VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

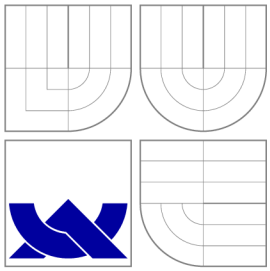
VÝUKOVÝ NÁSTROJ PRO BARVENÉ PETRIHO SÍTĚ

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

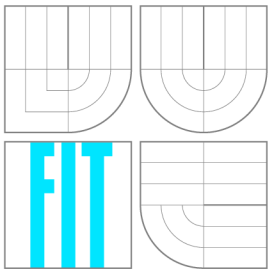
AUTOR PRÁCE
AUTHOR

Bc. ONDŘEJ NAVRÁTIL

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

VÝUKOVÝ NÁSTROJ PRO BARVENÉ PETRIHO SÍTĚ

EDUCATIONAL TOOL FOR COLOURED PETRI NETS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. ONDŘEJ NAVRÁTIL
AUTHOR

VEDOUCÍ PRÁCE — Mgr. ADAM ROGALEWICZ, Ph.D.
SUPERVISOR

BRNO 2013

Abstrakt

Barvené Petriho sítě (CPN) jsou rozšířením klasických place-transition Petriho sítí (P/T PN). Všechny tokeny a místa zde mají svůj typ (příp. i hodnotu), do sítě lze dále vkládat různé typy inskripcí. CPN vynikají svou snadnou čitelností a skvělou vyjadřovací schopností. Zároveň disponují pevnými formálními podklady, jenž usnadňují jejich počítačovou simulaci a v omezené míře umožňují verifikovat některé vlastnosti. Motivací tohoto projektu je fakt, že v současnosti je veřejně dostupný pouze jediný nástroj pro úpravu a simulaci CPN, a sice `CPNTools` vyvíjený na univerzitě v Aarhusu. Program je však komplikovaný a pro nezainteresovaného uživatele obtížně uchopitelný. Cílem projektu je prozkoumat možnosti a vlastnosti CPN a nástroje `CPNTools` a na základě získaných znalostí navrhnout a implementovat didaktickou aplikaci se svižným a intuitivním rozhraním, která umožní uživatelům bez hlubších teoretických vědomostí získat přehled o problematice CPN.

Abstract

Coloured Petri nets (CPN) are an extension of a standard place-transition Petri nets (P/T PN). Every token and place have its type (and eventually a value) and various inscriptions can be inserted into the net. CPN excel with great readibility and expresivity. At the same time, they carry a well-defined formal basis, which eases its computer simulation and allows limited verification of certain attributes to be performed. Motivation for doing this project is the simple fact that currently only one public software tool is available for CPN creation and simulation – `CPNTools` developed on the Aarhus university. The program, however, is quite complicated and hard to handle for an unexperienced user. The goal is to research capabilities and properties of both CPNs and `CPNTools` and on this basis design and implement a didactic application with swift and intuitive interface that helps users without deeper theoretical insight to get a grasp of the problematics.

Klíčová slova

Petriho sítě, barvené Petriho sítě, CPNTools, Qt, Flex, Bison

Keywords

Petri nets, coloured Petri nets, CPNTools, Qt, Flex, Bison

Citace

Ondřej Navrátil: Výukový nástroj pro barvené Petriho sítě, diplomová práce, Brno, FIT VUT v Brně, 2013

Výukový nástroj pro barvené Petriho sítě

Prohlášení

Prohlašuji, že jsem tento semestrální projekt vypracoval samostatně pod vedením Mgr. Adama Rogalewicze, Ph.D. Další informace mi poskytl Prof. RNDr. Milan Češka, CSc. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Ondřej Navrátil
22. května 2013

Obsah

Kapitola 1

Úvod

Petriho sítě poprvé definoval Carl Adam Petri roku 1962 ve své disertační práci „Kommunikation mit Automaten“ [15]. Koncept navržený v tomto dokumentu si ihned získal popularitu a představuje významný mezník v návrzích diskrétních paralelních a distribuovaných systémů. Petriho sítě nabízejí též řadu modifikací a rozšíření umožňující optimalizaci vyjadřovacích prostředků různým případům použití.

Tento projekt byl realizován za účelem vývoje didaktické aplikace pracující s barvenými Petriho sítěmi, které jsou významným a z popisného hlediska velmi silným derivátem Petriho sítí. Narozdíl od konkurenčních nástrojů specializovaných pro design komplikovaných průmyslových modelů si výsledná aplikace klade za cíl poskytnout co nejjednodušší a nejintuitivnější rozhraní pro demonstraci klíčových výrazových prostředků CPN. Výsledný program by měl být využit při výuce v kurzu Petriho sítě na FIT VUT.

Následující text je členěn do kapitol ilustrujících přechod od deklarací formálních vlastností sítí přes výzkum možností existujících simulačních nástrojů až k návrhu a implementaci cílového softwaru. Součástí práce je i CD s výslednou aplikací a příklady vymodelovaných sítí.

Kapitola 2 se zabývá Petriho sítěmi, jejich vznikem, vývojem, modifikacemi a použitím. V kapitole 3 jsou představeny barvené Petriho sítě, jejichž vlastnosti jsou hlavním tématem projektu. Kapitola 4 popisuje nástroj `CPNTools` jakožto vzorovou aplikaci pro práci s barvenými Petriho sítěmi. V kapitole 5 je představen návrh aplikace, která by měla zastoupit `CPNTools` pro výukové účely a odstranit některé neduhy spojené s jeho používáním. Kapitola 6 popisuje implementaci a stěžejní koncepty použité při vývoji aplikace `CPNSimulator`.

Kapitola 2

Petriho síť

Tato kapitola si klade za cíl vymezit výchozí pojmy spojené s Petriho sítěmi a jejich aplikacemi. Následující text v žádném případě není a ani se nesnaží být jakkoli vyčerpávajícím co do vlastností a definic popisovaných pojmů. Snahou je spíše podtrhnout základní koncepty, které budou posléze konfrontovány se svými alternativami v oblasti barvených Petriho sítí. Shrnutí vychází z [22, 23], kde lze též dohledat podrobnější infromace.

2.1 Síť

Petriho sítě vynikají zejména svou formální bází. Pro výstavbu matematického modelu je třeba nejprve definovat síť samotnou:

Definice 2.1 (Síť)**.** Síť je trojice $N = (P, T, F)$, kde:

P, T jsou disjunktní množiny

$F \subset (P \times T) \cup (T \times P)$ je binární relace

P zveme množinou míst, T množinou přechodů přechody a F je tzv. toková funkce (flow function)

Grafem sítě nazýváme grafovou reprezentaci této trojice – jedná se o orientovaný bipartitní graf, v němž uzly reprezentují místa (places) a přechody (transitions) a orientované hrany definované tokovou funkcí spojují vždy místo s přechodem či naopak. Jinými slovy se jedná o graf $G = (P \cup T, F)$. Zde je konvencí zobrazovat místa kružnicemi a přechody čtverci či svislými úsečkami.

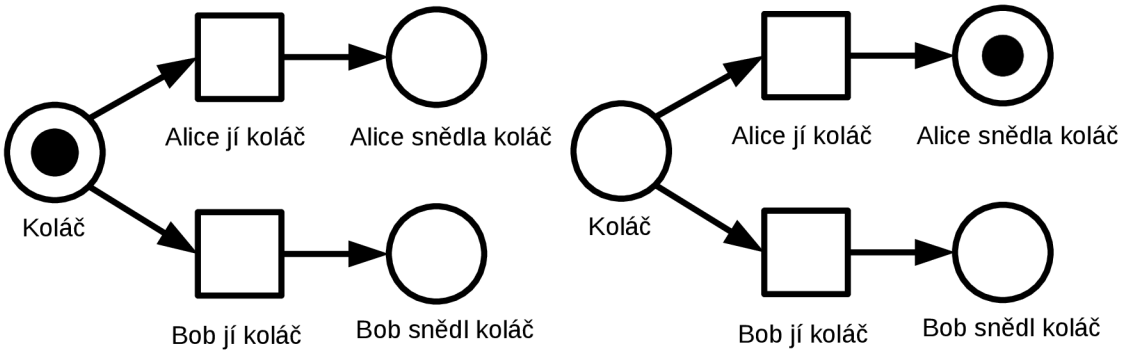
Při zkoumání sítě (zejména při výpočtu v síti) budeme často pracovat s prvky, které spolu sousedí z hlediska tokové relace. Definujeme proto:

Definice 2.2 (Preset, postset)**.** $\forall x \in P \cup T$ nazveme:

$\bullet x = \{y | yFx\}$ jako preset x

$x\bullet = \{y | xFy\}$ jako postset x

Lze se setkat též s českými ekvivalenty vstupní množina a výstupní množina.



Obrázek 2.1: Příklad kroku v C/E síti

2.2 C/E sítě

Asi nejjednodušší variantou Petriho sítí jsou C/E sítě (conditions/events, sítě s podmínkami a událostmi). Místa v těchto sítích reprezentují podmínky, přechody symbolizují události. U C/E sítí většinou užíváme namísto $N = (P, T, F)$ z definice 2.1 značení $N = (B, E, F)$ (z německého „die Bedingung“ – podmínka a „das Ereignis“ – událost).

V síti dále definujeme:

Definice 2.3 (C/E síť)**.** V C/E síti $N = (B, E, F)$ nazveme:

Libovolnou podmnožinu $c \subseteq B$ jako případ (case)

Pro případ c nazveme událost e c-proveditelnou (c-enabled), pokud $\bullet e \subseteq c \wedge e\bullet \cap c = \varnothing$

Je-li událost e c-proveditelná, případ $c' = (c \setminus \bullet e) \cup e\bullet$ nazýváme následníkem případu c při výskytu události e

V grafu značíme jednotlivé případy tečkou v odpovídajících místech. Intuitivně vidíme, že výpočet probíhá pomocí výskytů událostí, které modifikují daný případ.

Jednoduchý příklad C/E sítě vidíme na obrázku 2.2. Zobrazuje změnu případu při provedení události Alice jí koláč. Jsou zde patrné též dva základní jevy, se kterými se v Petriho sítích setkáváme. V prvním případě jsou zároveň proveditelné obě události, tedy Alice/Bob jí koláč. Zároveň však po výskytu jedné z událostí přestane být ta druhá proveditelná. Říkáme, že jsou tyto události **v konfliktu**. U C/E sítí lze tyto situace snadno detekovat – rozhodující je informace, jestli události mají společné prvky ve výstupních či vstupích množinách. Naopak jsou-li množiny disjunktní, tedy formálně $\bullet e \cap \bullet e' = e\bullet \cap e'\bullet = \varnothing$, nazýváme e, e' **nezávislé**. Tuto vlastnost lze zobecnit na libovolně mohutnou množinu událostí.

Při provádění událostí z nezávislé množiny nezáleží na pořadí jejich výskytu – výsledný případ bude vždy totožný. Díky tomu lze definovat **krok** v C/E síti obecně jako výskyt jakékoli množiny nezávislých, pro daný případ proveditelných událostí. Kromě zvýšení pohodlí v praktickém použití přináší tato definice výhody též z hlediska analýzy verifikace, kdy sjednocením akcí do jedné množiny uspoříme značnou část stavového prostoru oproti situaci, kdy bychom simulovali všechny permutace.

2.3 P/T sítě

P/T sítě (z anglického „Places and Transitions nets“ – sítě s místy a přechody) rozšiřují základní myšlenku C/E sítí tím, že do míst umožňují ukládat více tokenů. U míst dále povoluje uvádět jejich maximální kapacitu a u hran množsví přenesených tokenů. Namísto podmínek a událostí mluvíme v tomto případě o místech a přechodech.

K formální definici si nejdříve zavedeme rozšíření množiny přirozených čísel o prvek ω, který bude reprezentovat neomezenou kapacitu místa. Zavedeme si pro něj relaci $<$ a operace $+, -$:

Definice 2.4 (Supremum přirozených čísel)**.** Nad $\mathbb{N} \cup \{\omega\}$ definujeme:

$$\forall n \in \mathbb{N} : n < \omega$$

$$\forall m \in \mathbb{N} \cup \{\omega\} : m + \omega = \omega + m = \omega$$

$$\forall A \subseteq \mathbb{N} :$$
$$sup(A) = \begin{cases} a & a \in A \ \wedge \forall a' \in A - \{a\} : a' < a \\ \omega & \forall a \in A : \exists a' \in A : a < a' \end{cases}$$

P/T sítě nyní definujeme rozšířením sítě o sémantické informace:

Definice 2.5 (P/T síť)**.** P/T síť je šestice $N = (P, T, F, W, K, M_0)$, kde:

$N = (P, T, F)$ je konečná síť

$W : F \to \mathbb{N}^+$ je funkce určující **váhy** hran

$K : P \to \mathbb{N} \cup \{\omega\}$ je funkce určující **kapacitu** míst

M_0 je **počáteční značení** sítě (viz dále)

Obdobou případu u C/E sítí je **značení**, které už bylo použito v definici 2.5. Značení musí respektovat kapacity sítě, jak vyjadřuje následující definice:

Definice 2.6 (Značení P/T sítě)**.** Značením P/T sítě nazveme zobrazení $M : P \to \mathbb{N} \cup \{\omega\}$, pokud:

- $\forall p \in P : M(p) \leq K(p)$

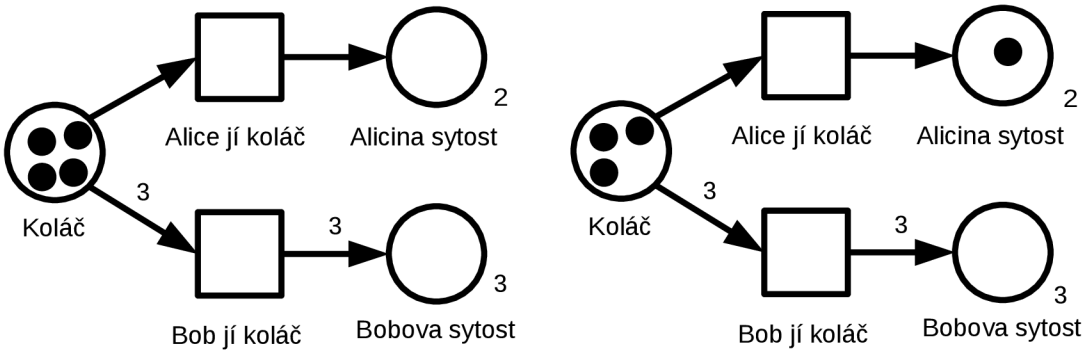
Při rozhodování o proveditelnosti přechodů nyní zkoumáme též kapacity míst a aktuální značení dle definice 2.7.

