektů najednou. Kromě grafického znázornění je v naší aplikaci součástí procesu výběru objektů také jejich funkční obohacení. To je prováděno pomocí zvláštního typu modulů, které nazýváme dekorační moduly. Tuto myšlenku lze obecně demonstrovat na situaci, kdy je žádoucí, aby konkrétní funkcionalita byla dostupná až v okamžiku, kdy je objekt vybrán. Rozhraní dekoračních modulů se skládá ze tří metod.

   Metoda *canDecorateObject* jako svůj jediný argument očekává libovolný grafický objekt. Vrací poté hodnotu *true* nebo *false* v závislosti na tom, zda je tento modul schopen daný objekt dekorovat. Metoda *decorateObject* provádí samotné rozšíření grafického objektu, zatímco metoda *removeDecorations* přidanou funkcionalitu odstraňuje a uvádí objekt do původního stavu.

Proces výběru objektů tedy začíná vyhledáním modulů, které jsou schopny vybraný objekt dekorovat. Následně je tento objekt dekorován všemi takovými moduly. Při jeho deselekci jsou tyto nadstavby odebrány. Jelikož se moduly o tom zda mohou vybrané objekty dekorovat rozhodují v závislosti na jejich typu, vzniká tak potřeba nějak jednotně reprezentovat množinu všech vybraných objektů. Tato skutečnost vedla ke vzniku třídy *PNObjectSet*.

5. **ObjectMoover**
   Jeden z nejkomplexnějších přítomných modulů je *ObjectMoover*. Jak již jeho název napovídá, zprostředkovává logiku přesouvání grafických elementů v plátně. Jedná se o implementaci dekoračního modulu, který vybrané objekty rozšiřuje v závislosti na jejich typu. V případě míst a přechodů, přidává posluchače na události spojené s tažením myší na těchto objektech. Hrany jsou dekorovány tak, že je pro každý jejich bod vytvořen pomocný objekt, jehož tažením může být upravena podoba hrany, nebo dokonce změněny propojené objekty za pomocí modulu *ObjectConnector*. Veškeré provedené změny jsou předány k synchronizaci modulu *PNConstructor*.

6. **Highlighter**
   V určitých situacích může být žádoucí dočasně upravit podobu objektů v plátně. Těmto dočasným úpravám budeme nadále v rámci této práce říkat jednoduše zvýraznění. Jako typický příklad můžeme uvést odlišení vybraných objektů od těch ostatních nebo indikaci, zda je koncový objekt validní při jejich propojování.

   Za účelem správy zvýraznění jsme implementovali modul *Highlighter*. Podporovaná zvýraznění jsou v něm uchovávána uvnitř kolekce typu *Table* pod klíči, které reprezentují typ zvýraznění a typ ovlivněného objektu. Hodnota uchovávaná pod těmito klíči představuje změny, které budou na objekt aplikovány při použití daného typu zvýraznění. Má-li být například jiným způsobem vykresleno označené místo sítě, *Highlighter* vyhledá pod klíči *selection* a *place* objekt změn a ten na místo posléze aplikuje. Při svém překreslování takový objekt využívá informace z předaného zvýraznění.

   Zvýraznění objektů je jev dočasný. *Highlighter* nemá k dispozici mechanismus jak zjistit, kdy konkrétní zvýraznění již není platné. Za to je vždy odpovědná část systému, která jej inicializovala. Při aplikaci jednotlivých zvýraznění je proto generován token, asociovaný se změnami, které byly v jejich rámci provedeny. Tento token je využíván k odebrání aplikovaných zvýraznění.

   V průběhu práce s aplikací může nastat situace, kdy je potřeba aplikovat nové zvýraznění na již upravený objekt. *Highlighter* si proto interně uchovává historii aplikovaných zvýraznění na konkrétních objektech. Je-li zvýraznění na objektu odebráno, je tato historie prohledána a v případech, kdy je nalezen odpovídající záznam je aplikováno zvýraznění předešlé.

4.2.3.5 Paleta nástrojů

Nástroje poskytují mechanismus pro aktivaci a deaktivaci konkrétní množiny dovedností implementovaných v prostředí *EaselCanvasManagera*. Mají přístup pouze ke dvěma jeho vnitřním komponentám a sice správci událostí a sběrnici modulů. V daném okamžiku může být aktivní pouze jeden nástroj. Pří své aktivaci zpravidla inicializuje posluchače na zainteresované události, ve kterých využívá moduly systému k provedení požadované operace.

Jako příklad nástroje uvedeme třídu *PlaceTool*. Tento nástroj po své inicializaci reaguje například na událost kliknutí na pozadí scény. Pokud tato událost nastane, je pomocí modulu *ObjectInsertor* vloženo na danou pozici nové místo sítě. Každý z nástrojů může události interpretovat odlišně anebo je zcela ignorovat.

Aktivace vybraného nástroje je prováděna uvnitř komponenty *ToolManager*, který stejně jako ostatní sběrnice obsahuje všechny registrované nástroje. Ty jsou do něj při jeho inicializaci načítány pomocí třídy *ToolLoader*. Výběr aktivního nástroje provádí uživatel aplikace.

4.2.4 Navigace v plátně

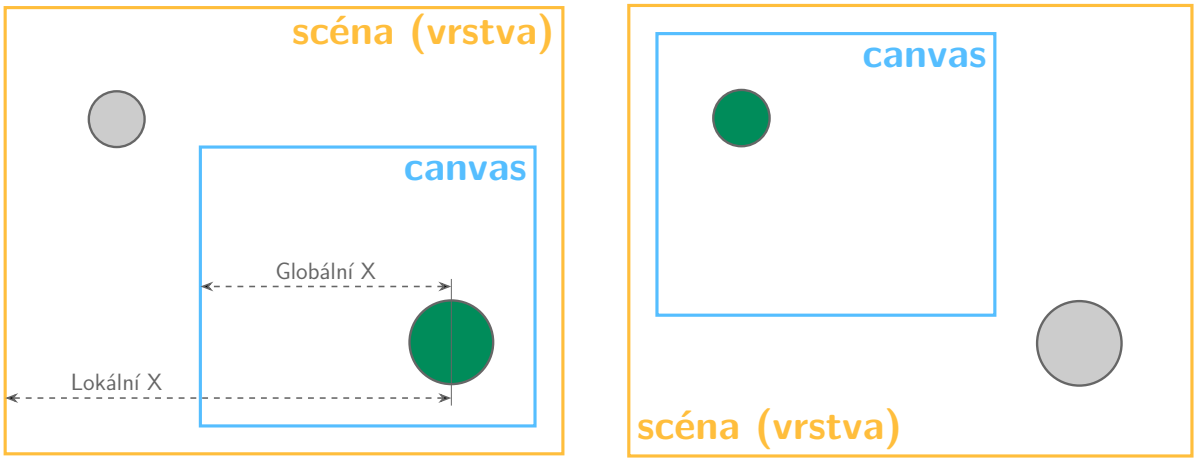
Velikost oblasti určené pro grafický editor je v HTML stránce omezená. Plocha plátna tak nemusí být schopna zobrazit celou scénu najednou. První řešení problému, které připadá v úvahu, je automatická úprava rozměrů samotného canvas elementu. Jelikož ale není žádoucí spolu s plátnem expandovat zbylou část uživatelského rozhraní, muselo by se plátno nacházet v obalujícím elementu, který by při přetečení jeho obsahu zobrazil posuvníky. U tohoto přístupu se ve spojení s použitými frameworky vyskytovaly komplikace. Rozhodli jsme se proto implementovat vlastní mechanismus, který v důsledku reprezentuje potenciálně nekonečně velikou scénu při zachování konstantních rozměrů plátna. Tato funkcionalita je obsažena v komponentě s názvem *CanvasNavigator*.

4.2.4.1 Změna zobrazené oblasti scény

Souřadnice grafických objektů jsou vzhledem ke kontejneru, ve kterém jsou umístěny, považovány za lokální. Jejich globální pozice je vypočtena s ohledem na pozici všech kontejnerů, ve kterých se daný objekt nachází. O tom, zda-li bude objekt na plátno vykreslen, rozhoduje právě jeho globální pozice. Tuto situaci demonstruje příklad na obrázku 7a.

Počátek souřadnicového systému je u canvas elementu i EaselJS kontejnerů umístěn v levém horním rohu. Šedý objekt není na plátno vykreslen jelikož jeho globální souřadnice neleží v intervalech stanovených velikostí plátna. Jedna z užitečných vlastností kontejnerů je ta, že při transformaci jejich souřadnic jsou automaticky upravovány i globální koordináty všech objektů v nich obsažených. Tato vlastnost nám umožňuje implementovat změnu zobrazené oblasti scény jako jednoduché posuvy jejích vrstev. Myšlenku zachycuje obrázek 7b, kde jsou role zobrazených objektů oproti situaci na obrázku 7a vyměněny.

Ve scéně mohou existovat vrstvy, které by neměly být operacemi navigátora ovlivněny. Pokud je konkrétní vrstva zamýšlena jako kontejner pro například pomocné objekty, není žádoucí měnit



(a) Před změnou pozice kontejneru

(b) Po změně pozice kontejneru

Obrázek 7: Vztah mezi globálními a lokálními koordináty objektů ve scéně

jejich pozici spolu se změnami zobrazené oblasti zbylé scény. V registračním záznamu vrstev je za tímto účelem definován atribut *navigationEnabled*. Dalším důležitým faktorem je stanovení hranic, ve kterých se může pozice vrstev pohybovat. Pokud by toto omezení neexistovalo, mohl by uživatel posunout vrstvy tak, že by už nikdy nemusel být schopen jejich obsah nalézt.

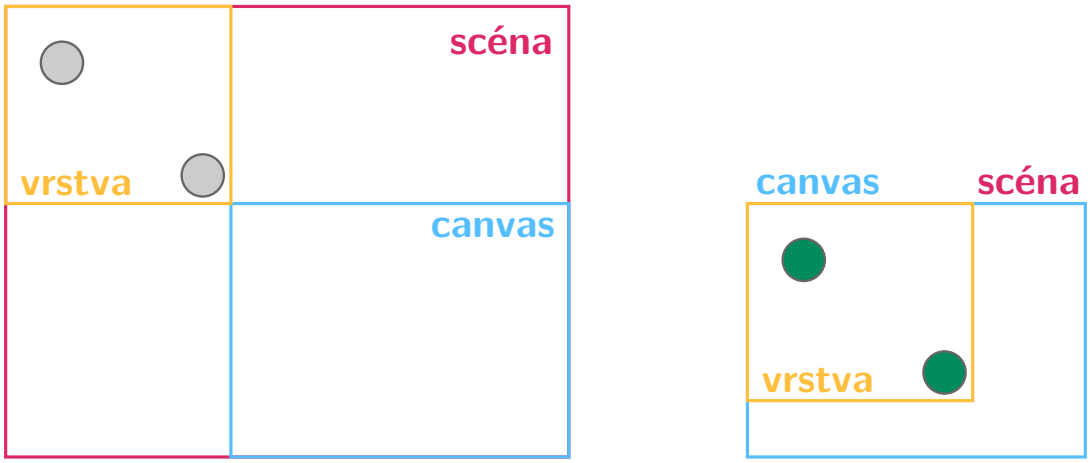
Hranice jsou vypočteny z rozměrů navigátorem ovlivňovaných vrstev. Jelikož kontejnery v EaselJS frameworku představují bezrozměrné kolekce, je jejich pomyslná velikost určena objekty, které obsahují. Pro pozici vrstev jsme definovali tato dvě kritéria:

- souřadnice levého horního bodu vrstvy musí být vždy menší nebo rovny nule,
- souřadnice pravého spodního bodu vrstvy musí být vždy větší nebo rovny nule.

Dvě extrémní situace, které výše uvedeným pravidlům vyhovují jsou zobrazeny na obrázku 8. V jeho variantě 8a lze také pozorovat celkovou velikost zobrazené scény, která je rovna součtu maxima z rozměrů vrstev a šířky respektive výšky plátna. Na obrázku 8b scéná splývá s plátnem. Je-li změněna množina objektů, které vrstva obsahuje, aktualizuje se také její hranice. Pokud bychom přidali objekt do kontejneru na obrázku 8a, budou hranice vrstvy posunuty směrem dovnitř plátna, což umožní jeho další posun ve směru doleva. Tuto skutečnost jsme měli na mysli, když jsme plátno označili za potenciálně nekonečné.

4.2.4.2 Úroveň přiblížení pohledu na plátno

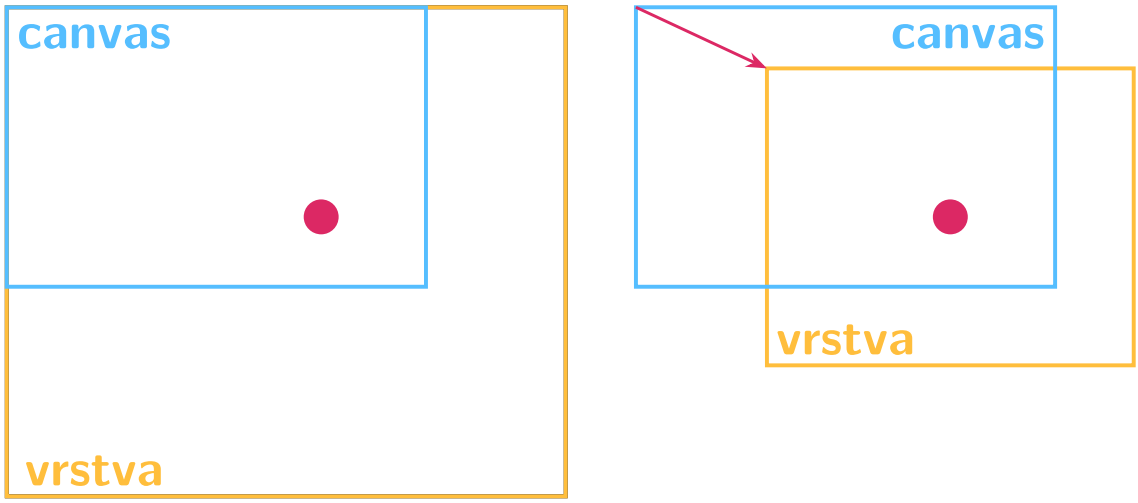
Velikost zobrazené oblasti scény lze ovlivňovat také změnou úrovně přiblížení pohledu na plátno neboli takzvaným zoomováním. EaselJS tuto funkcionalitu umožňuje simulovat pomocí měřítek, která jsou u každého grafického objektu dostupná pod atributy *scaleX* a *scaleY*. Stejně jako u transformací pozice je i úprava měřítka kontejnerů propagována dále na jeho obsah. Řešení pro zoomování se tedy nabízí podobné jako u posuvů scény. Měřítko je měněno v rámci povoleného intervalu pouze u vrstev ovlivňovaných navigátorem.



(a) Povolené minimum (b) Povolené maximum

Obrázek 8: Extrémní případy pozice vrstev vzhledem k plátnu

V důsledku změny velikostí objektů při zoomování jsou také nevyhnutelně aktualizovány rozměry vrstev, ve kterých jsou tyto objekty situovány. Jelikož je změna úrovně pohledu prováděna vzhledem ke konkrétnímu středovému bodu, může nastat situace, kdy daná vrstva po provedení zoomu porušuje podmínky pro její přípustnou pozici. V takovém případě je ještě dodatečně prováděna korekce. Popsaná situace je uvedena na obrázku 9. Červená tečka reprezentuje středový bod, ke kterému byla operace provedena, úsečka zase vzniklou chybu v pozici vrstvy.



(a) Před změnou úrovně pohledu (b) Po změně úrovně pohledu

Obrázek 9: Chyba pozice vrstvy po změně úrovně pohledu na scénu

4.2.4.3 Indikátory pozice zobrazené oblasti

Nepostradatelnou součástí navigace ve scéně je indikace toho, jaká část z celku je aktuálně zobrazena. Tyto indikátory jsme implementovali po vzoru klasických posuvníků a skládají se

tedy ze dvou částí. První část představuje tělo indikátoru a slouží jako kolejnice pro část druhou. Druhá komponenta vystupuje v roli táhla, jejíž změny pozice ovlivňují aktuálně zobrazenou oblast scény a naopak.

Poměr velikosti táhla k velikosti celého indikátoru je roven poměru velikosti plátna k rozměrům celé scény. Pozice táhla v rámci indikátoru je vypočtena obdobně vzhledem k pozici vrstev ve scéně. Indikátory *CanvasNavigator* přidává do scény dva: vertikální a horizontální. Jsou umístěny ve zvláštní vrstvě situované nade všemi ostatními. Tento kontejner je také dobrým příkladem vrstvy, u které není žádoucí provádět operace posuvů a zoomování. Vrstva je tak ve sběrnici registrována s hodnotou atributu *navigationEnabled* = *false*.

4.2.5 Simulace chování Petriho sítí

Součástí většiny nástrojů uvedených v přehledu 3 je simulátor chování Petriho sítě. Přestože taková simulace nemůže plně nahradit analytické metody v oblasti sběru informací o modelu, je neocenitelným nástrojem pro vizualizaci a snadnější pochopení jeho chování. Simulátor je nejčastěji implementován v podobě takzvané hry tokenů. Stejný formát jsme využili i ve zde popisované aplikaci. Termín hra tokenů (Token game) je vysvětlován například v [7]. My jej zde využíváme na základě jeho výskytů v popisech simulačních dovedností ostatních nástrojů pro práci s Petriho sítěmi.

4.2.5.1 Režimy EaselCanvasManagera

Početnou část komponent potřebných pro implementaci simulátoru jsme měli k dispozici už v okamžiku jeho návrhu. Síť je zobrazena ve stejné podobě jako při jejím modelování. Rozdíl je pouze v množině dostupných operací, které je s modelem možné provádět. Využili jsme tak vysoké škálovatelnosti *EaselCanvasManagera* a rozšířili jej o takzvané režimy. Režimy lze chápat jako množinu přípustných stavů, ve kterých se manažer může nacházet.

Spolu s každým režimem jsou vázány konkrétní vrstvy, moduly a nástroje, které definují množinu dovedností, jež manažer v daném stavu uživateli poskytuje. Registrace těchto komponent tak byla rozšířena o atribut *modes*. Jedná se o pole s názvy všech režimu, ve kterých má být komponenta k dispozici. Není-li tento atribut uveden, je komponenta využívána vždy. Při změně režimu jsou znovu inicializovány sběrnice uvedených komponent a to takovým způsobem, kde:

- komponenty, které jsou v současném režimu aktivní, ale nejsou registrovány pro nový stav manažera, jsou odebrány,
- komponenty, které nejsou v současném režimu aktivní, ale jsou registrovány pro nový stav manažera, jsou inicializovány a přidány do sběrnice,
- komponenty, ktere jsou v současném režimu aktivní a jsou současně registrovány pro nový stav manažera, jsou ponechány beze změny.

