FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Implementace Petriho sítě v hradlovém poli
Student:	Bc. Zbyněk Jakš
Vedoucí:	doc. Ing. Hana Kubátová, CSc.
Studijní program:	Informatika
Studijní obor:	Návrh a programování vestavných systémů
Katedra:	Katedra číslicového návrhu
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

Navrhněte interaktivní převod modelu systému popsaného Petriho sítí do HW/SW implementace.
Vyberte vhodný systém, který bude možné a vhodné modelovat Petriho sítí a tento model realizujte.
Použijte existující modelovací nástroj (např. CPNTools) pro simulaci a analýzu navrženého modelu.
Vyjděte z výsledků obhájených diplomových a semestrálních prací na podobné téma (např. překladač PNML2VHDL).
Výsledkem bude syntetizovatelný popis systému ve VHDL a/nebo v C a zhodnocení použitelnosti zvoleného řešení.

Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Hana Kubátová, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 14. února 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Implementace Petriho sítě v hradlovém poli

Bc. Zbyněk Jakš

Katedra číslicového návrhu
Vedoucí práce: doc. Ing. Hana Kubátová, CSc.

7. května 2018

Poděkování

V první řadě bych nejvíce rád poděkoval vedoucí své práce doc. Ing. Haně Kubátové, CSc. za její praktické rady a pomoc v průběhu psaní. Dále bych rád poděkoval své rodině, přátelům a kolegům za jejich morální podporu, která je také jedním ze stavebních kamenů studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 7. května 2018 .....

Odkaz na tuto práci

Jakš, Zbyněk. *Implementace Petriho sítě v hradlovém poli*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Tato práce se zabývá metodikou převodu popisu Petriho sítí z jazyka PNML do syntetizovatelného zápisu pro programovatelná hradlová pole v jazyce VHDL. V úvodní části představí samotný koncept Petriho sítí, varianty a praktické využití. Další část bude věnována nástrojům pro modelování Petriho sítí a jejich výhodám a nevýhodám. Následně bude vysvětlen popis v normalizovaném jazyce PNML, a poté také, jak lze sítě realizovat v programovatelných hradlových polích. Samotný převod mezi oběma formáty bude poté proveden pomocí vytvořené aplikace a předveden na zvoleném modelu Petriho sítě. V závěru práce zhodnotí funkčnost a využitelnost tohoto řešení.

Klíčová slova Petriho sítě, převod, PNML, VHDL, FPGA

Abstract

Primary focus of this thesis is to find a solution for converting a Petri net description in PNML language into synthesizable code for FPGA devices in VHDL language. The first part introduces a concept of Petri nets themselves, their variations and practical examples. Next part deals with available tools for modeling Petri nets and explains their positives and negatives. Third part presents the PNML standard and shows a possible method of implementing

a Petri net in FPGA platform. The conversion itself will be carried out by created application and demostrated on a Petri net model example. At the end the thesis comes with a conclusion of functionality and utilization of this solution.

Keywords Petri nets, conversion, PNML, VHDL, FPGA

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V průběhu vývoje vědy v lidské historii se vědci neustále setkávali se složitějšími a složitějšími systémy, jevy, procesy, analýzami, chováními a stavy. Ať již to byly obory technické, přírodní, ekonomické nebo sociální, všechny měly tu vlastnost, že pro svá bádání potřebovaly vhodného pomocníka pro popis svých objevů a návrhů. Tím pomocníkem, který vědcům umožnil uspokojivě jejich objevy a nápady popsat, se staly matematické modely.

Jedním z těchto diskrétních modelů, kterým lze popsat řídící toky, informační závislosti a asynchronní systémy, jsou tzv. Petriho sítě.

K čemu by ale byly pouze modely systémů, kdyby se nemohly stát skutečností. Každý navržený systém je určen k tomu, aby našel své reálné praktické využití. Aby se ale toto mohlo stát, je zapotřebí si k systému určit, jak přesně chceme, aby výsledek vypadal. Jednou z možností je modelování systémů pomocí hardwarových součástek, kde bude možné jeho chování detailně pozorovat.

Moderní doba umožňuje navrhovat digitální obvody na univerzálních platformách. Pro modelování systému zapsaného jako Petriho síť je tedy zapotřebí jej převést do podoby zápisu číslicového obvodu. Trendem moderní doby je posouvat úrovně implementace na stále snažší, a proto bude vhodné, když převod modelu obstará automatizovaný nástroj. Návrháři krom práce s návrhem a implementací také navíc odpadne nutnost znát jazyk pro popis hardwaru.

Aby tedy bylo možné s převodem začít, bude ze začátku zapotřebí si zadefinovat, co vlastně Petriho síť je, jak se reprezentuje, a přidat několik důležitých pojmů. Následně se v další kapitole představí modelovací nástroje. Ty dnes již dokáží velkou řadu funkcí, včetně klasifikací sítí, simulací, analýzy a různých možností výstupních popisů včetně toho hlavního, se kterým bude tato práce

dále pracovat. Tím popisem ja jazyk PNML. Ten, jakožto většina jazyků, má svůj vlastní standard, syntaxi a metodiku zápisu, kterou je zapotřebí uvést.

Po teoretické stránce přijde stránka praktická. Jednotlivé prvky Petriho sítě je zapotřebí navrhnout jako hardwarovou součástku a k tomu navíc navrhnout takový řadič, aby se i celý obvod choval jako Petriho síť. Tomu se bude věnovat první praktická kapitola. Druhá kapitola se již bude věnovat softwarovému převodu z popisu jednoho modelu na druhý. Celý proces převodu bude předveden na předem vybraném, navrženém a zanalyzovaném modelu. Výsledek následně ukázán na hardwarové platformě a celková práce zhodnocena.

Kapitola 1

Cíl práce

Práce je členěna do 3 hlavních kapitol a k nim příslušných podkapitol. První kapitola se zabývá čistě teorií nezbytnou k vypracování druhé části – praktické. Prvním cílem první kapitoly je zadefinovat kompletní nezbytnou teorii, která se týká matematického modelu Petriho sítí. Kromě definic je zapotřebí znát i značení, vlastnosti pro analýzu a rozšiřující koncepty. Dalším cílem teoretické části je seznámit se s reálnými možnostmi modelování Petriho sítí v podobě vyvinutých nástrojů, které poskytnou vhodný výstup pro další zpracování. Posledním cílem kapitoly 1 je představit detailně princip jednoho z těchto výstupů – PNML. A to hlavně způsob jeho zápisu, implementace a způsob, jakým rozšiřuje standardní model o koncept Petriho sítí, který se bude v praktické části převádět na číslicový obvod.

Cílem praktické části je navrhnout a realizovat samotný převod z PNML do VHDL. Zprvu je třeba ukázat, jak lze vůbec Petriho síť hardwarově realizovat, a následně pak, jak lze její možnosti dále rozšiřovat. V další části je nutno se podívat, jak lze chování Petriho sítě poskládat dohromady ve funkční celek. Následující část si klade za cíl ukázat a vysvětlit, jak funguje nástroj, neboli výsledná aplikace, která dokáže vyčíst informace z PNML souboru a data následně využít pro vygenerování syntetizovatelného VHDL řešení. Cílem poslední části praktické stránky práce je otestovat na alespoň 2 modelech systému správné chování převedeného modelu.

Kapitola 2

Analýza

2.1 Petriho sítě

Petriho sítěmi, jak uvádí zdroj [1], je označována široká škála diskrétních matematických modelů, které umožňují popsat specifickými prostředky řídicí toky a informační závislosti uvnitř modelovaných systémů. První koncept se objevil v dizertační práci německého matematika Carla Adama Petriho v roce 1962, ve které se snažil popsat vzájemné závislosti mezi podmínkami a událostmi modelovaného systému.

Petriho síť je speciálním typem orientovaného grafu, který využívá 2 typy uzlů: místa (angl. places) a přechody (angl. transitions). Koncept se postupně vyvíjel tak, aby vývojářům vyhovoval po stránce modelování systémů v praxi. Využití nachází např. v asynchronních systémech jako např. dopravní struktury, synchronizace procesů (problém večeřících filosofů) aj.

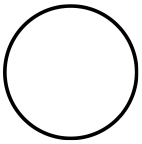
Zdroj [2] definuje, že trojici $N = (P, T, F)$ nazýváme sítí jestliže:

1. P a T jsou disjunktní množiny
2. $F \subseteq (P \times T) \cup (T \times P)$ je binární relace

- P nazýváme množinou míst (places)
- T nazýváme množinou přechodů (transitions)
- F nazýváme tokovou relací (flow relation)

Šestici $N = (P, T, F, W, K, M_0)$ nazýváme P/T Petriho sítí (Place/Transition Petri net) jestliže:

- (P, T, F) je konečná síť



Obrázek 2.1: Grafická reprezentace místa

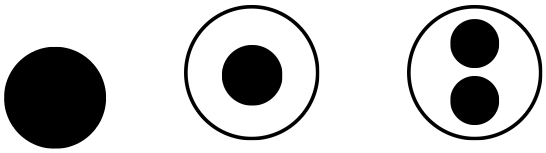
Obrázek 2.2: Grafická reprezentace přechodu

- $W : F \to \mathbb{N}$ je ohodnocení hran grafu, určující kladnou váhu každé hrany sítě
- $K : P \to \mathbb{N} \cup \{\omega\}$ je zobrazení určující kapacitu každého místa
- $M_0 : P \to \mathbb{N} \cup \{\omega\}$ je počáteční značení míst Petriho sítě takové, že $\forall p \in P : M_0(p) \leq K(p)$
- ω značí supremum množiny přirozených čísel $\mathbb{N}$ s vlastnostmi:
  - $\forall n \in \mathbb{N} : n < \omega$
  - $\forall m \in \mathbb{N} \cup \{\omega\} : m + \omega = \omega + m = \omega - m = \omega$

Pro grafickou reprezentaci Petriho sítí se používají následující symboly:

- pro znázornění míst se používají prázdné kruhy
- pro znázornění přechodů se používají obdélníky vyplněné černou barvou
- orientace míst a přechodů se znázorňuje pomocí šipky
- pro zobrazení ohodnocení místa se používá tzv. token. Token se zobrazuje jako plný černý kruh a jejich počet v místě nemusí být omezen.
- výchozí kapacita místa není omezena. Pokud chceme maximální počet tokenů v místě omezit, stačí k němu připsat číselnou poznámku.

Obrázek 2.3: Grafická reprezentace hrany



Obrázek 2.4: Grafická reprezentace tokenů

Obrázek 2.5: Grafická reprezentace násobné hrany

- výchozí kapacita hrany je jeden token na odpal. Pro specifikaci kapacity průtoku tokenů lze nad hranu (šipku) připsat číslo, které kapacitu určuje.

2.1.1 Přechody stavů

Okamžitý stav systému je v Petriho sítích udáván hodnotami dílčích stavů jednotlivých míst. V grafické podobě zobrazeno jako počet tokenů v jednotlivých místech, jak říká zdroj [1]. Každý přechod má definovanou množinu vstupních a výstupních míst. Vstupní a výstupní podmínky definují počty odebíraných a umisťovaných tokenů, což je v grafu Petriho sítě specifikováno

ohodnocením orientovaných hran. Hrana, která není v grafu explicitně ohodnocena má implicitně přiřazenou váhu 1. Přechod lze provést pouze při splnění všech vstupních a výstupních podmínek. Vstupními podmínkami je to, že pro každé místo P vstupní množiny přechodů platí, že obsahuje alespoň tolik tokenů, kolik činí násobnost hrany vedoucí z místa P do přechodu T. Výstupními podmínkami je to, že pro všechna místa P z výstupní množiny musí platit, že počet tokenů zvětšený o násobnost vstupní hrany nesmí překročit jejich kapacitu.

Zdroj [2] matematicky definuje přechod takto:
Nechť $N = (P, T, F, W, K, M_0)$ je Petriho síť.

1. Zobrazení $M : P \to \mathbb{N} \cup \{\omega\}$ se nazývá značení (marking) Petriho sítě N, jestliže $\forall p \in P : M(p) \leq K(p)$

2. Nechť M je značení Petriho sítě N. Přechod $t \in T$ je proveditelný(enabled) při značení M (stručněji M-proveditelný), jestliže

   - $\forall p \in t_{in} : M(p) \geq W(p,t)$
   - $\forall p \in t_{out} : M(p) \leq K(p) - W(t,p)$

3. Je-li $t \in T$ $M-$proveditelný, pak jeho provedením získáme následné značení M' ke značení M, které je definováno takto:

   $$\forall p \in P : M'(p) = \begin{cases} p \in t_{in}/t_{out} & \implies M(p) - W(p,t) \\ p \in t_{out}/t_{in} & \implies M(p) + W(t,p) \\ p \in t_{in} \cap t_{out} & \implies M(p) - W(p,t) + W(t,p) \\ jinak & M(p) \end{cases}$$

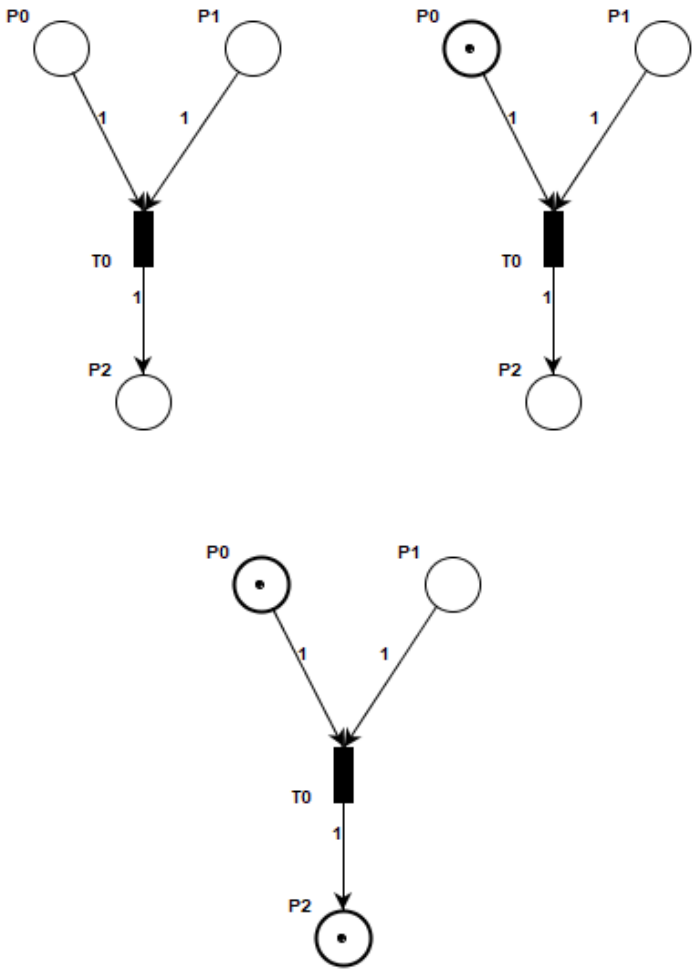
   Odpálení přechodu t (transition firing) ze značení M do značení M' zapisujeme symbolicky $M[t\rangle M'$..

4. Označme $M[t\rangle M'$ nejmenší množinu různých značení Petriho sítě N, pro kterou platí:

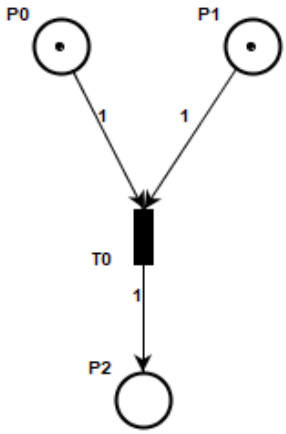
   - $M \in [M\rangle$
   - Je-li $M_1 \in [M\rangle$ a pro nějaké $t \in T$ platí $M_1[t\rangle M_2$, pak $M_2 \in [M\rangle$

   Množina $[M\rangle$ se nazývá množinou dosažitelných značení (reachability set) ze značení M.
   Množina $[M_0\rangle$ se nazývá množinou dosažitelných značení sítě N.

   Množina $[M_0\rangle$ reprezentuje stavový prostor Petriho sítě. Může být buď konečná nebo nekonečná spočetná.



Obrázek 2.6: Nevalidní přechody



Obrázek 2.7: Validní přechod

Zdroj [2] dále definuje přechodovou funkci takto:

Nechť $N = (P, T, F, W, K, M_0)$ je Petriho síť a $[M_0\rangle$ její množina dosažitelných značení. Přechodovou funkcí Petriho sítě N nazveme funkci δ:

$\delta : [M_0\rangle \times T \to [M_0\rangle$, pro kterou

$$\forall t \in T : \forall M, M' \in [M_0\rangle : \delta(M, t) = M' \overset{def.}{\iff} M[t\rangle M'$$

Přechodová funkce δ může být zobecněna na posloupnost přechodů:

$\delta : [M_0\rangle \times T^* \to [M_0\rangle$ takto:

- $\delta(M, t\tau) = \delta(\delta(M, t), \tau), \tau \in T$
- $\delta(M, \epsilon) = M$, kde ϵ je prázdný symbol

Řetězec $\tau \in T^+$ nazveme výpočetní posloupností Petriho sítě, je-li $\delta(M_0, \tau)$ definována (+ případné další podmínky).

Jazyk Petriho sítě je množina výpočetních posloupností Petriho sítě.

2.1.2 Analýza sítí

Podle [2] definujeme 4 základní problémy analýzy Petriho sítí. Jsou jimi:

- bezpečnost
- omezenost
- konzervativnost
- živost

2.1.2.1 Bezpečnost

Místo $p \in P$ Petriho sítě $N = (P, T, F, W, K, M_0)$ s počátečním značením M_0 je bezpečné, jestliže pro všechna značení $M \in [M_0\rangle$ je $M(p) \leq 1$. Petriho síť je bezpečná, jsou-li všechna místa bezpečná.

2.1.2.2 Omezenost

Místo $p \in P$ Petriho sítě $N = (P, T, F, W, K, M_0)$ se nazývá k-bezpečné, jestliže pro všechna značení $M \in [M_0\rangle$ je $M(p) \leq k$. Je-li místo p' k-bezpečné pro nějaké k, nazývá se omezené (bounded). Petriho síť, jejíž všechna místa jsou omezená, se nazývá omezená Petriho síť.