Definice 2.7 (Proveditelnost přechodu v P/T síti)**.** V síti $N = (P, T, F, W, K, M_0)$ a značení M označíme přechod $t \in T$ za M-proveditelný právě tehdy, když:

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

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

Zbývá ještě upravit sémantiku provádění přechodů, které je definováno takto:



Obrázek 2.2: Příklad provedení přechodu v P/T síti

Definice 2.8 (Provedení přechodu v P/T síti)**.** Buď M značení, $t \in T$ M-proveditelný přechod. Následné značení k M při provedení t definujeme jako M', pro které:

$$\forall p \in P : M'(p) = \begin{cases} M(p) - W(p,t) & p \in {\bullet t} \setminus t{\bullet} \\ M(p) + W(t,p) & p \in t{\bullet} \setminus {\bullet t} \\ M(p) - W(p,t) + W(t,p) & p \in {\bullet t} \cap t{\bullet} \\ M(p) & \text{jinak} \end{cases}$$

P/T sítě lze definovat i jinými způsoby, použitím komplementace či multimnožin. V následujícím textu však budeme uvažovat výše uvedený koncept. Konkrétní algoritmy pro převod a důkazy ekvivalence lze nalézt například v [23].

Příklad P/T sítě včetně přechodu je na obrázku 2.3. U míst bez explicitně uvedené kapacity uvažujeme pro tuto veličinu hodnotu ω, obdobně implicitní hodnotou pro váhu hran je 1. Opět si lze všimnout, že oba dva přechody jsou konkurentně proveditelné. Zároveň však v daném značení provedení jednoho nevylučuje proveditelnost druhého a to i přesto, že jejich vstupní a výstupní množiny nejsou disjunktní. Pokud však uvažujeme situaci, kdy máme k dispozici pouze 3 koláče, provedení obou přechodů již nebude dále možné. Narozdíl od C/E sítí nám tudíž k detekci konfliktů nepostačí pouze popis sítě, ale je třeba uvažovat i konkrétní značení, které stav modelovaného systému popisuje.

2.4 Další koncepty

V sekcích 2.2 a 2.3 byly představeny dva základní koncepty Petriho sítí. V praxi se však uplatnily i nejrůznější modifikace. Barveným Petriho sítím se detailně věnuje kapilota 3, tato sekce stručně představuje některé významné modely odvozené od P/T sítí. Další varianty Petriho sítí popisuje např. [22].

Zajímavou aplikací je využití P/T sítí pro generování jazyků, podobně jako se tomu běžně děje pomocí automatů či gramatik. Princip spočívá v přiřazení symbolů dané abecedy k přechodům v P/T síti. Vygenerovaný řetězec je potom určen symboly u přechodů v pořadí jejich provádění v dané výpočetní posloupnosti. Jednotlivé podtypy takovýchto sítí se dále liší omezeními, která kladou na funkci přiřazující symboly k přechodům či množinou koncových značení. Nejobecnější jazyky Petriho sítí, tzv. jazyky typu **L**, dosahují

z hlediska Chomského hierarchie vyjadřovací schopnosti mezi regulárními a kontextovými jazyky, s bezkontextovými jsou z hlediska inkluze neporovnatelné (důkazy viz [23]).

Za zmínku stojí též Petriho sítě rozšířené o tzv. inhibiční hrany.

Definice 2.9 (Síť s inhibitory)**.** Síť $N = (P, T, F, W, K, M_0, F')$ je P/T síť s inhibitory, pokud:

$N = (P, T, F, W, K, M_0)$ je P/T síť

$F' \neq \varnothing$

$F' \subseteq F \cap P \times T$

$\forall f \in F' : W(f) = 1$

F' nazýváme množinou inhibičních hran či množinou inhibitorů.

Inhibitory, jak již jejich název napovídá, mohou bránit proveditelnosti přechodů. Musíme tudíž této skutečnosti uzpůsobit odpovídající definici:

Definice 2.10 (Proveditelnost v síti s inhibitory)**.** V síti $N = (P, T, F, W, K, M_0, F')$ je $t \in T$ M-proveditelný, pokud:

$\forall p \in t\bullet : M(p) \leq K(p) - W(t, p)$ (shodně s 2.7)

$$\forall p \in \bullet t : \begin{cases} M(p) \geq W(p, t) & \langle p, t \rangle \in F \setminus F' \\ M(p) = 0 & \langle p, t \rangle \in F' \end{cases}$$

Významnou vlastností Petriho sítí s inhibitory je ekvivalence jejich výpočetní síly s výpočetní silou Turingových strojů (důkaz skrze převod registrového stroje viz [23]).

Kapitola 3

Barvené Petriho sítě

Barvené Petriho sítě (angl. „Coloured Petri nets“, CPN) jsou významným derivátem Petriho sítí. Tato kapitola deklaruje jejich základní atributy a principy a konfrontuje je s koncepty uvedenými v sekcích 2.2 a 2.3. Zdrojem informací byly knihy [3] a [4].

O vznik CPN se zasadil Kurt Jensen v Aarhuské univerzitě v Dánsku roku 1981. Lze je zařadit do kategorie vysokoúrovňových Petriho sítí, pro kterou je charakteristická kombinace sítě s programovacími jazyky, které v tomto kontextu zveme též inskripčními jazyky. Sítě jsou typicky využívány při návrzích a verifikaci komunikačních protokolů, datových sítí, distribuovaných algoritmů a vestavěných systémů. Stěžejní motivací pro vznik a použití CPN je stejně jako u dalších druhů PN flexibilita míry abstrakce a pevná teoretická báze umožňující automatizovanou simulaci a v omezené míře také verifikaci vlastností modelu.

3.1 Barvy a barevné množiny

V CPN nově zavádíme u všech tokenů tzv. barvy (angl. „colours“). Barva může být libovolná hodnota, kterou si s sebou token nese a která může dále ovlivňovat výpočet sítě. Sémantickým významem může být například obsah zprávy putující po síti či pořadové číslo paketu.

Dále zavádíme barevné množiny (colour sets). Rozumíme jimi prakticky jakoukoli neprázdnou množinu barev, konečnou či nekonečnou. Každé místo má přesně definovanou barevnou množinu, která omezuje barvy tokenů v něm obsažených. Sémantický význam barevných množin je tedy podobný významu datových typů u programovacích jazyků. Proto se v praxi nejčastěji setkáváme s datovými typy jako např. `integer`, `real`, `enum`, případně s kompozitními typy v podobě struktur a polí. Barevné množiny obsahující současně barvy napříč různými datovými typy lze modelovat pomocí unií.

3.2 Multimnožiny, značení

Vzhledem k faktu, že tokeny v CPN si nejsou "rovny"jako u P/T sítí, nestačí nám již k popisu míst celá čísla. Potřebujeme znát též barvy jednotlivých tokenů. Zároveň však nelze použít ani klasických množin, neboť v jednom místě se může nacházet zároveň několik tokenů se stejnou barvou a přítomnost dvou či více stejných prvků by porušovala matematickou definici množin.

Obsah míst je proto modelován pomocí multimnožin, které vícenásobný výskyt prvků umožňují. Úplná formální definice multimnožiny včetně potřebných operací je uvedena dále

v této kapitole, intuitivně si ji však lze představit jako množinu dvojic $(barva, pocet)$, kde barva je unikátní v rámci multimnožiny a je obsažena v barevné množině místa.

Značením CPN tedy rozumíme zobrazení přiřazující všem místům (i prázdnou) multimnožinu tokenů odpovídající barevné množiny. Značení CPN podobně jako u jiných zmíněných Petriho sítí reprezentuje stav modelu v nějakém okamžiku výpočtu. Zvláštním případem je opět počáteční značení definující stav modelu před zahájením výpočtu.

Vzhledem k potřebě zapisovat různá značení a tím i multimnožiny je třeba v automatizovaných simulačních nástrojích rozšířit programovací jazyk o patřičné syntaktické elementy.

3.3 Inskripce

CPN dále povolují vkládat k místům, přechodům a hranám sítě několik druhů inskripcí. Tyto prvky dále omezují proveditelnost přechodů, výsledné značení po jejich provedení atp. Dva typy inskripcí jsme si již představili – barevné množiny omezující barvy tokenů u míst a multimnožiny reprezentující jejich počáteční značení.

Další možnou inskripcí jsou hranové výrazy (angl. „arc expressions“). Jak název napovídá, jedná se o výrazy v programovacím jazyce, které se vážou k hranám v síti. Jejich vyhodnocením je multimnožina barev omezených barevnou množinou incidentního místa, která definuje, jaké tokeny se při provedení přechodu z místa odstraní či naopak v místě přibudou. Výrazy mohou být konstantní, mohou však obsahovat i proměnné nabývající hodnotu až v okamžiku vyhodnocování v závislosti na značení sítě. Proměnné jsou chápány jako globální a jejich konkrétní hodnota je tudíž konzistentní napříč inskripcemi v rámci jednoho provedení přechodu. Přiřazování konkrétních hodnot proměnným není zcela triviální záležitostí a bude dále diskutováno.

Přechody samotné dále mohou disponovat tzv. strážemi (angl. „guards“). Jedná se o výrazy, jejichž vyhodnocením je logická hodnota true/false a v případě negativního výsledku zabraňují proveditelnosti přechodu. Tyto výrazy mohou opět obsahovat proměnné se stejným významem jako u hranových výrazů.

Konečně, samotná síť obsahuje globální deklarace proměnných včetně jejich barevných množin či definice pomocných funkcí použitých v ostatních inskripcích.

3.4 Proveditelnost, výpočet v síti

V sekci 3.3 byl definován pojem značení pro CPN. Výpočet je zde definován stejně jako např. u P/T sítí jako posloupnost kroků, které obsahují provádění přechodů. Rozdíly však vznikají při rozhodování o proveditelnosti přechodů a také v určování následného značení po jejich provedení.

Potřebujeme nejprve vysvětlit pojem navázání proměnných (angl. „binding“). Navázání množiny proměnných V přiřazuje všem proměnným $v \in V$ určité barvy z jejich barevných množin. Vyskytuje-li se například ve výrazech pouze proměnné $V = \{m, n\}$ typu integer, navázáním můžeme nazvat například $b : m \to 3, n \to 4$. Stejně tak však můžeme m, n přiřadit jakékoli jiné celočíselné hodnoty. Podstatné však je, že je třeba pokrýt všechny použité proměnné, tzn. pouhé $b' : m \to 3$ není v tomto případě navázáním.

Přechod v CPN je proveditelný pro dané značení právě tehdy, když existuje takové navázání všech proměnných vyskytujících se v relevantních inskripcích, pro které:

- vyhodnocením stráže přechodu je hodnota true

- pro všechny hrany ze vstupní množiny platí, že multimnožina získaná vyhodnocením hranového výrazu je podmnožinou značení incidentního místa

Lze pozorovat značné odlišnosti oproti P/T sítím – určení proveditelnosti nyní zdaleka není triviální. Pokud bychom naivně chtěli prozkoumat všechna možná navázání abychom vyloučili proveditelnost, už při použití jedné neomezené proměnné (např. celočíselného typu) získáme nekonečné množství potenciálních navázání. Z tohoto důvodu bude nutno přistupovat k volbě navázání sofistikovaněji.

Značení m_2 následující po m_1 provedením přechodu t při navázání b získáme takto:

značení míst, která nejsou v $t \bullet \cup \bullet t$ je shodné pro m_1 i m_2

značení míst z $\bullet t$ je dáno rozdílem multimnožiny značící toto místo v m_1 a multimnožiny vzniklé vyhodnocením odpovídajícího hranového výrazu při navázání b

značení míst z $t\bullet$ je dáno sjednocením multimnožiny značící toto místo v m_1 a multimnožiny vzniklé vyhodnocením odpovídajícího hranového výrazu při navázání b

Předpokladem je samozřejmě proveditelnost t při m_1, b.

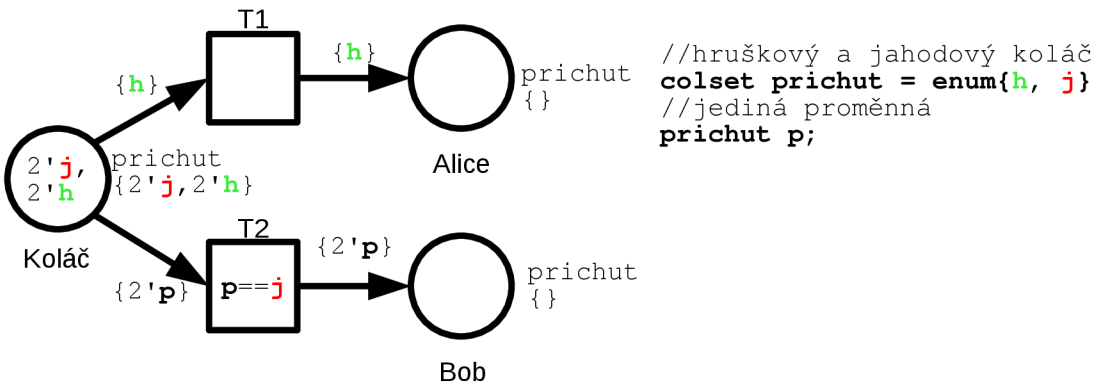
Podstatné je, že vyhodnocením jediného přechodu lze dosáhnout různých následných značení a to v závislosti na zvoleném navázání proměnných, významném zejména pro určení tokenů odebraných z míst ve vstupní množině přechodu a přidaných do míst v množině výstupní. Chceme-li tedy prozkoumat veškeré stavy, do kterých se modelovaný systém může dostat, potřebujeme určit všechna navázání, pro která jsou přechody proveditelné. Zde opět narážíme na neomezenost množiny všech možných navázání.