4.2.5.2 Simulační moduly

Implementace *EaselCanvasManagera* pro práci s Petriho sítěmi připouští celkem dva režimy: modelování a simulaci. V simulačním módu zůstávají aktivní moduly *PNConstructor* a *Highlighter*. O veškerou aplikační logiku, spojenou se simulací chování sítě, se stará speciálně navržený modul *SimulationPerformer*. Simulaci je možné provádět krokově (manuálně a automaticky) nebo dávkovaně. Jeden krok simulace tvoří následující posloupnost operací:

1. identifikace proveditelných přechodů a jejich zvýraznění,
2. výběr konkrétního přechodu z množiny proveditelných přechodů,
3. výpočet nového značení po provedení přechodu z předchozího bodu,
4. aplikace nového značení na model Petriho sítě.

Přechod, který bude proveden vybírá uživatel nebo je zvolen v případě automatického krokování náhodně. Provedené kroky simulace jsou uchovávány v historii a lze se k nim vracet. Simulaci je také možné kdykoliv uvést do počátečního stavu.

4.2.5.3 Dávkovaná simulace

Kromě krokované simulace umožňuje editor provádět také konkrétní počet kroků najednou v rámci simulační dávky. Oproti krokovému přístupu je uživatelské rozhraní aktualizováno až po dosažení finálního značení. Simulační dávka končí v okamžiku, kdy byl proveden zadaný počet kroků nebo se v průběhu simulace dostala síť do stavu uzamčení.

Maximální počet kroků, které lze najednou provést je omezen a to z důvodu výpočetních možností webových prohlížečů. Vykonání simulační dávky může být časově náročná operace. JavaScript jakožto programovací jazyk nepodporuje koncept více vláken a tudíž nelze tento výpočet provádět na pozadí při současném zachování responsivity uživatelského rozhraní. Asynchronní volání tento problém také nevyřeší, bude-li výpočetní kód prováděn v rámci jediného funkčního bloku.

V prostředí JavaScriptu provádí instrukce jediné exekuční vlákno. Jednotlivá volání jsou řazena do fronty, odkud toto vlákno instrukce odebírá a následně je provádí. V okamžiku, kdy by tento proces dospěl do fáze, ve které bude prováděn kód pro výpočet simulační dávky, jeví se aplikace jako zamrzlá. Žádné další instrukce nemohou být zpracovány (včetně těch reprezentující interakci s uživatelským rozhraním), dokud není dokončena právě probíhající úloha.

Tento problém řešíme rozdělením simulační dávky na menší části, které jsou řazeny do fronty instrukcí s časovým odstupem. Je tak zachována responsivita uživatelského rozhraní a umožněno například přerušení simulační dávky uprostřed jejího vykonávání. Přistup rozdělení, který využíváme, s sebou ale nese malou časovou prodlevu mezi vykonáním jednotlivých částí výpočtu. Celková prodleva roste v závislosti na počtu kroků v dávce.

4.2.6 Správa projektů

Každá Petriho síť je v systému reprezentována jako samostatný projekt. Implementovali jsme podporu pro veškeré standardní CRUD operace. Aplikační logiku nad projekty realizuje několik různých objektů. Na vrcholu jejich pomyslné hierarchie se nachází třída *ProjectManager*, která se zaměřuje především na operace spojené s uživatelským rozhraním nástroje. Skrze tento objekt je tak například zobrazován seznam dostupných projektů.

Samotná aplikační logika této oblasti je ponechána na třídu *ProjectPersister*. Hlavní funkcionalita pro práci s projekty je obsažena v implementacích rozhraní *NetPersister*. Každá taková implementace představuje jedno konkrétní úložiště projektů a definuje celkem čtyři metody:

- metoda *loadNetList* slouží pro načtení projektů, které jsou v daném úložišti uchovávány,
- konkrétní Petriho síť je načtena pomocí metody *loadNet*,
- ukládání je prováděno voláním metody *saveNet*,
- projekty lze odstranit užitím metody *deleteNet*.

ProjectPersister představuje prostředníka mezi manažerem projektů a implementovanými úložišti. Při volání konkrétní operace na vybraný projekt je vyhledán adekvátní *NetPersister* a požadavek je dále delegován na něj. Před uložením Petriho sítě je její objekt rozšířen o informace nezbytné pro její zpětné načtení.

Součástí systému jsou dvě různá úložiště projektů. První z nich využívá technologii HTML5 Local Storage, která uchovává data uvnitř webového prohlížeče. Projekt je dostupný pouze tehdy, kdy uživatel k práci s aplikací využívá stejnou instalaci prohlížeče, jako ve které projekt uložil. Projekty jsou ale ztraceny v okamžiku, kdy uživatel vymaže data prohlížeče. Výhoda tohoto přístupu spočívá v jednoduchosti jeho použití. Druhé úložiště je realizováno v podobě databázového serveru. Vzhledem k velikosti grafů Petriho sítí nevyužíváme speciální databáze určené k uchovávání grafů, ale objekt sítě ukládáme v podobě BLOBu. Databázový server a aplikace, která nad ním zprostředkovává rozhraní, fungují jako třetí samostatná součást systému. Toto úložiště jsme implementovali v základní podobě, kde přístup k uloženým projektům není vázán na konkrétního uživatele. Libovolné další úložiště lze do aplikace zakomponovat jako novou implementaci rozhraní *NetPersister*.

4.2.6.1 Importy & Exporty

Kromě načtení z úložiště je možné Petriho sítě do systému také importovat. Tuto operaci provádí objekt typu *PNImportManager*, který pracuje velice podobně jako třída *ProjectPersister*. Logiku importu vykonávají implementace rozhraní *PNImporter* uchovávané uvnitř import manažera. Vstupní data musí být v textové podobě.

Po načtení souboru je nejdříve provedena identifikace importéru, který je schopen tento soubor zpracovat. *PNImporter* za tímto účelem deklaruje metodu *canImportFile*. V okamžiku, kdy

je nalezen vhodný importér, proces pokračuje provedením samotného importu pomocí metody *parsePetriNet*. Tato metoda vrací instanci třídy *PetriNet* sestavenou z dat předaných v jejím argumentu.

Modely Petriho sítí je kromě importů aplikace schopná také exportovat. Hierarchie objektů realizujících tuto funkci je srovnatelná s tou pro importování. Mezi podporované formáty pro import patří nativní formát, který je v systému využíván pro reprezentaci Petriho sítí, a PNML. Množina formátu pro export je oproti importu rozšířena o TikZ.

4.2.7 Prezentace výsledků analýzy

Analýza je prováděna ve zvláštním režimu webového klienta. Pracovní prostor je rozdělen na dvě poloviny, kde první zobrazuje editor modelu sítě, zatímco ta druhá slouží pro vizualizaci výsledků jeho analýzy. Prostor pro vizualizaci výsledků je tvořen panely v podobě přepínatelných karet, které korespondují s konkrétní užitou analytickou metodou a jsou zobrazeny v závislosti na nastavení analýzy. První vždy zobrazený panel je vyhrazen pro výsledky rozhodování vlastností modelu.

Užitá distribuce prostoru aplikace umožňuje jednoduché znázornění výsledků analýzy přímo v modelu Petriho sítě. Jako příklad uvedeme zvýraznění nalezených cyklů v sítí nebo zobrazení vybraného značení z grafu jejího stavového prostoru. Pro zachování přehlednosti uživatelského rozhraní jsou výsledky analýzy často zobrazeny v podobě seznamu. Detaily jeho položek lze zobrazit v podobě samostatného dialogu.

Před provedením analýzy má uživatel možnost specifikovat množinu metod, které budou při analýze modelu aplikovány a dále jaké vlastnosti sítě mají být ověřovány. Každá z analytických metod může obsahovat vlastní parametry. Veškeré toto nastavení je spolu s modelem Petriho sítě odesíláno na server.

4.2.7.1 Vizualizace stavového prostoru

Jedna z implementovaných analytických metod konstruuje stavový prostor Petriho sítě. Tuto množinu dosažitelných značení lze graficky zobrazit v podobě orientovaného grafu, jehož tvorbě a využití se věnujeme v kapitole 5.1. Vizualizace stavového prostoru je stejně jako editor modelu Petriho sítí realizována nad canvas elementem. Logiku spojenou s vykreslováním a interakcí uživatele s grafem obstarává zvláštní implementace třídy *EaselCanvasManager*.

Tento potomek definuje vlastní množinu grafických objektů, vrstev, událostí a modulů. Nejdůležitější funkce jsou obsaženy v modulu *SSGraphBuilder*, který provádí konstrukci grafických objektů stavového prostoru a převody mezi jeho stromovou a grafovou reprezentací. Operace spojené s výběrem konkrétního značení poté zprostředkovává modul *MarkingSelector*. Při výběru konkrétního značení je daný stav sítě promítán přímo do jejího modelu.

Distribuce tokenů v místech sítě může být zobrazena také jako seznam v samostatném dialogu. Z důvodů snadnějšího čtení grafu jsou jeho uzly barevně odlišeny dle jejich významu. Dostupný je také mechanismus navigace ve scéně, jak byl popsán v kapitole 4.2.4.

4.2.8 Layoutování Petriho sítí

Grafická podoba Petriho sítě je závislá na editoru, ve kterém je její model zobrazen. Rozmísťování elementů provádí při jeho tvorbě uživatel. Analytické nástroje jsou na grafické podobě Petriho sítě nezávislé a pracují pouze s jejich objektovou nebo matematickou reprezentací. Modely sítí proto nemusí obsahovat žádné informace o podobě komponent a jejich vzájemném uspořádání. Pokud má být takový model zobrazen, musí existovat mechanismus, který jeho prvky automaticky rozmístí do struktury, která by měla být co nejlépe čitelná pro člověka. Takové rozmístění budeme nadále v této práci označovat jako layout.

Některé nástroje uvedené v kapitole 3 v sobě mají tuto funkcionalitu zakomponovánu. Většinou ovšem k modelu sítě přistupují jako k obecnému grafu. Jedna z takových aplikací je Renew. Přístup pro tvorbu layoutu je v tomto nástroji založen na principu přitažlivých a odpudivých sil mezi částicemi. O vhodnosti jeho použití pro Petriho sítě si lze učinit představu pohledem na obrázek v příloze B. V [13] je uvedeno několik estetických kritérií, které by měl splňovat ideálně navržený layout. Jedná se například o:

- minimalizaci křížení hran a oblasti, kterou graf zabírá,
- omezení počtu ohybů v hranách grafu,
- dodržení minimální velikosti úhlu, které mezi sebou dvě hrany svírají.

Obecně není možné všechna tato kritéria naplnit současně. Algoritmy jsou proto vyvíjeny pro konkrétní typy grafů s cílem naplnit vybranou množinu kritérií a to často v omezeném rozsahu. V případě Petriho sítí se jako nejvhodnější řešení jeví kategorie hierarchických layoutů, které komponenty grafu umisťují dle schématu představeného Sugiyamou v [32].

4.2.8.1 Layout API

Funkci layoutování grafů vykonává další ze součástí *EaselCanvasManagera*. Jedná se o třídu *LayoutPerformer*, do níž jsou objekty registrovány stejným způsobem, jaký je využit u ostatních sběrnic. Kromě spouštění layout algoritmů je v kompetencích této součásti také validace použitého rozvržení komponent, čehož využíváme při importu Petriho sítí. Pokud její dosavadní layout není vyhovující, je aplikován jeden z implementovaných v systému.

Layout algoritmus je v aplikaci reprezentován třídou implementující rozhraní *CanvasLayout*. Toto rozhraní deklaruje metody pro samotné vykonávání algoritmu a také pro získání výchozích hodnot jeho nastavení. Kromě těchto algoritmů jsou při inicializaci *LayoutPerformera* dynamicky načítány také objekty, zodpovědné za provádění již zmíněné validace modelů.

Pro layoutování Petriho sítí využíváme Javascriptovou knihovnu Dagre. Její hlavní výhodu představuje nezávislost na použité technologii pro vykreslování grafů. Zakomponování této metody v kontextu webového klienta znamenalo pouze vytvořit adaptér mezi naší reprezentací Petriho sítě a grafovou strukturou, kterou knihovna využívá. V okamžiku, kdy algoritmus skončí,

jsou souřadnice uzlů a body hran jeho grafové reprezentace aplikovány zpětně na model Petriho sítě spolu s drobnými korekcemi. Dagre implementuje hierarchický layout dle dříve zmíněné Sugiyamovy metody, která se skládá z několika kroků:

1. vstupní graf je transformován do jeho acyklické podoby změnou orientace vybraných hran,
2. vrcholy grafu jsou rozděleny do vrstev tak, aby hrany vedly pouze z vyšších vrstev do těch nižších,
3. hrany přesahující několik vrstev jsou rozděleny na menší segmenty. Každý průsečík hrany s pomyslnou horizontální čarou vrstvy je nahrazen pomocným vrcholem grafu,
4. mezi jednotlivými vrstvami je redukováno křížení hran úpravou pořadí jejich vrcholů,
5. pro každý vrchol grafu jsou vypočteny jeho souřadnice. Hrany, jejichž orientace byla v úvodním kroku algoritmu obrácena, jsou navráceny do původního stavu. Pomocné vrcholy jsou odstraněny a jejich koordináty jsou přidány mezi body příslušných hran.

Problémy spojené s každým krokem této metody jsou výpočetně náročné. Proto pro ně byly vyvinuty efektivní heuristiky, nicméně je praktičnost algoritmu stále limitována velikostí vstupního grafu. Knihovnu Dagre využíváme také pro layoutování grafů pokrytí. Elementy stromu pokrytí rozmísťujeme pomocí Reingold-Tilfordova algoritmu.

4.2.9 Sestavení klientské aplikace

Přestože Javascript spadá mezi interpretované jazyky a není tak nutné kompilovat jeho zdrojové kódy, musí být webová aplikace do své výsledné podoby sestavena. Pod pojmem sestavení označujeme v kontextu webové aplikace například vkládání referencí na Javascriptové soubory a soubory kaskádových stylů do primární HTML stránky (index.html). Důležitým faktorem je pořadí, ve kterém jsou odkazy vloženy. U projektů většího rozsahu jsou operace spojené s jejím sestavením časové náročné a náchylné k chybám.

Webový klient popisovaného systému je sestavován pomocí nástroje GUL. Gulp je ve své podstatě pouhý exekutor úloh v prostředí *Node.js*. Jeho hlavní síla spočívá v možnosti jejich řetězení, kde je výsledek jedné úlohy předán jako vstup úloze další. Dostupná je také bohatá množina zásuvných modulů.

V rámci této práce jsme vytvořili úlohu, která umožňuje aplikaci sestavit ve dvou režimech. V režimu zamýšleném pro použití v ostrém nasazení jsou zdrojové soubory Javascriptu i kaskádové styly minifikovány a spojeny do samostatných souborů. Hlavní HTML stránka poté obsahuje reference na tyto obsáhlé soubory. Druhý režim je zamýšlen pro vývoj aplikace, kdy jsou zdrojové soubory do stránky vkládány ve své originální podobě. Kromě již zmíněných dovedností v sobě úloha zahrnuje také například transformaci z SASS na CSS. Za účelem správy externích knihoven a komponent využíváme nástroj Bower.

4.3 Analytický server

Druhou z hlavních komponent vyvinutého systému je výpočetní server, jehož úkolem je analyzovat modely Petriho sítí. Skutečnosti, že server v systému vystupuje jako samostatný prvek, s sebou přináší několik výhod:

1. **Nezávislost na klientech**
   Analýza je prováděna nad objektovou reprezentací modelu Petriho sítě. Kompletně tak odpadá potřeba její vizualizace a tudíž není nezbytná ani existence aplikace obsahující grafický editor. Klient tak může být implementován jako obyčejná konzolová aplikace, která model sítě načte z nějakého úložiště a s výsledky jeho analýzy naloží po svém.

2. **Rozšířitelnost**
   Dovednosti serveru mohou být jednoduše rozšířeny, aniž by vznikla potřeba okamžité aktualizace klientských aplikací. Pokud byla zachována kompatibilita komunikačního rozhraní, mohou existující klienti ke své práci server využívat stejným způsobem jako před jeho aktualizací. Upraveni jsou až v okamžiku, kdy vzniká požadavek na začlenění nových dovedností serveru.

3. **Distribuce zátěže**
   Pokud výpočetní operace provádí samostatná komponenta, může jich v systému figurovat více najednou. Součástí takového prostředí je mechanismus, který v případech, kdy je jeden z výpočetních uzlů příliš zatížen, přesměrovává požadavky na jeho méně vytížený protějšek.

Analytický server jsme vytvořili pomocí programovacího jazyka Java. Objektová reprezentace Petriho sítě je na serveru takřka totožná s tou popsanou v kapitole 4.2.2. Třídy, reprezentující její elementy, jsou pouze jinak pojmenovány a k uchovávání informací o hranách byla využita přímo kolekce *com.google.guava.collections.Table* na rozdíl od její Javascriptové adaptace.

4.3.1 Komunikační rozhraní

Server své analytické dovednosti zpřístupňuje v podobě RESTful webové služby, kterou klienti konzumují. V prostředí REST architektury jsou data a funkcionalita považovány za zdroje, ke kterým se přistupuje pomocí URI (typickým příkladem jsou webové odkazy). Se zdroji se manipuluje s využitím několika jednoduchých dobře definovaných operací. Architektonický styl REST definuje aplikační architekturu jako architekturu klient/server a je navržen s cílem využit bezstavové komunikační protokoly, jakým je například HTTP [24].

JAX-RS je standard, který byl definován se záměrem zjednodušit vývoj RESTful webových služeb v Javě. Webovou službu jsme na serveru vytvořili pomocí frameworku Jersey, který slouží jako referenční implementace JAX-RS standardu. Dostupné API navíc obohacuje s cílem ještě více usnadnit práci při vývoji RESTful webových služeb [5].

4.3.1.1 Zdroje analytického serveru

Zdroje, které webová služba zpřístupňuje, jsou v rámci Jersey frameworku reprezentovány POJO třídami označenými anotací *@Path*, které obsahují alespoň jednu stejně anotovanou metodu. Hodnota této anotace představuje část URI identifikující konkrétní zdroj v rámci webové služby. Analytický server obsahuje jediný takový zdroj v podobě třídy *NetAnalysis* a metody *analyzeNet*.