2.1.2.3 Konzervativnost

Petriho síť $N = (P, T, F, W, K, M_0)$ je striktně konzervativní, jestliže platí:

$$\forall M \in [M_0\rangle : \sum_{p \in P} M(p) = \sum_{p \in P} M_0(p)$$

Konzervativnost vzhledem k váhovému vektoru $\underline{\omega} = (\omega_1, ..., \omega_n), \omega_i \geq 0$

$$\forall M \in [M_0\rangle : \sum_{i=1}^{n} \omega_i \cdot M(p_i) = \sum_{i=1}^{n} \omega_i \cdot M_0(p_i)$$

2.1.2.4 Živost

Nechť $N = (P, T, F, W, K, M_0)$ je Petriho síť a $t \in T$.

1. t se nazývá živý přechod, jestliže pro každé značení $M \in [M_0\rangle$ existuje značení $M' \in [M\rangle$ takové, že t je proveditelný při značení M'.

2. Síť N se nazývá živou, je-li každý její přechod živý.

2.1.2.5 P-invarianty

Důležitou součástí analýzy PN jsou tzv. P-invarianty. Zdroj [3] je definuje takto:
Funkce $f : (P \to N_0) \to N_0$ tak, že pro každé značení M a M_0 tak, že M_0 je výsledkem jednoho odpálení z M, $f(M) = f(M_0)$

Jedná se tedy o množiny míst, které v průběhu odpalů nemění počet svých tokenů.

2.1.2.6 T-invarianty

Dalším užitečným pojmem při analýze PN jsou tzv. T-invarianty. Ty k nějakému počátečnímu značení říkají, které přechody je třeba odpálit, aby se síť vrátila k počátečnímu značení. Definice podle [3] zní:

Funkce $f : T \to N_0$ tak, že pro každou posloupnost odpálení ze značení M do značení M_0, které odpálí každý přechod $t \in T$ přesně $f(t)$ krát, $M = M_0$.

2.1.3 Varianty Petriho sítí

S vývojem času se pojem Petriho síť postupně rozrůstal. Základní možnost modelování začala být nedostačující, a proto se v souvislostech s Petriho sítěmi začaly objevovat následující pojmy.

2.1.3.1 Petriho sítě řízené událostmi

Petriho sítě řízené událostmi jsou nejjednoduššími typy Petriho sítí, které zároveň poskytují nejméně informací. Jejich hlavní vlastností je omezenost všech míst na 1 možný token. Veškeré sítě tohoto typu lze převést na konečný automat.

2.1.3.2 Place-transition Petriho sítě

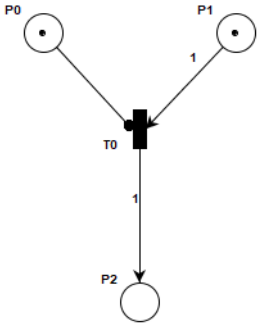
Place-transition Petriho sítě odpovídají přesné definici na začátku kapitoly 2.1. Hlavním rozdílem oproti Petriho sítím řízenými událostmi je neomezenost tokenů.

2.1.3.3 Petriho sítě s inhibičními nebo testovacími hranami

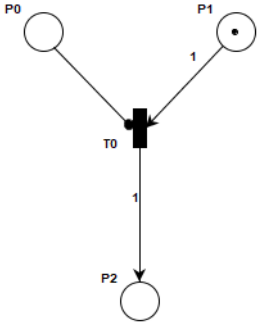
Podle [1] některé modely Petriho sítí zavádí koncept tzv. inhibičních a testovacích hran.

Cíl inhibičních hran je opakem hran klasických, tj. zabránění přechodu v jeho odpalu. Pokud tedy je místo p připojeno k přechodu t inhibiční hranou, pak může být přechod t odpálen pouze za předpokladu, že místo p obsahuje menší počet tokenů, než kapacita inhibiční hrany.

Obrázek 2.8: Grafické znázornění inhibiční hrany



Obrázek 2.9: Zakázaný přechod

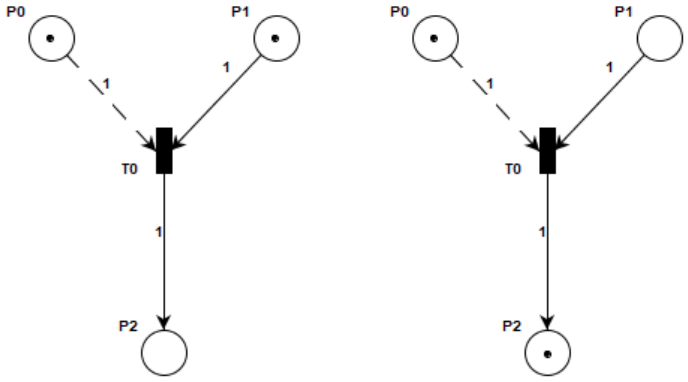


Obrázek 2.10: Povolený přechod

Speciální vlastností inhibičních hran je, že odpalování (pokud je možné) neodebírá tokeny z místa, které je inhibiční hranou připojeno k přechodu. Zavedením inhibičních hran se vyjadřovací hodnota Petriho sítě zvýši na úroveň Turingova stroje.

Cílem testovacích hran je zlepšení řiditelnosti vybraných přechodů tím, že se hraně zavede ta vlastnost, že odpal místa spojeného testovací hranou neodebírá jeho tokeny. Tato vlastnost je výhodná právě v případech testování chování daného modelu.

Obrázek 2.11: Grafické znázornění testovací hrany



Obrázek 2.12: Povolený přechod a stav po odpalu

2.1.3.4 Petriho sítě s prioritami

V klasických Petriho sítích se naprosto běžně stává, že v jednom okamžiku je aktivní vícero přechodů. Zdroj [3] uvádí, že odpaly v takových sítích jsou

nedeterministické. To tedy znamená, že v případě vícero aktivních přechodů se odpálí jeden náhodný.

V konceptu Petriho sítě s prioritami se každému přechodu přiřadí hodnota, která určuje jeho prioritu. V případě aktivnosti vícero přechodů se tedy uplatní ten s největší předností nebo v případě stejných priorit opět nedeterministicky.

2.1.3.5 Časované Petriho sítě

Zdroj [4] píše, že pro potřeby simulace, řízení, real-time a stochastické analýzy je třeba zavést do Petriho sítí vhodné časování. Časové značky lze přiřadit jak k místům, tak k přechodům. V případě místa hodnota času vyjadřuje, jakou minimální dobu musí token v místě existovat, aby mohl být odpálen. V případě přechodu časová značka vyjadřuje, že přechod lze odpálit pouze tehdy, je-li aktivní po stanovenou dobu.

2.1.3.6 Barevné Petriho sítě

Dalším z možných rozšíření modelů jsou tzv. Barevné Petriho sítě. Podle zdroje [5] mají tokeny v tomto konceptu také svojí hodnotu, často označovanou pojmem „barva“. Důvodů k zavedení barev je několik. Jedním z faktorů je, že klasický koncept PN se pro složitější modely může rozrůst do velikosti, kde se poté již obtížně řídí. Dalším důvodem je, že tokeny velmi často reprezentují různé objekty nebo zdroje v modelovaném systému. Tyto objekty mohou mít atributy, které jsou obyčejnými tokeny velmi složitě realizovatelné. Barevným Petriho sítím se také říká „high level nets,“ neboli sítě vysoké úrovně.

2.1.3.7 Hierarchické Petriho sítě

Zdroj [6] konstatuje, že hierarchické Petriho sítě umožňují členit vytvářenou síť na jednotlivé podsítě, které jsou navzájem propojeny. Hierarchickou Petriho sítí rozumíme částečně uspořádanou množinu nehierarchických Petriho sítí – tzv. stránek. Stránka B je pod stránkou A, jestliže síť na stránce B rozvíjí některý prvek ze stránky A.

Za tímto účelem se využívají hierarchizační konstrukty:

- Substituce přechodů – přechod v dané síti je nahrazen substitující sítí.
- Substituce míst – místo v dané síti je nahrazeno substitující sítí.
- Volání přechodů.
- Slučování přechodů.
- Slučování míst.

2.1.3.8 Objektově orientované Petriho sítě

S rostoucí popularitou objektově orientovaného programování začalo být zapotřebí umět modelovat i systémy, které objekty obsahují. Koncept Petriho sítí byl tedy k tomuto účelu rozšířen tak, aby dokázal reprezentovat atributy a metody objektu.

K reprezentování atribut v OOPN se nadále využívají tokeny, u kterých je specifikováno, jaký atribut v sobě nesou. Pro reprezentaci metod třídy se nově zavedl koncept tzv. stránek a podstránek, říká zdroj [7].

OOPN umožňují dále modelovat i paralelismus pomocí vláken a také dědičnost tříd.

2.2 Nástroje pro modelování PN

Z předchozí kapitoly víme, že možností, jak navrhovat modely v podobě Petriho sítí, existuje celá řada. Sítě v sobě nesou mnohé informace a různé koncepty mají různou vyjadřovací schopnost. Analýza sítě pak také otevírá a poukazuje na další možnosti a vlastnosti systému.

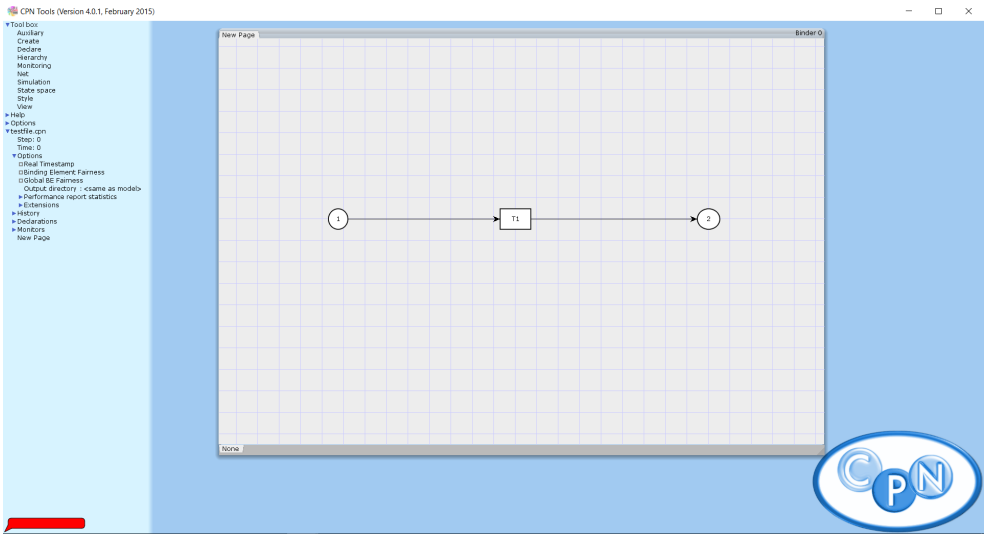
Pro zjednodušení lidské práce byly vytvořeny nástroje, které návrhy a analýzu podstatně ulehčují. Novodobé nástroje dokáží Petriho sítě nejenom modelovat, ale jsou schopné provádět např. i jejich simulace a analyzovat vlastnosti. Tato práce prozkoumala několik nástrojů. Následující sekce představí ty nejvhodnější kandidáty pro následné využití k modelování systému pro převod do HW platformy, a také ukáže různorodost existujících nástrojů.

2.2.1 CPN Tools

CPN Tools je poměrně mocným nástrojem, který dokáže modelovat, simulovat a analyzovat Petriho sítě. Kromě klasických PN program podporuje také barevné Petriho sítě (pro které byl primárně vytvořen) a také časované Petriho sítě. Původní vývoj nástroje probíhal v letech 2000 - 2010 na Aarthuské Univerzitě v Dánsku skupinou CPN Group. Hlavními architekty jsou Kurt Jensen, Søren Christensen, Lars M. Kristensen a Michael Westergaard. Od roku 2010 spadá vývoj pod skupinu AIS Group, která je součástí Technické univerzity Eindhoven v Nizozemí.

Tento nástroj má několik výhod. Tou hlavní je, jak již bylo zmíněno, široká škála modelování, analýzy a simulace. Dalším velkou výhodou je možnost exportování modelu PN do standardizovaného formátu PNML, který je přenosný mezi nástroji. Kromě exportování dokáže nástroj tento formát také přečíst.

Hlavní nevýhodou nástroje je uživatelské prostředí. Pro uživatele, který nástroj nikdy neviděl je vše velmi neintuitivní a možnosti pro modelování se musí buď složitě hledat v samotném prostředí, nebo s pomocí internetového vyhledávače.

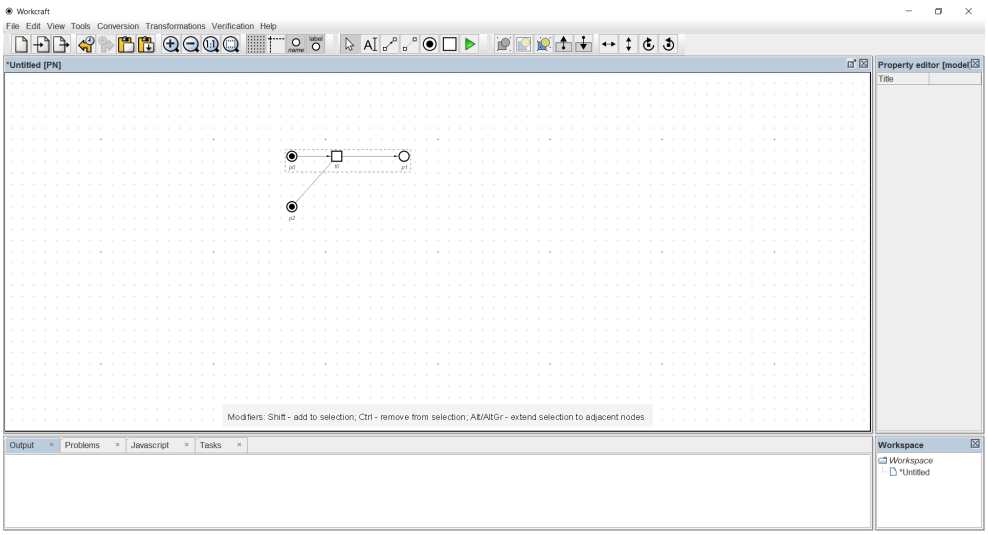


Obrázek 2.13: Prostředí nástroje CPN Tools

2.2.2 Workcraft

Workcraft je obecný nástroj pro interpretaci grafových modelů. Jednou ze sekcí jsou tedy i Petriho sítě. Nástroj se poprvé představil na konferenci DATE-2007 na univerzitě Booth 16. dubna 2007. První oficiální verze pak uzřela světlo světa o rok později. V současné době umožňuje modelovat a simulovat Petriho sítě, a k tomu nad nimi provádět některé operace.

Největší výhodou programu je oproti CPN Tools velmi jednoduché uživatelské rozhraní. Vše je přehledně uspořádáno, popsáno a velmi snadno se řídí. Nástroj také podporuje možnost sítí s testovacími hranami a hierarchické PN. Z možností analýzy a operací na PN dokáže nástroj v sítích detekovat deadlocky, a také (pokud je to možné) provést převod na konečný automat. Nevýhodou je, že lze modelovat pouze malý okruh PN. Program také sice podporuje širokou škálu výstupů od obrázků, graphviz, pdf nebo matematických modelů, nicméně výstup v obecném formátu PNML podporován není.

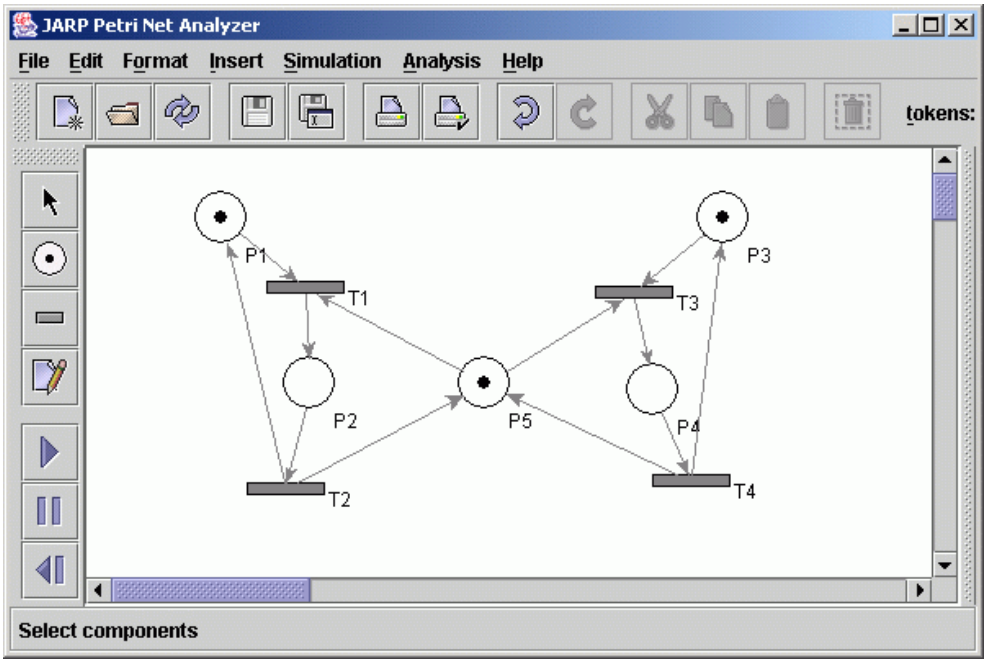


Obrázek 2.14: Prostředí nástroje Workcraft

2.2.3 JARP

Nástroj JARP je grafický editor a simulátor a jednoduchý analyzátor Petriho sítí. Napsaný je v jazyce Java a jeho vývoj byl ukončen v roce 2001.

Výhodou JARPu je podobně jako u předchozího nástroje Workcraft jednoduchost používání a možnost simulace. Nástroj také podporuje výstup v mnoha formátech, z čehož pro tuto práci je nejdůležitější výstup ve formátu PNML. Analýza je zde pouze omezená a obsahuje možnosti jako např. nalezení invariantů. Bohužel tento nástroj má v současné době více nevýhod než výhod. Tou největší je, že poslední zmínka o aktualizaci pochází z data 6. prosince 2001. Tato zastaralost způsobuje problémy se zprovozněním na novějších strojích s aktuálním Java Runtime Enviroment, který konstrukty obsažené v kódu nástroje JARP neumožňují spustit. Další nevýhodou je to, že podpora jakýchkoliv jiných konceptů, než klasických PN, zcela chybí.