Zbývá ještě prověřit, jak se CPN chovají vzhledem ke konkurenci a konfliktům přechodů. Vycházejme ze situace, kdy dva přechody jsou v daném značení a navázání oba současně povoleny. Detekce konfliktu není tak jednoduchá jako u P/T sítí, nestačí nám ani prostá kontrola počtu tokenů v jednotlivých místech. Jak již bylo zmíněno, tokeny v CPN nejsou rovnocenné a zpravidla záleží na tom, které z nich konkrétně pro daný přechod vybereme. Mají-li být dva přechody konkurentně (tedy paralelně) provedeny, musí být multimnožiny jejich vstupních tokenů disjunktní. V opačném případě, tedy když oba přechody potřebují pro své provedení jeden či více stejných tokenů, dochází ke konfliktu.

3.5 Příklad barvené sítě

Obrázek 3.1 ilustruje možnosti barvených Petriho sítí na jednoduchém příkladě. Síť umožňuje v jakémkoli značení pouze dvě možnosti navázání, a sice $p \to j, p \to h$. Přechod T1 má všechny hranové výrazy konstantní a jeho proveditelnost tudíž závisí pouze na přítomnosti tokenu ve vstupním místě. Stráž přechodu T2 oproti tomu omezuje přípustné navázání pouze na $p \to j$. Oba přechody jsou ve znázorněném značení (které odpovídá počátečnímu značení) a navázání $p \to j$ konkurentně proveditelné. Změnou stráže přechodu T2 na $p = h$ by však mezi nimi vznikl konflikt.

I přes jednoduchost příkladu lze též pozorovat značný nárůst informační hodnoty oproti srovnatelně velké P/T síti v obrázku 2.3. CPN nám v tomto případě umožnila snadno definovat různé příchutě koláčů a preference strávníků. Zároveň si lze všimnout variability vyjadřování vlastností sítě. U obou přechodů je požadovaná příchuť omezena jiným způsobem. Toto je typickým znakem CPN, často lze využít možnosti přesouvat informace mezi inskripcemi v síti, deklarační částí a grafem samotným v závislosti na tom, které vlastnosti sítě chceme čtenáři zdůraznit.



Obrázek 3.1: Příklad barvené Petriho sítě

3.6 Formální definice

Účelem této sekce je korektně definovat koncepty intuitivně popsané v předchozích sekcích a poskytnout tím formální bázi k prakticky podaným principům. Definice vychází z [3].

Základním kamenem pro výstavbu teorie CPN je multimnožina.

Definice 3.1 (Multimnožina)**.** Multimnožina m nad neprázdnou množinou S je funkce $S \to \mathbb{N}$. Tato funkce mapuje každý prvek z S na počet výskytů $m(s)$ prvku s v multimnožině m. Multimnožinu lze zapsat též jako sumu $\sum_{s \in S} m(s)`s$.

Množinu všech multimnožin nad množinou S budeme dále značit S_{MS}. Dále definujeme operace:

Definice 3.2 (Operace nad multimnožinami)**.** Nechť $m_1, m_2 \in S_{MS}$ pro nějaké S. Definujeme:

$$m_1 + m_2 = \sum_{s \in S} (m_1(s) + m_2(s))`s$$

$$n * m_1 = \sum_{s \in S} (n * m_1(s))`s$$

$$|m_1| = \sum_{s \in S} m_1(s)$$

$$m_1 \leq m_2 = \forall s \ \in S : m_1(s) \leq m_2(s)$$

$$m_2 \leq m_1 \Rightarrow m_1 - m_2 = \sum_{s \in S} (m_1(s) - m_2(s))`s$$

V CPN se dále setkáváme s proměnnými, výrazy a typy. Zavedeme proto konvenci:

$\mathbb{B} = \{true, false\}$ pro označení boolovského typu

$Type(x)$ pro typ proměnné/výrazu x

$Var(expr)$ pro množinu proměnných vyskytujících se ve výrazu $expr$ (lze použít i pro množinu výrazů jako sjednocení množin proměnných vyskytujících se v jednotlivých prvcích)

$expr\,\langle b\rangle$ pro hodnotu získanou vyhodnocením $expr$ při navázání b všech proměnných z $Var(expr)$

Nyní již lze zavést pojem sítě samotné:

Definice 3.3 (Barvená Petriho síť)**.** Barvená Petriho síť je n-tice $CPN = (\Sigma, P, T, A, N, C, G, E, I)$, kde

Σ je konečná množina typů, tedy barevných množin

P je konečná množina míst

T je konečná množina přechodů

A je konečná množina hran, $P \cap A = P \cap T = T \cap A = \varnothing$

N je uzlová funkce, $N : A \to P \times T \cup T \times P$

C je funkce barev, $C : P \to \Sigma$

G je funkce stráží zobrazující množinu T do výrazů tak, že $\forall t \in T : Type(G(t)) = \mathbb{B} \wedge Type(Var(G(t)) \subseteq \Sigma)$

E je funkce hranových výrazů zobrazující množinu A do výrazů tak, že $\forall a \in A : Type(E(a)) = C(p(a))_{MS} \wedge Type(Var(E(a))) \subseteq \Sigma$, kde $p(a)$ je místo z dvojice $N(a)$

I je inicializační funkce zobrazující P do výrazů tak, že $\forall p \in P : Type(I(p)) = C(p)_{MS} \wedge Var(I(p)) = \varnothing$

V tomto značení lze vnímat P, T, A, N jako popis sítě a jejího grafu, zatímco C, G, E, I popisují různé inskripce.

Rozšiřme si nyní definiční obor funkce Var také o přechody $t \in T$:

$$\forall t \in T : Var(t) = \{v | v \in Var(G(t)) \vee \exists a \in A(t) : v \in Var(E(a))\}$$

Definujme nyní navázání přechodu, které je esenciální pro následné zkoumání jeho proveditelnosti:

Definice 3.4 (Navázání přechodu)**.** Mějme síť $CPN = (\Sigma, P, T, A, N, C, G, E, I)$. Navázáním přechodu $t \in T$ nazveme $b : Var(t) \to \Sigma$ takové, že:

$\forall v \in Var(t) : b(v) \in Type(v)$

$G(t)\,\langle b\rangle = true$

Notací $B(t)$ budeme označovat všechna navázání pro t.

Zavedeme ještě pojmy token – rozumíme jím pár $(p \in P, c \in C(p))$ – a prvek navázání – pár $(t \in T, b \in B(t))$. Množinu všech tokenů budeme značit TE, množinu všech prvků navázání BE. Dále definujme značení sítě reprezentující aktuální stav výpočtu:

Definice 3.5 (Značení v CPN)**.** Značení M sítě je libovolná multimnožina nad množinou všech tokenů, tedy $M \in TE_{MS}$

Počáteční značení získáme vyhodnocením inicializační funkce

$$\forall(p,c) \in TE : M_0(p,c) = (I(p))(c)$$

Konečně definujme krok v síti:

Definice 3.6 (Krok v CPN)**.** Krok Y v CPN je neprázdná, konečná multimnožina nad množinou všech prvků navázání, tedy:

$$Y \in BE_{MS}$$

$$\sum_{b \in BE} Y(b) > 0$$

Pro simulaci sítě potřebujeme ještě pojmy proveditelnosti a provedení kroku:

Definice 3.7 (Proveditelnost kroku v CPN)**.** Krok Y prohlásíme za proveditený ve značení M právě tehdy, když:

$$\forall p \in P : \sum_{(t,b) \in Y} E(p,t)\,\langle b \rangle \leq M(p)$$

Tato podmínka formálně vystihuje intuitivní požadavek na dostatečné množství tokenů ve vstupních místech.

Definice 3.8 (Provedení kroku v CPN)**.** Mějme krok Y proveditelný ve značení M_1, poté následné značení M_2 k M_1 při provedení Y definujeme vztahem:

$$\forall p \in P : M_2(p) = (M_1(p) - \sum_{(t,b) \in Y} E(p,t)\,\langle b \rangle) + \sum_{(t,b) \in Y} E(t,p)\,\langle b \rangle$$

Tímto se nám povedlo zcela formálně a kompletně popsat syntaxi a sémantiku CPN ve smyslu uvedených intuitivních základů.

3.7 Hierarchie v CPN

Zvláště u rozsáhlejších projektů je prakticky nemožné modelovat kompletní úlohu jedinou, plochou sítí. Pro barvené Petriho sítě se používají celkem čtyři techniky pro zlepšení čitelnosti, znovupoužitelnosti a škálovatelnosti abstrakce. Tato sekce popisuje pouze jejich základní myšlenky, neb tyto postupy nebudou pro cílovou aplikaci významné. Bližší informace lze vyhledat například v [3, 23].

Substituce míst a substituce přechodů jsou dva blízké koncepty založené na nahrazování těchto elementů dalšími podsítěmi. Celkový model tím neztratí na detailnosti definice, nezajímavé či opakující se části jsou však substituovány jednoduchými prvky kompaktně zapouzdřujícími komplexnější strukturu a chování nahrazované podsítě. Podsíť obsahuje tzv. porty, které definují podobu vstupních a výstupních množin substitujícího prvku, podobně jako například funkce v programovacích jazycích definují počet a datové typy svých parametrů a návratovou hodnotu, čímž determinují formu jejich použití.

Invokace přechodů je velmi podobná substituci přechodů s tím rozdílem, že při každém provedení substituujícího přechodu je vytvořena nová kopie substituované podsítě. Změny provedené tímto přechodem v podsíti se tedy nijak neprojeví na dalším provádění přechodu.

Poslední, spíše kosmetickou metodou je fúze míst. Zvláště u větších sítí dochází často k situacím, kdy některá z míst jsou incidentní s více přechody umístěnými v graficky vzdálených částech sítě. Hrany modelující tyto závislosti by mohly narušit čitelnost grafu. Z těchto

důvodů lze pomocí fůze míst vytvořit více grafických prvků reprezentujících takováto místa s tím, že sémantické akce budou prováděny stejně, jakoby se jednalo o jediné místo a zachovají konzistenci obsahu míst.

3.8 Víceúrovňové CPN

Pro úplnost zmiňme ještě model víceúrovňových sítí. V tomto modelu jsou samotné tokeny reprezentovány vlastními sítěmi (tzv. sítě v sítích). Pro danou úroveň rozlišujeme síť systému a sítě objektů (tokenů), zpravidla má síť právě dvě úrovně. Toto rozšíření přináší nové možnosti a přístupy (interakce mezi jednotlivými sítěmi, jejich synchronizace atp). Aplikaci tohoto konceptu představuje např. prostředí PNTalk[2].

Kapitola 4

CPNTools

`CPNTools` [16] je nástroj pro vytváření, úpravu, simulaci a analýzu barvených Petriho sítí. Aplikace je vyvíjena na Aarhuské univerzitě od r. 2000 skupinou CPN Group, mezi hlavní vývojáře patří Kurt Jensen a Soren Christensen, tedy lidé, kteří se podíleli na vzniku barvených Petriho sítí samotných. Původním cílem bylo nahradit tehdejší nevyhovující nástroj `Design/CPN`. Aktuální verze 3.4.0 z června 2012 je dostupná zdarma na webových stránkách projektu. Pro platformy Linux a Mac je k dispozici pouze starší verze 2.3.5 a zdá se, že nejsou nadále podporovány. Frekvence aktualizací a datum vydání současné verze však dokládají živost a perspektivu projektu.

4.1 Inskripční jazyk

`CPNTools` používá pro deklarace a inskripce derivát jazyka Standard ML [13] zvaný CPN ML. Standard ML je funcionální jazyk s mnoha praktickými aplikacemi a jak již jeho název napovídá, je považován za standardní popisný jazyk v řadě průmyslových oborů. Drobnou nevýhodou je zmíněné funkcionální paradigma, které přes svou nepopiratelnou sílu a principiální jednoduchost není příliš blízké majoritě imperativně zaměřených programátorů.

`CPNTools` používá implementaci kompilátoru SML s názvem SML/NJ. Rozšíření CPN ML se týká zejména syntaktických konstrukcí pro popis barevných množin a multimnožin. SML je použit též pro implementaci analytických a simulačních součástí apliakce. Jak uvádí [4], jedním z důvodů pro využití právě SML byla též korespondence pojmů jako výraz, vyhodnocení, typ, proměnná a především **navázání** v CPN se základními koncepty funkcionálních jazyků, ke kterým se Standard ML řadí.

CPN ML je silně staticky typovaný. Datové typy výrazů a funkcí nejsou deklarovány uživatelem, nýbrž odvozeny překladačem. Umožňuje též definovat polymorfní funkce přijímající více typů parametrů.

Barevné množiny lze definovat několika způsoby:

- pomocí základních typů (string, int, bool, unit): `colset NUMBER = int;`
- jako součin typů (n-tici): `colset PACKET = NUMBER * DATA;`
- jako záznam (strukturu): `colset PACKET2 = record n:NUMBER * d:DATA;`
- jako sjednocení typů (unii): `colset DATA = union i:int + s:string;`
- jako výčet: `colset STATUS = with ok | error | unknown;`

- jako seznam: `colset ATTRIBUTES = list string;`
- jako podmnožinu typu dle funkce: `colset ODD = subset int by isOdd;`

Kompilátor automaticky vytvoří strukturám a uniím konstruktory pro inicializaci jejich složek. Pro přístup ke složkám je realizován skrze funkci `#slozka struktura`, tedy např. `#n packet`. U n-tic je místo názvu složky použito její pořadové číslo. Pro seznamy umožňuje použít konstrukci pro rozdělení na první prvek a zbytek seznamu, používaný pro jejich rekurzivní procházení.

Proměnné jsou definovány včetně své barevné množiny, např
`var m, n: NUMBER;`
Definice funkcí je uvozena klíčovým slovem `fun` a její návratový typ a omezení parametrů jsou odvozeny automaticky ze zadaného výrazu.

U výrazů podporuje CPN ML infixovou notaci operátorů a kromě standardních výrazových prostředků nabízí také konstukce `if-else` a `case-of`. V parametrech a definicích funkcí a `case` výrazech lze využít porovnávání vzorů.

Pro popis multimnožin v počátečním značení jsou k dispozici dva operátory:

‘ : $int \times \langle colset \rangle \rightarrow \langle colset \rangle_{MS}$ vytvoří multimnožinu obsahující pouze zadaný počet zadaného prvku

$++ : \langle colset \rangle_{MS} \times \langle colset \rangle_{MS} \rightarrow \langle colset \rangle_{MS}$ pro sjednocení multimnožin

4.2 Ovládání

Uživatelské rozhraní aplikace `CPNTools` je poněkud nekonvenční, dalo by se říci až kontroverzní. Uživatelský manuál [5] doporučuje pro ovládání použít v kombinaci myší namísto standardní klávesnice další pointing-device, např. trackball a vyzdvihuje také možnost práce více uživatelů na jednom zařízení zároveň.