Metoda *analyzeNet* jako argument očekává instanci třídy *AnalysisRequest* a vrací objekt typu *AnalysisResponse*. Součástí objektů žádosti je model Petriho sítě spolu s množinou analytických metod, které mají být na tento model aplikovány a seznamem vlastností, které mají být ověřovány. Každé analytická metoda s sebou nese také její nastavení.

4.3.1.2 Konverze datových reprezentací

Server přijímá i odesílá data v JSON podobě. Jersey tento datový formát podporuje pomocí externích modulů. My zde konkrétně využíváme soubor nástrojů pro zpracovávání dat s názvem Jackson. Jeho součástí jsou mimo jiné knihovny pro parsování a generování JSON objektů nebo knihovna pro jejich mapování na Java POJO objekty.

Konverze mezi JSON formátem a Java objekty je prováděna pomocí třídy *ObjectMapper*. Součástí Jacksonu je několik anotací, s jejichž pomocí je možné upravit způsob, jakým knihovna JSON data čte z a zapisuje do Java objektů. S jednoduchými datovými typy si *ObjectMapper* poradí sám. Pokud je ale transformovaná kompozice složitá a obsahuje objekty, jejichž třídy byly definovány v rámci vyvíjené aplikace, je nezbytné logiku jejich serializace a deserializace implementovat manuálně.

Jackson poskytuje možnost rozšířit transformační dovednosti *ObjectMapperu* pomocí abstraktních tříd *JsonSerializer* a *JsonDeserializer*. Jejich potomci jsou shromažďováni v modulech, které jsou dále registrovány uvnitř *ObjectMapperu*. Serializéry a deserializéry musí být asociovány s konkrétní třídou jejíž objekty mají transformovat. Jedním způsobem, jakým lze tuto asociaci provést, je označit třídu speciální anotací. My jsme jejich registraci prováděli s uvedením konkrétního typu. Tento přístup považujeme za vhodnější především u modulárně rozdělené aplikace. Není-li třída anotována, nevzniká u daného modulu ani závislost na knihovnách, ve kterých se definice anotace nachází.

Vlastní serializér jsme museli vytvořit například pro třídu *PetriNet*. S jeho pomocí je automaticky provedena konverze modelu Petriho sítě při konstrukci objektu *AnalysisRequest*. Ze stejných důvodů bylo definováno několik deserializérů, které jsou využívány při odesílání výsledků analýzy.

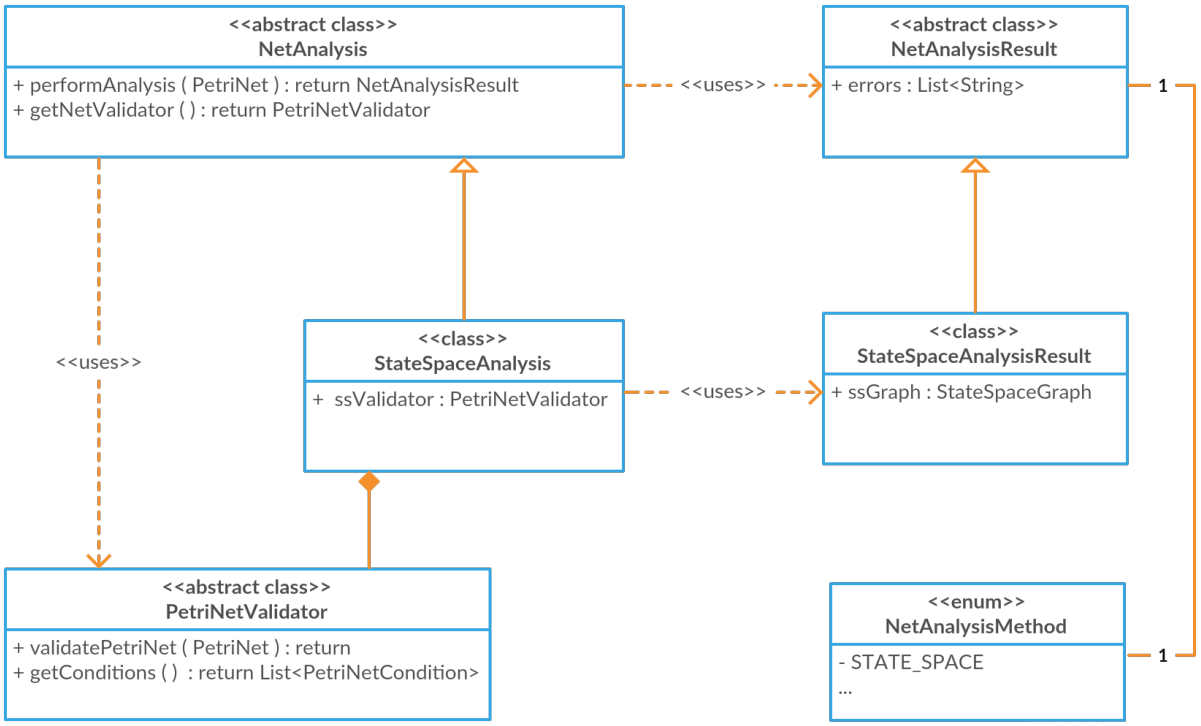
4.3.2 Analytický framework

Proces analýzy modelu Petriho sítě začíná v okamžiku volání metody *analyzeNet*. Z přijatých dat je sestavena žádost o analýzu, která je dále předána ke zpracování třídě *AnalysisRequestProcessor*. Ta daný model analyzuje s ohledem na nastavení v žádosti. Celkový proces analýzy se skládá ze dvou kroků, které nyní podrobněji popíšeme.

4.3.2.1 Provádění analytických metod

V prvním kroku jsou spuštěny zvolené analytické metody. Každá taková metoda je na serveru realizována jako potomek abstraktní třídy *NetAnalysis* a je jednoznačně identifikována pomocí výčtového typu *NetAnalysisMethod*. Objekt *AnalysisRequest* kromě modelu Petriho sítě obsahuje také seznam textových řetězců, které určují množinu použitých analytických metod. Tyto řetězce jsou konvertovány na jednu z *NetAnalysisMethod* konstant, pomocí níž je dohledána adekvátní implementace třídy *NetAnalysis*.

Výsledek analytické metody je reprezentován pomocí abstraktní třídy *NetAnalysisResult*. Ta v sobě obsahuje typ analytické metody a seznam chyb v podobě textových řetězců, které během vykonávání dané metody nastaly. Další atributy jsou v kompetenci konkrétních potomků této třídy. Vypočtené výsledky jsou předány do další fáze analýzy modelu. Vztah mezi uvedenými třídami spolu s ukázkou jedné z jejich implementací zobrazuje diagram na obrázku 10.



Obrázek 10: Diagram tříd využívaných při procesu provádění analytických metod.

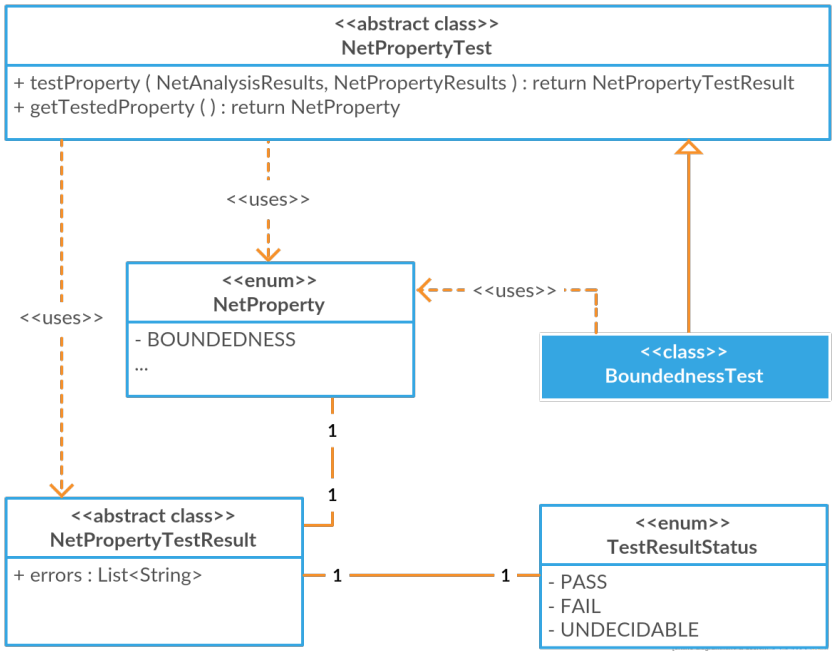
4.3.2.2 Validace modelu Petriho sítě

Před spuštěním každé analytické metody je provedena kontrola validity zkoumaného modelu. Kritéria jeho správnosti se u jednotlivých metod mohou lišit. Definovali jsme proto abstraktní třídu *PetriNetValidator*, jejíž potomci specifikují sadu podmínek, které musejí být splněny, aby byl model označen jako korektní. Každá taková podmínka je reprezentována jako implementace jednoduchého rozhraní *PetriNetCondition*.

Hlavní myšlenka validace spočívá v zastavení propagace chybného modelu v rámci operací prováděných při exekuci analytické metody. Algoritmy tak nemusely být ošetřeny pro tyto krajní případy a při vývoji nám bylo umožněno soustředit se čistě na jejich logiku.

4.3.2.3 Vyhodnocení vlastností Petriho sítě

V okamžiku, kdy jsou vypočteny výsledky analytických metod, proces pokračuje rozhodováním, zda má zkoumaná síť vybrané vlastnosti (uvedené v kapitole 2.7). Framework pro ověřování vlastností funguje na podobném principu jako ten pro provádění analytických metod. Každá vlastnost sítě je opět jednoznačně určena danou konstantou. Výčtový typ, který je k tomuto určen, se nazývá *NetProperty*. Logika ověřování vlastností je poté implementována v potomcích abstraktní třídy *NetPropertyTest*. Jejich vyhodnocování je prováděno na základě výsledků analytických metod a výsledků ostatních vlastností sítě, jejichž ověření bylo provedeno dříve. Vyhodnocování je tedy závislé na pořadí. Vztahy mezi třídami, které popsanou funkcionalitu zprostředkovávají, zachycuje diagram 11.



Obrázek 11: Diagram tříd využívaných při ověřování vlastností Petriho sítě

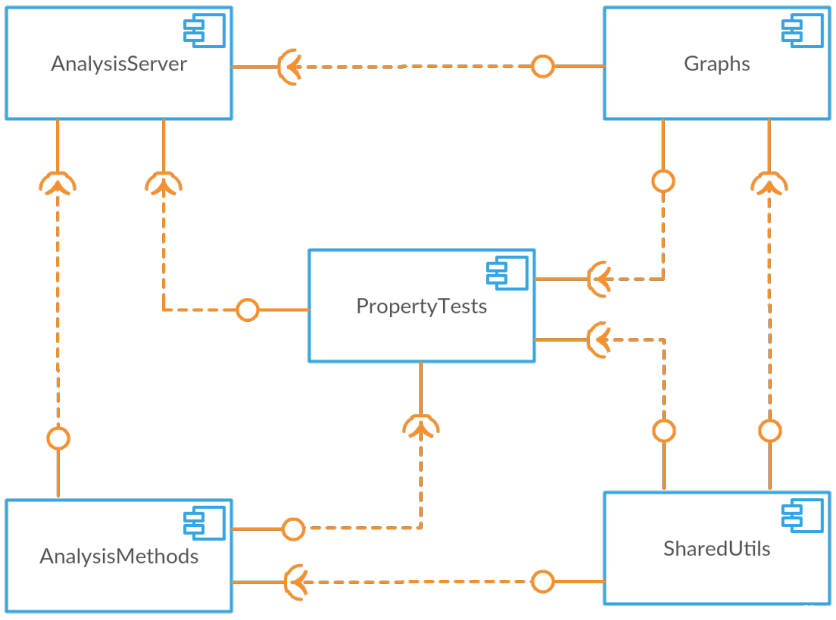
Výsledky ověření vlastností jsou reprezentovány objekty typu *NetPropertyTestResult*. Mezi jejich atributy patří typ ověřované vlastnosti, výsledný status, důvody pro učiněné rozhodnutí a případně seznam chyb, pokud v průběhu testu nějaké nastaly. Finální status je realizován opět pomocí výčtové konstanty a může nabývat celkem tří hodnot. Pokud je možné jednoznačně rozhodnout, zda zkoumaná síť danou vlastnost má, je výsledek roven hodnotě *PASS*. V opačném případě atribut status nabývá hodnoty *FAIL*. Nemá-li framework k učinění rozhodnutí dostatek informací, je výsledný status vlastnosti označen za nerozhodnutelný.

4.3.3 Sestavení serveru

Zdrojový kód analytické aplikace je rozdělen do několika modulů. Ke správě jejich vzájemných a externích závislostí stejně jako k jejímu sestavování využíváme nástroj Maven. Moduly, ze kterých se serverová aplikace skládá jsou:

- *Graphs* - zde jsou situovány třídy, které reprezentují grafové struktury, konvertory mezi jednotlivými grafy a implementace algoritmů nad nimi prováděných,
- *AnalysisMethods* - tento modul obsahuje veškerý zdrojový kód spjatý s prováděním analytických metod,
- *PropertyTests* - zahrnuje třídy spojené s ověřováním vlastností Petriho sítí,
- *SharedAssets* - slouží jako sběrnice pomocných nástrojů pro zbylou část aplikace,
- *AnalysisServer* - reprezentuje samotnou serverovou aplikaci. Tento modul je jako jediný závislý na knihovnách, potřebných pro realizaci komunikačního API.

Toto rozdělení umožňuje snadnou použitelnosti vytvořených modulů v jiných projektech. Například knihovna *AnalysisMethods* není závislá na konkrétní implementaci serveru a lze ji použít v libovolné další aplikaci. Celkovou představu o závislostech mezi jednotlivými komponentami serveru si čtenář může udělat pohledem na obrázek 12. Aplikace je sestavena do podoby webového archivu a vyžaduje pro svůj běh servlet kontejner například Apache Tomcat.



Obrázek 12: Diagram komponent analytického serveru

5 Metody analýzy Petriho sítí

Primárním cílem analýzy Petriho sítí je určit, jaké má zkoumaný systém vlastnosti. Za tímto účelem bylo vyvinuto několik technik, které lze rozdělit do dvou kategorií. První z nich zkoumá síť z hlediska jejího stavového prostoru, zatímco metody spadající do druhé skupiny se zabývají pouze strukturou jejího modelu a jsou proto nezávislé na počátečním značení.

Sílu každé analytické aplikace definuje množina metod, které má při zkoumání modelu k dispozici. Přestože jsou tyto techniky použitelné samostatně, mnohdy je k rozhodování, zda-li má daná síť konkrétní vlastnost, využita kombinace jejich výsledků. V této kapitole se budeme podrobněji věnovat metodám, které jsou součástí analytické části zde popisovaného systému. Pokud to konkrétní technika vyžaduje, rozšíříme teoretickou bázi, ze které jsme při její implementaci vycházeli. V takovém případě jsme informace převzali z [18] nebude-li uvedeno jinak.

5.1 Stavový prostor

První z metod, kterou jsme do analytické aplikace zakomponovali, zkoumá vlastnosti Petriho sítě v závislosti na její množině dosažitelných stavů. Množina dosažitelnosti je označována také jako stavový prostor. Jedna z užívaných grafických reprezentací stavového prostoru se nazývá graf dosažitelnosti.

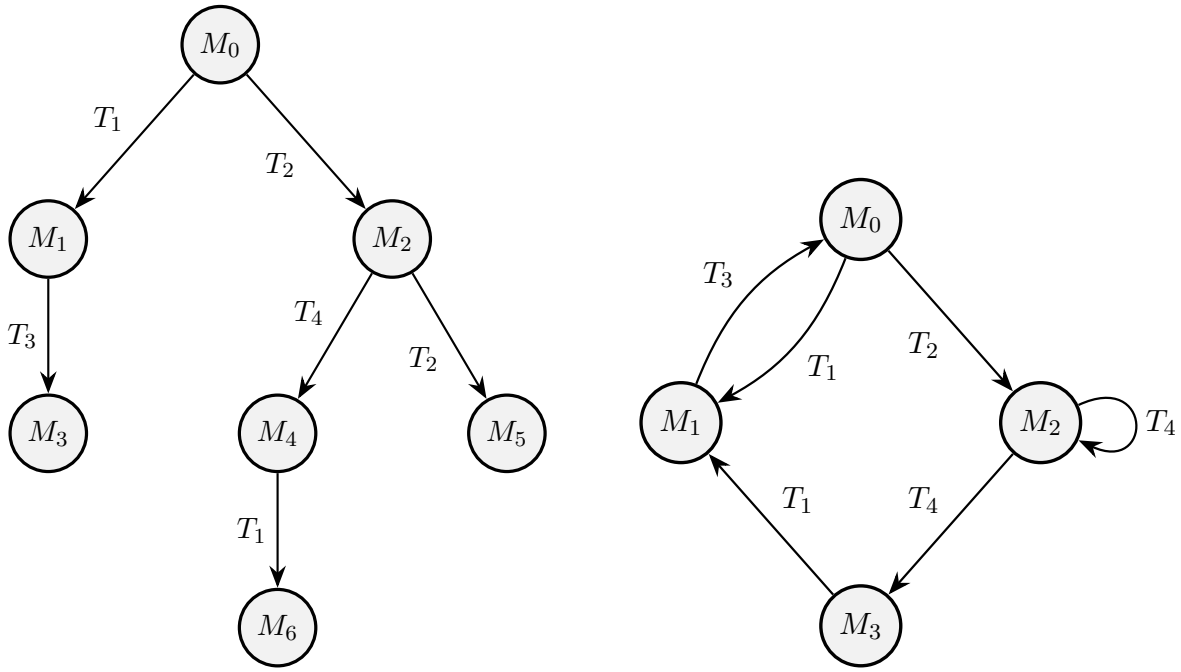
Definice 15 *Graf dosažitelnosti systému Petriho sítě s množinou dosažitelnosti $RS(M_0)$ je hranově ohodnocený orientovaný graf, kde:*

- *$RS(M_0)$ je množina uzlů grafu,*
- *$A \subseteq RS \times RS \times T$ je množina hran grafu, kde platí: $\langle M_i, M_j, t\rangle \in A \Leftrightarrow M_i \xrightarrow{t} M_j$*

Vrcholy grafu dosažitelnosti tedy představují jednotlivá dosažitelná značení sítě zatímco hrany realizují relaci bezprostřední dosažitelnosti mezi nimi. Při konstrukci stavového prostoru je častěji využívána jeho stromová reprezentace nazývaná strom dosažitelnosti. Graf dosažitelnosti a strom dosažitelnosti jsou dvě navzájem ekvivalentní reprezentace stavového prostoru Petriho sítě. Ve stromu dosažitelnosti může být ale stejné značení zastoupeno několika vrcholy, které jsou v grafu dosažitelnosti slučovány. Ukázku obou reprezentací množiny dosažitelnosti znázorňuje obrázek 13.