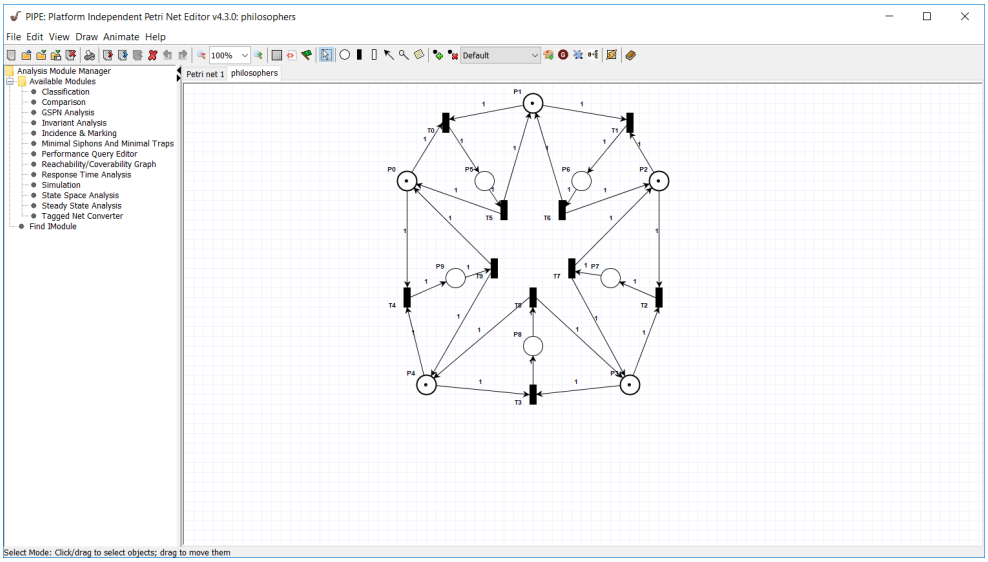


Obrázek 2.15: Prostředí nástroje JARP

2.2.4 PIPE

Jak je uvedeno ve zdroji [8], je PIPE open source projekt, nezávislý na platformě, který slouží pro modelování a analýzu Petriho sítí. Pro dobrou přenositelnost je celý program napsán v jazyce Java. Vznik nástroje se datuje na pomezí let 2002 a 2003 v rámci projektu MSc Group na katedře počítačů londýnské Imperial College.

PIPE má podobně jako někteří předchozí kandidáti výborně uzpůsobené GUI, které je i pro začínajícího uživatele velmi intuitivní a snadné. Zároveň také podporuje velké spektrum konceptů PN. Začíná na klasických Petriho sítích, dále také sítě inhibičními hranami, barevné a dokonce i časované PN. Hierarchická podpora je zde částečná. Další velkou výhodou je poměrně detailní analýza, která podporuje např. klasifikaci sítě, invarianty, graf dosažitelnosti, analýzu časovaných Petriho sítí aj. PIPE podporuje, podobně jako další zmíněné nástroje, mnoho možností exportu výstupu popisu PN. Ať už se jedná o grafický výstup, textový popis nebo datový soubor, nejdůležitějším aspektem pro tuto práci je výstup ve formátu PNML. Za nevýhodu bych považoval právě zmíněnou omezenost hierarchie a také složitější export výsledků analýzy. Tyto mínusy ale ani zdaleka nepřevažují veškeré výhody, a tudíž se tento nástroj stal nejvhodnějším pro modelování systémů použitých v této práci.



Obrázek 2.16: Prostředí nástroje PIPE

2.2.5 Ostatní

Veškeré prozatím zmíněné nástroje mají jednu hlavní společnou vlastnost. Tou je, že všechny jsou vyvinuté jako samostatný a nezávislý program. Existují ovšem i další typy nástrojů, které jsou vyvinuté jako doplňky pro různá vývojová prostředí. Z toho důvodu, že při zkoumání nástrojů se tyto typy neprojevily jako vhodní kandidáti pro tuto práci, budou uvedeny pouze 2 příklady takových doplňků.

Prvním zmíněným doplňkem bude nástroj zvaný **ePNK**. ePNK je plugin pro vývojové javovské prostředí jménem Eclipse. Hlavním účelem nástroje je možnost modelování Petriho sítí v tomto prostředí, a zároveň k němu poskytnout grafický editor. Nástroj ve svém základu podporuje pouze kontrolu modelu, ale umožňuje vývojářům rozšíření funkcí pomocí pluginů. Rozšířit lze jednak podporované typy Petriho sítí a jednak funkce pro analýzu a práci nad vytvořenými modely.

Druhým zmíněným doplňkem bude nástroj jménem **SimHPN**. Jedná se o doplňek vyrobený pro prostředí MATLAB. SimHPN nabízí sbírku nástrojů pro simulaci a analýzu. Dále také podporuje import funkcí z jiných grafických editorů Petriho sítí. Nástroj poskytuje uživateli příjemné grafické prostředí, které umožňuje intuitivní využití všech dostupných funkcí. Vyjma některých procedur jsou všechny procedury pro simulaci a analýzu z grafického prostředí dostupné.

2.3 PNML

PNML, neboli Petri net markup language, je značkovací jazyk pro popis Petriho sítí, založený na platformě XML. Zdroj [9] píše, že tato platforma přináší velkou řadu výhod, jako např. přenositelnost nebo rozšířítelnost pro nové koncepty Petriho sítí. Další výhodou uvedenou v [9] je existence mnoha programových nástojů a API pro zpracování a validaci formátu XML. Výhodou může být i výměna modelů Petriho sítí mezi různými nástroji. Za účelem podpory různých typů Petriho sítí se PNML zaměřuje na univerzálnost a flexibilitu.

2.3.1 Vlastnosti PNML

První vlastností, kterou zmíníme podle zdroje [9], je flexibilita. Tato vlastnost znamená, že je jazyk schopný reprezentovat libovolnou Petriho síť se všemi specifickými rozšířeními a vlastnostmi. PNML nesmí omezovat vlastnosti nějakého druhu Petriho sítě nebo dokonce vyžadovat upuštění od specifických informací Petriho sítě při konverzi do PNML. Za účelem dosažení této flexibility je Petriho síť považována za orientovaný graf se značkami, kde všechny specifické informace mohou být uloženy právě ve značkách [9]. Značka může být spojena s uzlem, hranou nebo se sítí samotnou. Jednotlivé prvky Petriho sítě jsou reprezentovány pomocí jednotlivých XML elementů a atributů. Síť tak může být popsána v přehledném formátu. PNML podporuje definici různých typů Petriho sítí. Podobně jako v jazyce XML existují definiční soubory (s koncovkou *.xsd*), u jazyka PNML jsou to soubory s koncovkou PNTD (*Petri net type definition*), někdy nazývány také jako meta modely. Vymezují platné značky pro patřičný typ Petriho sítě [9].

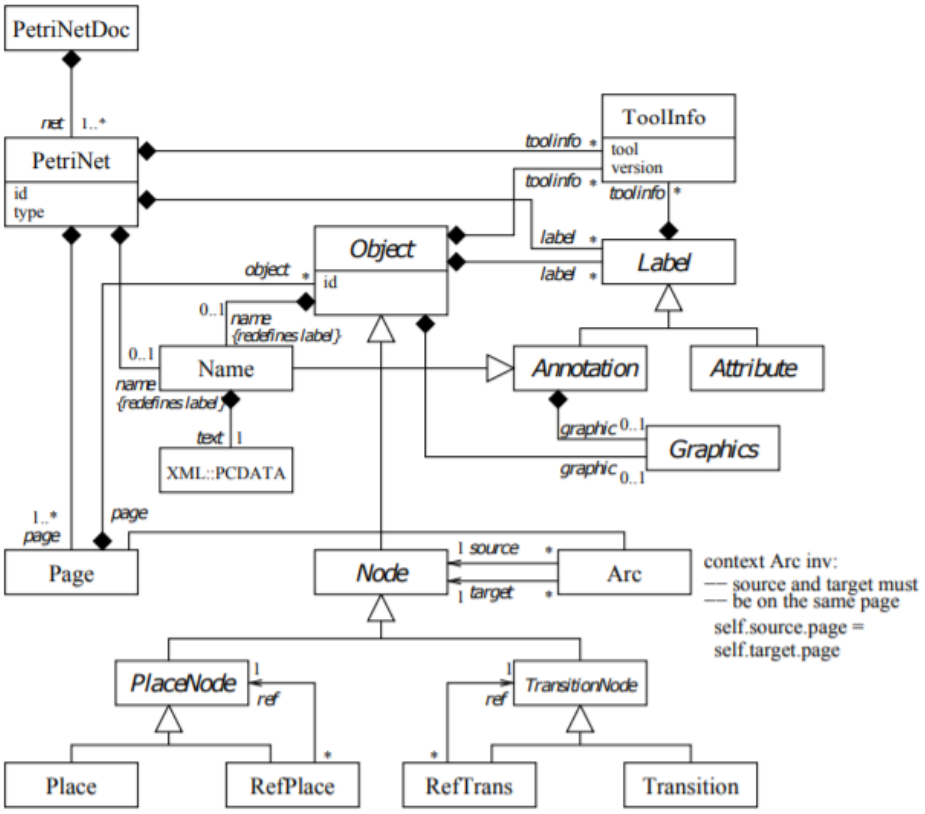
2.3.2 Struktura PNML

Pro poskytnutí a definování XML syntaxe PNML používá UML meta modely [9]. Základ PNML se dělí na 3 části:

- Meta model
- Feature Definition Interface
- Type Definition Interface

Účelem Meta modelu je definice základní struktury PNML souboru a všechny koncepty, které jsou sdíleny všemi druhy Petriho sítí. Pro definice nových vlastností Petriho sítí slouží Feature Definition Interface. Pro definice nových typů Petriho sítí slouží Type Definition Interface. Konkrétní XML syntax je pak definována mapováním konceptů těchto UML modelů na XML elementy [9].

PNML je standardizován podle ISO/IEC 15909 jako syntax pro tři druhy Petriho sítí: Place/Transition-Nets, High-level Petri Nets a Symetric Nets, jejichž definiční meta modely jsou přístupné na referenčních stránkách PNML [9].



Obrázek 2.17: Struktura PNML

2.3.3 Meta model

Meta model, také nazývaný jako PNML Core Model, je jádro PNML, ze kterého vychází a dědí další meta modely specifikující konkrétní typy Petriho sítí. Na nejvyšší úrovni meta modelu se nachází Petri net file (nebo také PetriNetDoc), který splňuje požadavky meta modelu. Tento dokument může obsahovat libovolný počet Petriho sítí [9].

2.3.3.1 Petriho sítě a objekty

Jak již bylo zmíněno, Petriho síť je interpretována grafovou strukturou. Tato struktura je reprezentována pomocí objektů (míst, přechodů, hran). Každý objekt má unikatní identifikátor, který může být využit pro odkazování se na tento objekt [9]. Objekty také mohou pro různé nástroje nést grafickou informaci v podobě souřadnice, velikosti, barvy, tvaru aj.

2.3.3.2 Stránky a odkazující uzly

Pro podporu hierarchičnosti Petriho sítí je v PNML navíc zavedený ještě jeden objekt s názvem Stránka. Ta v sobě pak obsahuje další objekty včetně dalších stránek. Lze tedy takto nadefinovat jednotlivé úrovně modelu. Standard PNML říká, že hrana může propojovat místa a uzly pouze na jedné stránce. Důvod pro toto omezení je takový, že hrana propojující uzly na různých stránkách nemůže být graficky zobrazena na jedné stránce [9].

2.3.3.3 Značení

Ke každému objektu v modelu lze přiřadit jeho popisek neboli *label*. Typicky značení reprezentuje jméno uzlu; počáteční značení místa; podmínku, časování či stráž přechodu nebo anotaci hrany [9]. Popisky může obsahovat i samotný model a stránky v něm obsažené. Takovým popiskům se říká *global labels*.

Značení jsou rozlišovány na 2 typy:

- anotace
- atributy

Anotace zahrnuje informace, které jsou typicky zobrazeny jako text u korespondujícího objektu. Narozdíl od anotace, atribut není zobrazen jako text u korespondujícího objektu, nýbrž nese informaci o grafické podobě objektu. Třídy pro značení, anotaci a atribut jsou v meta modelu pouze abstraktní [9].

2.3.3.4 Grafická informace

Každý objekt a každá anotace obsahuje grafickou informaci. Pro prvky typu uzel (místo a přechod) se jedná o absolutní pozici. Hrana může být tvořena buď úsečkou mezi dvěma uzly, nebo lomenou čárou skládající se z více bodů, v tom případě grafická informace obsahuje seznam těchto bodů. Anotace objektu se zobrazuje vedle patřičného objektu - grafickou informací je pak relativní pozice ke korespondujícímu objektu. Dalším případem může být uchování informace o tvaru objektu, velikosti a barvě [9].

2.3.4 Mapování meta modelu na elementy

Každá konkrétní třída PNML meta modelu je přeložena na odpovídající XML element. Mapování překladu je zobrazeno v tabulce 3.1. Tyto XML elementy tvoří klíčová slova jazyka PNML. Nenachází se zde žádné elementy pro značení, protože meta model pro ně nedefinuje konkrétní třídy. Konkrétní třídy pro značení jsou definovány konkrétními typy Petriho sítí [9].

Class	XML Element	XML Attributes
PetriNetDoc	*<pnml>*	xmlns: anyURI
PetriNet	*<net>*	id: ID type: anyURL
Place	*<place>*	id: ID
Transition	*<transition>*	id: ID
Arc	*<arc>*	id: ID source: IDRef (Node) target: IDRef (Node)
Page	*<page>*	id: ID
RefPlace	*<referencePlace>*	id: ID ref: IDRef (Place or RefPlace)
RefTrans	*<referenceTransition>*	id: ID ref: IDRef (Transition or RefTrans)
ToolInfo	*<toolspecific>*	tool: string version: string
Graphics	*<graphics>*	

Tabulka 2.1: Překlad PNML meta modelu na PMNL elementy

Parent element class	Sub-elements of <graphics>
Node, Page	*<position>*(required)
	<dimension>
PetriNet	*<fill>*
	<line>
Arc	*<position>*(zero or more)
	<line>
	<offset>(required)
Annotation	*<fill>*
	<line>
	<font>

Tabulka 2.2: Elementy v *<graphics>* elementu v závislosti na rodičovském elementu

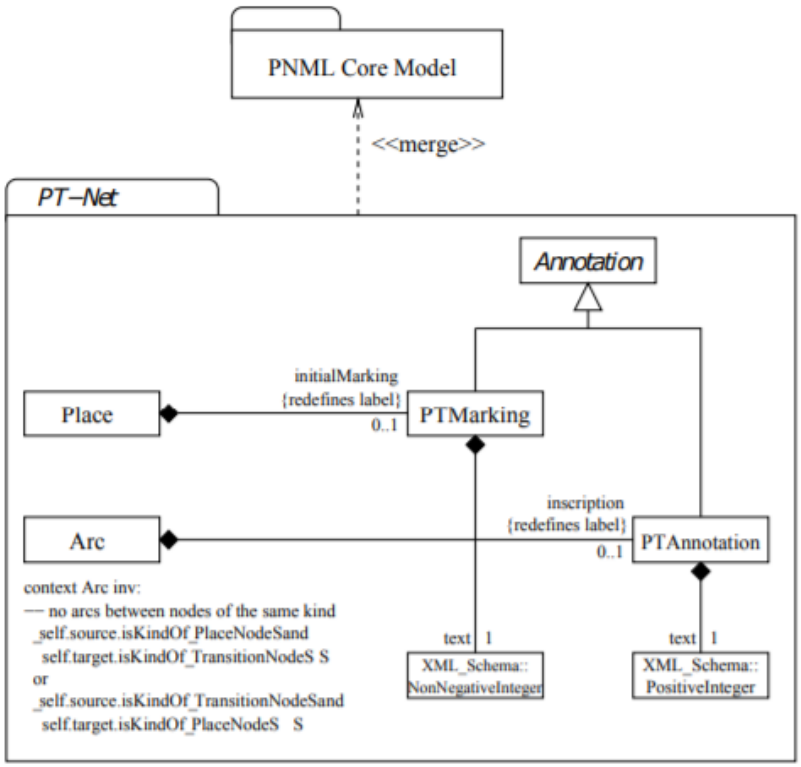
XML Element	Attribute	Domain
<*position*>	x	decimal
	y	decimal
<*offset*>	x	id:ID
	y	decimal
<*dimension*>	x	nonNegativeDecimal
	y	nonNegativeDecimal
<*fill*>	color	CSS2-color
	image	anyURI
	gradient-color	CSS2-color
	gradient-rotation	vertical, horizontal, diagonal
<*line*>	shape	line, curve
	color	CSS2-color
	width	nonNegativeDecimal
	style	solid, dash, dot
<*font*>	family	CSS2-font-family
	style	CSS2-font-style
	weight	CSS2-font-weight
	size	CSS2-font-size
	decoration	underline, overline, line-through
	align	left, center, right
	rotation	decimal

Tabulka 2.3: Grafické PNML elementy

PNML elementy a značení obsahují grafickou informaci. Struktura <*graphics*> elementu závisí na elementu, v němž se nachází. Konkrétní mapování zobrazuje tabulka 3.2. V každém uzlu je vyžadován element <*position*> značící absolutní pozici. Naopak element offset definuje relativní pozici (ke korespondujícímu uzlu) a je vyžadován u anotací. Ostatní elementy jsou volitelné. Pro hranu může být přítomno 0 a více elementů position, protože hranu může tvořit lomená čára. Každý element position udává jeden z prostředních bodů. Každá absolutní i relativní pozice představuje bod v kartézských souřadnicích (x, y). Tabulka 3.3 zobrazuje atributy, které se mohou vyskytovat uvnitř jednotlivých grafických elementů. Sloupec Domain obsahuje datový typ XML schématu, Kaskádových stylů (CSS2) nebo vyjmenování přípustných hodnot pro daný atribut. Element <*dimension*> definuje výšku a šířku uzlu. V závislosti na poměru výšky a šířky bude místo zobrazeno jako elipsa nebo kruh, přechod zase jako čtverec nebo obdélník. Elementy <*fill*> a <*line*> definují vnitřek a obrys korespondujícího elementu. Pro text značení slouží element <*font*> [9].

2.3.4.1 PNML pro P/T Petriho sítě

Pro P/T Petriho sítě zavádí PNML do svého jádra nový element zvaný *PTMarking*. Úkolem tohoto elementu je zavést do modelu počáteční značení v podobě přirozeného čísla. V modelu musí jak místa, tak hrany mít své označení. Hrany se v P/T konceptu označují elementem *PTAnnotation*. P/T modely se vyznačují tou vlastností, že místo může být spojeno pouze s přechodem a přechod také pouze s místem. Nelze tedy připojit jednou hranou 2 místa ani 2 přechody.