Samotná pracovní plocha je organiozována se snahou usnadnit práci s mnoha rozsáhlými sítěmi, což odpovídá použití v klasických průmyslových projektech. Obsahuje libovolné množství oken - binderů, které se dále dělí na karty s jednotlivými sítěmi, nástroji a deklaracemi (viz obr. 4.1 vlevo).

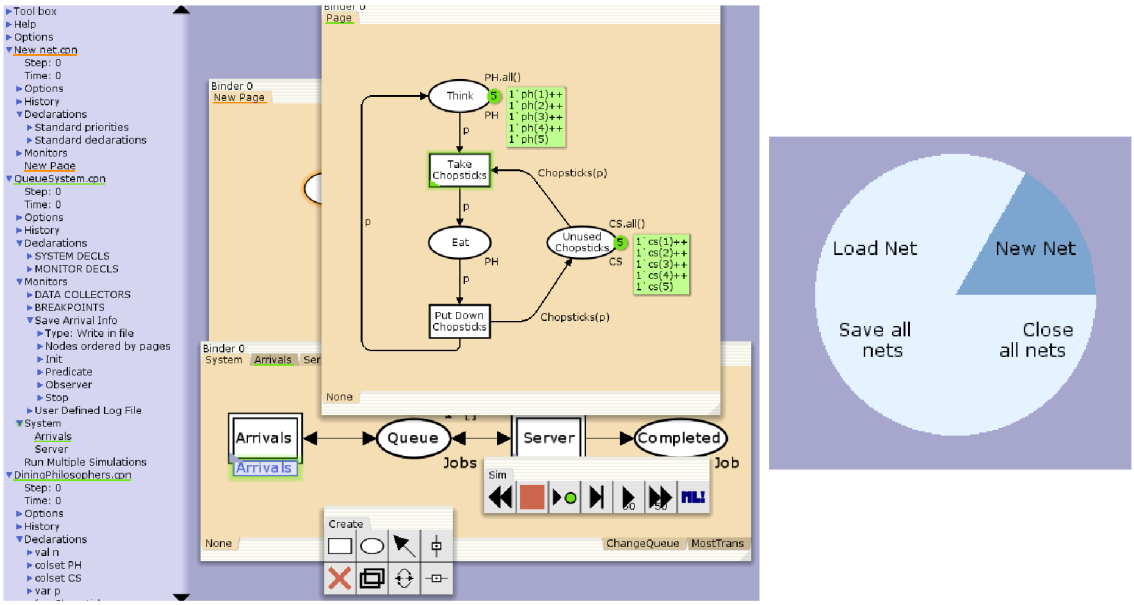
Akce jsou realizovány pomocí různých typů vstupů (typicky je možné provést je více způsoby) – přímou manipulací myší, obouruční manipulací s použitím dvou myší, pomocí klávesnice, klasických paletových nástrojových nabídek (které však nejsou v aplikaci nijak pevně uchyceny), tzv. indexu v levé části obrazovky či diskovitých kontextových nabídek (viz obr. 4.1 vpravo). Inskripce jsou upravovány přímo v síti/indexu s podporou zvýraznění syntaxe.

Celkově lze ovládání aplikace hodnotit jako matoucí pro nezainteresovanou osobu. Určitě se nejedná o aplikaci, kterou by dokázal nový uživatel ovládat bez přečtení manuálu. Příliš velké množství inovativních a nestandardních prvků brání v použití běžných pracovních postupů naučených z jiných aplikací.

4.3 Simulace

Pro účely simulace poskytuje `CPNTools` sadu nástrojů (viz obr. 4.1 vlevo dole):

- „rewind“: návrat modelu zpět do počátečního značení

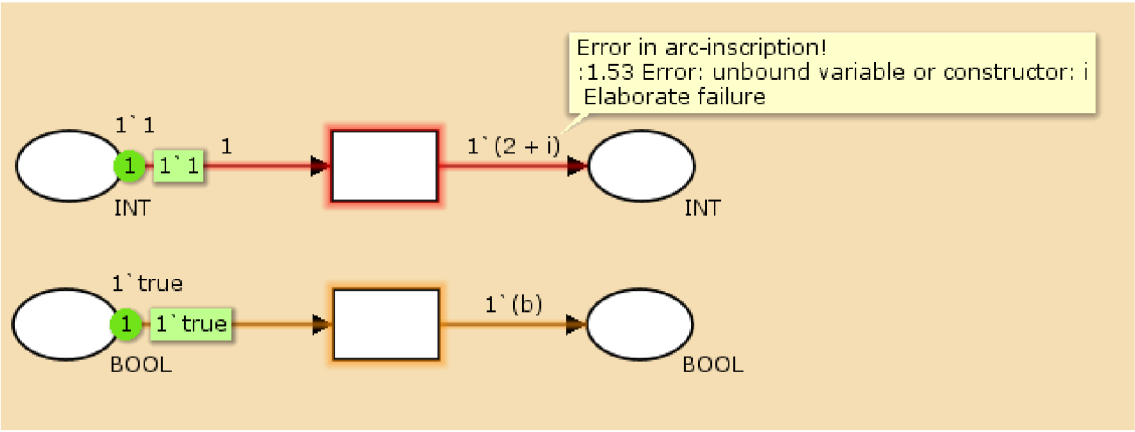


Obrázek 4.1: Nástroj `CPNTools`: index a pracovní plocha s bindery a toolboxy (vlevo), diskovitá kontextová nabídka (vpravo)

- „single-step“: provedení náhodného/zvoleného proveditelného přechodu
- „play“: provedení zvoleného počtu přechodů s grafickým výstupem mezi jednotlivými kroky
- „stop“: zastavení simulace
- „fast forward“: instantní provedení zvoleného počtu přechodů

U přechodů lze definovat také jejich prioritu, která je rozhodující pro automatický výběr provedeného přechodu v případě automatizované simulace.

Důležitým, avšak pochopitelným omezením je nemožnost pracovat s neomezenými proměnnými. Takováto proměnná generuje nekonečný počet možných navázání a je proto nesmyslné síť simulovat. Hodnotu neomezené proměnné nelze určit ze zvolených vstupních tokenů a hranových výrazů vstupní množiny navazovaného přechodu a je tudíž omezena pouze svou barevnou množinou. Aplikace při detekci takovéto proměnné zobrazí odpovídající hlášení. Obrázek 4.2 ilustruje takovouto situaci. Lze si též všimnout, že tuto situaci může vyvolat pouze proměnná nekonečné barevné množiny (např. int), navázání proměnných konečných barevných množin (např. bool) je program schopen enumerovat.



Obrázek 4.2: Příklad neomezené proměnné konečné a nekonečné barevné množiny

Kapitola 5

Návrh aplikace

Tato kapitola se věnuje architektuře aplikace využitelné pro výuku CPN. Popisuje zvolené nástroje a postupy, zvýrazňuje odlišnosti s referenčním `CPNTools`. Cílem je deklarovat požadované vlastnosti a vytvořit kompaktní návrh aplikace.

5.1 Uživatelské rozhraní

Pro výběr frameworku bylo podstatných několik požadavků definovaných zadáním a vyplývajících z nároků na aplikaci:

- multiplatformnost
- jednoduchost instalace
- dostatečná paleta standardních prvků pro tvorbu uživatelských rozhraní
- efektivita cílového programu (vyplývá z výpočetní náročnosti plánované analýzy sítě)

Na základě těchto požadavků a zkušeností autora bylo pro implementaci GUI vybráno prostředí Qt.

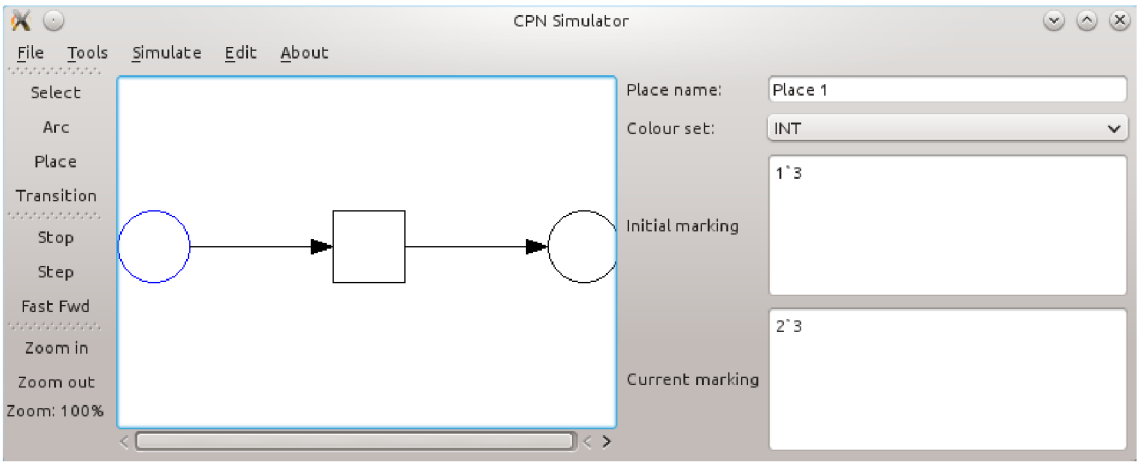
Qt je multiplatformní framework vyvíjený společností Digia. Qt podporuje OS Windows, Linux i Mac a také několik mobilních platforem. Je založeno na dialektu jazyka C++, rozšiřuje jej o několik klíčových slov a syntaktických konstrukcí. Výkonnostní efektivita jazyka C++ splňuje předpoklad pro implementaci analytických součástí aplikace. Qt vyniká svou dynamičností, širokou škálou vestavěných prvků a především sofistikovaným systémem komunikace mezi objekty zprostředkovaném pomocí signálů a slotů. Tyto prostředky umožňují realizaci tzv. loose couplingu, který minimalizuje závislost reakce objektu na vlastnostech původce události. Objekty je též možné za běhu přihlašovat/odhlašovat k/z odběru všech událostí (signálů) a tím dosáhnout maximální flexibility uživatelského rozhraní. Framework poskytuje též prostředky pro implementaci multilinguálních aplikací.

Qt je distribuováno s vlastním SDK s názvem QtCreator. QtCreator je moderní vývojové prostředí s řadou užitečných vlastností, jako je kontextově závislý code assistent, vestavěný debugger či podpora několika programů pro verzování projektu. Qt disponuje též živou uživatelskou komunitou[1], kvalitní dokumentací [18] a řadou volně přístupných tutoriálů.

Při bližším návrhu budeme vycházet z postupů popsaných v ukázkovém projektu DiagramScene[2]Pro samotné zobrazení sítě využijeme tříd `QGraphicsScene` a `QGraphicsView`,

[1] QtCentre, http://www.qtcentre.org/

[2] Qt DiagramScene example, http://doc.qt.digia.com/qt/graphicsview-diagramscene.html



Obrázek 5.1: První verze návrhu uživatelského rozhraní aplikace `CPNSimulator` v Qt

které jsou vhodné pro zobrazování 2D grafických objektů a poskytuje nástroje pro kreslení jednoduchých tvarů. Poskytují též rozhraní pro detekci a zpracování událostí myši.

Jednotlivé uživatelské akce budeme reprezentovat třídou `QAction`, díky čemuž můžeme abstraktně sjednotit jejich vyvolání z různých zdrojů. Standardní kontextové nabídky vytvoříme jako `QMenu`, panely s jednotlivými nástroji (akcemi) flexibilně zobrazuje `QToolBar`.

Úpravy vlastností prvků sítě zprostředkujeme pomocí standardních formulářových prvků `QTextEdit`, `QLineEdit`, `QComboBox` atp. Vlastnosti definované v inskripčním jazyce lze upravovat s využitím generického syntaktického zvýrazňovače `QSyntaxHighlighter`.

Prvotní verzi uživatelského rozhraní ilustruje obrázek 5.1. Oproti nástroji `CPNTools` (obr. 4.1) je patrná snaha o střízlivost, jednoduchost a přehlednost. Veškeré ovládací prvky a postupy jsou intuitivní a notoricky známé z jiných aplikací, důraz je kladen na pochopení samotných principů CPN, nikoli na experimentování s inovativními metodami ovládání.

5.2 Inskripční jazyk

Z důvodů popsaných v předchozích kapitolách bylo učiněno rozhodnutí nerespektovat funkcionální paradigma inskripčního jazyka nástroje `CPNTools`, nýbrž se snažit zjednodušit práci s programem uživatelům navyklým na imperativní jazyky. Jako vzor pro inskripční jazyk uvažujeme jazyk C s několika odlišnostmi:

- V reakci na zadání uvažujeme pouze datové typy int, bool a zvláštní typ unit. Dále potřebujeme definovat též typy reprezentující multimnožiny nad těmito typy, nazveme je multiint, multibool a multiunit. Kompozitní datové typy zcela zanedbáme.
- Vzhledem k tomu, že výsledný kód bude interpretován a nikoli překládán do strojového kódu, zanedbáme preprocesorová makra a podmínky, vkládání souborů atp.
- Není třeba implementovat dynamickou správu paměti.
- Výrazy a operátory rozšíříme o operace nad multimnožinami – zde budeme uvažovat syntaxi podobnou z `CPNTools`, tedy operátor ‘ pro vytvoření multimnožiny, dále $+$ pro jejich sjednocování, $*$ pro skalární násobení a $-$ pro rozdíl.

- Syntaktická konstrukce popisující multimnožinu musí splňovat požadavek revertovatelnosti, tedy aby libovolné dosažené značení bylo popsatelné inskripčním jazykem
- Poskytneme standardní konstrukce pro větvení a smyčky, tedy příkazy if-else, while, do-while, for.

Počáteční značení, současné značení, hranové výrazy a stráže definujeme jako výraz odpovídajícího datového typu v tomto derivátu jazyka C. Deklarační část pro celou síť dále obsahuje deklarace a definice funkcí dle standardů jazyka C. Dále obsahuje deklarace globálních proměnných využitelných ve výrazech.

5.3 Kompilace kódu

Pro sestavení kompilátoru využijeme kombinací nástrojů Flex (Lex) a Bison (Yacc).