Jelikož je množina dosažitelnosti neomezených Petriho sítí nekonečná, jsou nekonečné i její grafové reprezentace. Takové sítě nelze na základě jejich grafu dosažitelnosti analyzovat. Z toho důvodu byly a stále jsou vyvíjeny nové techniky, které umožňují aproximovat nekonečnou množinu dosažitelnosti a znázornit ji pomocí konečné grafové reprezentace. Základy této oblasti pokladli Karp s Mullerem v [17], kde představili takzvaný ω symbol, který slouží pro reprezentaci nekonečných komponent vektoru značení.

Množina dosažitelnosti je tak rozdělena na konečný systém tříd takových, že v každé třídě se nacházejí všechna značení, která jsou pokryta určitým omega značením. Omega značením



(a) Strom dosažitelnosti (b) Graf dosažitelnosti

Obrázek 13: Vizualizace ukázkového stavového prostoru

budeme rozumět vektor, jehož alespoň jedna složka je vyjádřena pomocí ω symbolu. Grafová reprezentace množiny dosažitelnosti, s jejíž pomocí jsme schopni zobrazit i stavový prostor neomezených sítí, se nazývá strom pokrytí. Dříve než uvedeme algoritmus jeho konstrukce, upřesníme pojem pokrývání značení a uvedeme operace, které jsou uplatňovány při počítání s ω symbolem.

Definice 16 *Značení M pokrývá značení N, jestliže* $(\forall i)[m_i \geq n_i] \wedge (\exists i)[m_i > n_1]$

Připomeňme, že značení systému Petriho sítě je dáno vektorem $M = (m_1, m_2, m_3, ..., m_{|P|})$, kde P je množina míst sítě.

Definice 17 *Pro počítání s ω symbolem platí:*

- $(\forall k \in N)[\omega + k = \omega = \omega - k]$,
- $(\forall k \in N)[\omega > k]$,

kde N zastupuje množinu přirozených čísel včetně nuly.

Nyní můžeme uvést samotný algoritmus pro konstrukci stromu pokrytí, ze kterého jsme při realizaci této analytické metody vycházeli. Oproti zde uvedenému originálu jsme v naší implementaci provedli několik změn, kterým se budeme podrobněji věnovat v jedné z dalších částí této kapitoly. Původní algoritmus zde uvádíme pro úplnost a jako referenci využívanou při popisu našeho přístupu.

1. Kořenem stromu určíme uzel, který reprezentuje počáteční značení.

2. Pro každé doposud nezpracované značení (uzel stromu) M jsou vypočtena všechna bezprostředně dosažitelná značení $M' \in \{M_r : M \xrightarrow{t} M_r, t \in E(M)\}$ dle následujících pravidel:

   (a) Je-li $m_i = \omega$ pro libovolné $i \in \{1, 2, 3, ..., |P|\}$, pak je také $m'_i = \omega$.

   (b) Jestliže existuje na cestě od kořene stromu včetně do nově vypočteného vrcholu M' značení N, které je značením M' pokrýváno, pak $m'_i = \omega$ pro každé $i \in \{1, 2, 3, ..., |P|\}$ takové že, $m'_i > n_i$.

   (c) V ostatních případech je značení M' vypočteno dle pravidla provedení přechodu $M \xrightarrow{t} M'$

3. Rozvoj větve stromu je ukončen v případech, kdy dané značení M' představuje uzamčení systému $E(M') = \emptyset$, nebo se už stejné značení na cestě od kořene stromu po vrchol M' již vyskytuje.

Analýza stavového prostoru je obecně aplikovatelná na všechny třídy Petriho sítí. Její složitost ovšem roste s mohutností množiny dosažitelnosti. Ta může být i u sítí s malým počtem prvků obrovská a exponenciálně růst v závislosti na počátečním značení. Tento problém bývá označován jako exploze stavového prostoru. Pokud nastává u sítě, kterou chceme analyzovat, ve většině případů se uchylujeme k analýze její struktury.

5.1.1 Interpretace výsledků analýzy stavového prostoru

V okamžiku, kdy je stavový prostor Petriho sítě popsán grafem dosažitelnosti, lze na její analýzu nahlížet jako na analýzu orientovaného grafu. Vlastnosti, které je analytická aplikace schopná pomocí stavového prostoru rozhodovat jsou:

1. **Omezenost**
   O Petriho sítí lze na základě jejího grafu dosažitelnosti tvrdit že je omezená, pakliže je tento graf konečný. Je-li k reprezentaci jejího stavového prostoru využit graf pokrytí, je síť omezená, pokud žádný z vrcholů grafu neobsahuje ve svém značení ω symbol.

   Jelikož je bezpečnost zvláštním případem omezenosti, lze k jejímu rozhodování stavový prostor také využít. Petriho síť je bezpečná, jestliže žádná složka vektoru značení každého uzlu grafu, není větší než jedna.

2. **Živost**
   Petriho síť je živá, jestliže pro každou finální silně souvislou komponentu jejího grafu dosažitelnosti platí, že každý přechod sítě ohodnocuje alespoň jednu hranu této komponenty. Silně souvislá komponenta grafu je definována jako podmnožina jeho uzlů V taková, že z libovolného vrcholu $v \in V$ existuje cesta, vedoucí do každého dalšího uzlu $v' \in V - \{v\}$.

Pokud navíc platí, že $(\forall v \in V)(!\exists u \notin V)[P(v,u)]$, kde P představuje relaci existence cesty z vrcholu v do vrcholu u, je komponenta označována jako finální.

V kontextu grafu dosažitelnosti si lze finální silně souvislou komponentu představit jako množinu dosažitelných stavů, mezi kterými se může síť libovolně přepínat. Změna značení je prováděna přechody, které jsou v grafu zachyceny v podobě ohodnocení hran. Takové přechody jsou v důsledku vždy proveditelné. Z definice živosti 12 poté vyplývá, proč je nezbytné, aby byl součástí ohodnocení hran takové komponenty každý přechod sítě.

Graf dosažitelnosti může obsahovat více než jednu finální silně souvislou komponentu. Pokud se v průběhu rozvoje sítě v takové komponentě ocitneme, nemůžeme ji už nikdy opustit. Aby byla síť živá, musí výše uvedená podmínka ohodnocení hran platit pro každou takovou komponentu.

Za účelem vyhledávání silně souvislých komponent využíváme vlastní implementaci Tarjanova algoritmu. Ověřování, zda je nalezená komponenta rovněž finální, není v jeho režii a je proto prováděno samostatně při vyhodnocování živosti sítě.

3. **Existence uzamčení**
   Uzamčení systému Petriho sítě je definováno jako dosažitelné značení, jehož množina proveditelných přechodů je prázdná. V grafu dosažitelnosti je takové značení zachyceno vrcholem, který nemá žádné následníky. Při vyhodnocování této vlastností sítě, jsou proto takové uzly grafu vyhledávány a pokud je alespoň jeden nalezen, pak obsahuje zkoumaný systém uzamčení.

4. **Reverzibilnost**
   Skutečnost, že je značení M' dosažitelné ze značení M, je v grafu dosažitelnosti zachycena v podobě orientované cesty z vrcholu M do vrcholu M'. Aby byla Petriho síť reverzibilní, musí být dle definice reverzibilnosti počáteční značení vždy dosažitelné. V kontextu grafu dosažitelnosti to znamená, že do uzlu, který reprezentuje počáteční značení, musí vést orientovaná cesta z každého dalšího vrcholu.

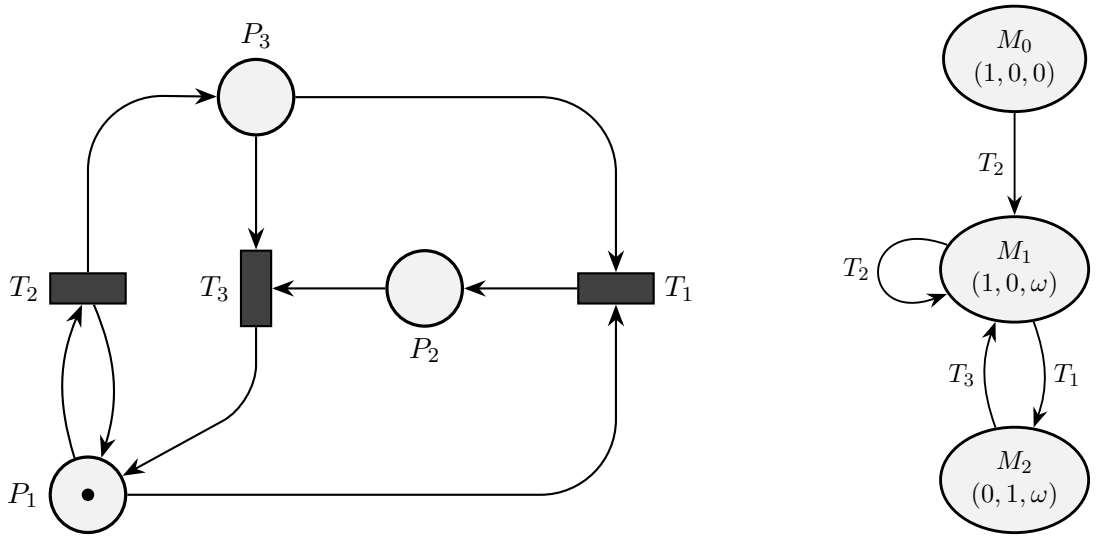
   Důsledek popsané situace je takový, že je celý graf dosažitelnosti silně souvislý. Pro rozhodování zde proto opět využíváme Tarjanův algoritmus. Pokud je jeho výstupem jediná silně souvislá komponenta, je síť reverzibilní.

5.1.2 Stavový prostor neomezených sítí

Analýza určitých vlastností neomezených Petriho sítí, jako jsou živost, reverzibilita nebo dosažitelnost, se ukazuje být problematickou oblastí, která byla a je předmětem stálého výzkumu. Hlavní problém zde představuje nekonečnost stavového prostoru. Pro jeho grafovou reprezentaci tedy nelze použít graf dosažitelnosti, ale je nezbytné využít graf pokrytí. Jelikož graf pokrytí představuje konečnou reprezentaci nekonečně velké množiny značení, dochází k určité ztrátě in-

formací. V důsledku tato reprezentace stavového prostoru nemusí obsahovat dostatek informaci pro správnou analýzu výše uvedených vlastností.

Nejprve se zmíníme o problémech spojených s rozhodováním živosti. Příklad Petriho sítě, u které analýza živosti na základě grafu pokrytí není korektní, je uvedena na obrázku 14a. Uvažovaná síť je očividně neomezená, jelikož kdykoliv je v místě P_1 přítomen alespoň jeden token, je možné prováděním přechodu T_2 generovat libovolný počet tokenů v místě P_3. Pro další analýzu této sítě bude využit její graf pokrytí zobrazen na obrázku 14b.



(a) Model sítě (b) Graf pokrytí

Obrázek 14: Petriho síť č. 1

V počátečním značení je proveditelný pouze přechod T_2, který převádí síť do značení $M_1 = (1, 0, 1)$. Toto značení pokrývá počáteční značení v místě P_3 a tudíž může počet tokenů v tomto místě neomezeně růst. Dle pravidel algoritmu pro generování stromu pokrytí je počet tokenů v místě P_3 ve značení M_1 označen symbolem ω.

Pro vlastní rozhodování živosti je uplatněn stejný postup jako u grafu dosažitelnosti. Výsledný graf pokrytí obsahuje jedinou finální silně souvislou komponentu tvořenou vrcholy M_1 a M_2 a hranami, které je bezprostředně spojují. Vzhledem ke skutečnosti, že ohodnocení těchto hran obsahuje všechny přechody, lze o síti tvrdit, že je živá. To ovšem není pravda.

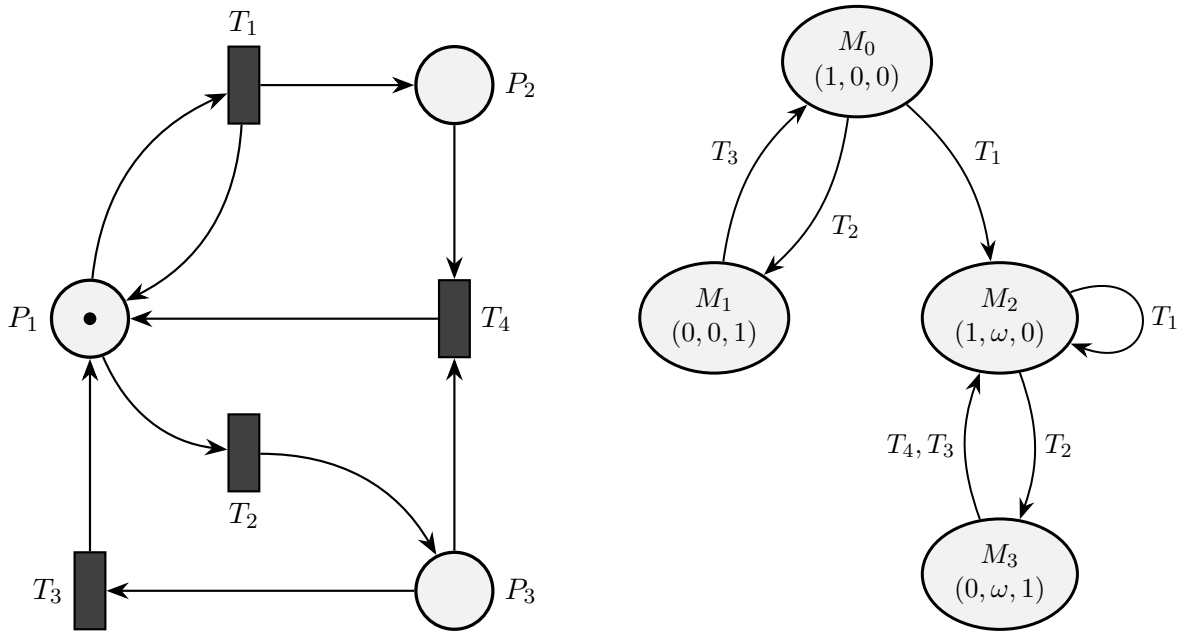
Opomeneme-li při konstrukci stavového prostoru zkoumané sítě pokrývání značení, potom se provedením sekvence přechodů T_2 a T_1 dostáváme do značení $M' = (0, 1, 0)$. Toto značení představuje uzamčení systému. Síť tedy nemůže být živá a tato informace byla v grafu pokrytí ztracena. Značení M' náleží do třídy značení, ve které jsou všechna značení pokryta značením M_2. Hodnota ω byla v místě P_3 převzata při konstrukci stromu pokrytí z rodičovského značení M_1. Problém chybné identifikace je zde způsoben konkrétně hranou grafu vedoucí ze značení M_2 do značení M_1. Přechod T_3 je jistě ve značení M_2 proveditelný, pokud uvažujeme pravidla, která platí pro počítání s ω symbolem. Jak již bylo ale ukázáno, do množiny značení, která značení

M_2 pokrývá, náleží také M', ve kterém přechod T_3 proveditelný není. Živost této sítě není možné z jejího grafu pokrytí správně rozhodnout.

Problémy při analýze reverzibilnosti předvedeme na Petriho sítí na obrázku 15a. Vlastnosti této sítě lze díky její jednoduchosti odečítat přímo z její grafické reprezentace. Opakované provádění přechodu T_1 generuje tokeny v místě P_2 a síť je tedy neomezená. Přechody T_2 a T_3 umožňují neustálou výměnu tokenu mezi místy P_1 a P_3. Pomocí přechodu T_4 lze z místa P_2 odebírat vygenerované tokeny a to až do stavu, kdy je síť uvedena opět do počátečního značení. Jelikož se lze do počátečního značení dostat z každého dosažitelného značení, je uvedená síť reverzibilní.

Graf pokrytí uvažované sítě je zobrazen na obrázku 15b. Aby byla síť reverzibilní, musí být její graf dosažitelnosti (pokrytí) silně souvislý. Při bližším pohledu na uvedený graf pokrytí jsme schopni takové komponenty identifikovat dvě. Síť tedy dle výsledků grafové analýzy jejího stavového prostoru není reverzibilní.

Pokud bychom se rozhodli stavový prostor této sítě znázornit jen částečně grafem dosažitelnosti a neuvažovali tak pokrývání značení, potom bychom se provedením sekvence přechodů T_1, T_2 a T_4 ocitli zpět v počátečním značení. V grafu pokrytí by tuto skutečnost představovala hrana vedoucí z vrcholu M_3 do M_0. Tato hrana se v grafu pokrytí nevyskytuje, protože je informace o konkrétním počtu tokenů v neomezených místech popsána pomocí ω symbolu. Kvůli pravidlům algoritmu pro tvorbu stromu pokrytí, se bude ω symbol vyskytovat vždy alespoň na stejných pozicích vektoru značení M' jako ve značení M, ze kterého bylo M' dosaženo. V důsledku tak v uvedeném příkladu není ze značení M_3 dosažitelné $M_0 = (1, 0, 0)$, ale místo P_2 převezme ω symbol. Provedením přechodu T_4 se tak dostáváme zpět do značení M_2.



(a) Model sítě

(b) Graf pokrytí

Obrázek 15: Petriho síť č. 2

Každá neomezená Petriho síť s konečným počátečním značením bude pomocí grafové analýzy jejího stavového prostoru, označena jako nereverzibilní. Předešlá věta se opírá o fakt, že stavový prostor bude reprezentován grafem pokrytí, který byl zkonstruován pomocí algoritmu popsaného v části 5.1. Z každého ω značení se nelze dostat zpět do žádného značení, které ω symbol neobsahuje. Za předpokladu, že je počáteční značení M_0 konečné (tj. neobsahuje ω symbol), je tedy toto značení nedosažitelné z žádného ω značení. Tato skutečnost vylučuje souvislost grafu pokrytí a tím také možnost, že by síť byla označena jako reverzibilní.