Obrázek 2.18: PNML pro P/T Petriho sítě

2.4 Existující řešení

V celé historii světa již pokusy o tvorbu nástroje pro převod z PNML do VHDL samozřejmě probíhaly. Nicméně je nutno podotkout, že jich zase také

nebylo příliš mnoho. Během průzkumu ohledně předchozích řešení byly nalezeny 2 významné zdroje.

2.4.1 Universidade Nova de Lisboa

Prvním výrazným nálezem je stránka

http://www.uninova.pt/fordesign/PNML2VHDL.htm

fungující k datu 28.4.2018 a patřící portugalské vysoké škole Universidade Nova de Lisboa. Stránka s titulem „PNML2VHDL – A translator from PNML to VHDL“, která byla vytvořena 20.1.2009, ve svém abstraktu sděluje, že se věnuje nástroji pro převod z jazyka PNML do syntetizovatelného popisu v jazyce VHDL. Koncept slibuje podporu P/T Petriho sítí s podporou testovacích hran a možností přidělování priorit jednotlivým přechodům.

Nástroj má mít relativně vyšší softwarové nároky na knihovny a frameworky v OS Windows, jako jsou .NET Framework a .NET Framework SDK. Pod toolem jsou samozřejmě také uvedena jména autorů. Bohužel při důkladném surfování po stránce si lze všimnout, že není dokončena a mnoho míst je stále uváděno jako „Under contruction“. Jedním z nich je bohužel i místo, kde by měl být nástroj k dispozici, a proto lze předpokládat, že nástroj možná nikdy nebyl dokončen, a pokud ano, tak nebyl zveřejněn.

2.4.2 Fakulta Elektrotechnická Českého Vysokého Učení Technického

Dalším nálezem je výrazné množství prací z FEL ČVUT. Mnoho akademiků se zabývalo modelováním nebo výzkumem Petriho sítí. Největši inspirací mi byla práce Doc. Hany Kubátové s názvem *Direct implementation of Petri net in FPGA*, k přečtení zde:

https://ddd.fit.cvut.cz/publ/2004/Kubatova_DESDEs.pdf

Práce popisuje metodiky, jak lze v HW realizovat a implementovat Petriho sítě a uvádí příklady modelů včetně zhodnocení chování komponent v obvodu.

Dále je známo, že fakulta disponovala nástrojem, který tento převod dákazal uskutečnit. Byl taktéž pojmenován názvem PNML2VHDL a byl součástí závěrečné práce studenta fakulty. Nicméně práci ani nástroj se mi již nepodařilo dohledat.

Kapitola 3

Návrh

3.1 Návrh Petriho sítě jako číslicového obvodu

V teoretické části práce je uvedeno, že konceptů Petriho sítí existuje mnoho typů na základě jejich vyjadřovacích schopností. Je tedy v první řadě před samotným návrhem vhodné si určit, jaký konkrétní typ PN budeme realizovat. Vybraný typ bude po vysvětlení všech variant konkrétně specifikován.

Před samotným návrhem je zapotřebí si uvědomit, že každý rozšiřující koncept sítě musí být schopný realizovat i síť úrovně s nižší vyjadřovací schopností. Každé rozšíření si nicméně žádá výrazný zásah do číslicové logiky, a proto není vhodné začínat od těch nejsložitějších konceptů, ale naopak postupně rozšiřovat od nejjednoduššího. V následujících podkapitolách budou postupně rozebrány návrhy jednotlivých prvků dané PN a průběžně zobrazena rostoucí složitost jejich implementace, počínaje od nejjednodušší až po nalezenou rozumnnou mez složitosti.

Bylo zvoleno, že všechny návrhy budou z důvodu pokusu o co nejvěrohodnější kopii modelu vycházet z principu realizace jak místa, tak přechodu jako samostatné součástky s říditelným tokem informace.

3.1.1 Petriho síťě řízené událostmi

V Petriho sítích řízených událostmi je pro návrh nejdůležitější ta informace, že maximální počet tokenů v místě může být maximálně jedna.

3.1.1.1 Požadavky na realizaci místa

Nutným požadavkem pro realizaci místa je tedy schopnost udržení tokenu na dobu nezbytně nutnou. Zároveň po odpalu je zapotřebí token z místa odstranit, tedy místo musí na požadavek token uvolnit. Místa v tomto typu sítě

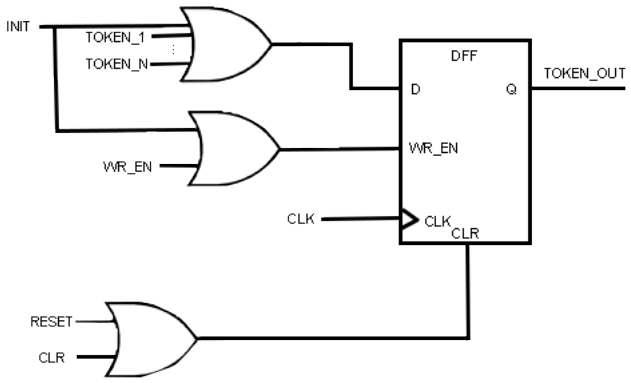
mohou při inicializaci sítě obsahovat token. Obvod tedy musí umožňovat inicializovat sám sebe. Jakýkoliv koncept Petriho sítě neomezuje počet vstupních a výstupních hran pro místa a přechody. Kdykoliv tedy se na některé hraně token objeví, místo musí na tuto událost správně zareagovat.

3.1.1.2 Realizace místa

Ideální komponenta, která splňuje požadavek na udržení 1 tokenu na dobu nezbytně nutnou je klopný obvod typu D s povolením zápisu. Ten v základu obsahuje 5 portů:

- D - vstupní data
- Q - výstupní data
- WR_EN - povolení zápisu dat
- CLK - vstup hodinového signálu
- CLR - vymazání hodnoty

Hlavním využitím portu CLR je vynulování hodnoty na datovém výstupu. Toho lze využít jak pro reset obvodu, tak pro vymazání tokenu. Stačí pouze příslušné signály sloučit pomocí hradla logického součtu (OR). Pro zpracování tokenu na datovém vstupu lze podobným způsobem sloučit všechny vstupní datové hrany od přechodů. Inicializace má poté takovou speciální vlastnost, že sama dokáže uložit token přímo na výstup. Jedná se tedy o signál, který dokáže zavést hodnotu logické 1 jak na datový vstup, tak na povolení zápisu hodnoty. Toto je také řešeno s pomocí hradla logického součtu. Celé schéma návrhu místa pro Petriho síť řízenou událostmi je na následujícím obrázku.



Obrázek 3.1: Návrh místa pro Petriho sítě řízené událostmi

3.1.1.3 Požadavky na realizaci přechodu

Základním požadavkem pro přechod je umožňovat přesun tokenů mezi místy. Je k tomu ale zapotřebí splňovat několik podmínek. První z nich je, že přechod musí být aktivní, tudíž na všech datových vstupech musí být token. Druhou je nutnost brát v úvahu tu možnost, že nebude jediný aktivní v celém modelu, takže musí odesílat signál o své aktivitě a čekat, jestli bude vybrán pro odpal. Třetí tedy je, že musí obsahovat logiku, která na základě předchozích dvou podmínek umožní propuštění tokenu na výstup.

3.1.1.4 Realizace přechodu

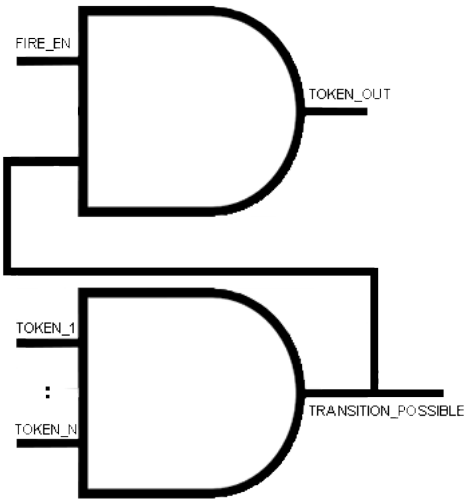
Zde není díky paměti z míst zapotřebí žádná sekvenční logika. Je pouze třeba detekovat 2 podmínky které zní:

1. Jsou všechny datové vstupy v logické 1?

2. Platí předchozí podmínka a je signál pro povolení v logické 1?

Z těchto podmínek plyne, že je lze detekovat s pomocí logického součinu. Počet hradel nicméně závisí na konkrétním počtu vstupů. Pro 1 vstup je detekce aktivnosti pouze příslušný vodič. Pro 2 vstupní tokeny se jedná o 1 hradlo typu AND. Pro více než 2 vstupní tokeny lze použít buď hradlo AND o stejném

počtu vstupů, nebo strom AND hradel o 2 vstupech. Výstup detekce aktivnosti zapojený do hradla AND společně se signálem aktivace přechodu poté vytvoří výstupní token. Schéma přechodu pro Petriho síť řízenou událostmi je vyobrazeno na následujícím obrázku.



Obrázek 3.2: Návrh přechodu pro Petriho sítě řízené událostmi

3.1.1.5 Realizace hran

Díky faktu, že hrana může nést pouze 1 token, se jedná vždy o informaci o velikosti 1 bit. Na realizaci tedy stačí pouze jedna vodivá cesta. Hrany jsou reprezentovány přímým spojením příslušných datových vstupů a výstupů z míst a přechodů na základě realizovaného modelu.

3.1.2 P/T Petriho sítě bez násobných hran

Prvním rozšířením základního konceptu bude přidání možnosti, aby místa mohla obsahovat neomezený počet tokenů, a zároveň mohla disponovat kapacitami, které v případě překročení mezní hodnoty zablokují vstupní přechod.

3.1.2.1 Požadavky na realizaci místa

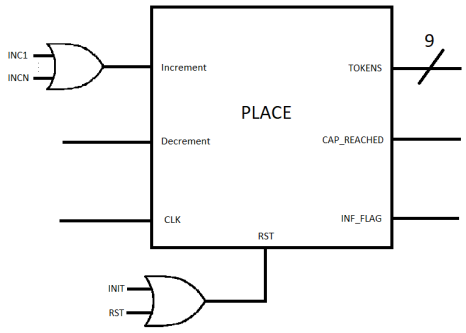
Místo musí tentokrát být schopné uschovat tokenů neomezené množství. Zároveň ale také musí být schopno dodržovat stanovenou kapacitu. Při odpalu je tedy třeba správně zareagovat na přidávání a odebírání tokenu. Místa i zde

mohou mít svou počáteční hodnotu. Narozdíl od Petriho sítí řízených událostmi ale může být tokenů od počátku neomezeně. Stejně jako v předchozím případě, i zde může být vstupních hran neomezené množství. Je tedy i zde třeba správně zareagovat na vstup dat.

3.1.2.2 Realizace místa

První věcí, kterou je zde zapotřebí brát v úvahu, je fakt, že hardware nedisponuje neomezenými prostředky. Je proto nutné brát v úvahu určité limity na počet tokenů. Srdcem součástky tedy zde musí být komponenta, která si dokáže nejen pamatovat nějaké celé číslo, ale také je schopná jej na povel přidat nebo odebrat. Takovou ideální komponentou je n-bitový obousměrný čítač. V implementaci jsem se omezil na maximální hodnotu 255 tokenů, vyšší hodnota poté již nastaví flag signalizující nekonečný počet. Tento limit byl sice nastaven pevně, ale lze jej implementačně zvýšit. Flag nekonečno způsobí, že jakékoliv další přičítání a odčítání je nadále ignorováno.

Princip inicializace a resetu zde byl změněn. Součátka reprezentující místo bude mít taktéž 2 resetovací signály (init a reset), které oba uvedou hardware do výchozího počtu tokenů. Tyto signály jsou oddělené hradlem logického součtu, a stejně jako v předchozím návrhu, i zde jeden slouží pro lokální reset a druhý pro globální. Pro zjištění příchozího tokenu z výstupních hran přechodů je taktéž použita redukce typu OR pro všechny vstupní hrany signálu pro inkrementaci. Součástka poté v případě, že při náběžné hraně hodinového signálu nedochází k resetu, kontroluje, zdali při aktivním inkrementačním signálu nejsou zároveň aktivní signály pro překročení kapacity a příznak nekonečna. V případě splnění podmínky dále součástka kontroluje, zda-li čítač nedosáhl maximální hodnoty. Pokud dosáhl, nastaví se příznak nekonečno na hodnotu logické 1. Pokud ne, čítač zvýší svou hodnotu o 1. Příznak pro signalizaci kapacity se aktivuje v případě, že čítač dosáhl limitní kapacitní hodnoty. Ta je implementačně realizovaná porovnáním s vloženou konstantní hodnotou, což lze implementovat snadno s pomocí kombinační funkce. Při aktivním signálu pro dekrementaci musí součátka brát v potaz 2 další faktory. Pro odečtení nesmí být nastaven flag pro nekonečno, a zároveň hodnota čítače nesmí být nulová. V případě splnění podmínky se čítač sníží o jedna. Celkové zapojení pomocí hradel a klopných obvodů by v tomto případě bylo již složité, takže bude součátka zobrazena na následujícím obrázku pouze jako blok s chováním popsaným v tomto odstavci.



Obrázek 3.3: Návrh místa pro P/T Petriho sítě bez násobných hran

3.1.2.3 Požadavky na přechod

Úkolem přechodu je zde aktivovat přičítací signál pro výstupní místa. Pro možnost odpalu musí být přechod samozřejmě aktivní, tudíž musí obsahovat logiku pro detekci aktivnosti. I zde je třeba brát v potaz možnost aktivnosti vícero přechodů, tím pádem je nutno upřesnit, který přechod se bude odpalovat formou indikace a vyčkání na odpal.

3.1.2.4 Realizace přechodu

Stejně jako v případě přechodů v předchozím případě, i zde si vystačíme s kombinační logikou. Podmínky pro odpal jsou zde 3 a od prvního případu se i původní 2 mírně liší:

1. Poskytují všechna vstupní místa nějaký token?

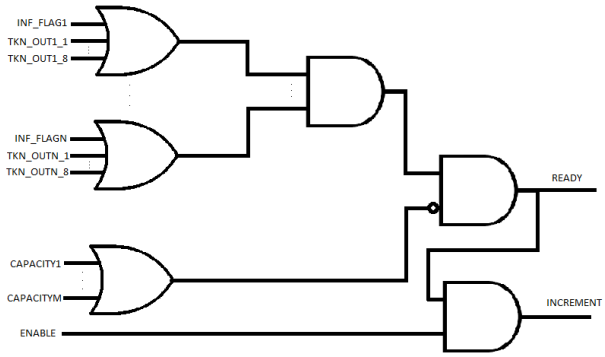
2. Platí předchozí podmínka a nejsou překročeny kapacity výstupních míst?

3. Platí předchozí 2 podmínky a je odpal aktivován?

Jelikož každé místo produkuje 8bit výstupní hodnotu, je zapotřebí zjistit, zda-li tato hodnota je nenulová. To lze provést redukcí typu OR. Zároveň ale v případě, že místo indikuje nekonečný počet tokenů, tak místo produkuje tokeny vždy a hodnota datového výstupu bude nulová. Je tedy zapotřebí k OR redukci zavést i příslušný flag pro nekonečno. Jelikož vstupních míst může být neomezené množství, je zapotřebí zjistit, zda-li tokeny obsahují všechna. Takovou operaci můžeme provést redukcí typu AND všech tokenových indikací.

Pro doplnění první podmínky o druhou, musíme posbírat údaje z flagů pro překročení kapacity od výstupních míst. Ty poté také projdou redukcí typu OR, aby bylo jasné, zdali všechny jsou nastavené na hodnotu 0. Pro indikaci aktivnosti tedy musí platit, že podmínka 1 a zároveň negace OR redukce kapacit musí být v logické 1.

Pro aktivaci odpalu pak tedy musí platit podmínka 2 a navíc být aktivní signál pro povolení odpalu, což je pouze operace logického součinu (AND).



Obrázek 3.4: Návrh přechodu pro P/T Petriho sítě bez násobných hran

3.1.2.5 Realizace hran

Hrany jsou zde realizovány pomocí propojení vstupních portů míst a přechodů k příslušnému výstupnímu portu místa/přechodu podle modelu. Jedinou výjimku zde tvoří port pro vstupy kapacit u vstupu přechodů, které je zapotřebí zapojit k příslušným portům výstupních přechodů.

3.1.3 Rozšíření o násobnost hran

Posledním návrhem, o který jsem se v rámci této práce pokusil, bylo rozšíření na plnohodnotnou P/T síť. I přes to, že teoretický návrh se podařil, implementační složitost by značně ztížila tvorbu výsledného nástroje na převod do VHDL, a proto jsem se rozhodl zůstat u předchozí varianty návrhu.

3.1.3.1 Požadavky na realizaci místa

Zde jediným rozdílem oproti předchozímu návrhu bude nutnost reagovat na možnost přičítání a odčítání vícero tokenů. Zbývající model chování zůstává

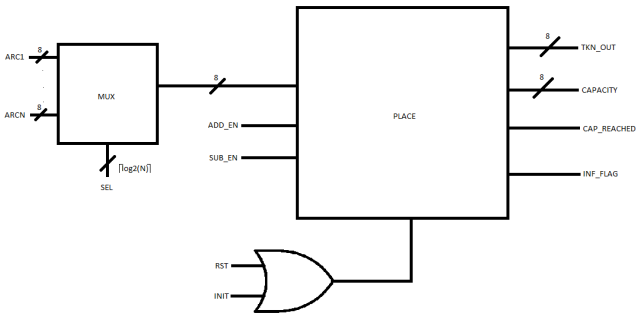
stejný.

3.1.3.2 Realizace místa

Jak již bylo zmíněno v předchozím návrhu, bylo zapotřebí omezit maximální počet tokenů v místě. V tomto návrhu by toto omezení mělo být bráno i pro násobnost hrany, jelikož nedisponujeme neomezenými prostředky. Je samozřejmě také nesmyslné udávat maximální násobnost vyšší než omezení kapacity pro místo. Násobnost hrany je proto vhodné limitně omezit na stejnou hodnotu, čili 8 bitů. Znamená to tedy, že všechny vstupy pro součet a vstup pro rozdíl bude o velikosti 8 bitů.