Flex je unixový nástroj s dlouhou historií. Jeho prací je sestavení lexikálního analyzátoru (scanneru), který převádí zdrojový text na seznam parametrizovatelných tokenů. Podobu tokenů vymezujeme regulárními výrazy, Flex na základě takovéhoto zdrojového popisu sestaví zdrojový kód v jazyce C/C++ implementující konečný automat rozdělující vstupní text na jednotlivé tokeny.

Bison je historicky svázán s Flexem a velmi často jsou tyto dva programy používány ve vzájemné spolupráci pro zpracování zdrojového textu. Bison na základě zdrojového textu ve formátu podobném LR-gramatikám vytvoří zdrojový kód v jazyce C/C++ pro LR-automat přijímající specifikovaný jazyk s možností provádění sémantických akcí spojených s aplikací jednotlivých pravidel.

Přestože oba nástroje nativně preferují spíše generování kódu v jazyce C, pomocí vhodných přepínačů a direktiv lze vygenerovat i adekvátní C++ kód s podporou objektové orientace v rámci akcí prováděných v jednotlivých pravidlech.

Detailnější informace o použití těchto nástrojů, typech automatů, gramatik a celkově výstavbě kompilátorů poskytuje [8] či [7].

5.4 Formát uložení

Vzhledem k široké škále aplikací Petriho sítí vzniklo mnoho programů umožňujících vytvářet a analyzovat různé druhy sítí. Pluralita softwarových nástrojů vedla k potřebě převoditelnosti modelů mezi jednotlivými programy. Výsledkem je značkovací jazyk PNML[3], který je standardizován ISO/IEC [10, 11, 12].

PNML pojímá několik .rng specifikací pro jednotlivé deriváty Petriho sítí. K dispozici jsou i validační soubory pro vysokoúrovňové Petriho sítě. Zápis umožňuje také vkládání komentářů, popisků a především grafických dat pro vizualizaci sítě.

Specifikace PNML však uvažuje použití funkcionálního inskripčního jazyka. Co více, jednotlivé výrazy a podvýrazy jsou mapovány přímo na XML elementy (jako např. element pro součet, násobení atp) což značně ztěžuje možnosti rozšíření a variability jazyka. Pro ratifikaci vlastního schématu je nutné nový koncept odůvodnit a projít složitým procesem schvalování. Podstatným nedostatkem je též fakt, že ani nástroj `CPNTools` s tímto formátem neumí pracovat (jeho podpora je pouze v seznamu uvažovaných rozšíření). Převoditelnost sítí, která je zásadním důvodem pro využití unifikovaného značkovacího jazyka, tímto ztrácí význam. Ve vyvíjené aplikaci bylo na základě těchto argumentů učiněno rozhodnutí PNML

[3] angl. „Petri Nets Markup Language“, http://www.pnml.org/

nevyužít a vytvořit vlastní XML schéma šité na míru možnostem programu. Export a import souborů v PNML formátu se tedy obdobně jako u `CPNTools` přesunuje do sféry volitelných rozšíření.

Kapitola 6

Implementace

Tato kapitola popisuje stěžejní prvky a postupy využité při vývoji nástroje `CPNSimulator`. Architektura programu vychází z návrhu prezentovaném v kapitole 5.

Aplikace byla dle plánu implementována v prostředí Qt s využitím deklarovaných externích nástrojů. Okno aplikace je vytvořeno pomocí třídy `QMainWindow` a využívá standardních prvků pro tvorbu uživatelského rozhraní. Ovládací prvky jsou umístěny v roletových nabídkách `QMenu`, k nejpoužívanějším funkcím usnadňuje přístup několik sad nástrojů založených na třídě `QToolBar`. Pro omezení duplicit kódu obsluhujícího stejné události z těchto dvou různých umístění je využito třídy `QAction` a napojení signálů z jejích instancí na sloty hlavního okna.

Aplikace umožňuje editaci více sítí současně. Přepínání mezi sítěmi stejně jako přehled otevřených sítí zajišťuje `QTabWidget`. Obsah jednotlivých záložek závisí na konkrétním stavu práce a přechází mezi třídami `CPNetEditor` a `CPNetSimulator` pro úpravu, resp. simulaci sítě. O exkluzivní zobrazení těchto widgetů se stará třída `QStackedWidget` poskytující rozhraní pro zobrazení právě jednoho widgetu z mnoha.

Následující sekce se detailněji věnují konkrétním prvkům programu `CPNSimulator`.

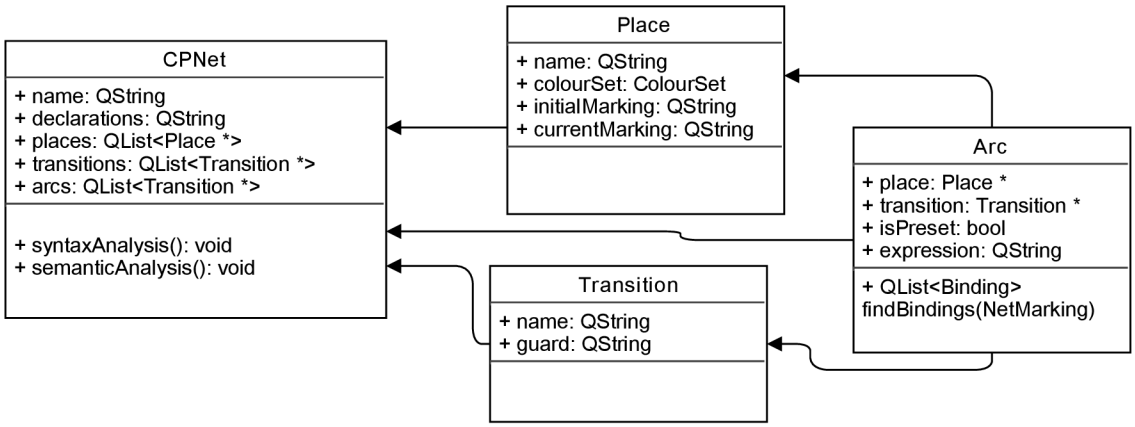
6.1 Model sítě

Snahou autora bylo co nejlépe oddělit datovou část sítě od grafické, stejně jako z důvodů zvýšení výkonnosti a znovupoužitelnosti minimalizovat množství tříd dědících z `QObject` (podpora signálů, slotů a závislostí objektů s sebou nese jistou režii).

Model sítě zachycený na obrázku 6.1 znázorňuje základní atributy tříd, korespondující s formální definicí sítě. Barvená Petriho síť je tvořena globálními deklaracemi, množinami míst, přechodů a jejich spojnic. Tyto třídy si s sebou nesou odpovídající inskripce. Vlastnosti vizualizačního charakteru záměrně nejsou součástí těchto objektů v reakci ke zvolené filozofii.

Model nezachycuje všechny metody a atributy přítomné ve výsledné implementaci – jeho účelem není plně dokumentovat a specifikovat třídy, nýbrž poskytnou zevrubný náhled na datové struktury. Význam některých metod bude osvětlen v následujících sekcích.

Pro vizualizaci sítě v editoru, resp. simulátoru je využit Graphics View Framework[17]. Jedná se o sadu tříd poskytujících užitečné a přitom výkonné nástroje pro vytváření, úpravu a zobrazení 2D grafických objektů. Pro jednotlivé entity přítomné v sítí byly implementovány třídy dědící z `QGraphicsItem` (či jejich potomků `QGraphicsRectItem` a `QGraphicsEllipseItem`). Jejich zobrazení zajišťují `QGraphicsView` a `QGraphicsScene` či třídy z nich



Obrázek 6.1: Zjednodušený model barvené Petriho sítě

dědící. Nastavením příznaku `QGraphicsItem::ItemIsMovable` poskytneme uživateli možnost předměty jednoduše přemisťovat tažením myši.

Třídy `PlaceItem` a `TransitionItem` zobrazující místa, resp. přechody obsahují kromě odkazu na odpovídající objekt též seznam typu `ArcItem`, tedy hran incidentních s danou entitou. Pro korektní aktualizaci pozice hran při pohybu s přilehlými objekty je totiž nutné hranám tuto událost sdělit. Nastavením příznaku `QGraphicsItem::ItemSendsGeometryChanges` si vynutíme volání metody `itemChange` při změně pozice objektu. V těle metody dále předáme zprávu o změně polohy.

Třída `ArcItem` sloužící k zobrazování hran vykresluje zaoblené šipky pomocí kvadratických beziérových křivek [6]. V závislosti na směru šipky jsou nejdříve vypočítány průsečíky s místem/přechodem, kontrolní bod křivky je následně umístěn na ose úsečky několik pixelů ve směru normálového vektoru. Pro lepší ovladatelnost je vhodně implementována i metoda `shape()` definující plochu, ze které hrana přijímá události myši použité pro označení šipky. Tato plocha je při vykreslování prvku opsána pomocí `QPainterPathStroker` pakliže je šipka zvolena.

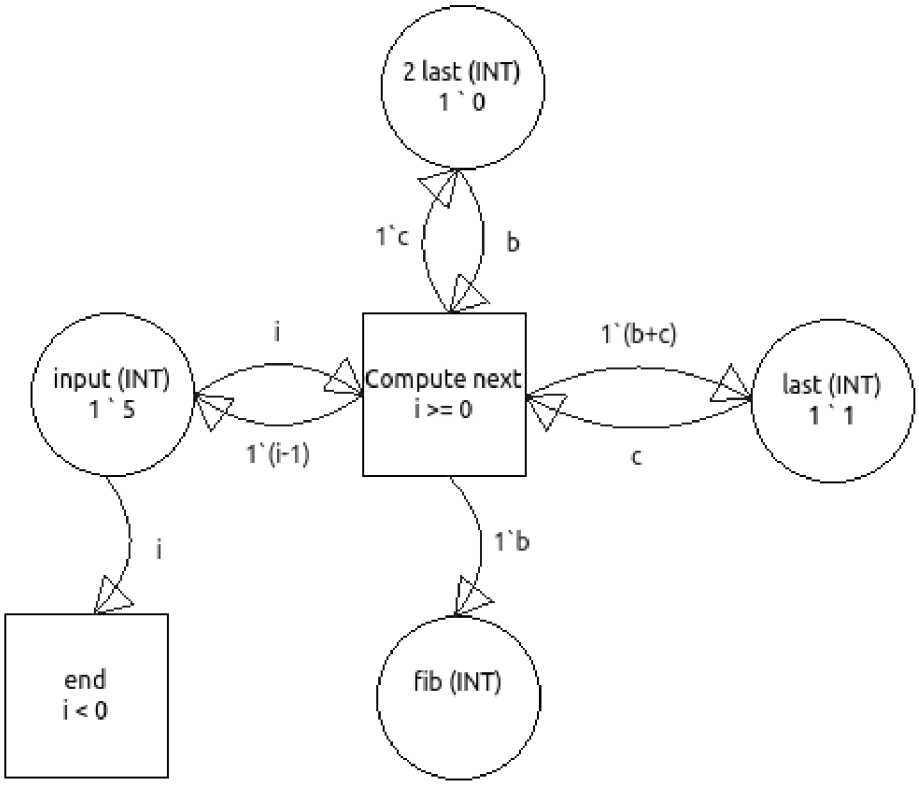
Všechny zmíněné třídy dále reimplementují metodu paint tak, aby kromě základního tvaru zobrazovaly též inskripce či názvy objektů. Použití ilustruje obrázek 6.1.

6.2 Editor sítě

Widget pro editaci sítě implementuje třída `CPNetEditor`. Obsahuje mřížkový layout s náhledem scény – `QGraphicsView` s `EditorScene` – a dále několik formulářů pro úpravu inskripcí jednotlivých entit a sítě samotné. Zobrazení konkrétního formuláře je řízeno dle aktuálně vybrané entity a iniciováno signálem `QGraphicsScene::selectionChanged`. Widget dále obsahuje tabulku s popisem chyb nalezených při kompilaci inskripcí sítě.

Pro přehlednější úpravu inskripcí je využita třída `InscriptionEdit` dědící ze standardní `QTextEdit`. K základní funkcionalitě přidává ještě zvýrazňování syntaxe (klíčových slov, operátorů atp.) pomocí potomka třídy `InscriptionHighlighter`. Widget má také přetíženou metodu `keyPressEvent()` tak, aby při odřádkování zachovával odsazení, jak je tomu zvykem u většiny editorů programovacích jazyků.

Třída `EditorScene` dědící z `QGraphicsScene` dále zprostředkovává interakci se sítí. Disponuje několika módy podle aktuálně zvoleného nástroje. Umožňuje přidávat místa,



Obrázek 6.2: Zobrazení barvené Petriho sítě

přechody a hrany, dále tyto entity mazat či označovat. Všechny tyto akce jsou realizovány pomocí přetížených metod `mousePressEvent()`, `mouseReleaseEvent()` a `mouseMoveEvent()`. Z výkonnostních důvodů totiž prvky scény nedědí ze třídy `QObject` a neposkytují proto rozhraní k zapojení signálů a slotů. Scéna musí při těchto operacích zachovávat konzistenci mezi datovým modelem sítě znázorněným v obrázku 6.1 a odpovídajícími grafickými prvky.

6.3 Zpracování inskripčního jazyka

Zpracování zdrojového textu je dle běžné praxe rozděleno do dvou základních fází – lexikální analýzy realizované nástrojem `Flex` a syntaktické analýzy zajištěné nástrojem `Bison`. Pro překlad těchto souborů současně se zbytkem zdrojových kódů tak, aby bylo možno pro kontrolu verzí využít `qmake` a využívat celý framework ve scanneru/parseru, byly do projektu (.pro souboru) includovány dva vnořené soubory (`flex.pri`, `bison.pri`) definující parametry zpracování zdrojových *.y a *.l souborů. Díky této úpravě lze projekt jednoduše přeložit pouze kombinací `qmake && make` bez nutnosti použití dalších příkazů.

V případě, že některá z částí kompilátoru narazí při zpracování vstupu na chybu, oznámí toto síti samotné voláním funkce `addError`. Všechny posbírané zprávy včetně odkazu na odpovědné objekty a inskripce jsou poté zobrazeny ve výpisu kompilace v editoru sítě. Po kliknutí na zprávu editor automaticky zvolí dotyčný prvek a přesune kurzor na problematické místo v odpovídající inskripci.

Datové struktury potřebné pro zpracování jazyka (`Command`, `Expression`, ...) jsou deklarovány, resp. definovány v souboru `compiler.h`, resp. `compiler.cpp`. Následující sekce se blíže věnují vytvořenému lexeru a parseru.