Strom pokrytí, sestavený dle algoritmu v [17], bývá v literatuře označován také jako Finite Reachability Tree (FRT). Výzkum v této oblasti vedl k několika jeho rozšířením. V [39] byl představen takzvaný Modified Reachability Tree (MRT), který namísto ω symbolu využívá pro vyjádření složek vektoru značení výraz $a + bn_i$ navržený Petersonem v [26]. Oproti FRT tak zachycuje více informací a lze jej dle autorů práce využít při rozhodování problémů živosti, dosažitelností nebo existence uzamčení. MRT v sobě ovšem stále zahrnuje nedosažitelná značení, jak bylo ukázáno v [28] a selhává proto při rozhodování problémů dosažitelnosti a existence uzamčení.

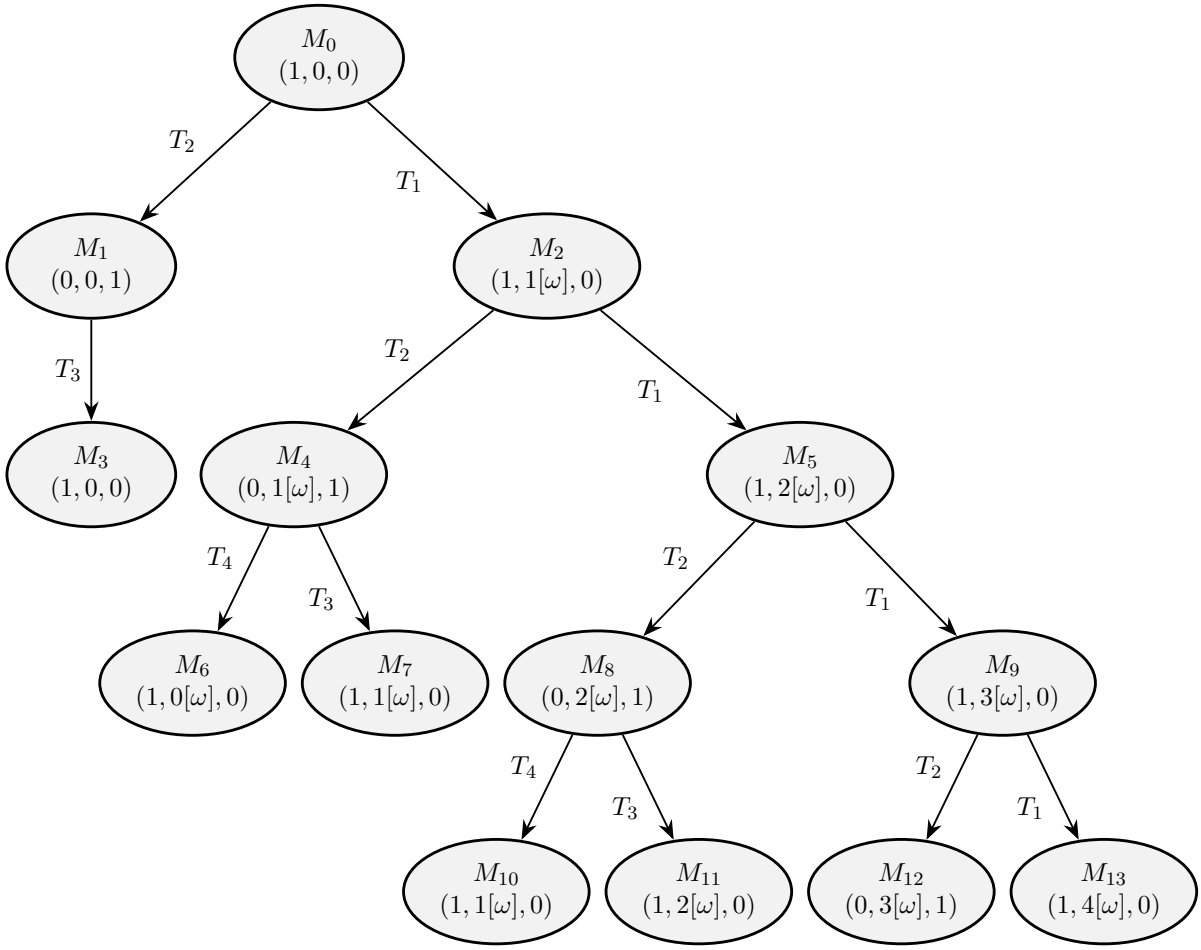
Výzkumné skupiny se často zaměřují na vybrané třídy Petriho sítí. Pro třídu, do které náleží sítě s pouze jediním neomezeným místem, byl v [14] vyvinut takzvaný Augmented Reachability Tree (ART). Jeho hlavní myšlenka spočívá v kalkulaci minimálního značení pro každý vrchol stromu. Kromě živosti lze dle autorů tuto metodu aplikovat také na rozhodování reverzibilnosti. Nevýhodou je však limitovaná oblast její aplikace.

Wang později navázal na výsledky své práce v [39]. Vzniklo tak vylepšení původního MRT se jménem New Modified Reachability Tree (NMRT). Cílovou skupinou této metody jsou takzvané ω-*independent* Petriho sítě. NMRT obsahuje pouze dosažitelná značení a je tak schopen na rozdíl od svého předchůdce správně rozhodovat problém existence uzamčení systému. Měřítko použitelnosti této metody bylo větší než u doposud známých technik. Problémem zde zůstává potenciálně vysoký počet uzlů stromu. Optimalizace tohoto problému byla předmětem práce [40]. Vznikla tak další grafická reprezentace stavového prostoru označovaná jako ωRT. Tento přístup efektivně snižuje počet vrcholů NMRT, odebráním jeho duplicitních uzlů. Obě tyto reprezentace stavového prostoru lze využít k rozhodování vlastností jako jsou živost, dosažitelnost a existence uzamčení.

5.1.3 Líný graf pokrytí

Metody zmíněné v předešlé části práce s sebou přináší novou grafickou reprezentaci stavového prostoru. Přístup, který jsme zvolili my je poněkud odlišný. Metoda, kterou v této kapitole popíšeme, vychází z algoritmu pro konstrukci FRT uvedeném v úvodní části kapitoly 5.1. Finální grafická reprezentace stromu pokrytí zůstává totožná. K jeho konstrukci ovšem využíváme postup, u kterého nedochází ke ztrátám informací, nezbytných pro správné určení problémových vlastností Petriho sítí, jako jsou živost a reverzibilnost.

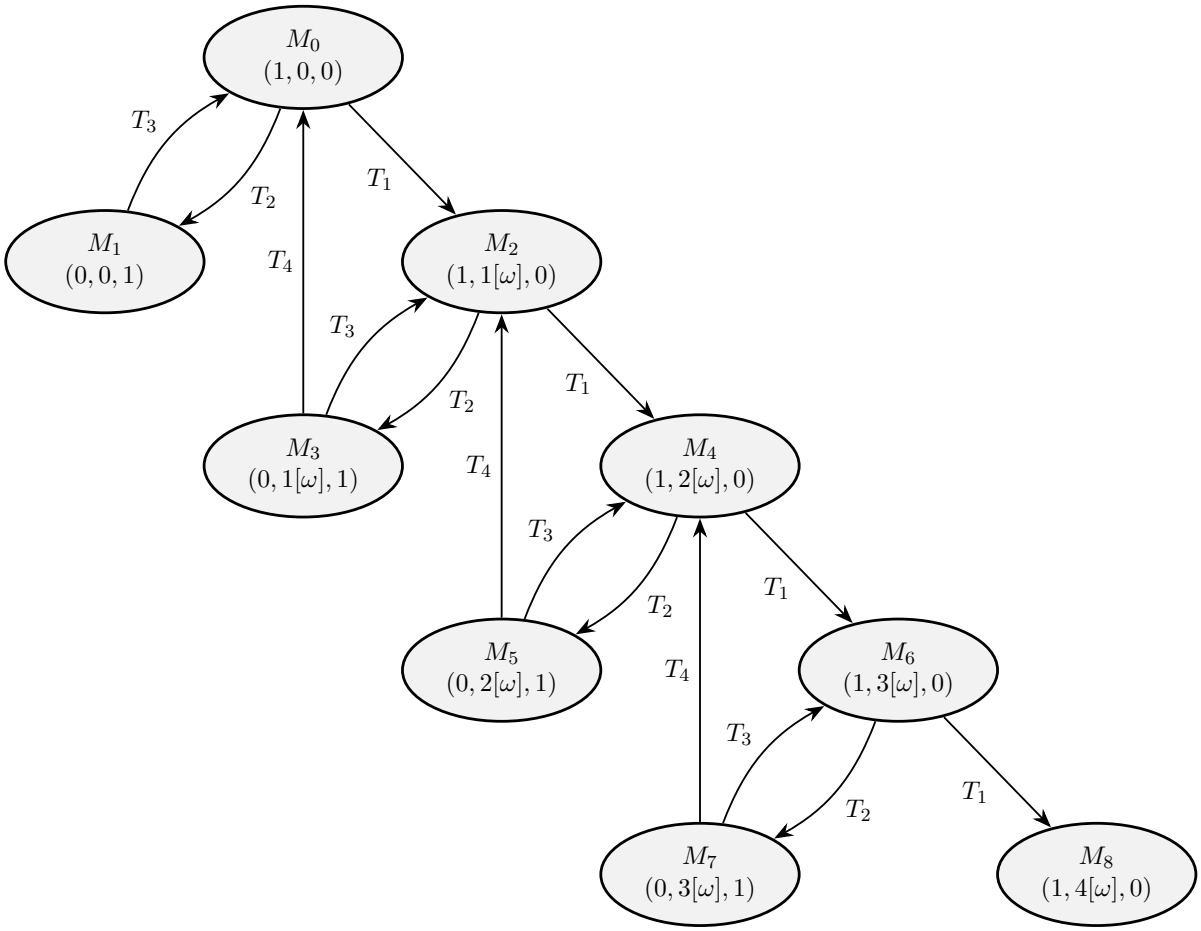
Hlavní myšlenka vznikla při pozorování struktury grafu dosažitelnosti neomezených sítí, při jeho postupném rozvíjení. Jak jsme ukázali na předešlých příkladech, je ztráta informací u FRT způsobena substitucí skutečného počtu tokenů v neomezených místech sítě pomocí ω symbolu. Náš přístup tuto informaci v první fázi konstrukce stromu pokrytí zachovává paralelně s ω příznakem v příslušných komponentách vektorů značení. Obrázek 16 zachycuje část takto sestrojeného stromu dosažitelnosti pro Petriho síť z obrázku 15a.



Obrázek 16: Část stromu dosažitelnosti Petriho sítě č. 2

Popisovaný přístup rovněž zachovává struktury, které v původním grafu pokrytí zanikají. Na obrázku 17 je ukázána část grafu dosažitelnosti, který vznikne ze stromu na obrázku 16 odebráním duplicitních vrcholů na cestě od kořene stromu po jeho listy a přidáním zpětných hran do původního značení.

Můžeme zde pozorovat opakující se struktury, které jsou v grafu pokrytí sjednoceny do jedné množiny ω vrcholů. Značení v této množině pokrývají všechna značení v opakujících se komponentách grafu dosažitelnosti. Důležitá je hrana mezi vrcholy M_3 a M_0, která v původním grafu pokrytí chybí a která zaručuje reverzibilnost celé Petriho sítě.



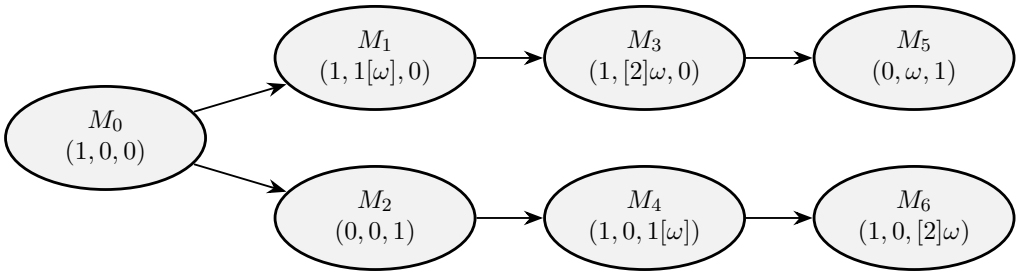
Obrázek 17: Část grafu dosažitelnosti Petriho sítě č. 2

Způsob identifikace proveditelných přechodů zůstává stejný jako u původního algoritmu. Přestože je u vypočtených značení k dispozici informace o skutečném počtu tokenů, musí být rozhodování o proveditelnosti přechodů prováděno také s ohledem na ω příznak. Pokud bychom tento symbol neuvažovali, mohli bychom chybně identifikovat množinu proveditelných přechodů v situacích, kdy přechod sice není proveditelný na základě skutečného počtu tokenů, ale jeho proveditelnost umožňuje právě ω symbol na daných pozicích vektoru značení. Takovéto přechody budeme dále v rámci této práce označovat jako podmínečně proveditelné.

Nové značení je vypočteno standardním způsobem dle váhy vstupních a výstupních hran prováděného přechodu. Počet tokenů v neomezených místech sítě bude po provedení podmínečně proveditelného přechodu záporný. V takových značeních a analogicky ve značeních z nich dosažitelných jsou vynechány informace o skutečném stavu sítě a počet tokenů v neomezených místech je rovnou substituován pomocí ω symbolu. Pravidla pro počítání s ω symbolem zůstávají stejná jako u algoritmu pro konstrukci FRT.

Způsob jeho aplikace na nově vypočtená značení jsme ovšem upravili tak, aby bylo možné tento příznak zavést s konkrétním zpožděním. V původním algoritmu je ω symbol vložen do

příslušných složek vektoru nově vypočteného značení M ihned, pokud značení M pokrývá jiné značení M' na cestě od kořene stromu po M. Náš přístup ω příznak zavádí až v okamžiku, kdy značení M takto pokrývá alespoň k dalších značení M'. Číslo $k \in (1, 2, 3, ..., \infty)$ tedy udává, při kterém výskytu pokrývání značení má být na příslušnou pozici vektoru značení vložen ω příznak. Toto zpožděné zavádění ω symbolu budeme označovat jako líné pokrývání a konstantu k jako jeho úroveň. Princip je demonstrován na obrázku 18.



Obrázek 18: Ukázka aplikace líného pokrývání značení. Úroveň líného pokrývání $k = 2$.

Informace o prováděných přechodech je vynechána, jelikož není pro účely této ukázky nijak důležitá. Úroveň líného pokrývání k byla stanovena na hodnotu dvě. Jeho efekt lze pozorovat například u značení M_1, které pokrývá počáteční značení M_0. Jelikož se jedná o první výskyt pokrývání této složky vektoru značení, není ω příznak v tuto chvíli zaveden a je v obrázku uveden pouze pro ilustraci. Ve skutečnosti je aplikován až ve značení $M3$. Jakmile je ω příznak jednou do značení zaveden, je dále předáván jeho potomkům stejným způsobem jako u algoritmu konstrukce FRT. Tuto skutečnost v obrázku zachycuje značení M_5. Je-li konstanta $k = 1$, je pokrývání značení prováděno stejně jako u FRT.

Rozvoj větve stromu pokrytí je ukončen, jestliže není v aktuálním zpracovávaném značení proveditelný žádný přechod (a to ani podmínečně) anebo je tento vrchol duplicitní. Identifikace stejných značení je prováděna s ohledem na ω příznaky, které mají přednost před případnou skutečnou hodnotou tokenů v daném místě. Značení $M_1 = (1, 1(\omega), 0)$ a $M_2 = (1, 2(\omega), 0)$ jsou proto považována za totožná. Vyhledávání duplicitních značení je možné provádět dvěma způsoby:

- na cestě od kořene stromu včetně po aktuální značení,
- globálně v celém doposud vygenerovaném stromu pokrytí.

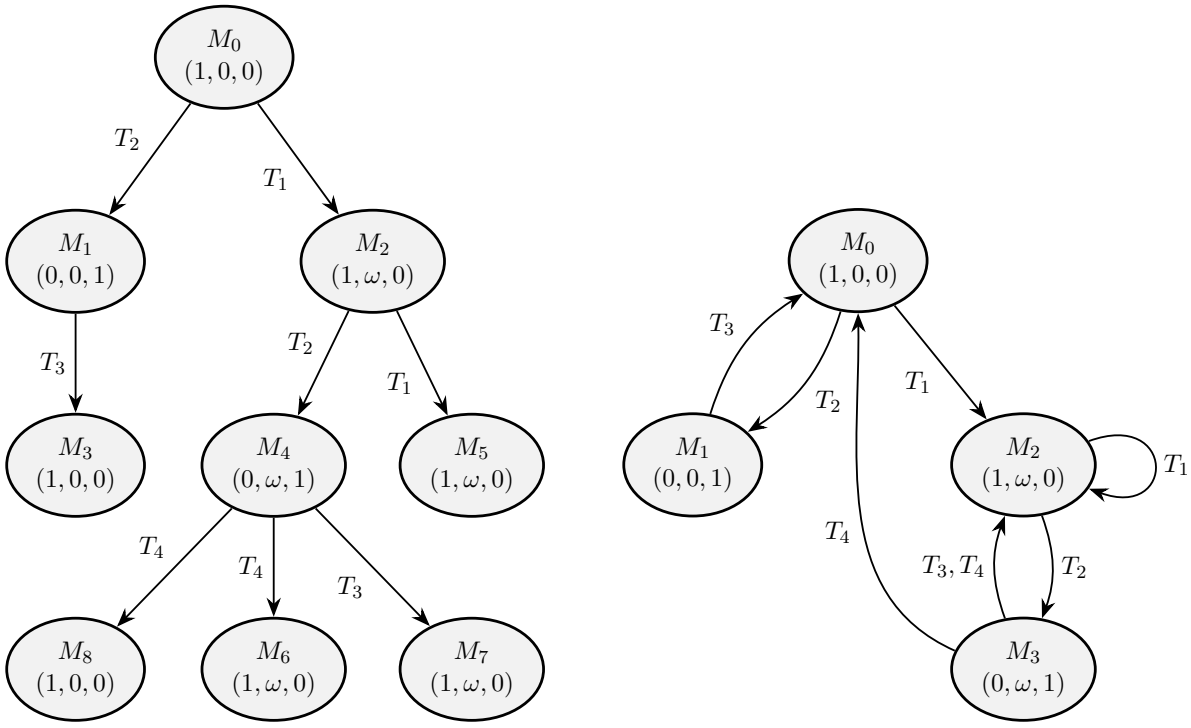
Volbu této varianty má možnost provést uživatel pomocí parametrů této analytické metody. Varianta, kdy jsou duplicitní vrcholy stromu vyhledávány globálně, přináší výhodu menšího počtu vygenerovaných uzlů a v důsledku rychlejší exekuci algoritmu. Aplikace výše uvedených pravidel při konstrukci stromu pokrytí na obrázku 16 způsobí ukončení rozvíjení větví například na cestě $M_0 \to M_2 \to M_4 \to M_6$ a $M_0 \to M_1 \to M_3$.