Výběr tentokrát nemůže fungovat stejným způsobem jako v předchozím případě. Původně byla totiž OR redukce pouze signalizace přičtení jedničky. Zde je třeba vybrat číslo z jednoho konkrétního vstupu. K takovému účelu ideálně poslouží multiplexor datových vstupů jak pro součet, tak pro rozdíl. Dále pak bude zapotřebí říct, že součet nebo rozdíl má proběhnout. To lze provést pomocí řídících vstupů pro povolení sčítání a rozdílu.

Reset a inicializace jsou zde totožné jako v předchozím případě. Stejně tak výstup počtu tokenů, flagu pro nekonečno a dosažení kapacity. Co zde musí být nové, je výstup, který udává kapacitu konkrétního místa. Konkrétní význam bude uveden v návrhu přechodu.



Obrázek 3.5: Návrh místa pro P/T Petriho sítě

3.1.3.3 Požadavky na realizaci přechodu

Úkolem přechodu je zde přenášet správný počet tokenů po výstupních hranách na všechna výstupní místa. Přechod pro možnost odpalu musí být ak-

tivní, nicméně aktivnost zde obsahuje výraznější omezení než v předchozím případě. Vyčkání na výběr odpalu je také nutností.

3.1.3.4 Realizace přechodu

Protentokrát již nastává případ, že podmínky pro odpal začínají být složitější. Stále nicméně si lze vystačit s kombinační logikou. Těmi podmínkami tentokrát jsou:

1. Každá vstupní hrana do přechodu musí obsahovat minimální požadované množství tokenů.
2. Přechod nesmí být aktivní v případě, že kapacita kteréhokoliv výstupního místa by po odpalu byla překročena.
3. Při splnění předchozích 2 podmínek a následného povolení přechodu se musí po každé výstupní hraně přenést správný počet tokenů.

Na počátku musí tedy každý konrétní přechod vědět, jaká minimální hodnota musí z každé vstupní hrany přijít. Z důvodu slova minimální bude tedy zapotřebí implementovat vyhodnocení výrazu pro $n \in \mathbb{N}$ vstupních míst. Pojmenujeme-li vstupní port přechodu detekující počet tokenů TKN_IN a násobnost vstupní hrany ARC_CONST, musíme vyhodnotit výraz, pro všechny $k \in \{1, ..., n\}$:

$$TKN\_IN_k \geq ARC\_CONST_k$$

Aby nebylo třeba používat sekvenční komparátor, budou se čísla vhodně znaménkově interpretovat. Ideální komponenta, která dokáže podmínku detekovat, je sčítačka/odčítačka v doplňkovém kódu s detekcí přetečení. V případě, že po vypočítání výrazu $TKN\_IN_k - ARC\_CONST_k$ pro všechna k zůstane flag pro přetečení nulový, znamená to, že hodnota TKN_IN_k musí být větší nebo rovna násobnosti hrany, a proto je podmínka splněna.

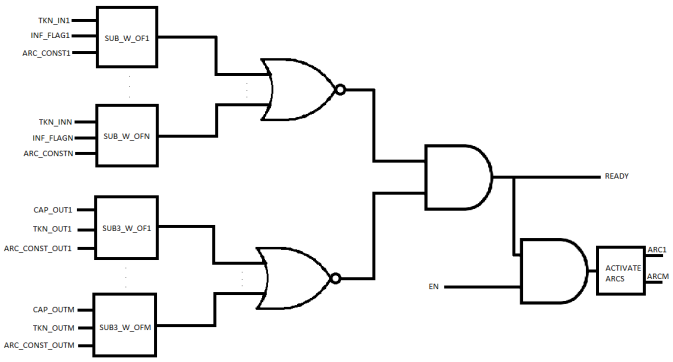
Podobnou logiku využijeme i při zjišťování druhé podmínky. Ta říká, že počet tokenů ve výstupním místě po přičtení hodnoty výstupní hrany nesmí překročit kapacitu. Musí být tedy implementován výraz pro $m \in \mathbb{N}$ výstupních míst. Pojmenujeme-li signál pro počet tokenů ve výstupním místě TKN_OUT, signál pro kapacitu výstupního místa CAP_OUT a násobnost výstupní hrany přechodu ARC_CONST_OUT, musíme vyhodnotit výraz pro všechny $l \in \{1, ..., m\}$:

$$TKN\_OUT_l + ARC\_CONST\_OUT_l \leq CAP\_OUT_l$$

čili lépe přepsáno jako

$$CAP\_OUT_l - TKN\_OUT_l - ARC\_CONST\_OUT_l \geq 0$$

Na vyhodnocení je tedy třeba opět použít sčítačku/odčítačku s detekcí přetečení pro daný výraz. Platí-li oba vyhodnocené výrazy, pak se aktivuje signál aktivity přechodu do hodnoty logické 1. Přijde-li poté signál k aktivaci přechodu, každá výstupní hrana nabyde hodnoty násobnosti ze zadaného modelu.



Obrázek 3.6: Návrh přechodu pro P/T Petriho sítě

3.1.3.5 Realizace hran

Hrany jsou stejně jako v předchozích modelech reprezentovány spoji mezi příslušnými výstupními a vstupními signály mezi místy a přechody na základě zadaného modelu.

3.2 FPGA realizace

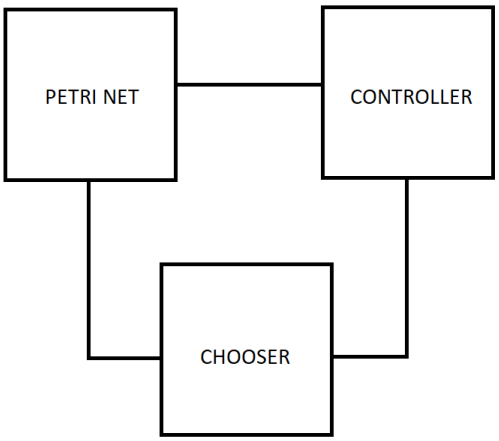
Návrh realizace míst, přechodů a hran umožňuje obvodu (realizovaném podle zadaného modelu) podporovat chování, které je dané definicí pro Petriho sítě. Aby bylo tohoto chování dosaženo, je zapotřebí správným způsobem stimulovat řídící vstupy. Hardware pro Petriho síť je tedy pouze datová cesta pro nějaký výsledný obvod.

Pro stimulaci řídících signálů je zapotřebí do obvodu přidat komponentu řadič. Podmínkou pro každý řadič Petriho sítě je znalost topologie sítě kvůli správnému přesunu tokenů.

V poslední řadě je třeba stále myslet na fakt, že v případě aktivnosti vícero přechodů musí být odpálen pouze jeden náhodně zvolený. Řadič tuto funkci bohužel zvládnout nedokáže, a proto se do návrhu musí zavést komponeta pro náhodný nebo pseudonáhodný výběr. Tuto komponentu jsem nazval výrazem

„Chooser“ a jejím hlavním úkolem je na základě vstupních signálů vybrat jeden z těch, které jsou uvedeny ve stavu logické 1.

Pro další návrh se bude uvažovat návrh sítě pro koncept P/T Petriho sítě bez násobných hran. Blokový návrh celkové FPGA realizace je na následujícím obrázku.



Obrázek 3.7: Blokové schema kompletního systému běžícím v FPGA

3.2.1 PETRI NET blok

Tento blok je srdcem celého obvodu. Obsahuje přesný počet správně propojených míst a přechodů na základě modelu systému.

Při $n \in \mathbb{N}$ místech a $m \in \mathbb{N}$ přechodech bude u vícebitových signálů každému místu a přechodu přiřazen jeden konkrétní bit. Vstupní porty bloku jsou tyto:

- n-bitový signál Decrement, který na základě aktivace příslušného bitu ubere z místa, které mu je přiřazeno, právě 1 token
- signál CLK, který slouží jako hodinový vystup pro paměť tokenů
- signál INIT, který uvede místa do výchozího stavu
- signál RST, který slouží k uvedení celého obvodu do výchozího stavu (v našem případě místa a řadič)

- m-bitový signál EN, který na základě aktivace příslušného bitu odpálí přechod, kterému je signál přiřazen za podmínky, že je přechod aktivní

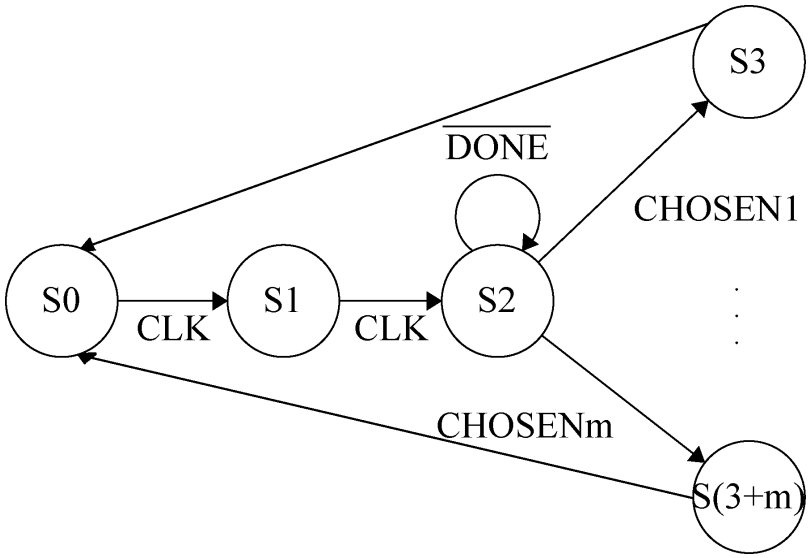
Výstupem budou tyto porty:

- 8*n bitový signál TOKENS, který bude po n-ticích ukazovat stav tokenů na všech místech
- n-bitový signál CAP_REACHED, který bude na příslušném bitu signalizovat překročení kapacity místa
- n-bitový signál INF_FLAG, který na příslušném bitu signalizuje, že místo obsahuje neomezený počet tokenů

3.2.2 CONTROLLER blok

CONTROLLER, neboli česky řadič, je mozkem celého obvodu. Jedná se o konečný automat, který má za úkol správně aktivovat řídící signály pro datovou cestu ve správném časovém okamžiku za účelem dosažení požadovaného chování.

Každý řadič musí být uzpůsoben konkrétnímu případu využití, a proto nelze uvést přesný předpis, jak přesně musí být realizován. Lze ale uvést příklad toho, jak by mohl vypadat. Následující obrázek zobrazuje kostru obecného řadiče, který by při implementaci prováděl běh Petriho sítě největší maximální rychlostí.



Obrázek 3.8: Ukázka modelu možného řadiče

Významy jednotlicých stavů, když předpokládáme $n \in \mathbb{N}$ míst a $m \in \mathbb{N}$ přechodů jsou:

- S0: Stav, který aktivuje signál pro inicializaci a uvede tím místa do výchozího stavu.
- S1: Stav, který pošle signál, že CHOOSER má začít vybírat přechod.
- S2: Stav, který čeká na odpověď CHOOSERU. V momentě, kdy CHOOSER vybere stav, ohlásí, že výběr byl dokončen a společně s ním bude i platný signál, který uvádí vybraný přechod.
- S3 - S(3+m): Stavy, které aktivují odpálení vybraného přechodu a zároveň aktivují odečtení tokenu ze vstupních míst.

Porty pro vstup do řadiče v takovéto kostře běhu musí být:

- hodinový signál CLK
- signál pro uvedení do výchozího stavu RST
- signál DONE, který upozorňuje na hotový výběr CHOOSERU
- m-bitový signál CHOSEN, který v případě aktivního signálu DONE udává vybraný aktivní přechod

Porty pro výstup do řadiče v takovéto kostře běhu musí být:

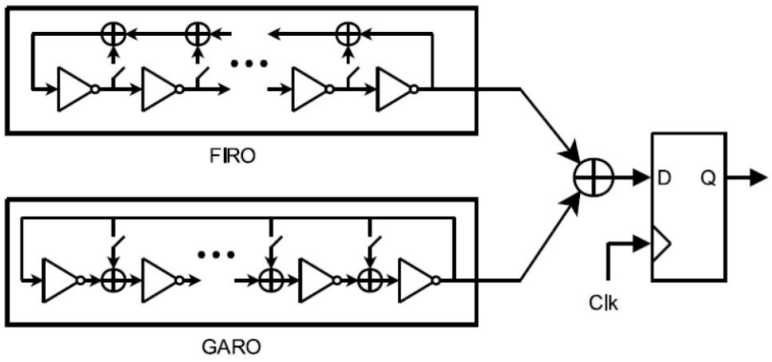
- inicializační signál INIT
- signál CHOOSE, který aktivuje komponentu CHOOSER
- n-bitový signál Decrement, který určuje, ze kterého místa se má odebrat token
- m-bitový signál EN, který odesílá vybranému přechodu signál k odpálení

3.2.3 CHOOSER blok

Pro výběr náhodného přechodu je zapotřebí implementovat mechaniku, která dokáže na základě signálů indikujících aktivitu přechodů určit, který má řadič odpálit. Je k tomu tedy zapotřebí implementovat generátor buď náhodných, nebo pseudonáhodných čísel s další logikou. Stejně jako tomu bylo v případě řadiče, i zde je implementace čistě v rukách návrháře. Následující popis uvádí pouze příklad implementace, který byl použit pro jiný projekt a je obecně vhodný. Skládá se z následujících bloků:

3.2.3.1 RANDOM

Zvolený generátor je TRNG zvaný FIGARO. Ten vzniká tak, že výstupy generátorů typu FIRO a GARO projdou hradlem typu XOR a uloží se do LSFR. Všechny 3 komponenty pracují s polynomy nad GF(2^x). Každá „buňka“ FIRO a GARO funguje jako invertor a každá „buňka“ LFSR funguje jako registr. Pro zlepšení náhodnosti se nepoužívá inicializace.



Obrázek 3.9: Generátor náhodných čísel FIGARO

Z důvodu, že komponenta nevyužívá inicializaci ji není možné simulovat. Problémy nastávají také se syntézou, protože komponenta obsahuje vstupy pouze pro resetování a hodinový signál a „nějaký“ výstup. Mnoho syntézních nástrojů proto při běhu takové komponenty ignoruje. Z toho důvodu, pokud je to možné, je třeba upravit nastavení syntézního nástroje.

3.2.3.2 COUNTER

Counter, neboli česky čítač, je klasická součástka, která při náběžné hraně hodinového signálu zvýší svou vnitřní hodnotu o 1. Důležitou vlastností counteru v komponentě CHOOSER je, že je inicializován na hodnotu vygenerovanou z komponenty RANDOM.

3.2.3.3 SHIFTER

Komponenta SHIFTER funguje tím způsobem, že pracuje s tolika registry, kolik přechodů obsahuje model. Na počátku své práce nastaví jeden zvolený registr na hodnotu logické 1. Dále pak na základě signálu, který určuje povolené přechody, zašle signál k příslušným registrům k aktivaci povolení zápisu. Následně probíhá posuv logické 1 z prvního registru přes pouze ty registry, ve kterých je povolený zápis. Počet posuvů je dán hodnotou čítače. Výsledkem je signál, který obsahuje náhodně vybranou hodnotu ze vstupu.

3.3 Návrh softwaru pro převod

První věc, kterou bylo potřeba si rozmyslet, bylo použití programovacího jazyka na tvorbu nástroje. Na základě vlastních zvyklostí z průběhu vysoké školy mi k tomuto účelu byl nejsympatičtější jazyk C++.

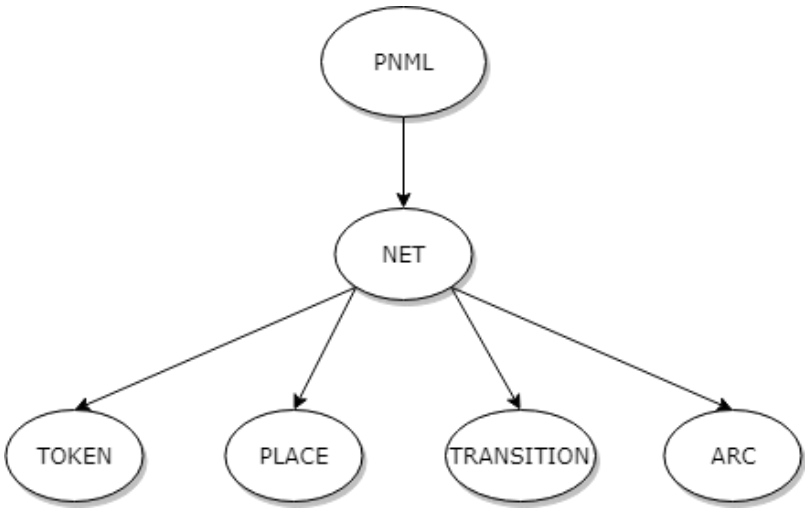
3.3.1 LibXML2

Jelikož formát PNML je založený na formátu XML, bylo by vhodné v rámci zjednodušení práce s parsováním XML dokumentů podle standardu najít již hotový nástroj. K tomuto účelu ideálně poslouží LibXML2. Jedná se o volně dostupný nástroj, který je přenositelný na většinu známých platforem, a cílem autorů je implementovat co nejstriktnější dodržování specifikací XML.

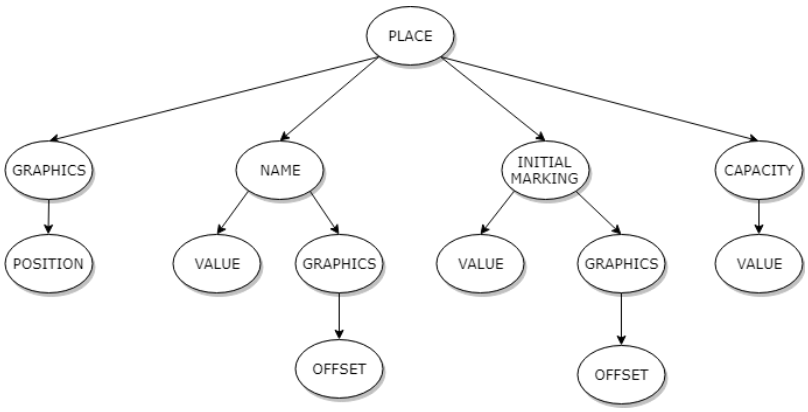
Nástroj poskytuje funkce a datové struktury, které dokáží jak ověřit/validovat vstupní formát souboru, tak velmi usnadnit průchod XML strukturou a získávání dat z ní. Napsaný je čistě v jazyce C, ale tvůrci poskytují wrappery pro další jazyky, kterými jsou např. C++, C#, Python a další.