6.3.1 Scanner

Scanner implementovaný v soboru `scanner.l` definuje regulární výrazy pro zpracování jednotlivých tokenů. Pro potlačení některých ne příliš vhodných výchozích vlastností `Flexu` (jak je vysvětleno v [7]) deklaruje soubor pomocí direktivy `%option` tři vlastosti výsledného scanneru:

- `noyywrap` – pro zamezení volání wrapovací funkce při dosažení konce souboru
- `nodefault` – pro možnost zachytávání lexikálních chyb (jinak by byl neznámý symbol nahlášen až v parseru)
- `yylineno` – pro uchovávání čísel řádků

Direktivou `%x` dále deklarujeme jeden exkluzivní (narozdíl od obdobné syntaxe `%s`) stav pro situaci, kdy se scanner nachází uprostřed blokového komentáře.

Scanner tedy detekuje celkem dva druhy lexikálních chyb:

- neukončený blokový komentář – reprezentovaný výrazem `<COMMENT><<EOF>>`
- neznámý symbol – reprezentovaný výrazem `.` jako posledním pravidlem.

Zbytek pravidel tvoří standardní regulární výrazy zpracovávající různé druhy tokenů a jejich obsah. Pro možnost parsovat vstup definovaný přímo pomocí řetězce `QString` je implementována funkce `parseQString` využívající typu `YY_BUFFER_STATE` a souvisejících funkcí pro podsunutí daného textu na vstup.

Operátor	Význam	Asociativita
`++ -- - !`	inkrement, dekrement, unární minus, negace	nonassoc
`‘`	vytvoření multimnožiny	left
`* / %`	multiplikativní operátory	left
`+ -`	aditivní operátory	left
`> < >= <=`	porovnání	left
`== !=`	rovnost, nerovnost	left
`&&`	logický součin	left
`||`	logický součet	left
`= ? :`	přiřazení, ternární operátor	right

Tabulka 6.1: Precedence operátorů inskripčního jazyka (sestupně)

Soubor obsahuje ještě dvě pomocné direktivy využité pro rozlišení syntaxe jednotlivých typů parsovaných inskripcí. Jejich význam a účel je osvětlen v následující sekci, neb úzce souvisejí s parserem samotným.

6.3.2 Parser

LR parser je popsán v souboru `parser.y`. V deklarační sekci je podstatná redefinice funkce `yyerror` tak, aby zpracovávala chyby způsobem zmíněným v sekci 6.3 namísto prostého výpisu do konzole a definice `%error-verbose` direktivy pro získání detailního popisu syntaktických chyb. Dále je zde poněkud netradičním způsobem vyřešen klasický shift/reduce konflikt větveých výrazů `if` a `if/else`. Tokenu `ELSE` je přiřazena priorita před pseudotokenem `THEN`, čímž se, narozdíl od direktivy `%expect`, která má potenciál zakrýt jiný konflikt, bezpečně zbavíme výstrahy při překladu parseru.

Příkaz `%union` deklaruje všechny přípustné datové typy, které mohou být v tokenech obsaženy. Jelikož jazyk C++ z pochopitelných důvodů neumožňuje v uniích přítomnost prvků s konstruktory/destruktory, veškeré objekty a složitější datové struktury jsou reprezentovány odkazem a dynamicky alokovány a dealokovány v jednotlivých pravidlech. Podstatné je uvést pro tyto tokeny pomocí `%destructor` příkazy pro dealokaci paměti v případě, že parser narazí na syntaktickou chybu. Neuvedení těchto vlastností by mohlo vést k únikům paměti při vzpamatovávání z chyb.

Precedence operátorů vychází z definice jazyka C++ [9] a je implementována pomocí kontrolních sekvencí `%left`, `%right`, `%nonassoc` a `%prec`. Konkrétní skupiny operátorů v sestupném pořadí ilustruje tabulka 6.1. Ternární operátor a logický součin a součet navíc podporují zkrácené vyhodnocení, hodnoty operandů jsou tudíž spočteny pouze pokud na nich skutečně závísí výsledek operace.

Kromě definice tokenů, jejich typů a gramatických pravidel je třeba zmínit ještě jednu podstatnou vlastnost parseru. Vzhledem k tomu, že jednotlivé typy inskripcí se liší svou syntaxí, avšak základní principy zůstávají stejné a mnoho pravidel je sdílených, vytváření zvláštního .y souboru pro každou skupinu zvlášť by vedlo k znatelné redundanci kódu. Bison neumožňuje definovat více než jeden startovací symbol, tuto skutečnost se však autorovi aplikace podařilo obejít. Část s pravidly pro scanner umožňuje vložení speciálního `%{ ...%}` bloku, který dle dokumentace [14] bude vložen do výsledné `yylex()` funkce. Vytvoříme si tedy speciální tokeny reprezentující jednotlivé typy inskripcí, definujeme globální proměnnou `startSymbol` a do scanneru vložíme v sekci s pravidly:

```
if (startSymbol)
{
    int t = startSymbol;
    startSymbol = 0;
    return t;
}
```

Před voláním funkce `yyparse()` stačí nastavit hodnotu startSymbol na požadovaný typ inskripce a upravit startovací pravidlo parseru:

```
start: START_DECLARATION declarationList
    | START_EXPRESSION expression
    | ...
```

Tímto způsobem obejdeme nutnost mít jeden startovací symbol a můžeme parsovat všechny druhy inskripcí stejným parserem. Syntaktické chyby jsou odchytávány na globální úrovni pravidlem `start: error`.

6.4 Omezení inskripcí

Vzhledem k faktu, že barvené Petriho sítě jsou standardně popisovány jazyky pracujícími nad funkcionálním paradigmatem, narážíme při snaze použít imperativní jazyky na jistá omezení. Tato sekce rozebírá jak celkový přístup k sémantickým kontrolám v implementovaném jazyce, tak i syntaktickým/sémantickým omezením kladeným na inskripce samotné.

Jazyk samotný pracuje se syntaxí řízeným překladem se silnou statickou kontrolou datových typů [8]. Tabulka symbolů je implementována jako zásobník tabulek jednotlivých rozsahů platnosti proměnných. Tabulky samotné pak využívají třídu `QMap` implementovanou jako červeno-černý strom. Alternativou by mohlo být použití třídy `QHash` používající tabulky s rozptýlenými položkami (jak doporučuje [8]), avšak vzhledem k absenci překladačových proměnných a k faktu, že strom je v případě malého počtu položek dosahuje lepší výkonnosti[1] by touto úpravou k žádnému zvýšení efektivity nedošlo. Při interpretaci dále rozlišujeme dva druhy tabulek – tabulku pro proměnné a tabulku pro funkce. Jazyk provádí v případě možnosti několik typů implicitních konverzí, dále je možné uvést i explicitní přetypování, pakliže je podporováno. Jazyk podporuje datové typy s klíčovými slovy `unit`, `bool`, `int`, mytextttmultiunit, `multibool`, `multiint`. Datový typ `unit` nemá žádné klíčové slovo pro označení typu – `unit` reprezentuje přímo jedinou dostupnou hodnotu. Implementované typové konverze popisuje tabulka 6.4.

6.4.1 Deklarace sítě

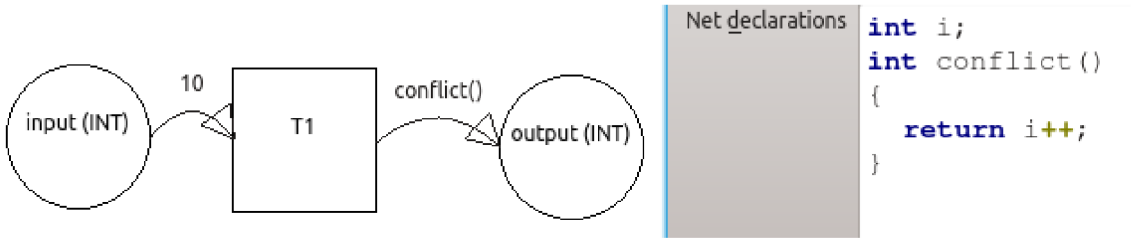
V deklarační části se mohou vyskytovat tři druhy výrazů:

- deklarace globálních proměnných
- deklarace funkcí
- definice funkcí

[1] http://woboq.com/blog/qmap_qhash_benchmark.html

z/na	unit	bool	int	multiunit	multibool	multiint
unit	-	true:false	0:1	1‘unit:nounit	nedef.	nedef.
bool	unit:nounit	-	0:1	nedef.	1‘true:false	nedef.
int	unit:nounit	true:false	-	nedef.	nedef.	1‘int
multiunit	nedef.	nedef.	nedef.	-	nedef.	nedef.
multibool	nedef.	nedef.	nedef.	nedef.	-	nedef.
multiint	nedef.	nedef.	nedef.	nedef.	nedef.	-

Tabulka 6.2: Tabulka dostupných konverzí a jejich význam



Obrázek 6.3: Nejednoznačnost vzniklá použitím globální proměnné

Datové typy globálních proměnných jsou vzhledem ke konceptu simulace omezeny na jednoduché (nikoli multimnožinové) datové typy, konkrétně tedy `int` a `bool` (datový typ `unit` není z důvodů popsaných v sekci 6.4 možné použít).

Deklarace funkcí je nutno uvést v případě, kdy chceme použít (cyklickou) rekurzi. V tomto případě se kontroluje jednak shoda v typech, názvech a počtech parametrů a jednak definice všech deklarovaných funkcí.

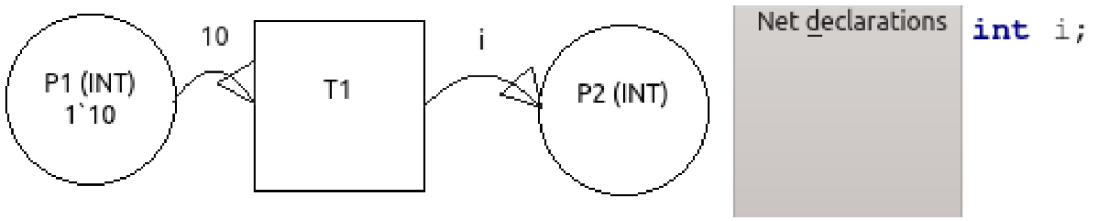
Všechny parametry funkcí jsou předávány hodnotou. Definice funkcí navíc podléhají oproti svým vzorům v jazyce C jednomu významnému omezení – nemohou pracovat s globálními proměnnými, přiřazovat do nich ani číst jejich hodnotu. Důvodem je skutečnost, že hodnoty nejsou globálním proměnným nijak explicitně přiřazovány. Jejich hodnota je udána až při zvolení konkrétního navázání při simulaci sítě a v mnoha případech ani není určena. Manipulace s nimi by navíc vyžadovala určení pořadí vyhodnocování jednotlivých výrazů v síti což se neshoduje s koncepty barvených Petriho sítí. Z těchto důvodů je jakákoli práce s nimi jakožto s globálními proměnnými zcestná a kompilátor ji vyhodnotí jako chybnou. Příklad takovéto situace ilustruje obrázek 6.3.

6.4.2 Značení míst

Vzhledem k faktu, že výrazy pro značení musejí být konstantní a jejich vyhodnocení se provádí bez jakéhokoli navázání globálních proměnných, je jakékoli použití proměnných či přiřazení v těchto inskripcích vyhodnoceno jako chyba. Ve výrazech se však stále může vyskytovat volání definovaných funkcí a celkově libovolný konstantní výraz konvertovatelný na multimnožinu typu daného místa.

6.4.3 Presetové hrany

Výrazy uvedené u presetových hran (z hlediska přechodů) mají v síti významnou úlohu. Právě na jejich základě se totiž určuje navázání proměnných tak, aby jejich vyhodnocením vznikla multimnožina neostře menší (dle definice této relace z definice 3.2) než aktuální



Obrázek 6.4: Porušení podmínky pokrytí proměnných

ohodnocení incidentního místa. Nalezení takovéhoto navázání není v žádném případě triviální záležitostí, vyžaduje jistou míru revertovatelnosti výrazů a velmi jednoduše lze narazit na situaci, kdy je množina všech možností nekonečná. Příkladem budiž použití jakékoli periodické funkce (hranový výraz 6.1) či eliminace významu dané proměnné (hranový výraz 6.2). Často je také nutné zkoumat ostatní hranové výrazy, obsah místa či stráž příslušného přechodu, které mohou poskytovat další informace o množině hodnot, jenž lze k dané proměnné navázat (hranový výraz 6.3).

$$E(a) = \sin(x) \tag{6.1}$$
$$E(a) = 1`10 + 0`x + 3`(x - x) \tag{6.2}$$
$$E(a) = \sin(x), \quad G(t) = x \geq 0 \wedge x \leq 2\pi \tag{6.3}$$

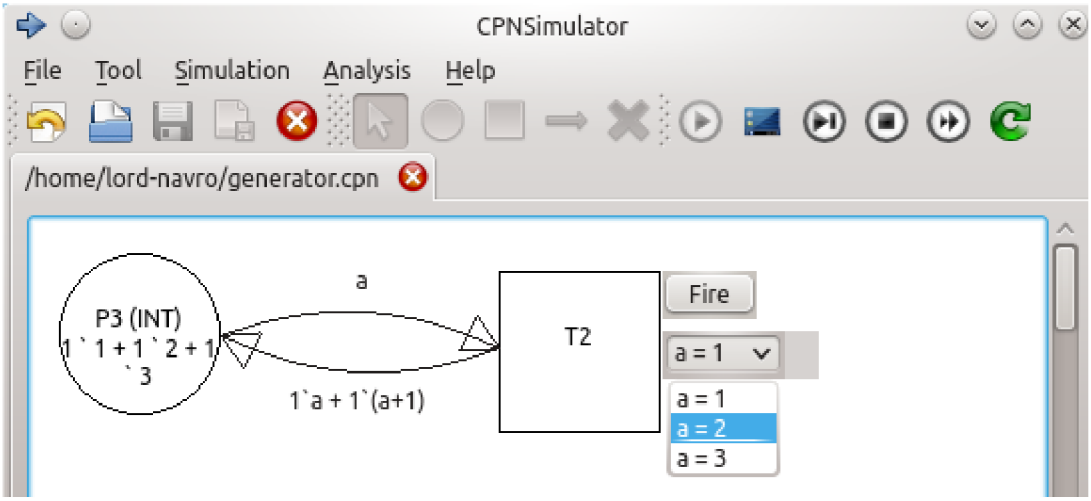
V reakci na deklarované problémy byla zvolena vhodná omezení pro hranové výrazy tak, aby bylo navázání z hranových výrazů jednoznačně určitelné. Každý výraz musí odpovídat jednomu z následujících vzorů:

1. $\langle colour \rangle$
2. $\langle variable \rangle$
3. $\langle int \rangle$ ` $\langle colour \rangle$
4. $\langle int \rangle$ ` $\langle variable \rangle$

kde $\langle colour \rangle$ je libovolná barva z barevné množiny místa, $\langle variable \rangle$ je proměnná odpovídající barevné množiny a $\langle int \rangle$ je numerická konstanta. V uvedeném výčtu mají body 1, 2 totožný význam s body 3, resp. 4, uvažujeme-li $\langle int \rangle = 1$.