V okamžiku, kdy je vygenerován celý strom pokrytí, algoritmus pokračuje krokem, ve kterém je u všech značení, jejichž vektor obsahuje v alespoň jedné složce ω symbol, tímto příznakem

nahrazen skutečný počet tokenů. Pokud značení M obsahuje informaci o skutečném počtu tokenů ve všech místech sítě, je z něj při této substituci extrahováno značení M', u kterého je vynechán ω symbol. Značení M' je vytvořeno pouze pokud:

- se již ve stromě vyskytuje v podobě vrcholu bez ω příznaků,
- představuje uzamčení systému.

Nechť N je rodičovský vrchol značení M, pak je spolu se značením M' do stromu pokrytí přidána také hrana $N \rightarrow M'$, jejíž ohodnocení je rovno ohodnocení hrany $N \rightarrow M$. Konkrétně u stromu pokrytí na obrázku 16 bylo značení $M_5 = (1, 2(\omega), 0)$ finalizováno jako $(1, \omega, 0)$, jelikož se značení $(1, 2, 0)$ ve stromu nevyskytuje. Užitím stejných pravidel bude z vrcholu $M_6 = (1, 0(\omega), 0)$ extrahováno značení $M_8 = (1, 0, 0)$. Finální strom a graf pokrytí jsou znázorněny na obrázcích 19a a 19b.



(a) Strom pokrytí (b) Graf pokrytí

Obrázek 19: Grafy stavového prostoru Petriho sítě č. 2 zkonstruované dle zde popisované metody

Zmíněné rozdvojování vrcholů stromu je hlavním rozdílem mezi zde popisovaným přístupem a původním algoritmem. Extrahované vrcholy grafu nejsou pohlceny v žádném z ω značení a mohou ovlivňovat výsledky analýzy. Finální graf pokrytí na obrázku 19b tak na rozdíl od původního grafu z obrázku 15b obsahuje chybějící hranu vedoucí ze značení M_3 do M_0. Grafová analýza reverzibilnosti je tomto případě korektní. Aplikací stejného algoritmu na Petriho síť z obrázku 14a obdržíme graf pokrytí, ve kterém figuruje značení $(0, 1, 0)$ jako samostatný vrchol.

Analýza živosti tento uzel vyhodnotí jako finální silně souvislou komponentu, která neobsahuje v ohodnocení svých hran všechny přechody. Síť tedy není živá.

Důkaz korektnosti našeho přístupu byl naneštěstí nad naše síly a uvědomujeme si, že tato metoda není v matematické podstatě tedy validní. Její správnost zde demonstrujeme na příkladech. Budeme zde prezentovat úvahy, které nás k jeho implementaci vedly.

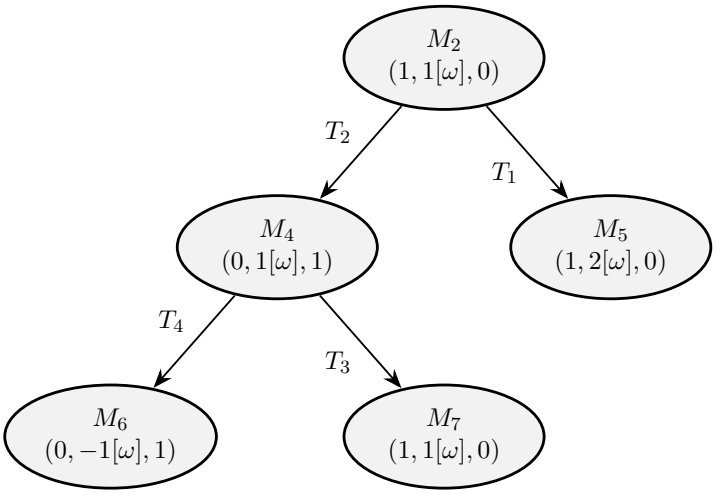
Předpokládejme, že zkoumaná síť je obyčejná, tedy že váha každé její hrany je rovna jedné a že duplicitní uzly jsou při generování jejího stromu pokrytí vyhledávány na aktuální větvi. Ke správné analýze živosti je nutné odhalit značení, které reprezentují uzamčení systému. V případě vrcholů, které neobsahují ω symbol, je tato operace triviální. Uvažujeme-li možné varianty rozvoje ω vrcholů stromu, může skutečný počet tokenů v neomezených místech buďto růst anebo klesat. Díky monotónnostií sítě se u takového rozvoje stromu pokrytí může uzamčení vyskytovat pouze v případech, kdy má počet tokenů klesavou tendenci. Tyto větve stromu je náš algoritmus schopen zachytit, díky uchovávání skutečného počtu tokenů v ω značeních. Situace, kdy by nemuselo být uzamčení takto odhaleno, je ta, kdy je rozvoj větve, která zachycuje pokles tokenů v neomezeném místě, ukončen díky duplicitnímu výskytu stejného ω značení.

Vrchol, ze kterého tato sekvence započala, již ovšem také obsahoval ω příznak v alespoň stejných složkách vektoru značení. V důsledku by tak tento rozvoj stromu byl zachycen na vyšší úrovni stromu pokrytí. Samotná extrakce uzlu uzamčení pak výsledky analýzy negativně neovlivňuje, jelikož se jedná o regulérní dosažitelné značení. Takový vrchol má zajisté také vliv na rozhodování reverzibilnosti. Petriho síť je na základě jejího grafu pokrytí vyhodnocena jako nereverzibilní, což je správně s ohledem na skutečnost, že obsahuje uzamčení jiné než počáteční značení.

U reverzibilnosti představují potenciální problém přidané zpětné hrany z ω značení do těch, které tento příznak neobsahují. Hrana je jistě korektní pro značení, které bylo v průběhu konstrukce stromu reprezentováno skutečným počtem tokenů v příslušném ω vrcholu. Je ale nutné ukázat, že lze platnost těchto hran zobecnit pro všechna značení, které náleží do dané třídy ω značení. Aby mohla být neomezená Petriho síť reverzibilní, musí existovat sekvence přechodů, která způsobuje konzumaci tokenů z jejich neomezených míst. Tato sekvence musí být proveditelná již v při prvním výskytu vrcholu, který spadá do dané třídy ω značení. Díky monotonnosti Petriho sítí pak musí být tato sekvence proveditelná také ve značeních, které toto úvodní značení pokrývají. V kontextu grafové reprezentace zde hovoříme o již zmíněných opakujících se strukturách.

Uvedený postup není možné bez úprav aplikovat na Petriho sítě, které nejsou obyčejné. Komplikace nastává u podmíněně proveditelných přechodů, kde zanikají informace o konkrétním stavu sítě. Uvažujme stejnou Petriho síť jako na obrázku 15a pouze s tou úpravou, kdy je hodnota váhy hrany mezi místem P_2 a přechodem T_4 rovna dvěma. Problémovou část stromu pokrytí ilustruje obrázek 20.

V okamžiku, kdy je aktuálně zpracovávaným vrcholem stromu značení M_4, je přechod T_4 proveditelný pouze podmínečně, protože vyžaduje alespoň dva tokeny v místě P_2. Značení dosa-



Obrázek 20: Část stromu pokrytí uvažované modifikované Petriho sítě č. 2

žené po provedení tohoto přechodu je $M_6 = (1, -1, 0)$. Takové značení je očividně nedosažitelné a tudíž je informace o skutečném stavu sítě vynechána a počet tokenů v místě P_2 označen pomocí ω symbolu. Oproti původní sítí tak z tohoto grafu nevyplyne duplicitní značení $(1, 0, 0)$ a síť nebude označena jako reverzibilní. Větev stromu na cestě $M_2 \rightarrow M_5$ by tuto informaci odhalila, nicméně je její rozvoj dle pravidel algoritmu ukončen dříve, než tato situace nastane.

Naskýtají se zde dvě možnosti řešení. Váhu hran Petriho sítě lze nahradit rozšířením její struktury. Tato operace ovšem způsobí nárůst počtu prvků sítě a tím i zvýšení složitosti její analýzy. Druhou možností je navýšení úrovně líného pokrývání našeho algoritmu na hodnotu dvě a duplicitní vrcholy detekovat na každé větvi stromu zvlášť. V kontextu stromu pokrytí na obrázku 20 tato změna způsobí to, že značení M_2 není zařazeno do třídy značení $(1, \omega, 0)$, jelikož se jedná o první výskyt pokrývání druhé složky jejího vektoru. V důsledku tak není rozvoj větve $M2 \rightarrow M5$ ukončen na úrovni značení $M5$. Mezi potomky tohoto vrcholu patří i značení $(1, 0(\omega), 0)$, ze kterého je v pozdější fázi algoritmu extrahováno značení $(1, 0, 0)$ a nalezena tak potřebná zpětná hrana.

Nezbytná hodnota úrovně pokrývání se na základě našeho testování jeví jako maximum z ohodnocení hran Petriho sítě. Pokud by byla takto ohodnocena hrana mezi přechodem a jeho vstupním místem, je při rozvoji stromu pokrytí umožněno zachycení situace, kdy je tento přechod proveditelný vzhledem ke skutečnému počtu tokenů v ω značení a ne pouze podmínečně. Jedná se ovšem o pouhý popis našeho pozorování. Příklad je demonstrován v příloze A.

5.1.4 Objektová reprezentace stavového prostoru

Každého programátora, který má zkušenosti s objektově orientovanými jazyky, jako první pravděpodobně napadne přístup, kdy jsou jednotlivé prvky grafu reprezentovány samostatnými objekty. Objekty s sebou přináší tu výhodu, že je v nich možné jednoduše uchovávat veškeré informace, které potřebujeme o dané komponentě grafu zaznamenat. Vezmeme-li jako příklad strom

pokrytí, může být jeho reprezentace tvořena klasickou kompozicí objektového stromu. Každý vrchol je zastoupen samostatným objektem, který má reference na své potomky a případně také na rodičovský uzel.

Grafy, které reprezentují stavový prostor Petriho sítě, mají tu nevýhodu, že v sobě musí obsáhnout poměrně velké množství informací. Například pro realizaci propojení dvou vrcholů stromu nestačí pouze reference na jiný uzel, ale je nutné také uchovávat informaci o přechodu, kterým bylo značení dosaženo. Tento požadavek lze vyřešit například pomocí další třídy, jejíž objekty představují hrany grafu. Zmíněný způsob přináší jednoduché rozhraní pro práci s grafy a poměrně dobrou čitelnost kódu.

Tato původní implementace se ovšem ukázala jako nevhodná. Problém spočíval v počtu objektů, které bylo nutné vytvořit při konstrukci stromu pokrytí. Programovací jazyk Java neumožňuje ve svých kolekcích uchovávat hodnoty primitivních datových typů. Problém je řešen existencí obalujících tříd jako jsou *java.lang.Integer* pro celočíselné hodnoty anebo *java.lang.Double* pro ty reálné. K uchovávání informací o například skutečném počtu tokenů v každém značení, jsme využívali kolekce typu *java.util.Map*. Klíčem byl identifikátor konkrétního místa sítě a hodnota počet tokenů v tomto místě. Neustálá tvorba objektů pro tyto hodnoty primitivních typů, způsobila neakceptovatelnou časovou a paměťovou náročnost algoritmu.

Naše finální objektová reprezentace stavového prostoru využívá knihovnu Trove. Ta obsahuje speciální implementace *java.util.Collections* API a přináší podporu pro uchovávání hodnot primitivních datových typů v kolekcích. Odpadá tak potřeba užití obalových tříd, jako je *java.util.Map.Entry*. V důsledku kolekce zabírají méně paměti a vyznačují se lepším výkonem než jejich standardní protějšky v JDK [34].

Vrchol grafu je namísto objektu zastoupen pomocí celočíselné hodnoty. Veškeré informace, které byly v původní implementaci obsaženy v daném objektovém uzlu, jsou nyní indexovány v několika Trove mapách. Například distribuce tokenů v místech sítě je pro dané značení popsána polem celých čísel a je uchovávána v kolekci typu *gnu.trove.map.hash.TIntObjectHashMap*. Konkrétní komponenty vektoru značení jsou na místa sítě mapována pomocí indexů. Při mapování se předpokládá, že je kolekce míst setříděna podle jejich ID. Na podobném principu jsou uchovávány všechny informace v grafu pokrytí.

5.2 Invarianty

Další implementované metody se již spadají do skupiny, která obsahuje techniky zabývající se analýzou struktury Petriho sítě. První z nich využívá její maticovou reprezentaci, která se nazývá incidenční matice. Ta obsahuje informace o změnách počtu tokenů v jednotlivých místech sítě při provádění konkrétních přechodů.

Definice 18 *Incidenční matice PN-struktury* $\langle P, T, I, O, H \rangle$ *je matice* $C = \{C(p,t)\}$ *typu* $|P| \times |T|$, *definována jako* $C = O^T - I^T$, *kde* $O = \{O(t,p)\}$, $I = \{I(t,p)\}$ *jsou matice typu* $|P| \times |T|$, *které reprezentují vstupní respektive výstupní funkci* O *a* I.

Z definice incidenční matice 18 ihned vyplývá, že na její podobu nemá vliv existence inhibičních hran. Stejně tak v ní není zachycena informace o přítomnosti takzvaných testovacích hran. Testovací hranou je popsána situace, kdy je hrana, vedoucí z místa do přechodu, doplněna zpětnou hranou (s identickým ohodnocením) z přechodu do původního místa. Incidenční matice Petriho sítě na obrázku 14a vypadá následovně:

$$C = O^T - I^T = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} - \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & -1 \end{bmatrix}$$

V kapitole 2.7 jsme se krátce zmínili o vektorech, které se nazývají invarianty. V této části práce tento pojem vysvětlíme podrobněji, uvedeme algoritmus, dle kterého jsou v aplikaci počítány a ukážeme jak jsou využívány při analýze vybraných vlastností Petriho sítí.

Definice 19 *P-invariantem struktury* $\langle P, T, I, O \rangle$ *nazýváme vektor* $Y = (y_1, y_2, y_3, ..., y_{|P|})^T$, *kde* $y_i \in N = \{0, 1, 2, ...\}$, *který zleva anuluje incidenční matici C. Tedy vektor pro který platí:* $Y^T \cdot C = 0^T$

Množinu $P_Y = \{p_i \in P : y_i > 0\}$ poté budeme nazývat nosičem P-invariantu Y. Jestliže každé místo Petriho sítě spadá do nosiče nějakého P-invariantu, pak říkáme, že je P-invarianty pokryta. Dříve než uvedeme algoritmus pro výpočet P-invariantů, je nezbytné uvést ještě několik poznatků:

- jestliže jsou Y_1, Y_2 P-invarianty a k_1, k_2 celá nezáporná čísla potom je i lineární kombinace $k_1 \cdot Y_1 + k_2 \cdot Y_2$ P-invariantem. Jeho nosičem je sjednocení nosičů P-invariantů Y_1 a Y_2,
- P-invariant Y je v kanonickém tvaru, jestliže největší společný dělitel jeho složek je roven jedné,
- P-invariant Y nazýváme minimální, pokud neexistuje žádný další P-invariant Y' stejné struktury Petriho sítě takový, že $Y' \leq Y \wedge Y' \neq Y$.

Nyní popíšeme algoritmus, který analytická aplikace využívá při výpočtu úplného systému minimálních P-invariantů. Zbývá ještě dodat, že systém P-invariantů je úplný, pakliže je každý P-invariant dané struktury vyjádřitelný jako lineární kombinace P-invariantů, náležících do tohoto systému. Kroky algoritmu jsou následující:

1. Inicializuje se složená matice $[A|B]$, kde $A = C$ (incidenční matice) a $B = I_m$ (jednotková matice řádu $m = |P|$.
2. Pro $j \in (1, \ldots, |T|)$ jsou provedeny následující operace:
   - Ke složené matici $[A|B]$ jsou přidány všechny řádky, které reprezentují lineární kombinace řádků matice $[A|B]$ s přirozenými koeficienty a které zároveň anulují j-tý sloupec matice A.

- Z matice $[A|B]$ jsou vyškrtány všechny řádky s nenulovým prvkem v j-tém sloupci matice A.

3. Řádky matice B jsou převedeny do kanonického tvaru.

4. Z matice B jsou odstraněny všechny neminimální P-invarianty.

5. Zbylé řádky matice B představují úplný soubor minimálních P-invariantů PN struktury zadané incidenční maticí C.

P-invarianty vypočtené dle uvedeného algoritmu jsou označovány také jako strukturální P-invarianty Petriho sítě. Pro každý takový P-invariant Y struktury $\langle P, T, I, O\rangle$ poté platí: $(\forall M \in RS(M_0))[Y^T \cdot M = Y^T \cdot M_0]$ a to pro každý systém Petriho sítě $\langle P, T, I, O, M_0\rangle$, který z její struktury vznikne přidáním libovolného počátečního značení. Výraz $Y^T \cdot M = Y^T \cdot M_0$ je poté nazýván systémovým P-invariantem. Hodnota systémového P-invariantu udává konstantní počet tokenů v místech sítě, které jsou součástí nosiče příslušného strukturálního P-invariantu a je vypočtena dle vztahu $k = Y^T \cdot M_0 = \sum y_i \cdot m_{0i}$, kde $i = 1, 2, 3, ..., |P|$,

Definice 20 *T-invariantem struktury $\langle P, T, I, O\rangle$ nazýváme vektor $X = (x_1, x_2, x_3, ..., x_{|T|})^T$, kde $x_i \in N = \{0, 1, 2, ...\}$, který zprava anuluje incidenční matici C. Tedy vektor pro který platí: $C \cdot X^T = 0^T$*

Obdobně jako u P-invariantu hovoříme i u T-invariantu o jeho nosiči. Ten představuje množinu přechodů sítě $T_X = \{t_i \in T : x_i > 0\}$. V případě, kdy každý přechod sítě spadá do nosiče nějakého T-invariantu, říkáme, že je síť T-invarianty pokryta. Pro tvorbu T-invariantů pomocí jejich lineárních kombinací platí stejná pravidla jako u P-invariantů. Stejným způsobem jsou definovány také pojmy kanonický tvar, minimální T-invariant a úplný systém T-invariantů. Pro každý T-invariant X struktury $\langle P, T, I, O\rangle$ existuje systém Petriho sítě $\langle P, T, I, O, M_0\rangle$ takový, že platí: $(\exists \sigma \in T^*)[M_0 \xrightarrow{\sigma} M_0 \wedge V_\sigma = X]$, kde V_σ představuje charakteristický vektor.