3.3.2 PNML strom

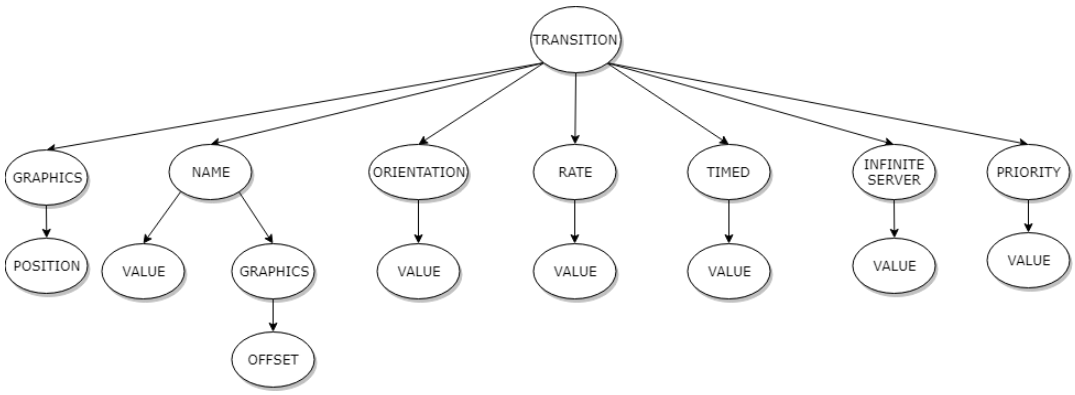
Aby mohla být data ze souboru správně získána, je zapotřebí znát jejich strukturu. U jazyků typu XML víme, že struktura značek je strom. Díky popisu formátu PNML známe také názvy uzlů, které hledáme. Funkce LibXML2 nicméně umožňují se ve stromu pohybovat nejen z rodiče na potomka, ale v rámci úrovně stromu také mezi vedlejšími uzly. Pro usnadnění pohybu je vhodné jednotlivé úrovně stromu vygenerovaného souboru znát. Struktura bude zobrazena na následujícím obrázku a pro zjednodušení bude od každého typu uzlu v úrovni ukazovat pouze jeden prvek i přesto, že jich může daná úroveň obsahovat nelimitované množství. Graf bude zároveň rozčleněn na několik částí vzhledem k jeho velikosti.



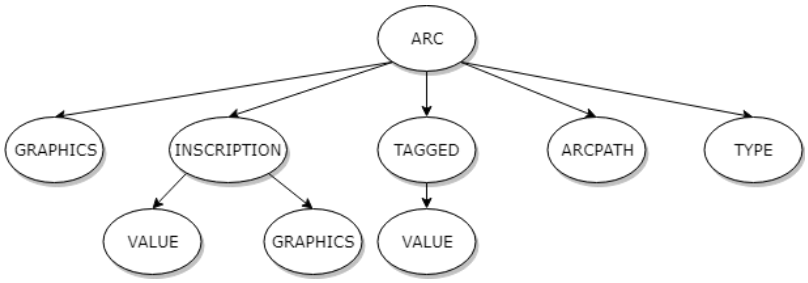
Obrázek 3.10: Kořen PNML stromu



Obrázek 3.11: Strom informací o místě



Obrázek 3.12: Strom informací o přechodu



Obrázek 3.13: Strom informací o hraně

Z návrhu, informací o PNML a struktuře PNML stromu je tedy zapotřebí dohledat, jaké informace budeme ke každé převáděné komponentě potřebovat. Pro implementaci míst musíme získat tyto informace:

- ID - získáme z atributu *id* u tagu *<place>*
- Inicializační hodnota - získáme z průchodu stromem cestou PLACE → INITIAL MARKING → VALUE (hodnota se nachází mezi tagy *<value>* a *</value>*)
- Kapacitu - získáme z průchodu stromem cestou PLACE → CAPACITY → VALUE (hodnota se nachází mezi tagy *<value>* a *</value>*
- Vstupní hrany a jejich počet - získáme z atributů *source* a *target* tagů *<arc>*

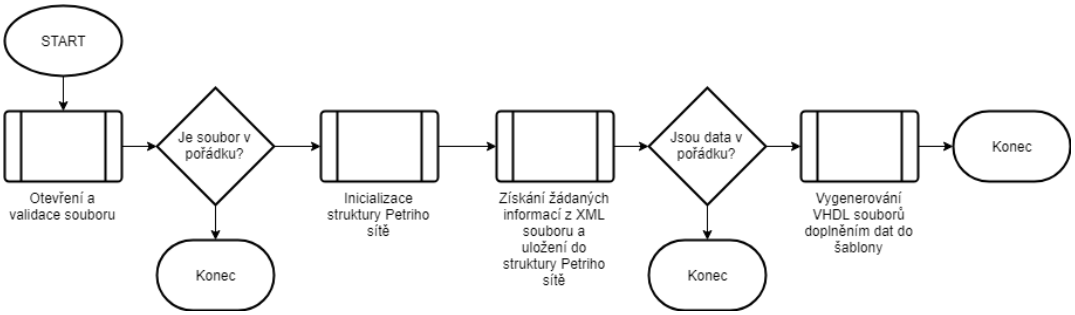
Pro implementaci přechodů musíme získat tyto informace:

- ID - získáme z atributu *id* u tagu *<transition>*

- Vstupní hrany a jejich počet - získáme z atributů *source* a *target* tagů *<arc>*
- Výstupní hrany a jejich počet - získáme z atributů *source* a *target* tagů *<arc>*

3.3.3 Postup převodu

Samotný převod se pak bude řídit následujícím postupem.



Obrázek 3.14: Vývojový diagram převodu

Kapitola 4

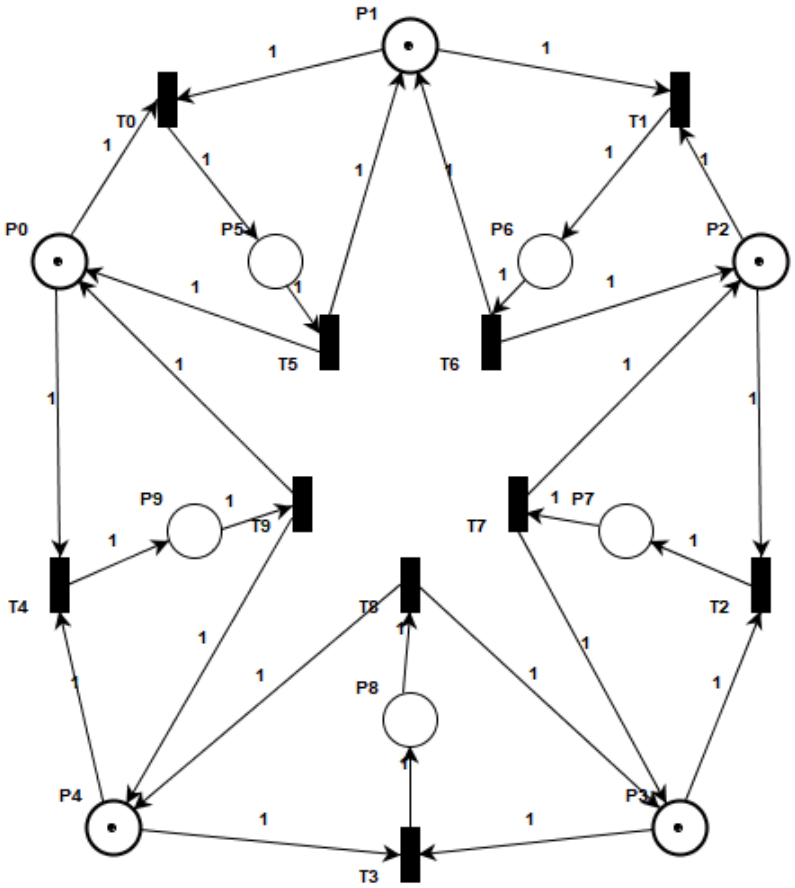
Realizace a testování

4.1 Příprava a analýza modelů

Před samotným převodem je třeba si namodelovat nějaké systémy pro převod. Pro účel této práce využiji 2 jednoduché - model problému večeřících filosofů a model producent-konzument. Pro zjištění vlastností modelů použiji vestavěné analytické funkce nástroje PIPE.

4.1.1 Večeřící filosofové

Tento model se využívá k simulování chování přístupu procesů k výpočetním zdrojům tak, aby nedošlo k deadlocku. Petriho síť řešící tento problém vypadá takto:



Obrázek 4.1: Petriho síť pro model Večeřících filosofů

Výstup z analýzy nástroje PIPE vypadá takto:

State Machine	false
Marked Graph	false
Free Choice Net	false
Extended Free Choice Net	false
Simple Net	false
Extended Simple Net	false

Obrázek 4.2: Klasifikace modelu Večeřících filosofů

Petri net incidence and marking

Forwards incidence matrix I^+

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9
P0	0	0	0	0	0	1	0	0	0	1
P1	0	0	0	0	0	1	1	0	0	0
P2	0	0	0	0	0	0	1	1	0	0
P3	0	0	0	0	0	0	0	1	1	0
P4	0	0	0	0	0	0	0	0	1	1
P5	1	0	0	0	0	0	0	0	0	0
P6	0	1	0	0	0	0	0	0	0	0
P7	0	0	1	0	0	0	0	0	0	0
P8	0	0	0	1	0	0	0	0	0	0
P9	0	0	0	0	1	0	0	0	0	0

Backwards incidence matrix I^-

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9
P0	1	0	0	0	1	0	0	0	0	0
P1	1	1	0	0	0	0	0	0	0	0
P2	0	1	1	0	0	0	0	0	0	0
P3	0	0	1	1	0	0	0	0	0	0
P4	0	0	0	1	1	0	0	0	0	0
P5	0	0	0	0	0	1	0	0	0	0
P6	0	0	0	0	0	0	1	0	0	0
P7	0	0	0	0	0	0	0	1	0	0
P8	0	0	0	0	0	0	0	0	1	0
P9	0	0	0	0	0	0	0	0	0	1

Combined incidence matrix I

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9
P0	-1	0	0	0	-1	1	0	0	0	1
P1	-1	-1	0	0	0	1	1	0	0	0
P2	0	-1	-1	0	0	0	1	1	0	0
P3	0	0	-1	-1	0	0	0	1	1	0
P4	0	0	0	-1	-1	0	0	0	1	1
P5	1	0	0	0	0	-1	0	0	0	0
P6	0	1	0	0	0	0	-1	0	0	0
P7	0	0	1	0	0	0	0	-1	0	0
P8	0	0	0	1	0	0	0	0	-1	0
P9	0	0	0	0	1	0	0	0	0	-1

Inhibition matrix H

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9
P0	0	0	0	0	0	0	0	0	0	0
P1	0	0	0	0	0	0	0	0	0	0
P2	0	0	0	0	0	0	0	0	0	0
P3	0	0	0	0	0	0	0	0	0	0
P4	0	0	0	0	0	0	0	0	0	0
P5	0	0	0	0	0	0	0	0	0	0
P6	0	0	0	0	0	0	0	0	0	0
P7	0	0	0	0	0	0	0	0	0	0
P8	0	0	0	0	0	0	0	0	0	0
P9	0	0	0	0	0	0	0	0	0	0

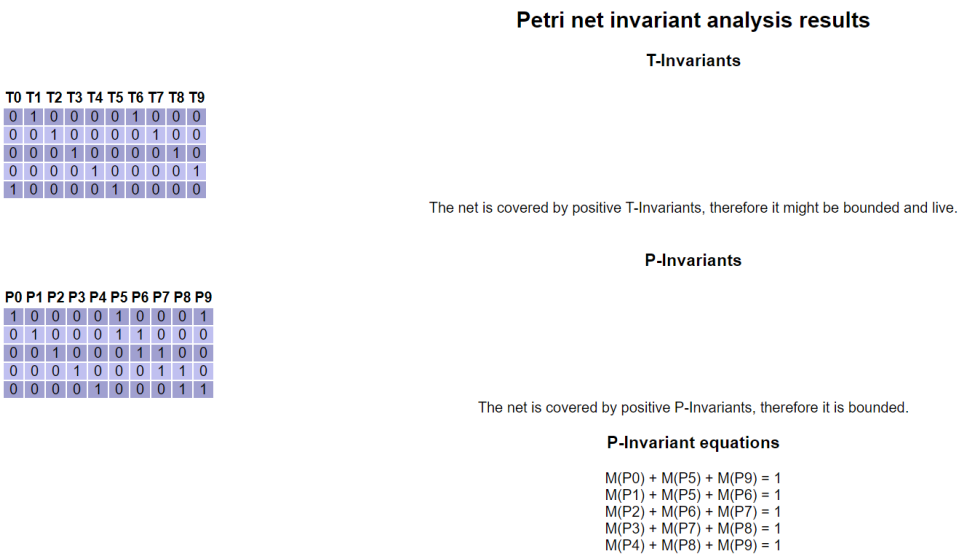
Marking

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9
Initial	1	1	1	1	1	0	0	0	0	0
Current	1	1	1	1	1	0	0	0	0	0

Enabled transitions

T0	T1	T2	T3	T4	T5	T6	T7	T8	T9
yes	yes	yes	yes	yes	no	no	no	no	no

Obrázek 4.3: Incidence modelu Večeřících filosofů



Petri net invariant analysis results

T-Invariants

T0	T1	T2	T3	T4	T5	T6	T7	T8	T9
0	1	0	0	0	0	1	0	0	0
0	0	1	0	0	0	0	1	0	0
0	0	0	1	0	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1
1	0	0	0	0	1	0	0	0	0

The net is covered by positive T-Invariants, therefore it might be bounded and live.

P-Invariants

P0	P1	P2	P3	P4	P5	P6	P7	P8	P9
1	0	0	0	0	1	0	0	0	1
0	1	0	0	0	1	1	0	0	0
0	0	1	0	0	0	1	1	0	0
0	0	0	1	0	0	0	1	1	0
0	0	0	0	1	0	0	0	1	1

The net is covered by positive P-Invariants, therefore it is bounded.

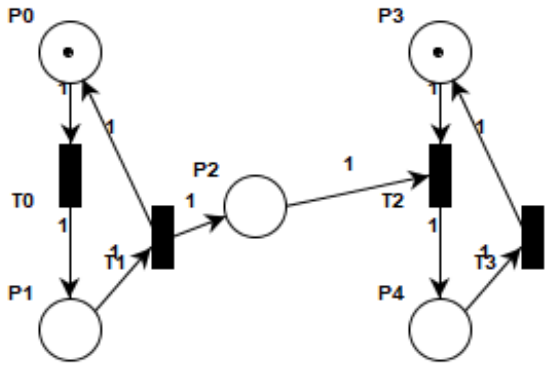
P-Invariant equations

M(P0) + M(P5) + M(P9) = 1
M(P1) + M(P5) + M(P6) = 1
M(P2) + M(P6) + M(P7) = 1
M(P3) + M(P7) + M(P8) = 1
M(P4) + M(P8) + M(P9) = 1

Obrázek 4.4: Invarianty modelu Večeřících filosofů

4.1.2 Producent-konzument

Tento model simuluje systém, kde jeden prvek vytváří výpočetní instance a druhý je zpracovává. V případě, že druhý prvek nestíhá instance zpracovat, ukládají se do bufferu úloh. Model takového systému jako Petriho síť vypadá takto:

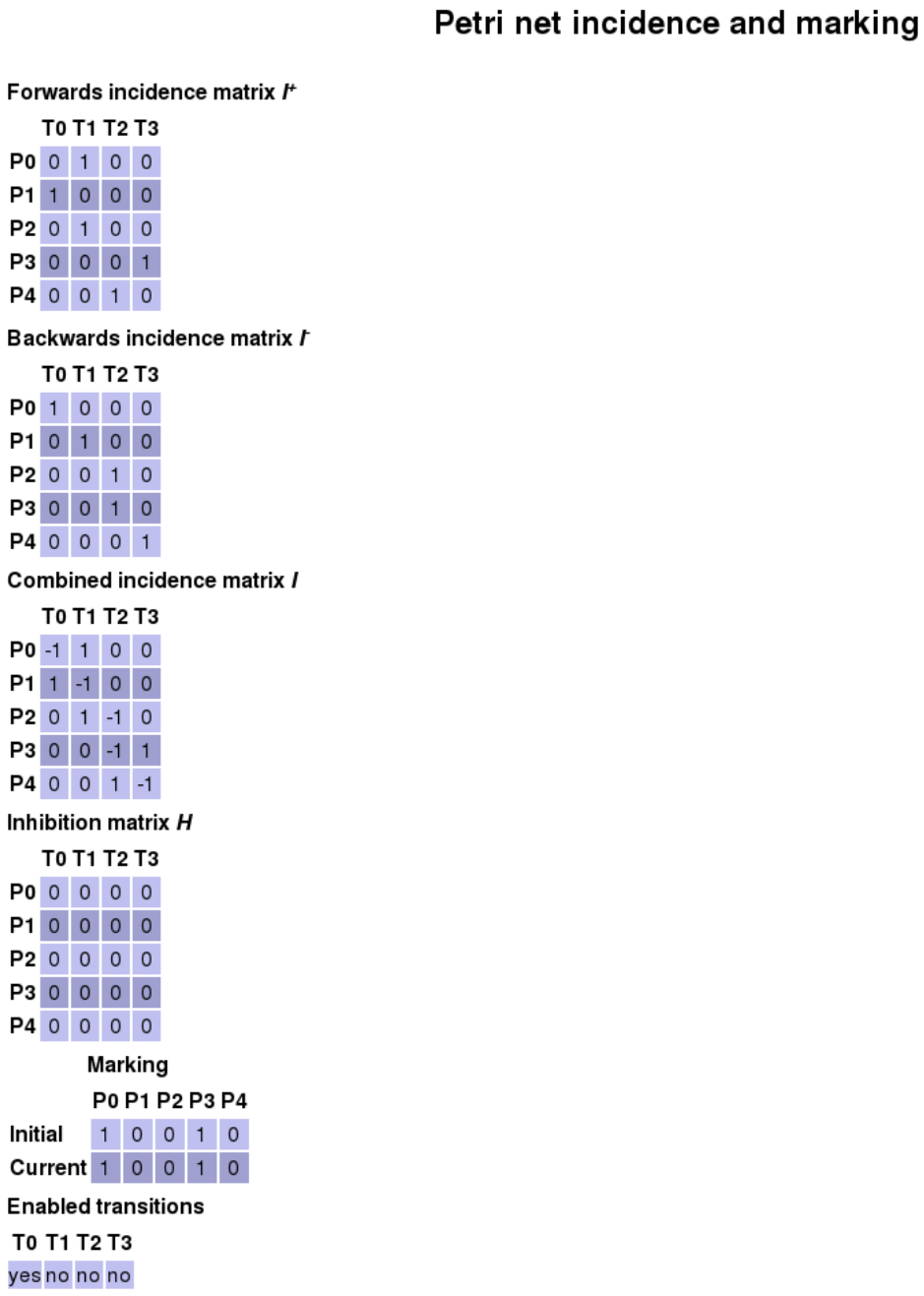


Obrázek 4.5: Petriho síť pro model Producent-konzument

Výstup z analýzy nástroje PIPE vypadá takto:

State Machine	false
Marked Graph	true
Free Choice Net	true
Extended Free Choice Net	true
Simple Net	true
Extended Simple Net	true

Obrázek 4.6: Klasifikace modelu Producent-konzument



Petri net incidence and marking

Forwards incidence matrix I^+

	T0	T1	T2	T3
P0	0	1	0	0
P1	1	0	0	0
P2	0	1	0	0
P3	0	0	0	1
P4	0	0	1	0

Backwards incidence matrix I^-

	T0	T1	T2	T3
P0	1	0	0	0
P1	0	1	0	0
P2	0	0	1	0
P3	0	0	1	0
P4	0	0	0	1

Combined incidence matrix I

	T0	T1	T2	T3
P0	-1	1	0	0
P1	1	-1	0	0
P2	0	1	-1	0
P3	0	0	-1	1
P4	0	0	1	-1

Inhibition matrix H

	T0	T1	T2	T3
P0	0	0	0	0
P1	0	0	0	0
P2	0	0	0	0
P3	0	0	0	0
P4	0	0	0	0

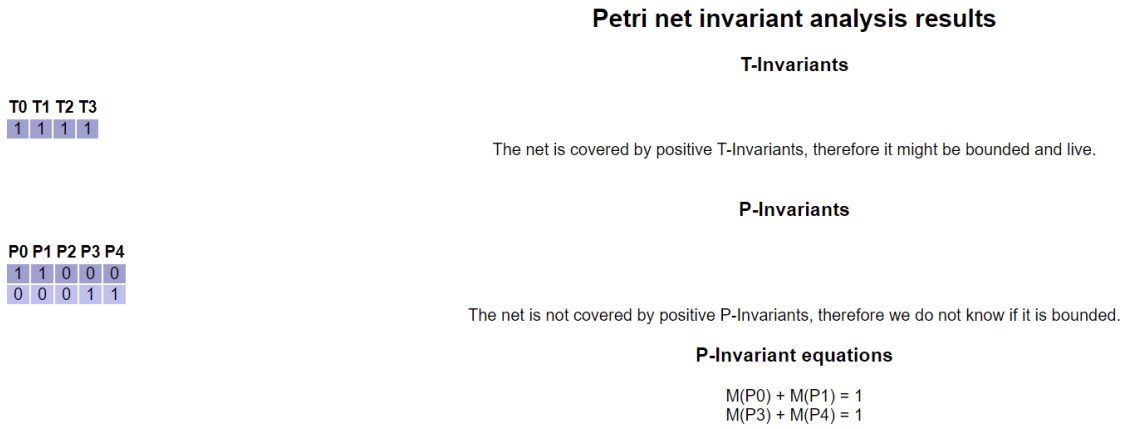
Marking

	P0	P1	P2	P3	P4
Initial	1	0	0	1	0
Current	1	0	0	1	0

Enabled transitions

T0	T1	T2	T3
yes	no	no	no

Obrázek 4.7: Incidence modelu Producent-konzument



Obrázek 4.8: Invarianty modelu Producent-konzument

4.2 Struktura nástroje PNML2VHDL

Základem nástroje jsou 3 struktury. Důvodem, proč byly využity struktury a ne objekty je to, že nejsou zapotřebí metody. V průchodu XML stromem totiž pouze doplňuji data na příslušné místo. První strukturou je **Place**, která obsahuje následující prvky:

- **ID** - řetězec s identifikátorem místa
- **initval** - inicializační hodnota místa
- **inedges** - pole řetězců s identifikátory vstupních hran
- **inedgessize** - velikost pole inedges
- **capacity** - číslo udávající kapacitu (v případě 0 se uvažuje neomezený počet)
- **inedgescount** - udává počet vstupních hran místa

Druhou strukturou je **Transition**, která obsahuje následující prvky:

- **ID** - řetězec s identifikátorem přechodu
- **inedges** - pole řetězců s identifikátory vstupních hran
- **outedges** - pole řetězců s identifikátory výstupních hran
- **inedgescount** - udává počet vstupních hran přechodu

- **outedgescount** - udává počet výstupních hran přechodu
- **inedgessize** - velikost pole inedges
- **outedgessize** - velikost pole outedges

Třetí strukturou je struktura **Petri_net**, která obsahuje následující prvky:

- **places** - pole míst
- **transitions** - pole přechodů
- **places_size** - velikost pole míst
- **trans_size** - velikost pole přechodů
- **placescount** - počet míst v poli
- **transcount** - počet přechodů v poli

Pro práci s XML souborem software využívá funkcí z knihoven *libxml/parser.h* a *libxml/tree.h*. Dále program obsahuje celkem 8 klíčových funkcí, z toho 7 je ručně implementovaných.

4.2.1 validateXML

validateXML je funkce z nástroje LibXML2, jejímž úkolem je kontrola vstupního souboru. Veškeré nedostatky, které nalezne, vypisuje na standardní výstup. Obsahuje jediný parametr, kterým je název kontrolovaného souboru.

4.2.2 initPN

Úkolem funkce InitPN je inicializace celkové struktury pro ukládání dat o celé Petriho síti. Alokuje místo v paměti pro všechny řetězce, jejichž velikost je omezena na 32 znaků. Dále pak nastavuje výchozí hodnoty pro velikosti polí míst, přechodů a hran. Výchozí hodnota je nastavena na 200 pro všechny velikosti, nicméně při překročení se pole dynamicky zvětší, takže výsledná velikost modelu je omezena pouze kapacitou hardwaru, na kterém nástroj běží. Jediným parametrem funkce je reference na strukturu Petriho sítě.

4.2.3 getNet

Tato funkce má za úkol provést inicializační rutiny pro parsování a průchod XML stromem pomocí funkcí *xmlReadFile*, která načte XML dokument a *xmlDocGetRootElement*, která nastaví ukazatel na kořen XML stromu. Dále zde pak probíhá volání funkce pro průchod celým stromem.

4.2.4 parseTree

Úkolem parseTree je kompletní průchod strukturou XML stromu. U každého uzlu nejprve zjišťuje typ XML prvku. Jedná-li se o tag (XML_ELEMENT_NODE), hledáme pak důležité typy známé ze struktury PNML jako jsou *place*, *transition*, *arc*, *initialMarking*, *capacity* a *value*. V případě, že je nalezen některý z hledaných uzlů, za pomocí dalších funkcí z knihovny LibXML se uloží požadovaná data do struktury Petri_net. LibXML umožňuje pohyb ve stromové struktuře pomocí struktury *xmlNode* a jejích prvků *children* (přesune ukazatel na potomka uzlu) a *next* (přesune ukazatel na další uzel ve stejné hloubce).

4.2.5 generateVHDL

generateVHDL je funkce, která na základě parametru struktury Petri_net začne sestavovat kompletní VHDL kód voláním 3 dalších funkcí.

4.2.6 generatePlaces

generatePlaces je funkce, která na základě pevné VHDL kostry (bude upřesněno dále) doplňuje do šablony informace ze struktury Petri_net. Výsledný soubor má název **PLACES.VHD**.

4.2.7 generateTransitions

generateTransitions je funkce, která na základě pevné VHDL kostry (bude upřesněno dále) doplňuje do šablony informace ze struktury Petri_net. Výsledný soubor má název **TRANSITIONS.VHD**.

4.2.8 generateTop

generateTop je funkce, která má za úkol vygenerovat Top entitu (detailní postup upřesněn dále). Výsledný soubor nese název **PETRI_NET.VHD**.

4.3 Převod míst

Při sběru dat pro implementaci míst hledá funkce *parseTree* několik klíčových uzlů pomocí atributu datového typu *xmlNode* s názvem *name*. Prvním kontrolovaným názvem uzlu je *place*. Tag *place* v sobě obsahuje atribut *id*, který získáme voláním funkce *xmlGetProp*. Dalším kontrolovaným je *arc*. Zde je zapotřebí s pomocí funkce *xmlGetProp* získat atribut *target*. Následně funkce zjišťuje, jestli atribut odpovídá některému z již nalezených míst, a pokud ano, uloží do něj vstupní hranu. Posledním důležitým kontrolovaným tagem je *value*. Z tohoto uzlu můžeme získat buď velikost kapacity nebo inicializačního počtu tokenů. Díky znalosti stromu dokážeme o příslušnosti hodnoty rozhodnout na základě předka uzlu. K tomu slouží v datovém typu *xmlNode* atribut

s názvem *parent*. Funkce tedy zjistí, jestli předkem není jeden z tagů buď *initialMarking* nebo *capacity*. Pokud ano, hledaná hodnota se nachází mezi tagy *<value>* a *</value>*. Pro získání hodnoty slouží funkce z knihovny LibXML2 s názvem *xmlNodeGetContent*.

Po sběru dat probíhá generování VHDL kódu. Na samém začátku je třeba vložit knihovny. Kromě standardní knihovny *IEEE.STD_LOGIC_1164* zde ještě využívám knihovny *IEEE.STD_LOGIC_UNSIGNED* pro zjednodušení aritmetických operací (inkrementace) a *IEEE.STD_LOGIC_MISC* pro logické redukce. Každá entita pak nese název **Place** společně s pořadovým číslem. Obsahuje celkem 8 portů:

- **increment** - při generování tohoto portu pro signalizaci inkrementace se program na základě počtu vstupních hran rozhoduje, zda-li se bude jednat o typ *std_logic_vector* nebo pouze *std_logic*
- **decrement** - vstup typu *std_logic* pro signalizaci dekrementace
- **rst,init** - vstupy typu *std_logic* pro uvedení místa do výchozího stavu
- **clk** - vstup hodinového signálu typu *std_logic*
- **tkn_out** - výstup typu *std_logic_vector* o velikosti 9 bitů pro signalizaci počtu tokenů v místě
- **cap_reached** - výstup typu *std_logic* pro signalizaci překročení kapacity místa
- **infinity** - výstup typu *std_logic* pro signalizaci nekonečného počtu tokenů

Architektura má pak za cíl implementovat chování popsané v návrhu místa. Při generování kódu si nástroj dává pozor na počet vstupních hran ve 2 místech. Pokud totiž místo obsahuje pouze jednu, není zapotřebí nad vstupem pro inkrementaci provádět žádnou kontrolní logiku, a lze jej zahrnout rovnou do logické funkce pro výstup. V opačném případě je nad vstupními signály třeba provést redukci typu OR, a až výsledný signál přivést do logické funkce pro výstup. Možný kód pro vygenerování místa může vypadat takto.

```
architecture Place0_body of Place0 is
signal tokens    : std_logic_vector(8 downto 0);
signal inf_flag : std_logic;
signal capacity : std_logic;
signal is_zero  : std_logic;
signal is_inc : std_logic;

begin

is_inc <= or_reduce(increment);

is_zero <= not(or_reduce(tokens));

cap_reached <= '1' when tokens = "000000001" else '0';

capacity <= '1' when tokens = "000000001" else '0';

tkn_out <= tokens;

infinity <= inf_flag;

PLACE: process(clk)
begin

 if(rising_edge(clk)) then
     if(init = '1' or rst = '1') then
         tokens  <= "000000001";
         inf_flag <= '0';
     else
         if(is_inc = '1' and inf_flag = '0' and capacity = '0') then
                 if(tokens="011111111") then
                     inf_flag <= '1';
                     tokens <= (others => '0');
                 else
                     tokens <= tokens + 1;
                 end if;
         else
             if(decrement = '1' and inf_flag = '0' and is_zero = '0') then
                 tokens <= tokens - 1;
             end if;
         end if;
     end if;
 end if;
end process PLACE;
end architecture Place0_body;
```

Obrázek 4.9: Možný výstup popisu architektury místa

4.4 Převod přechodů

Při sběru dat pro implementaci přechodů hledá funkce *parseTree* několik klíčových uzlů pomocí atributu datového typu *xmlNode* s názvem *name*. Prvním kontrolovaným názvem uzlu je *transition*. Tag *transition* v sobě obsahuje atribut *id*, který získáme voláním funkce *xmlGetProp*. Dalším kontrolovaným je *arc*. Zde je ale zapotřebí získat jak zdroje, tak cíle hran. Ty se nacházejí v atributech *source* a *target* a získáme je funkcí *xmlGetProp*. Funkce následně prohledává existující přechody ve struktuře Petri_net a doplňuje do nich vstupní a výstupní hrany.

Po získání všech dat a vygenerování míst nejprve funkce ověřuje, zda-li má generovaný přechod definovaný nějaký výstup. Pokud by neměl, přechod není platný. Pokud platný je, začíná generování VHDL kódu opět vložením knihoven. Zde krom standardní knihovny *IEEE.STD_LOGIC_1164* využívám ještě *IEEE.STD_LOGIC_MISC* pro funkce pro logickou redukci. Poté je zapotřebí rozlišovat přechody podle počtu vstupních a výstupních hran. Každá entita nese název **Transition** a pořadové číslo. V případě, že přechod obsahuje nějakou vstupní hranu bude entita obsahovat tyto porty.

- **tokens** - vstupní indikace tokenů typu *std_logic_vector* o velikosti 9*(počet vstupních hran)
- **inf_flags** - vstupní indikace neomezeného počtu tokenů typu buď *std_logic* pro jednu vstupní hranu nebo *std_logic_vector* pro více hran
- **en** - vstupní indikace pro odpálení přechodu typu *std_logic*
- **capacity** - vstupní indikace překročení kapacit buď typu *std_logic* pro jednu vstupní hranu nebo *std_logic_vector* pro více hran
- **increment** - výstupí indikace pro součet typu *std_logic*
- **ready** - výstupí indikace pro aktivnost přechodu typu *std_logic*

V případě, že přechod neobsahuje vstupní hrany, entita nebude obsahovat porty **tokens** a **inf_flags**. Architektura má pak za cíl implementovat chování popsané v návrhu přechodu. V případě, že počet vstupních hran je nulový, přechod bude vždy aktivní a bude čekat pouze na signál en. V případě, že vstupní hrana bude jedna, provede se redukce typu OR všech bitů signálu **tokens** společně se signálem **inf_flags**. V případě, že hran bude více, provede se redukce typu OR bitů vstupu **tokens**, které přísluší jedné hraně společně se vstupem **inf_flags**, který přísluší stejné hraně. Výstup těchto redukcí projde další redukcí, tentokrát typu AND. V případě, že výstupních hran je více, je zapotřebí provést redukci typu OR i na vstupní kapacity. Toto zpracování

vstupů se poté použije do výstupní logiky přechodu. Možný vygenerovaný kód přechodu může vypadat takto.

```
architecture Transition0_body of Transition0 is
signal has_token   : std_logic;
signal is_inf      : std_logic;
signal tokenplace  : std_logic_vector(1 downto 0);
signal cap_reached : std_logic;

begin

REDUCE_TOK: process(tokens, inf_flags)
begin
     for I in 0 to 1 loop
         tokenplace(I) <= or_reduce(tokens(I*9+8 downto I*9)) or inf_flags(I);
     end loop;

end process REDUCE_TOK;

has_token <= and_reduce(tokenplace);

cap_reached <= capacity;

ready <= has_token and not(cap_reached);

increment <= has_token and not(cap_reached) and en;

end architecture Transition0_body;
```

Obrázek 4.10: Možný výstup popisu architektury přechodu

4.5 Vytvoření top entity

Pro vytvoření top entity již není zapotřebí dalších dat z XML souboru. Veškeré informace v sobě nese struktura Petri_net. Top entita má 8 portů, které slouží buď pro vstup řídících signálů nebo výstup dat a kontrolních signálů. Těmi porty jsou:

- **decrement** - vstup signalizující dekrementaci zvoleného místa typu *std_logic_vector* o velikosti počtu míst
- **rst** - vstup pro uvedení Petriho sítě do výchozího stavu typu *std_logic*
- **clk** - vstup pro hodinový signál typu *std_logic*

- **init** - vstupní signál pro uvedení Petriho sítě do výchozího stavu typu *std_logic*
- **enable** - vstupní signál pro odpálení vybraného přechodu typu *std_logic_vector* o velikosti počtu přechodů
- **tokens** - výstupní signál ukazující stav všech tokenů typu *std_logic_vector* o velikosti 9*(počet míst)
- **capacities** - výstupní signál ukazující dosažení kapacit jednotlivých míst typu *std_logic_vector* o velikosti počtu míst
- **inf_flags** - výstupní signál ukazující dosažení nekonečného počtu tokenů jednotlivých míst typu *std_logic_vector* o velikosti počtu míst
- **fireable** - výstupní signál ukazující aktivnost přechodů typu *std_logic_vector* o velikosti počtu přechodů

Architektura si pak klade za cíl instanciaci komponent. Komponentami jsou zde všechna vygenerovaná místa a přechody. Tvorba spojů se pak řídí tím pravidlem, že propojovací signál se pojmenovává vždy podle výstupního portu. Vstupní porty komponent se poté na základě dat ze struktury Petri_net připojí k příslušnému signálu. Propojení vypadá například takto:

```
PLACE0_con: Place0 port map(
     increment(0) => increments(5),
     increment(1) => increments(9),
     decrement => decrement(0),
     rst => rst,
     init => init,
     clk => clk,
     tkn_out => tokens_out(8 downto 0),
     cap_reached => capacities_out(0),
     infinity => inf_flags_out(0)
);
```

Obrázek 4.11: Příklad výstupu připojení komponenty do výsledného obvodu

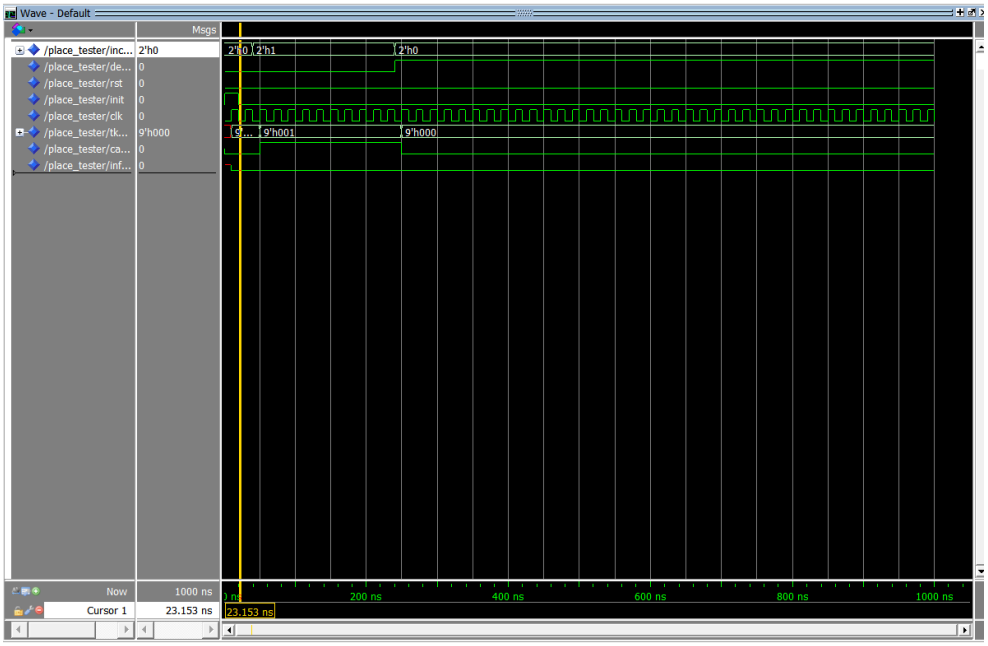
4.6 Testování

Pro otestování správnosti výstupu nástroje je zapotřebí všechny vygenerované komponenty vyzkoušet v simulátoru s pomocí testbenche. Pro svou

práci využívám simulátor Modelsim 10.4a PE Student Edition se studentskou licencí.