6.4.4 Postsetové hrany, stráže

Výrazy postsetových (z hlediska přechodu) hran a stráží podléhají výjma omezení svého typu (boolovská hodnota u stráží resp. multimnožina nad barevnou množinou incidentního místa u hran) ještě podmínce použití proměnných. Všechny proměnné vyskytující se v těchto inskripcích se musí nacházet též v presetových hranách daného přechodu. Tímto zaručíme, že při provedení přechodu k nim bude navázána konkrétní hodnota a dané výrazy budou tudíž jednoznačně vyhodnotitelné. Nejasnost vzniklou porušením této podmínky ilustruje obrázek 6.4.



Obrázek 6.5: Uživatelské rozhraní simulátoru

6.5 Simulace sítě

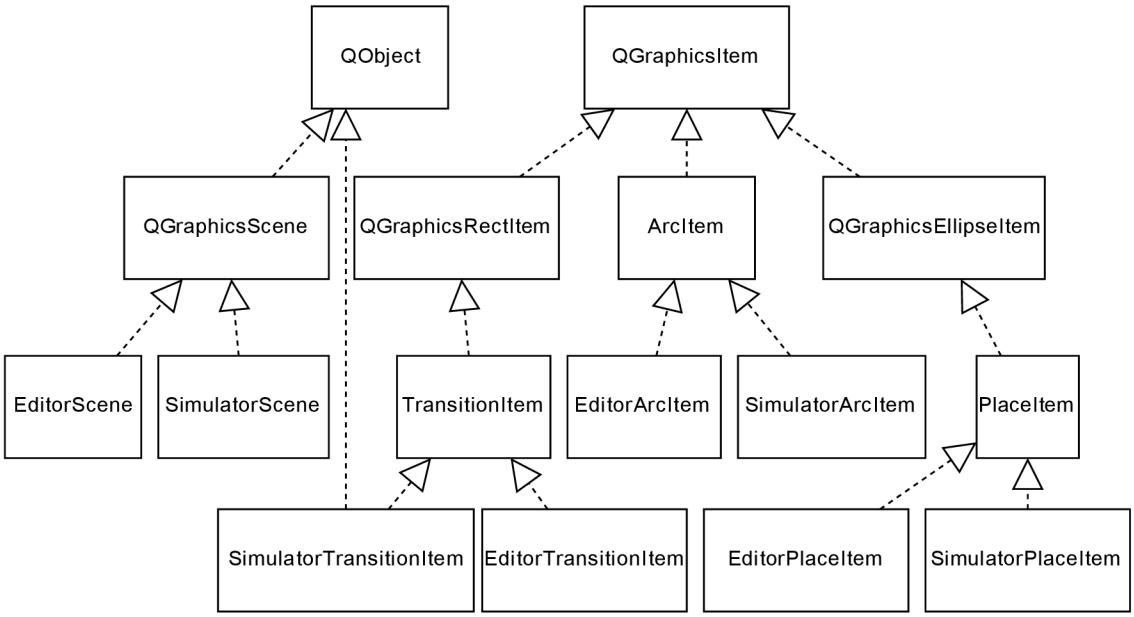
V případě úspěchu kompilace se nástroj `CPNSimulator` přepne do simulačního režimu. Místy a přechody lze stále pohybovat v případě, že jejich současná situace omezuje čitelnost sítě. Nástroje pro přidávání/mazání entit a úpravu inskripcí jsou však znepřístupněny. Při zahájení simulace se značení míst inicializuje na hodnotu uvedenou v poli „Current marking“ v editoru. Dále je vyhodnocena proveditelnost jednotlivých přechodů a v případě pozitivního výsledku je u přechodu znázorněna nabídka dostupných navázání a tlačítko pro provedení přechodu.

V simulátoru má uživatel možnost interagovat s programem následujícími akcemi, přístupnými skrze ikony v simulačním panelu nástrojů:

1. Návrat do editoru – ukončí simulační režim a zobrazí editor sítě
2. Provedení náhodného přechodu – z množiny proveditelných přechodů a odpovídajících navázání je zvolen náhodný prvek a proveden, následně je znovu spočtena proveditelnost jednotlivých přechodů
3. Návrat do původního značení – značení všech míst jsou změněna na hodnotu získanou vyhodnocením výrazu v poli „Initial marking“
4. Provedení několika náhodných přechodů – po zadání počtu má stejný efekt jako opakované provádění náhodného přechodu z bodu 2
5. Nalezení navázání – v případě, že hledání navázání nebylo ukončeno korektně, lze tuto akci reprodukovat manuálně
6. Vygenerování stavového prostoru – slouží k analýze sítě popsané v sekci 6.7
7. Manuální výběr navázání a provedení přechodu – skrze ovládací prvky přímo na scéně

Rozhraní simulátoru ilustruje obrázek 6.5.

Scéna uživatelského rozhraní je implementována obdobně jako v editoru v třídě `SimulatorScene` dědící z `QGraphicsScene`. Za zmínku stojí metoda `loadNetGraph` která do



Obrázek 6.6: Hierarchie tříd zobrazujících entity sítě

scény nakopíruje a umístí grafické prvky z instance třídy `EditorScene`. Prvky zobrazující přechody jsou instancemi třídy `SimulatorTransitionItem`. Tato třída jako jediná z použitých potomků `QGraphicsItem` dědí také ze třídy `QObject`, aby získala možnost používat ke komunikaci signály a sloty. `QComboBox` a `QPushButton` sloužící k výběru navázání, resp. provedení přechodu jsou na scéně zobrazeny skrze `QGraphicsProxyWidget`, který emuluje prostředí potřebné k zobrazení instance `QWidget` uvnitř `QGraphicsScene`. Tyto ovládací prvky jsou pomocí signálů a slotů propojeny s objektem zobrazujícím přechod, který stejným způsobem předává informace o událostech uživatelského rozhraní. Bližší náhled do třídní hierarchie jednotlivých grafických prvků poskytuje obrázek 6.6.

Následující sekce se blíže věnují implementaci konkrétních úloh prováděných při simulaci sítě.

6.6 Hledání proveditelných elementů navázání

Hledání dostupných navázání je patrně jedním z nejdůležitějších úkolů při implementaci simulátoru barvených Petriho sítí, jak uvádí i [4, s.73-75]. Při řešní tohoto problému musíme brát v úvahu tří základní elementy – ohodnocení vstupních míst, výrazy u vstupních hran a také stráž daného přechodu. Teoreticky dostupný stavový prostor, v němž prvky navázání hledáme, je často nekonečný a prohledávání silou tudíž nepřichází v úvahu. Jedinou možností jak tento stav změnit by bylo omezit množiny barev vlastností konečnosti. Tím by však modelu znatelně ubylo na eleganci a vyjadřovacích schopnostech, navíc v případě rozsáhlejších sítí (především u přechodů s mnoha vstupními hranami) či větších barevných množin by se i tento přístup projevil jako značně neefektivní.

Na rozdíl od referenčního nástroje `CPNTools` nepracuje `CPNSimulator` s kompozitními datovými typy a nepotřebuje tudíž pro tento úkol využívat pattern matchingu. Zároveň

však obsahuje neomezený datový typ `integer`[2], který znemožňuje naivní hledání navázání pouhou filtrací výčtu všech dostupných možností.

Prohledávání je však značně zjednodušeno syntaktickými omezeními aplikovanými na vstupní hrany, které mj. zabrání vzniku neomezených proměnných. `CPNSimulator` postupuje při hledání prvků navázání dle následujícího algoritmu:

Algoritmus 6.1 (Hledání elementů navázání)**.** Postupuj dle následujícího algoritmu:

1. Nalezni možná navázání pro všechny hrany, ulož do pole `arcBindingsList`.
2. Inicializuj pole `resultBindings` výsledných navázání na seznam obsahující jeden prvek – prázdné navázání
3. Je-li pole `arcBindingsList` či `resultBindings` prázdné, vrať aktuální pole `resultBindings` jako výsledná možná navázání
4. Vytvoř nové, prázdné pole `newResultBindings`
5. Vyber možná navázání `arcBindings` další hrany z `arcBindingsList`
6. Pro každý prvek z `arcBindings` a každý prvek z `resultBindings`:
   (a) Zkontroluj, zda-li nejsou navázání konfliktní, tedy zda-li nepřiřazují dvě rozdílné hodnoty jedné proměnné
   (b) Pokud nejsou konfliktní, sjednoť a přidej je do pole `newResultBindings`
7. Vezmi `newResultBindings` jako novou hodnotu `resultBindings` a vrať se k bodu 3

Nalezení možných navázání použité v bodě 1 algoritmu 6.1 pro jednotlivé hrany pak probíhá podle následujícího předpisu:

Algoritmus 6.2 (Hledání elementů navázání u hran)**.** Pro zadanou hranu:

1. Inicializuj prázdné pole `arcBindings`
2. Je-li hranový výraz ve tvaru $\langle konstanta \rangle$ ‘ $\langle konstanta \rangle$, zkontroluj, zda-li místo obsahuje dostatečné množství tokenů. Pokud ano, přidej do `arcBindings` prázdné navázání
3. Je-li hranový výraz ve tvaru $\langle int \rangle$ ‘ $\langle var \rangle$. Postupuj dle barevné množiny místa:
   (a) Boolovské místo: Je-li v místě alespoň $\langle int \rangle$ tokenů $true$, přidej do `arcBindings` navázání obsahující jediný prvek $\langle var \rangle$ $rightarrowtrue$, obdobně s $false$
   (b) Integerové místo: Najdi všechny hodnoty tokenů $\langle val \rangle$, jejichž počet v daném místě je alespoň $\langle int \rangle$, a pro každou takovou hodnotu přidej do `arcBindings` navázání obsahující jediný prvek $\langle var \rangle \rightarrow \langle val \rangle$
   (c) Jednotkové místo: Tato situace nemůže nastat, neboť nelze vytvořit proměnnou typu `unit`

[2]Proměnné tohoto typu mohou sice nabývat pouze omezeného množství hodnot, prohledávání kompletního stavového prostoru jejich možných navázání však není v rozumném časovém intervalu proveditelné, což má totožné důsledky jako v případě, pokud by množství hodnot omezeno nebylo

4. Vrať `arcBindings` jako výslednou hodnotu

Po nalezení všech možných prvků navázání pro daný přechod algoritmem 6.1 je ještě nutné zkontrolovat stráž, tedy aplikovat toto navázání na tabulku globálních proměnných a zkontrolovat, zda-li vyhodnocením stráže dostaneme hodnotu `true`.

Zmíněné algoritmy jsou implementovány v `Arc::findBindings`, `Computer::findBinding` a `Computer::mergeBindings` a tvoří jádro simulátoru.

6.6.1 Interpretace jazyka

Interpretace příkazů a výrazů je prováděna na základě datových struktur vytvořených parserem. Výrazy a příkazy nejsou překládány do tříadresného kódu, jak bývá zvykem zejména u překladačů do strojového jazyka, nýbrž udržovány ve strukturách přirovnatelných k výpočetním či abstraktním syntaktickým stromům. Důvodem je nutnost pozdější sémantické analýzy a hledání použitých proměnných, kterézto úkony jsou snáze proveditelné nad těmito objekty nežli nad jednolitým seznamem mezikódu s velkým množstvím dočasných proměnných. Z hlediska vyhodnocování výrazů je forma výpočetního stromu též velmi vhodná. Nelze opomenout také roztříštěnost jednotlivých úseků kódu napříč rozličnými inskripcemi, která by značně znepříjemňovala konvenční přístup.

Základním stavbním prvkem pro modelování výrazů je třída `Expression` reprezentující uzel výpočetního stromu. Jendotlivé instance obsahují typ prováděné operace (sčítání, násobení, proměnná atp.) a potřebné argumenty (odkazy na další uzly, id proměnné atp). Vzhledem k deklarované silné statické typové kontrole je výrazu v době kompilace přiřazen též datový typ výsledku. Vyhodnocení výrazu probíhá rekurzivním zanořováním k listům stromu a následným postupným výpočtem až ke kořeni.

Vzhledem k faktu, že implementovaný inskripční jazyk podporuje imperativní definici globálních funkcí, nevystačíme pouze s výpočetními stromy. Datovou strukturu pro těla funkcí tvoří třída `Command`, jejíž instance podobně jako u `Expression` obsahují typ příkazu (výraz, podmínka if, smyčka while atp) a parametry (odkazy na výrazy, příkazy atp). Při provádění příkazů (volání funkce při vyhodnocování výrazu) je kód nejdříve převeden do mezikódu reprezentovaného třídou `InterCode`. Z jejích instancí je vytvořen lineární spojový seznam s podmíněnými/nepodmíněnými skoky a návěštími ve formě podobné konvenčním překladačům.

Vzhledem k popsaným omezením aplikovaným na inskripce se při vyhodnocování výrazů a provádění příkazů předávají dvě tabulky symbolů – tabulka funkcí a tabulka proměnných. Tabulku funkcí vždy reprezentuje globální tabulka symbolů pro danou síť (obsahuje tudíž jen a pouze symboly uvedené v deklaraci sítě). Tabulka proměnných závisí na kontextu dané inskripce:

- Inskripce u entit – výrazy u hran, míst a přechodů – používají globální tabulku sítě
- Příkazy ve funkcích využívají lokální tabulku, která byla vytvořena při vyhodnocení výrazu „volání funkce“ a jejímž prostředníctvím byly předány parametry funkce
- Výrazy uvnitř funkcí využívají lokální tabulku, stejně jako u příkazů

Tato politika předávání kontextu zajišťuje ochranu globálních proměnných proti nežádoucím změnám hodnot, dodržování rozsahu platnosti proměnných a možnost jejich snadné redeklarace uvnitř složených příkazů.