Je důležité zmínit, že ne každá sekvence přechodů $\sigma \in T^*$ je pro dané počáteční značení M_0 realizovatelná. Pokud taková sekvence realizovatelná je, nazývá se systémovým T-invariantem Petriho sítě. Systémové T-invarianty aplikace počítá s pomocí stromu dosažitelnosti. V každém kroku je ověřováno, zda je přechod, který odpovídá konkrétní složce charakteristického vektoru, proveditelný. Pokud ano pokračuje rozvoj stromu provedením tohoto přechodu a snížením hodnoty příslušné složky charakteristického vektoru o jedničku. Systémovými T-invarianty jsou pouze ty sekvence přechodů, jež tímto způsobem dokázaly charakteristický vektor vynulovat.

Pro výpočet T-invariantů lze využít stejný algoritmus jako pro výpočet P-invariantů. V inicializační fázi je proveden pouze jeden krok navíc a sice transponování incidenční matice Petriho sítě. Korektnost tohoto přístupu byla ukázána v [18] na základě takzvaných duálních struktur.

5.2.1 Využití invariantů při analýze

Aplikace je schopna vypočíst strukturální a systémové P a T invarianty . Analytický význam této metody jsme již naznačili při popisu vybraných vlastností Petriho sítí. Způsob, jakým invarianty aplikujeme při analýze, zde podrobněji popíšeme stejně jako u předešlé metody na množině vlastností, které jsou na jejich základě rozhodovány.

1. **Konzervativnost**
   Zmínili jsme, že nosičem strukturálního P-invariantu rozumíme určitou podmnožinu míst sítě. Podsystém sytému $\langle P, T, I, O, M_0 \rangle$ indukovaný touto množinou, se nazývá konzervativní komponenta Petriho sítě. Do této komponenty náleží také všechny přechody, které jsou bezprostředně spojeny s alespoň jedním místem v nosiči spolu s hranami, které je propojují. Jestliže existuje P-invariant, jehož nosič je totožný s množinou všech míst Petriho sítě, je taková síť označována jako konzervativní.

   Vypočtený systém P-invariantů v sobě nemusí přímo obsahovat invariant, který pokrývá celou Petriho síť. Takový T-invariant nicméně může stále vzniknout lineární kombinací T-invariantů z tohoto systému. Aby byla síť označena jako konzervativní postačuje, aby bylo každé místo sítě součástí nosiče alespoň jednoho z minimálních P-invariantů.

2. **Omezenost**
   Omezenost systému vyplývá přímo z jeho konzervativnosti. Je-li Petriho síť konzervativní, pak existuje P-invariant, který indukuje konzervativní komponentu totožnou s celkovou strukturou Petriho sítě. Vzhledem ke skutečnosti, že je součet tokenů v konzervativní komponentě v každém dosažitelném značení konstantní, lze síť označit za bezpečnou za předpokladu, že je její počáteční značení konečné.

3. **Repetičnost**
   Postup ověřování repetičnosti je velice podobný tomu uplatňovaném při testování konzervativnosti. Namísto P-invariantů jsou využívány strukturální T-invarianty, jejichž nosiče indukují takzvané repetiční komponenty Petriho sítě. Existuje-li T-invariant takový, že jeho nosič je roven množině všech přechodů, je síť repetiční.

4. **Živost**
   Pomocí P-invariantů lze rozhodnou o neživosti Petriho sítí. Na rozdíl od ostatních vlastností zde jsou kromě strukturálních P-invariantů využívány také jejich systémové protějšky. Petriho síť není živá, jestliže existuje nenulový kladný strukturální P-invariant, jehož systémový protějšek nabývá hodnoty nula. Jak bylo ukázáno v [20], v žádném místě, které náleží do nosiče takového P-invariantu, se v žádném dosažitelném značení nenacházejí tokeny. V důsledku, tak jsou přechody, které náleží do sjednocení množin vstupních a výstupních přechodů míst nosiče, mrtvé při počátečním značení.

Rozhodování zda je Petriho síť neživá na základě T-invariantů vychází z nutné podmínky pro její živost, uvedené v [18]. Její znění lze formulovat následovně: PN systém $\langle P, T, I, O, M_0 \rangle$ je živý a omezený $\rightarrow$ existuje T-invariant, který pokrývá celou síť (neboli síť je repetiční). Tuto formuli lze jednoduchými úpravami převést do podoby, která říká, že omezená Petriho síť, která není repetiční, rovněž není živá.

Živost je příklad vlastnosti, u které je důležité pořadí jejího rozhodování. Jelikož využívá výsledků omezenosti, musí být ověřována až v okamžiku, kdy je dokončeno testování omezenosti. Omezenost by mohla být ověřena v průběhu testu živosti samostatně, jednalo by se ale o vícenásobné testování téže vlastnosti a zpomalení celkového procesu.

5.3 Pasti a zámky

Další z implementovaných analytických metod se rovněž zaměřuje na strukturu Petriho sítě. Její princip spočívá v nalezení takzvaných pastí a zámků.

Definice 21 *Podmnožina množiny míst Petriho sítě se nazývá zámek (cotrap), jestliže platí:* ${}^\bullet Q \subseteq Q^\bullet$. *Podmnožina množiny míst Petriho sítě se nazývá past (trap), jestliže platí:* $Q^\bullet \subseteq^\bullet Q$.
${}^\bullet Q$ *reprezentuje množinu vstupních přechodů všech míst v* Q *formálně zapsáno jako:*
${}^\bullet Q = \cup_{p \in Q} \{{}^\bullet p\}$, *kde* ${}^\bullet p = \{t \in T : O(t,p) > 0\}$.
$Q^\bullet$ *reprezentuje množinu výstupních přechodů všech míst v* Q *formálně zapsáno jako:*
$Q^\bullet = \cup_{p \in Q} \{p^\bullet\}$, *kde* $p^\bullet = \{t \in T : I(t,p) > 0\}$.

Neformálně lze zámek popsat jako množinu míst takových, kdy pokud přechod do této množiny tokeny přidává, musí je z ní také odebírat. V důsledku tak zámek nemůže nikdy znovu získat tokeny v okamžiku, kdy jsou z něj všechny odebrány. Tato část sítě se obrazně uzamkne. Past je poté možné analogicky popsat jako množinu míst takových, kdy pokud přechod z této množiny tokeny odebírá, musí je do ní také vracet. Pro pasti tedy platí, že pokud jednou získají nějaký token, nikdy jej už neztratí. Zámky a pasti mohou být označovány jako minimální. Tato skutečnost nastává pokud v sobě past (zámek) neobsahuje žádnou jinou past (zámek). Sjednocením dvou pastí (zámků) vzniká znovu past (zámek).

Při implementaci této analytické metody jsme vycházeli z poznatků v [16]. Její autoři uvádějí dva přístupy aplikovatelné pro detekci pastí a zámků. První z nich je založen na hledání řešení logických rovnic, zatímco druhá metoda spočívá v řešení systému lineárních rovnic a nerovnic. Metoda, kterou práce popisuje a kterou jsme implementovali v rámci zdejší analytické aplikace, je založena na první z uvedených technik. Kromě vlastního řešení je v [16] uvedeno několik dalších publikací, které se ale většinou zaměřují na minimální pasti a zámky.

V následující části práce popíšeme implementaci a principy metody publikované v [16]. Rovnice, vzorce a další poznatky, které zde uvedeme jsme z této práce převzali nebude-li uvedeno jinak.

5.3.1 Logická reprezentace zámku a pastí

Aby množina míst S představovala zámek, musí splňovat následující podmínku:
$(\forall t \in T)[t_i^\bullet \cap S \neq \emptyset \implies {}^\bullet t_i \cap S \neq \emptyset]$. Tuto podmínku je možné zapsat v podobě logické formule. Mějme množinu booleovských proměnných $B = \{b_1, b_2, b_3, ..., b_{|P|}\}$ a funkci g typu $P \to B$, která každému místu sítě jednoznačně přiřazuje booleovskou proměnnou z množiny B. X je množina booleovských proměnných taková, že $g(p) \in X \iff p \in Q$, kde Q je množina míst, vzhledem ke které je množina X konstruována.

Jako ${}^\bullet X_i$ a $X_i^\bullet$ budeme značit množiny X vytvořené vzhledem k ${}^\bullet t_i$ a $t_i^\bullet$ respektive. Disjunkce proměnných obsažených v ${}^\bullet X_i$ a $X_i^\bullet$ značíme jako $\vee[{}^\bullet X_i]$ a $\vee[X_i^\bullet]$. Podmínku pro přechod Petriho sítě t_i lze poté zapsat formulí $(\vee[X_i^\bullet] \implies \vee[{}^\bullet X_i]) = 1$.

Definice 22 *Všechny zámky Petriho sítě jsou definovány jako kořeny logické rovnice* $\wedge_{i=1}^{n}(\vee[X_i^\bullet] \implies \vee[{}^\bullet X_i]) = 1$, *kde* $n = |T|$.

Definice pastí je podobná té pro zámky. Jedinou změnu představuje záměna členů implikace ve všech konjunktech formule. Při dalším popisu algoritmu se tedy zaměříme výhradně na výpočet zámků. Aplikaci výše uvedeného postupu pro sestavení logické rovnice demonstrujeme na Petriho síti uvedené na obrázku 15a.

Funkci g jsme definovali tak, že místu $p_i \in P$ přiřazuje logickou proměnnou $b_i \in B$, jejíž index v seřazené posloupnosti prvků množiny B je totožný s indexem místa p v seřazené posloupnosti míst sítě. Ačkoliv aplikace provádí řazení míst dle jejich ID, budeme dále v rámci popisu pro jednoduchost uvažovat řazení podle názvu. Proměnné z množiny B budeme také označovat písmenem x z důvodu pojmenování příslušných množin.

Množiny ${}^\bullet t_1$ a $t_1^\bullet$ tedy indukují množiny ${}^\bullet X_1 = \{x_1\}$ a $X_1^\bullet = \{x_1, x_2\}$ respektive. Podmínka pro přechod t_1 je formulována jako $x_1 \vee x_2 \implies x_1$. Rovnice, jejíž řešení představují zámky sítě, vypadá dle definice 22 následovně:
$(x_1 \vee x_2 \implies x_1) \wedge (x_3 \implies x_1) \wedge (x_1 \implies x_3) \wedge (x_1 \implies x_2 \vee x_3) = 1$.

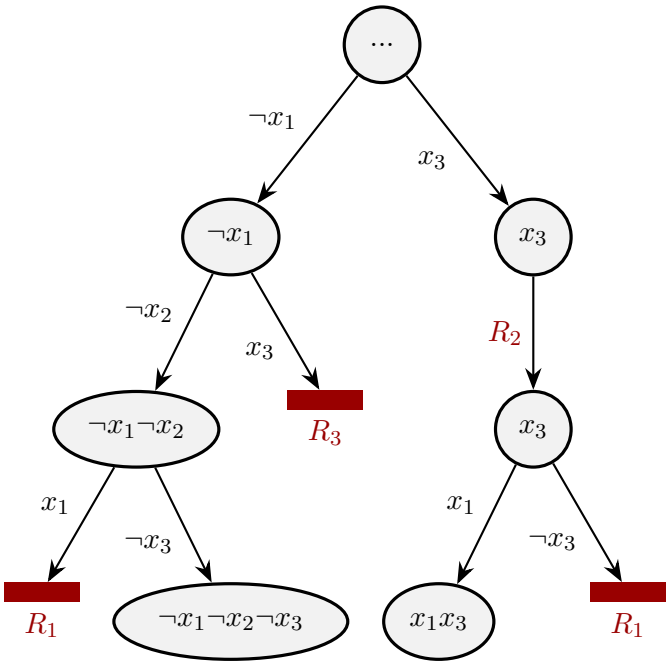
5.3.2 Thelenova metoda

K nalezení řešení logických rovnic využíváme Thelenovu metodu pro hledání primárních implikantů logické formule. Ta je navržena pro práci s formulemi v konjunktivní normální formě. Proto je nutné do této podoby formule popisující pasti a zámky nejprve převést.

Pro objektovou reprezentaci logických formulí využíváme knihovnu AIMA3e-Java [25], která je sbírkou implementací algoritmů popsaných v [29]. Součástí knihovny je implementace algoritmu pro převod zadané formule do CNF. Výsledná normální forma ovšem nebyla minimální a proto jsme tento postup obohatili o některá redukční pravidla, která například eliminují opačné literály.

Definice 23 *Implikant k logické formule F je konjunkce literálů taková, že platí: $k \implies F$. Primární implikant k_p logické formule F je implikant formule F takový, že odebráním libovolného literálu z k_p vzniká konjunkce k', která není implikant formule F.*

Thelenova metoda pro hledání primárních implikantů je založena na konstrukci vyhledávacího stromu, jehož každá úroveň odpovídá jedné klauzuli formule v CNF a ohodnocení hran představuje jednotlivé literály dané disjunkce. Hodnota vrcholu stromu je rovna konjunkci literálů, které se vyskytují jako ohodnocení hran na cestě od kořene stromu po daný vrchol. Listy stromu reprezentují elementární konjunkce, které jsou primárními implikanty formule nebo implikanty, které v sobě obsahují jiné primární implikanty nalezené dříve. Příklad vyhledávacího stromu sestaveného pro formuli $F = (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_3)$ je znázorněn na obrázku 21.



Obrázek 21: Ukázka vyhledávacího stromu využívaného v Thelenově metodě

Vyhledávací strom je konstruován do hloubky. Z ukázky 21 je zřejmé, že jeho kořen nemá žádný analytický význam. První klauzule formule je zachycena hranami vedoucími do uzlů $\neg x_1$ a x_3. Po zpracování poslední klauzule $x_1 \vee \neg x_3$, lze v listech stromu nalézt dva primární implikanty $\neg x_1 \wedge \neg x_2 \wedge \neg x_3$ a $x_1 \wedge x_3$. Velikost stromu exponenciálně roste s počtem klauzulí a jejich literálů. Existuje proto několik redukčních pravidel, jejichž aplikaci lze v obrázku pozorovat v místech označených $R1$, $R2$ a $R3$.

- **R1** - rozvoj větve je zastaven, jestliže konjunkce vrcholu, ze kterého počáteční hrana větve vychází, obsahuje negaci literálu, který tuto hranu ohodnocuje,

- **R2** - disjunkce není ve stromu zpracována, pokud obsahuje literál, který se vyskytuje v konjunkci rodičovského vrcholu,
- **R3** - rozvoj větve je zastaven, jestliže na vyšší úrovni stromu existuje jiná nerozvinutá větev, jejíž počáteční hrana má stejné ohodnocení jako hrana aktuální.

Za účelem efektivnějšího využití pravidel $R1 - R3$ byly v [3] představeny heuristiky aplikované na formuli v CNF, dříve než je pro ni strom sestaven. Jejich cílem je minimalizovat velikost vyhledávacího stromu a zamezit nalezení neprimárních implikantů. Popis těchto heuristik považujeme nad rámec této práce a pouze zmíníme, že je jejich implementace rovněž součástí naší analytické aplikace.

Výslední primární implikanti jsou reprezentováni pomocí ternární matice. Každý její řádek odpovídá jednomu primárnímu implikantu a reprezentuje jeden nebo více zámků Petriho sítě. Sloupce matice zastupují jednotlivá místa sítě. Nechť funkce h představuje inverzní funkci k funkci g, potom i-tý řádek v j-tém sloupci matice nabývá hodnoty:

- 1, jestliže i-tý implikant obsahuje literál, který funkce h mapuje na j-té místo sítě,
- 0, jestliže i-tý implikant obsahuje negovaný literál, který funkce h mapuje na j-té místo sítě,
- $-$, jestliže i-tý implikant neobsahuje literál ani jeho negaci, který funkce h mapuje na j-té místo sítě.

Takto sestavená matice pro dříve vypočtené primární implikanty má podobu:

$$A = \begin{array}{c} \begin{matrix} P_1 & P_2 & P_3 \end{matrix} \\ \begin{bmatrix} 0 & 0 & 0 \\ 1 & - & 1 \end{bmatrix} \end{array}$$

V prvním řádku matice (implikantu $\neg x_1 \wedge \neg x_2 \wedge \neg x_3$) nenalezneme žádné zámky sítě. Jelikož znak $-$ zastupuje libovolnou hodnotu z množiny $\{1, 0\}$, mohli bychom druhý řádek matice rozložit na dva řádky $(1, 0, 1)$ a $(1, 1, 1)$. Tyto řádky poté definují dva zámky sítě konkrétně množiny míst $S_1 = \{p_1, p_3\}$ a $S_2 = \{p_1, p_2, p_3\}$. Pasti a zámky jsou při analýze sítě v našem systému využívány ve spolupráci s klasifikací sítě. Popis jejich užití proto přenecháme na následující kapitolu.

5.4 Klasifikace

Mezi dovednosti analytického serveru patří rozpoznávání několika speciálních typů P/T Petriho sítí. Omezení kladená na tyto typy sítí snižují jejich modelovací sílu, ale zároveň usnadňují jejich analýzu. Všechny typy, které zde uvedeme jsou zvláštním případem obyčejných P/T Petriho sítí. Klasifikace zkoumané sítě nám v závislosti na dané třídě umožňuje okamžitě rozhodnout některé její vlastnosti.

5.4.1 Automatové sítě

Aby byla síť klasifikována jako automatová musí mít každý její přechod právě jedno vstupní a jedno výstupní místo. Musí tedy platit: $(\forall t \in T)[|^\bullet t| = |t^\bullet| = 1]$. Automatové sítě nejsou díky uvedenému omezení schopné vyjádřit paralelismus. Je-li Petriho síť klasifikovaná jako automatová, lze o ní bezprostředně usuzovat že:

- je omezená a konzervativní, jelikož se provedením libovolného přechodu sítě pouze přesouvá token z jednoho místa do druhého,
- pokud je součet všech tokenů v počátečním značení roven jedné, je síť bezpečná v důsledku konzervativnosti,
- je-li síť silně souvislá a počáteční značení není nulové, pak je síť živá.

Za účelem detekce silné souvislosti grafu Petriho sítě aplikace opět využívá Tarjanův algoritmus. Pro jeho použití bylo nutné vytvořit převaděč mezi grafovou reprezentací modelu sítě a formou, se kterou je schopna naše implementace tohoto algoritmu pracovat.

5.4.2 Synchronizační sítě

Synchronizační P/T Petriho sítě mají tu vlastnost, že každé jejich místo má právě jeden vstupní a jeden výstupní přechod. Pro tyto sítě platí: $(\forall p \in P)[|^\bullet p| = |p^\bullet| = 1]$. Oproti automatových sítím umožňují vyjádřit paralelismus, ale už ne možnost alternativy. Vlastnosti synchronizačních sítí souvisí s existencí elementárních cyklů v modelu sítě.