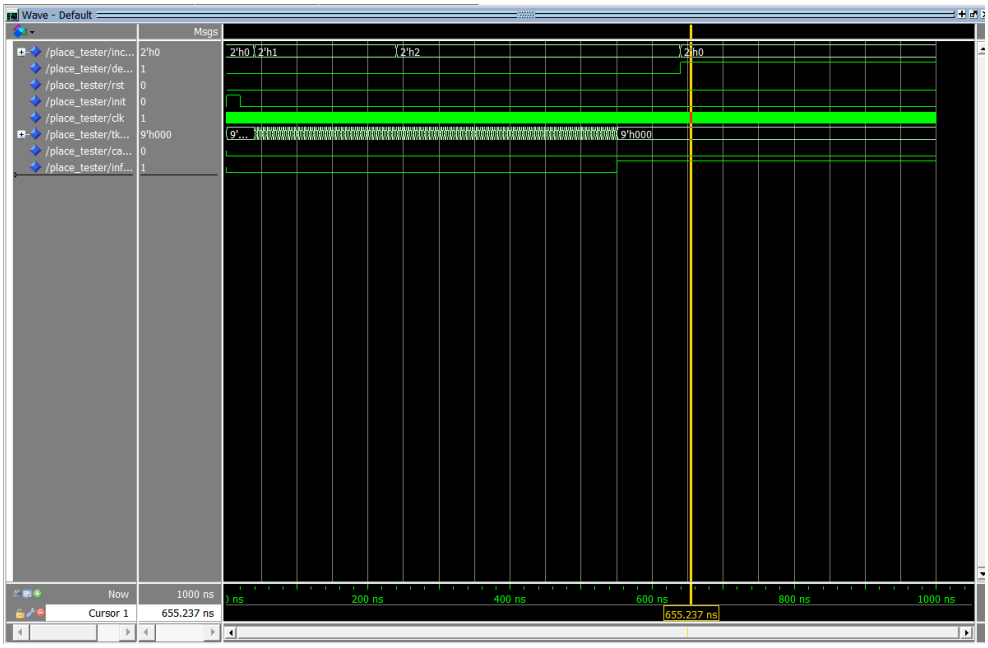
4.6.1 Test místa

U místa je zapotřebí otestovat, jestli indikace kapacity, nekonečného počtu tokenů a výstup signalizující počet tokenů zobrazují správnou hodnotu. K tomu jsou zapotřebí 2 testy, jelikož v případě omezené kapacity při správně implementaci nesmí dojít k dosažení flagu pro nekonečno. První test tedy kontroluje správnost reakce na překračování kapacity a druhý test sleduje správnost výstupů tokenů neustálým přičítáním hodnoty až do dosažení limitu. Při testu kapacity bude limit nastaven na hodnotu jednoho tokenu. Výsledky testů jsou zde:



Obrázek 4.12: Výsledek 1. testu místa

Na obrázku lze vidět, že v momentě, kdy signál cap_reached dosáhl hodnoty logické 1, tokeny i přes aktivní signál inicializace nezvyšují svou hodnotu. Dále pak při aktivním signálu decrement se hodnota dostane zpět na nulu a indikátor kapacity spadne do logické 0, což je správné chování.

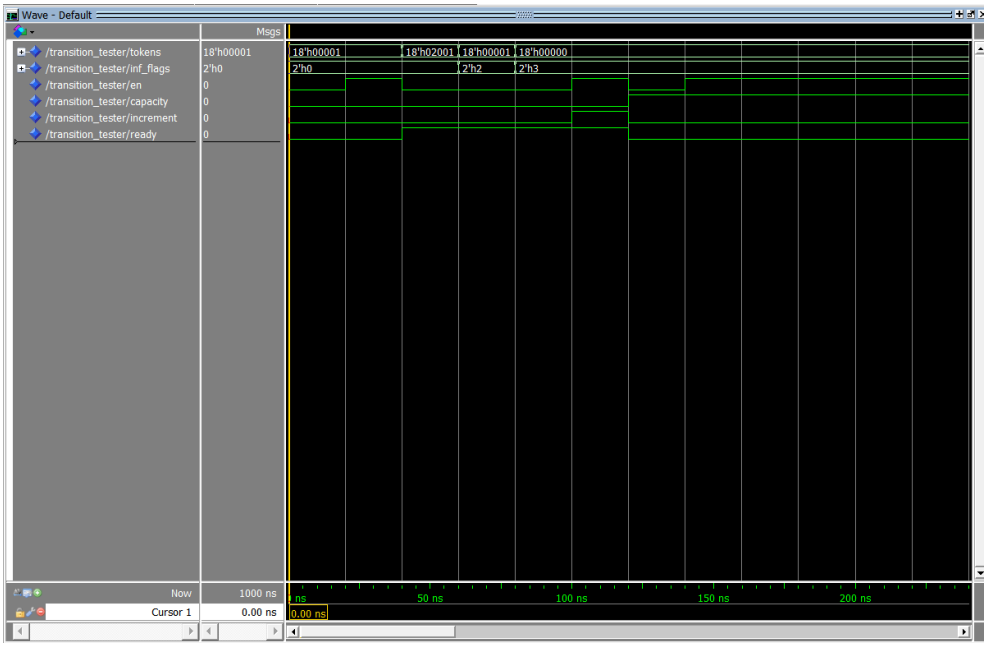


Obrázek 4.13: Výsledek 2. testu místa

Druhý test ukazuje, že místo správně reaguje na více typů aktivních signálů pro inkrementaci. Dále jakmile počet tokenů přesáhne limit, aktivuje se signál infinity. Při aktivním signálu infinity již místo nereaguje na aktivitu signálu pro dekrement, což je správné chování, a tudíž místo funguje podle požadavků.

4.6.2 Test přechodu

U přechodů je zapotřebí vytvořit testbench, který bude kontrolovat správnou indikaci aktivnosti a výstupu pro inkrementaci na základě různých vstupů. Vzhledem ke známosti vnitřní logiky, která není složitá, není třeba generovat všechny možné vstupy metodou hrubé síly, ale pouze vyzkoušet okrajové stavy.

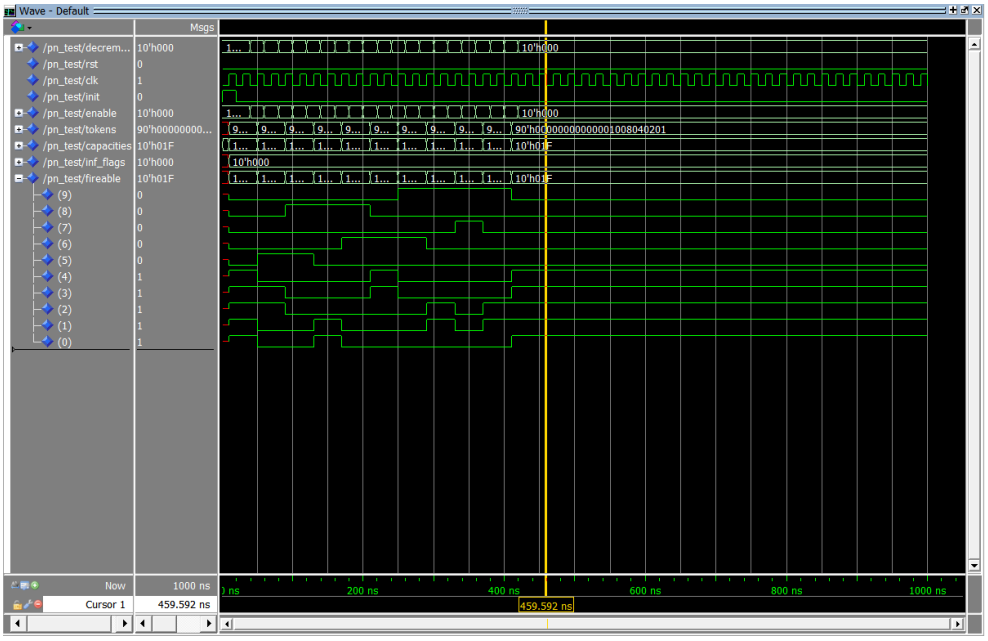


Obrázek 4.14: Výsledek testu přechodu

Testovací scénář probíhá tak, že nejprve nastaví nějakou hodnotu tokenu na jeden ze vstupů. Poté se zaktivuje signál povolení odpalu, u kterého je očekáváno, že neprovede žádnou změnu. Dále je signál pro aktivaci schozen a přidán token na druhý vstup. Zde se očekává aktivace signálu ready. Poté se se token opět na druhém vstupu vymění za signál inf_flag, kde se očekává, že aktivita se nezmění. Následuje výměna Tokenu za inf_flag na prvním vstupu a opět se nepředpokládá změna na výstupu. Poté se aktivuje znovu signál pro odpálení, kde se očekává aktivní signál increment. V poslední části se deaktivuje signál en a přivede logická 1 na vstup capacity. V tomto okamžiku se očekává změna signálu ready na hodnotu logická 0. Dle obrázku si lze všimnout, že test dopadl přesně podle očekávání.

4.6.3 Test modelu Večeřící filosofové

V tomto momentě je známo, že místa a přechody plní svou funkci správně. Je tedy třeba otestovat správné chování modelu. Zde bude využito toho, že tuto síť lze převést na konečný automat. Cílem tedy bude otestovat správné provedení přechodů za předpokladu správného „tahání za dráty“. Stav Petriho sítě lze sledovat díky signálu pro aktivnosti přechodů.

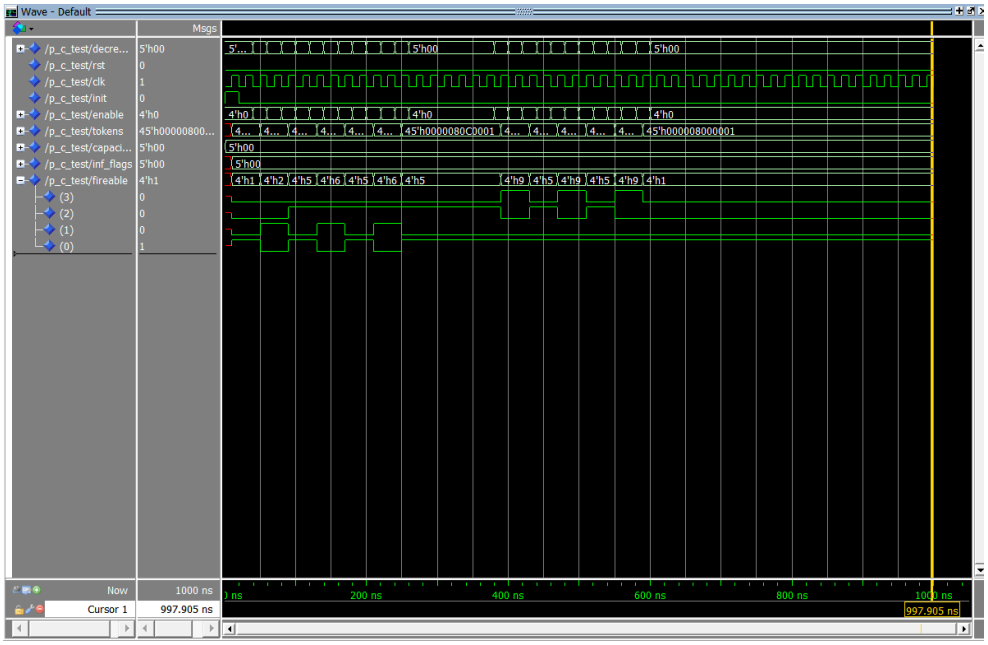


Obrázek 4.15: Výsledek testu modelu Večeřících filosofů

Test ukazuje průběh náhodně zvolenou nejkratší možnou cestou grafem, která odpálí všechny možné přechody. Průběh ukazuje správné změny stavů, a lze tedy říct, že model při správném řízení dokáže plnit svou funkci.

4.6.4 Test modelu Producent-konzument

Tento model bohužel na konečný automat převést nelze. Test tedy probíhá klasickou metodou pozorování, zda-li producent opravdu správně produkuje tokeny a konzument správně odebírá.



Obrázek 4.16: Výsledek testu modelu Producent-konzument

Na obrázku je tento test bohužel špatně pozorovatelný. Producentem byly nagenerovány 3 tokeny, které byly pro alespoň malou přehlednost pozdrženy, poté zpracovány konzumentem. I přes horší přehlednost obrázku test dopadl v pořádku a model producent-konzument reaguje podle očekávání.

Závěr

Tato práce se zabývala metodikou převodu modelů systémů reprezentovaných jako Petriho sítě a následně převedených do popisového jazyka PNML, do jazyka pro popis hardwaru s názvem VHDL. Nejprve se věnovala teoretickému základu Petriho sítí. Dále pak hledala nástroje, kterými lze generovat kýžený výstup, a popsala formát tohoto výstupu. Následně prošla předchozí výzkum stejného tématu. V návrhové části pak popsala různé možné formy implementace kompletního modelu jako číslicového obvodu a také nástroje, který dokáže na základě předchozích znalostí provést převod z formátu PNML do formátu VHDL. Tento převod byl po krocích popsán a otestován na 2 modelech systému - Večeřící filosofové a Producent-konzument.

Výsledkem práce je nástroj pojmenovaný PNML2VHDL. Na základě testů dokáže implementovat syntetizovatelný a správný VHDL kód. Nástroj je funkční, ale je limitovaný pouze na typ P/T Petriho sítí bez násobných hran. V dalším kroku lze tedy provést rozšíření o násobné hrany (tak, jak je popsáno v návrhu), a dále pak přidat inhibiční a testovací hrany, a následně priority přechodů. Další možností výzkumu v tomto tématickém poli by byla verifikace převodu nástroje pomocí formálního důkazu ekvivalence popisů v obou jazycích.

Literatura

[1] Fibiger, J.: *Implementace modelu křižovatky v hradlovém poli*. Diplomová práce, České Vysoké Učení Technické v Praze, jan 2009.

[2] Češka, M.: Úvod do Petriho sítí. Dostupné z: `http://www.fit.vutbr.cz/study/courses/TIN/public/Prednasky/tin-pr13-ps.pdf`

[3] Ratschan, S.: Petriho sítě. oct 2017. Dostupné z: `https://edux.fit.cvut.cz/courses/MI-TES.16/_media/lectures/lecture_10_petri_nets.pdf`

[4] Janoušek, V.: *Modelování objektů Petriho sítěmi*. Dizertační práce, Vysoké Učení Technické v Brně, dec 1998.

[5] Aalst, W.: Interval Timed Coloured Petri Nets and their Analysis. *Application and Theory of Petri Nets 1993*, ročník 691, 1993: s. 453–472.

[6] Dorda, M.: Úvod do Petriho sítí. Dostupné z: `http://homel.vsb.cz/~dor028/Nekonvencni_metody_1.pdf`

[7] Anonym: Objektová Petriho síť. Dostupné z: `https://akela.mendelu.cz/~petrj/opnml/opn.html`

[8] Bonet, P.; Llado, C. M.; Puijaner, R.; aj.: *PIPE v2.5: A Petri Net Tool for Performance Modelling (PDF format). Proc. 23rd Latin American Conference on Informatics (CLEI 2007)*. Costa Rica: San Jose, 2007.

[9] JOSEFÍK, M.: *NÁSTROJ PRO PRÁCI S OBJEKTOVĚ ORIENTOVANÝMI PETRIHO SÍTĚMI*. Diplomová práce, Vysoké Učení Technické v Brně, may 2016.

Příloha A

Seznam použitých zkratek

PNML Petri net markup language

VHDL VHSIC hardware description language

VHSIC Very high speed integrated circuit

FPGA Field programmable gate array

PN Petri net

OOPN Object oriented Petri net

XML Extensible Markup Language

PNTD Petri net type definition

UML Unified Modeling Language

TRNG True Random Number Generator

LFSR Linear Feedback Shift Register

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `exe` ..... adresář se spustitelnou formou implementace
  - `PNML2VHDL.exe`
  - `philosophers.xml` ..... Petriho síť modelu Večeřících filosofů
  - `producer-consumer.xml` ... Petriho síť modelu Producent-konzument
- `src`
  - `impl` ..... zdrojové kódy implementace
    - `main.cpp`
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
    - `thesis.tex`
    - `bibliography.bib`
    - `img` ..... adresář s přiloženými obrázky
      - `img.zip`
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `thesis.ps` ..... text práce ve formátu PS