Definice 24 *Cyklem PN struktury* $\langle P, T, I, O\rangle$ *nazýváme slovo* $c = p_{(1)}t_{(1)}p_{(2)}t_{(2)}...p_{(n)}t_{(n)}p_{(n+1)}$ *nad abecedou* $P \cup T$*, kde* $p_{(i)} \in P$*,* $t_{(i)} \in T$ *pro* $i = 1, 2, 3, ..., n$*, pro které platí:* $p_{(n)} = p_{(n+1)} \wedge (\forall i)[p_{(i)} \in I(t_{(i)})] \wedge (\forall i)[p_{(i+1)} \in O(t_{(i)})]$.

Detekce cyklů představuje samostatnou analytickou metodu systému. Jelikož je využívána převážně ve spojení s klasifikací synchronizačních sítí, zmiňujeme se o ní v této části práce. Za účelem detekce elementárních cyklů jsme implementovali algoritmus popsaný Johnsonem v článku [15]. Nalezené cykly jsou při analýze využívány kromě ověřování vlastností synchronizačních sítí také k rozhodování, zda je jimi daná síť pokryta. Zde jsou využívány takzvané nosiče cyklu.

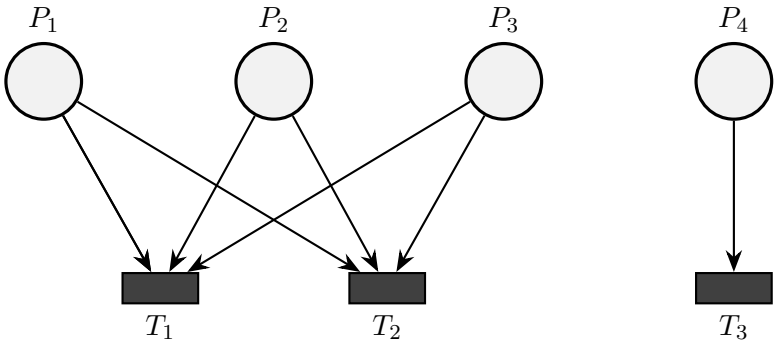
Definice 25 *P-nosičem cyklu označujeme množinu míst sítě*
$P_c = \{p \in P : p \text{ se vyskytuje v cyklu } c\}$.
T-nosičem cyklu označujeme množinu přechodů sítě
$T_c = \{t \in T : t \text{ se vyskytuje v cyklu } c\}$

Petriho síť je pokryta elementárními cykly, jestliže každé její místo náleží do P-nosiče a každý její přechod do T-nosiče nějakého elementárního cyklu. Pojem pokrývání je zde podobný jako u invariantů. Analytická aplikace dokáže u synchronizačních sítí detekovat jejich živost a bezpečnost na základě nalezených elementárních cyklů. O každé synchronizační sítí lze říci, že:

- je živá, jestliže počáteční značení umísťuje do každého jejího cyklu alespoň jeden token,
- je bezpečná, pokud každé místo sítě je součástí P-nosiče nějakého cyklu, kde součet tokenů v místech tohoto nosiče je roven jedné.

5.4.3 Sítě s volným výběrem

Poslední ze tříd, které je systém schopen rozpoznat, jsou takzvané sítě s volným výběrem. Aby Petriho síť mohla být takto klasifikována, musí mít všechna vstupní místa každého přechodu společnou množinu výstupních přechodů. Formálně lze tuto podmínku zapsat jako $(\forall t \in T)(\forall p, p' \in^\bullet t)[p^\bullet = p'^\bullet]$. Do této kategorie spadá každá automatová a synchronizační síť. Pro lepší představu zobrazuje obrázek 22 ukázku takové sítě.



Obrázek 22: Ukázka Petriho sítě s volným výběrem

U sítí s volným výběrem je ověřována jejich živost na základě takzvaného Commoner's teorému. Ten říká, že síť s volným výběrem je živá, právě když každý její zámek obsahuje past s alespoň jedním tokenem.

6 Závěr

Cílem této diplomové práce bylo vytvořit nástroj pro editaci a analýzu P/T Petriho sítí. Ačkoliv může být model sítě reprezentován několika různými způsoby, jednoznačně zde vítězí jeho grafová reprezentace. Nezbytnou součástí systému je tedy grafický editor, který jsme implementovali v podobě webové aplikace.

Uživateli jsou poskytnuty veškeré základní objekty a operace potřebné pro práci s modely P/T Petriho sítí. Aplikační logika editoru respektuje pravidla uvedená v kapitole 2. Model sítě je tvořen třemi základními prvky: místy, přechody a hranami. U všech těchto objektů může uživatel upravovat jejich vlastnosti jako je počet tokenů v místech sítě anebo násobnost hran.

Vykreslování a práce s modelem je realizována pomocí canvas elementu. Pro práci s canvasem jsme vytvořili vlastní API nadstavbu, kterou nazýváme *EaselCanvasManager*. Při jeho návrhu jsme kladli důraz na vysokou míru abstrakce, což nám umožnilo využít stejný přístup jak pro editor modelů Petriho sítí, tak pro vizualizaci grafů stavového prostoru. Konkrétní implementace *EaselCanvasManagera* v sobě nesou množinu dostupných grafických objektů, vrstev scény a funkčních modulů, které realizují aplikační logiku. Součástí naší nadstavby je rovněž mechanismus navigace ve scéně.

Implementovaná podpora pro režimy *EaselCanvasManagera* nám umožnila jednoduchou realizaci simulátoru chování Petriho sítí. Po uvedení editoru do simulačního módu jsou proveditelné přechody zvýrazňovány zeleně. Uživatel může model převést do nového značení provedením jednoho z aktuálně proveditelných přechodů. Kromě manuálního krokování lze simulaci spustit také kontinuálně, kdy je prováděný přechod vybírán náhodně. Poslední ze způsobů je provedení vybraného počtu přechodů najednou v rámci jedné simulační dávky.

Výsledky každé analytické metody jsou vizualizovány a mohou být promítány přímo do zkoumaného modelu. Jako příklad zde zmíníme zvýrazňování elementárních cyklů, pastí a zámků, konzervativních a repetičních komponent nebo aktuálně vybraného značení grafu/stromu pokrytí. Vizualizace stavového prostoru je jedna z oblastí potenciálního vylepšení systému. U velkých grafů obsahujících řádově tisíce prvků klesá rychlost jejich vykreslování. Přestože jsme implementovali řadu optimalizací, může vykreslení takového grafu (v závislosti na jeho zobrazené části) trvat stovky milisekund. Plátno pak není příliš responzivní, což se projeví například při změnách zobrazené oblastí scény.

Oblast dalšího vývoje jsou zajisté také algoritmy pro výpočet souřadnic vrcholů grafů. Výslednou podobu stromové reprezentace stavového prostoru považujeme za uspokojivou. Pro grafy pokrytí se jako nejvhodnější přístup jeví hierarchická rozvržení. Grafický editor obsahuje jejich dvě různé realizace. První z nich je zastoupena externí knihovnou s názvem Dagre, zatímco u druhé se jedná o naši vlastní implementaci. Výsledná podoba grafu závisí především na způsobu odebrání cyklů v grafu a rozmístění jeho vrcholů do jednotlivých vrstev. Ve srovnání s naší implementací jsou výsledky Dagre knihovny konzistentní a čitelnější především u velkých grafů.

Časová složitost její implementace je ale u takových grafů nepraktická. Náš hierarchický layout dokáže zpracovat i grafy, u kterých Dagre selhává, ale na úkor jejich čitelnosti.

Analytická část systému disponuje metodami, které pokrývají značnou část základní výuky Petriho sítí na zdejší univerzitě. V kapitole 5 jsme u každé analytické metody popsali detaily její implementace a způsob jakým její výsledky využíváme pro rozhodování vybraných vlastností Petriho sítí. Nejúčinnější z dostupných technik je metoda analýzy stavového prostoru. V kapitole 5.1.3 jsme rovněž prezentovali vlastní způsob konstrukce stromu pokrytí, který spočívá ve zpožděném zavádění ω příznaku do pokrývajících značení. Na příkladech jsme ukázali jakým způsobem si myslíme, že může být takto vzniklý strom pokrytí využíván pro rozhodování živosti a reverzibilnosti neomezených Petriho sítí. Možnosti vylepšení analytické aplikace spočívají především v rozšíření množiny implementovaných analytických metod, což její architektura jednoduše umožňuje.

Literatura

[1] Thomas Freytag Andreas Eckleder. Woped – a tool for teaching, analyzing and visualizing workflow nets. *Petri Net Newsletter. Newsletter of the Special Interest Groups on PetriNets and Related System Models*, 75, October 2008.

[2] Sunil Arora. Javascript frameworks: The best 10 for modern web apps. [online]`http://noeticforce.com/best-Javascript-frameworks-for-single-page-modern-web-applications`, 2016.

[3] Jacek Bieganowski and Andrei Karatkevich. Heuristics for thelen's prime implicant method, 2005.

[4] Correct system design group Carl von Ossietzky Universität Oldenburg. Petruchio. [online]`http://petruchio.informatik.uni-oldenburg.de/index.html`, 2013.

[5] Oracle Corporation. Jersey restful web services in java. [online]`https://jersey.java.net/`, 2016.

[6] Cpntools.org. Cpn tools homepage. [online]`http://cpntools.org/`, 2015.

[7] J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors. *Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness, REX Workshop, Mook, The Netherlands, May 29 - June 2, 1989, Proceedings*, volume 430 of *Lecture Notes in Computer Science*. Springer, 1990.

[8] Aalborg University Department of Computer Science. Tapaal: Tool for verification of timed-arc petri nets. [online]`http://www.tapaal.net/introduction/`, 2015.

[9] TU Eindhoven and Deloitte. Yasper: Yet another smart process editor, user guide. [online]`http://www.tapaal.net/introduction/`, 2015.

[10] Luis Gomes Fernando Pereira, Filipe Moutinho. Iopt tools user manual. [online]`http://gres.uninova.pt/iopt_usermanual.pdf`, 2014.

[11] inc. gskinner.com. Easeljs | a javascript library that makes working with html5 canvas element easy. [online]`http://www.createjs.com/easeljs`, 2016.

[12] Informatik.uni-hamburg.de. Petri nets world: Online services for the international petri nets community. [online]`http://www.informatik.uni-hamburg.de/TGI/PetriNets/`, 2015.

[13] Roberto Tamassia Isabel F. Cruz. *Graph Drawing Tutorial*. 1998.

[14] Mu Der Jeng and Mao Yu Peng. Augmented reachability trees for 1-place-unbounded generalized petri nets. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 29(2):173–183, Mar 1999.

[15] Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM J. Comput.*, 4(1):77–84, 1975.

[16] Andrei Karatkevich and Jacek Bieganowski. Detection of deadlocks and traps in petri nets by means of thelen's prime implicant method. *International Journal of Applied Mathematics and Computer Science*, 14:2004.

[17] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *J. Comput. Syst. Sci.*, 3(2):147–195, May 1969.

[18] Jaroslav Markl. Petri net sylabus. [online]`http://drazdilova.cs.vsb.cz/Data/Sites/5/petrinet/petrinetsylabus.pdf`.

[19] Microsoft. Svg vs canvas: how to choose. [online]`https://msdn.microsoft.com/en-us/library/gg193983%28v=vs.85%29.aspx#Introducing_the_Technologies`, 2016.

[20] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr 1989.

[21] Mozilla Developer Network. Inheritance and the prototype chain. [online]`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain`, 2016.

[22] W.J. Knottenbelt N.J. Dingle and T. Suto. Pipe2: A tool for the performance evaluation of generalised stochastic petri nets. *ACM SIGMETRICS Performance Evaluation Review (Special Issue on Tools for Computer Performance Modelling and Reliability Analysis)*, 36(4):34–39, March 2009.

[23] Chair of Software Engineering University of Würzburg. Welcome to the qpme project website. [online]`http://se.informatik.uni-wuerzburg.de/tools/qpme/`, 2014.

[24] Oracle. The java ee 6 tutorial. [online]`https://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html`, 2013.

[25] Ciaran O'Reilly. Aima3e-java. [online]`https://github.com/aimacode/aima-java`, 2016.

[26] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[27] Wolfgang Reisig. *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies*. Springer Publishing Company, Incorporated, 2013.

[28] Y. Ru, W. Wu, and C. N. Hadjicostis. Comments on "a modified reachability tree approach to analysis of unbounded petri nets. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 36(5):1210–1210, Oct 2006.

[29] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (3rd Edition)*. Prentice Hall, 3 edition, December 2009.

[30] IRT RWTH Aachen University. Netlab in general. [online]`http://www.irt.rwth-aachen.de/fileadmin/IRT/Download/NetLab/HilfeHTML/index.html`, 2015.

[31] IRT RWTH Aachen University. Petri net tool netlab. [online]`http://www.irt.rwth-aachen.de/en/fuer-studierende/downloads/petri-net-tool-netlab/`, 2015.

[32] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, Feb 1981.

[33] Department for Informatics University of Hamburg Theoretical Foundations Group. Renew, the reference net workshop. [online]`http://se.informatik.uni-wuerzburg.de/tools/qpme/`, 2015.

[34] trove.starlight systems.com. Trove high performance collections for java. [online]`http://trove.starlight-systems.com/overview`, 2016.

[35] DIPARTIMENTO DI INFORMATICA Università di Torino. Greatspn home page. [online]`http://www.di.unito.it/~greatspn/index.html`, 2008.

[36] DIPARTIMENTO DI INFORMATICA Università di Torino. The home of the new greatspn graphical editor. [online]`http://www.di.unito.it/~amparore/mc4cslta/editor.html`, 2013.

[37] W3.org. Ui events specification. [online]`https://www.w3.org/TR/DOM-Level-3-Events/#event-flow`, 2016.

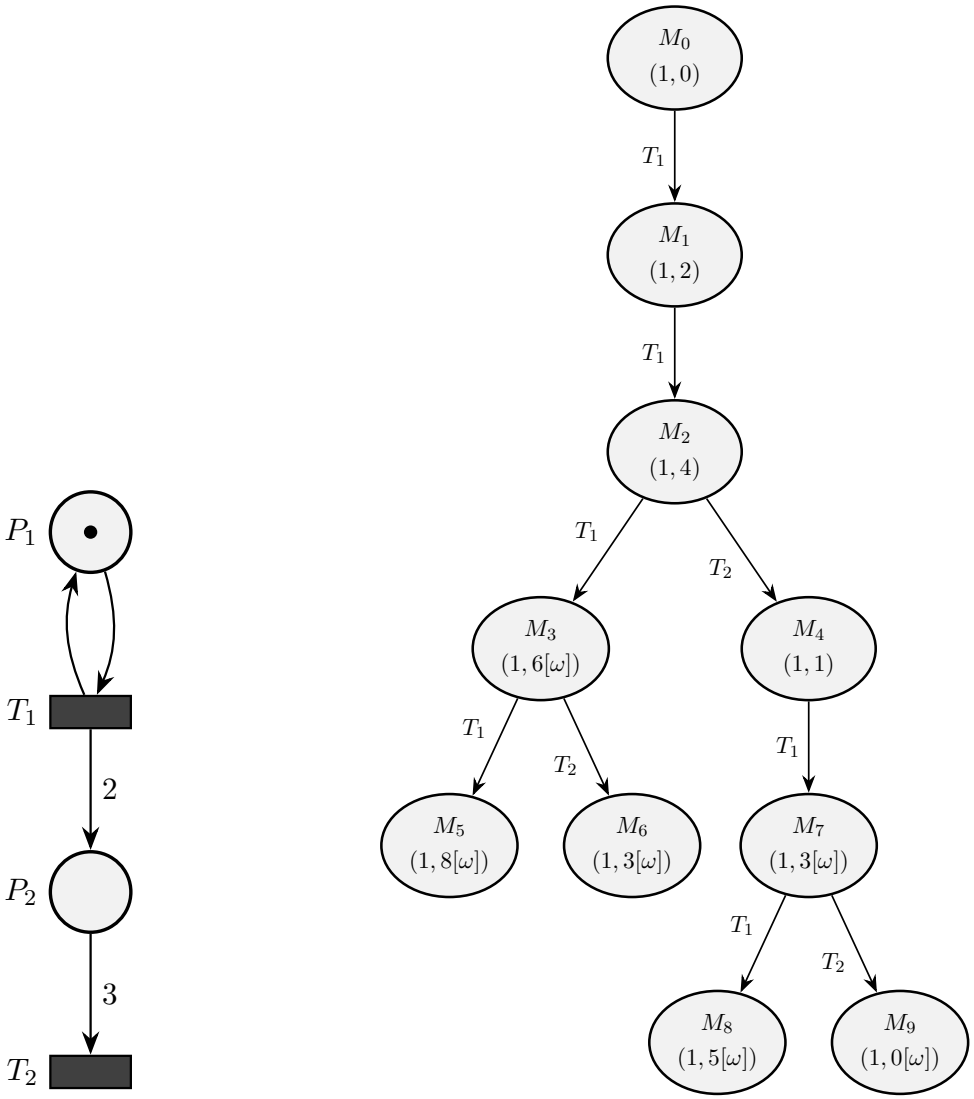
[38] W3Schools. Javascript object prototypes. [online]`http://www.w3schools.com/js/js_object_prototypes.asp`, 2016.

[39] Fei-Yue Wang, Yanqing Gao, and MengChu Zhou. A modified reachability tree approach to analysis of unbounded petri nets. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 34(1):303–308, Feb 2004.

[40] S. Wang, M. Gan, M. Zhou, and D. You. A reduced reachability tree for a class of unbounded petri nets. *IEEE/CAA Journal of Automatica Sinica*, 2(4):345–352, October 2015.

[41] Armin Zimmermann and Michael Knoke. Timenet 4.0 a software tool for the performability evaluation with stochastic and colored petri nets. [online]`http://www2.tu-ilmenau.de/sse_file/TimeNET/Documentation/TimeNET-UserManual40.pdf`, 2007.

A Neobyčejná Petriho síť



(a) Model sítě

(b) Strom pokrytí zkonstruovaný dle zde metody popisované v kapitole 5.1.3

Obrázek 23: Petriho síť č. 3. Demonstrace vlivu hodnoty úrovně líneho pokrývání

B Grafická reprezentace Petriho sítě v nástroji Renew

Obrázek 24: Petriho síť zobrazená v nástroji Renew