Veškeré hodnoty jsou ukládány v unii `Value` a zapouzdřeny v objektech třídy `Data`, která navíc udržuje informaci o datovém typu konkrétní instance (nutné pro správnou

alokaci/dealokaci datových struktur pro typ `multiint`). Aritmetické operace potřebné při interpretaci jsou jednoduše použitelné s využitím implementovaných přetížení operátorů a kopírovacího konstruktoru.

6.6.2 Provádění výpočtů a zotavování z chyb

Variabilita a volnost ve vyjadřování, kterou barvené Petriho sítě poskytují, má i svou stinnou stránku – uživatel může do sítě vnést sémantické chyby, neošetřit chybné vstupní hodnoty či vytvořit velmi neefektivní výpočetní model. Během simulace sítě lze tudíž narazit na několik problémů:

1. Výpočet je příliš náročný, zdlouhavý a blokuje smyčku uživatelského rozhraní
2. Nekonečný cyklus/rekurze v interpretovaném kódu
3. Nelze získat dostatek zdrojů pro provedení výpočtu
4. Při výpočtu se vyskytne výjimka (chybná aritmetická operace):
   - dělení/modulace nulou
   - odečítání multimnožin nesplňující podmínku relace $\leq$ z definice 3.2
   - násobení multimnožiny záporným číslem

Pro správnou funkci simulátoru je však nezbytné analyzovat procesy mající potenciál tyto události vyvolávat. Následně se budeme snažit jim předcházet, či je alespoň detekovat a vhodně eliminovat jejich následky.

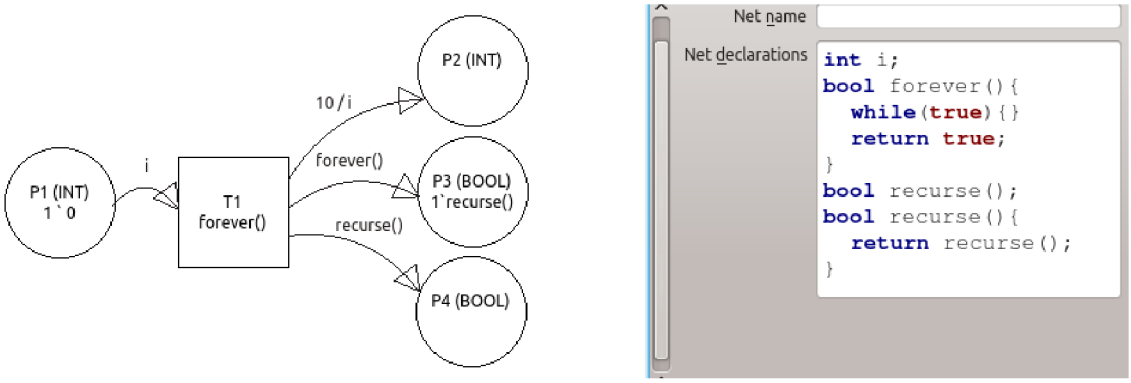
Deklarované obtíže jsou zpravidla spojeny s vyhodnocováním inskripcí a týkají se následujících úkolů:

1. Výpočet aktuálního/počátečního značení – výrazy mohou obsahovat libovolné konstantní výrazy a volání funkcí
2. Výpočet dostupných prvků navázání – algoritmus 6.1 je sice „terminating“, avšak vyhodnocení stráže pro kontrolu výrazu je zcela v rukou uživatele
3. Provádění událostí – problémy mohou nastat při vyhodnocení výrazů výstupních hran přechodu
4. Generování stavového prostoru – z velké části se skládá z operací 2 a 3

Vznik těchto problémů ilustruje obrázek 6.7.

Aplikace `CPNSimulator` využívá pro tyto nebezpečné operace oddělených vláken. Představují je instace třídy `Computer` dědící z `QThread`. Výpočet v externích vláknech zamezí zablokování smyčky uživatelského rozhraní a s ním spojené nemožnosti interakce s programem. Při spuštění vlákna je scéna simulátoru překryta poloprůhledným grafickým prvkem demonstrujícím nemožnost editace sítě před dokončením akce. Zároveň je zobrazen popis prováděného úkonu a tlačítko pro její zastavení. Stisknutím tlačítka je nastaven příznak `cancelRequest` běžícího vlákna a v dalším cyklu výpočtu je tento asynchronně ukončen.

`Computer` implementuje rozhraní pro spuštění všech deklarovaných operací. Pří dokončení práce vysílá `signalCompleted` nebo `signalFailed` v závislosti na tom, zda-li došlo k chybě či manuálnímu přerušení od uživatele. V rámci výpočtu je též evidována a kontrolována hloubka zanoření, aby nedošlo v důsledku rekurzivního vyhodnocování k přetečení



Obrázek 6.7: Uživatelem vnesené chyby v síti

zásobníku. Předávání informace o chybě je zajištěno pomocí systému výjimek a `catch` bloků.

Podstatné je též zpožděné přiřazování nově spočteného značení při přerušení operace provádění přechodu. Vlákno si následné značení uchovává v dočasné proměnné a skutečně jej na síť aplikuje až v případě, kdy běh úspěšně skončí. Účelem je zabránit stavu, kdy je vykonána jen část operací (např. odstranění tokenů ze vstupních míst bez přidání tokenů do míst výstupních). Síť by se tímto způsobem mohla z hlediska sémantiky modelu dostat do nekonzistentního či jinak nedosažitelného značení.

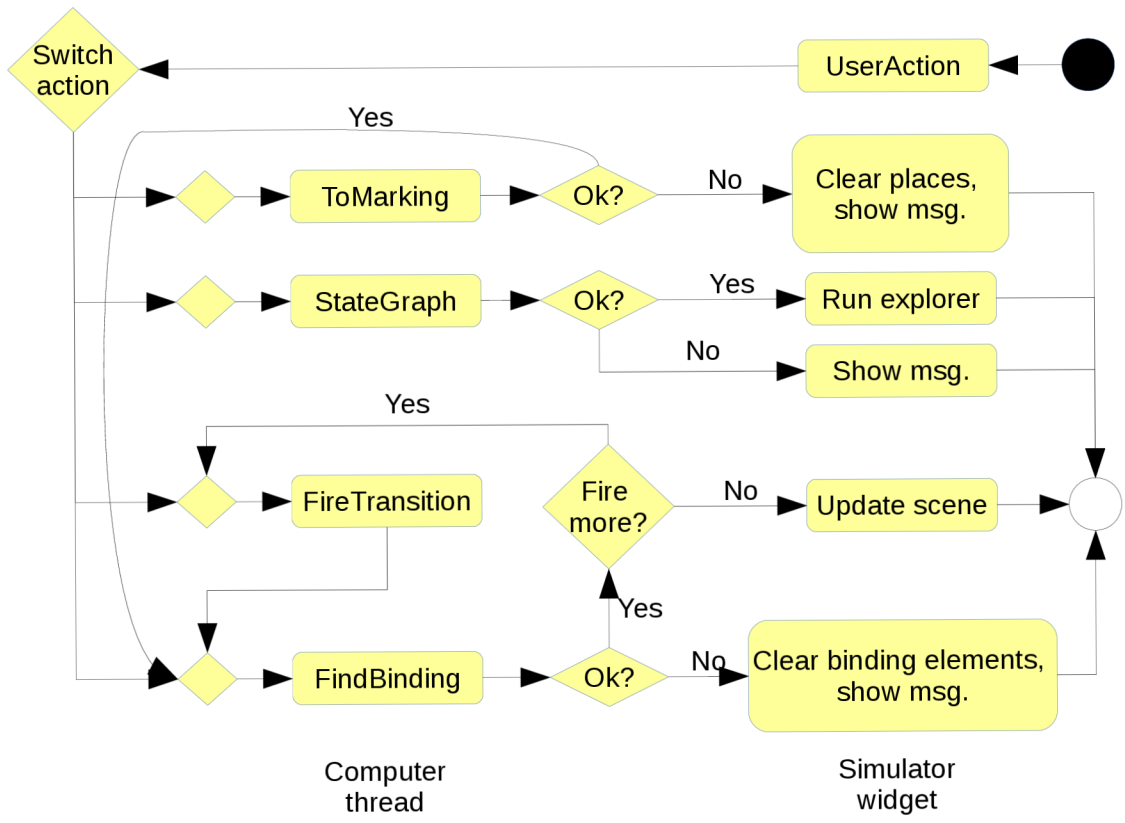
Zodpovědností objektu třídy `CPNetSimulator` je korektně reagovat na signály výpočetního vlákna tak, aby zobrazené značení a nabízené prvky navázání odpovídaly vzniklé situaci. Například po dokončení provádění události musí spustit akci hledání prvků navázání. Komplikace vznikají v případě, že se nepovede vyhodnotit přechod do původního/aktuálního značení. V tomto případě jsou místa vyprázdněna a uživateli zobrazeno varovné hlášení.

Kompletní model synchronizace mezi výpočetním vláknem a simulačním widgetem popisuje obrázek 6.8.

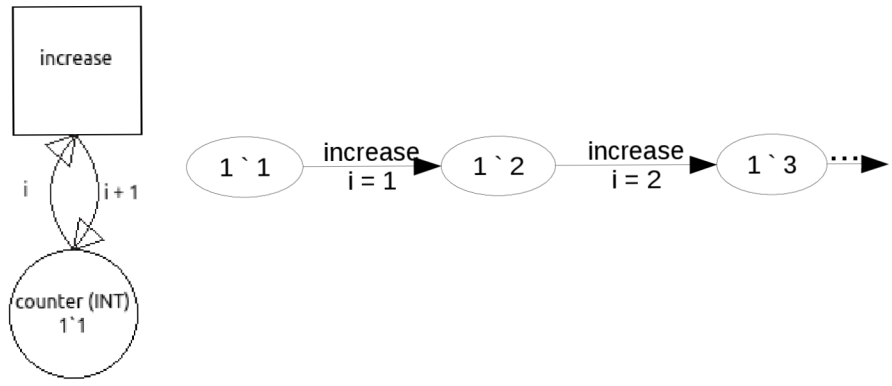
6.7 Nástroj pro zkoumání stavového prostoru

Graf stavového prostoru (*state space graph*, [4, s.154]) barvené Petriho sítě je definován jako orientovaný graf, jehož uzly tvoří všechna dosažitelná značení v síti. Hrana označená prvkem navázání(t, b) z uzlu M_1 do M_2 je součástí grafu právě tehdy, když (t, b) je povolen v M_1 a M_2 je následné značení M_1 při provedení (t, b). Narozdíl od P/T sítí, jejichž charakter nám umožňuje při rozbalování stavového prostoru „generalizovat“ potenciálně nekonečné generování tokenů a omezit tímto velikost grafu (viz [23]), u barvených Petriho sítí obecně nelze podobnou metodu uplatnit. I velmi jednoduché barvené sítě mohou navíc disponovat nekonečnou množinou dosažitelných značení a generovat tudíž nekonečné grafy stavového prostoru, jak ilustruje obrázek 6.9. Přestože všechny stavy ukázkového čítače by bylo možné vyjádřit jakýmsi parametrickým tvarem, tento jev ani zdaleka není pravidlem a úvahu potřebnou k nalezení podobného popisu nelze nijak jednoduše algoritmizovat.

Aplikace `CPNSimulator` umožňuje pro vymodelovanou síť vygenerovat graf značení dosažitelných v omezeném počtu kroků od aktuálního značení a následně jej zkoumat.



Obrázek 6.8: Synchronizace simulátoru a výpočetního vlákna



Obrázek 6.9: Síť modelující jednoduchý čítač (vlevo) a náznak odpovídajícího grafu stavového prostoru (vpravo)

6.7.1 Datový model grafu

Rozhraní pro práci s grafem poskytuje třída `StateSpaceGraph` a pomocné třídy `StateSpaceVertex` a `StateSpaceEdge` implementované v souborech `analyzer.h` a `analyzer.cpp`. Graf je reprezentován pomocí seznamu sousedů, použití matice sousednosti by vzhledem k potenciálnímu rozsahu grafu způsobila znatelné zvýšení paměťových nároků[1, s.24-31]. Graf si též pamatuje příznak, zda-li bylo při jeho generování dosaženo zadaného limitu hloubky zanoření a zda-li tudíž pokrývá celý stavový prostor sítě, či jen jeho část.

Jednotivé uzly si evidují odpovídající značení a stejnětak hrany nesou informaci o provedeném prvku navázání. Odkazy na výchozí, resp. cílová místa jsou z opimalizačních důvodů uchovávána jako celočíselné atributy vyjadřující index uzlu v rámci pole všech uzlů. Tato konvence mírně komplikuje situaci v případě, že by bylo nutné graf obměňovat či mazat místa, avšak vzhledem ke skutečnosti, že se graf po vygenerování již nijak nemění, není třeba se těmito nevýhodami zabývat.

6.7.2 Algoritmy pro generování a zkoumání grafu

Genrování stavového prostoru probíhá prohledáváním do hloubky se složitostí $O(m + n)$ [1, s.52-53]. Tento přístup je výhodnější, neb potřebujeme vždy projít celý stavový prostor a prohledávání do šířky by pouze zvýšilo paměťové nároky pro uložení neprozkoumaných stavů. Metoda backtrackingu by byla použitelná, avšak vyžadovala by opakované přepočítávání proveditelných elementů navázání což by u složitějších výrazů mělo za následek značné zpomalení algoritmu. Zpravidla nejnáročnější operací je zejména u rozsáhlých grafů identifikace dupliciních značení, které je v aktuální verzi implementováno prostým dotazem na výskyt tohoto prvku v poli dostupných stavů. Zde by bylo možno generování urychlit např. vhodným hashováním.

Aplikace `CPNSimulator` poskytuje navíc možnost hledání nejkratší cesty z původního značení do libovolného dosažitelného obsaženého v grafu. Pro tento úkol je použit algoritmus prohledávání do šířky[1, s.49-50], neb snahou je vždy nalézt nejkratší cestu tak, aby její rekonstrukce a analýza byla co nejjednodušší. Algoritmus dosahuje při zadání seznamem sousedů složitosti $O(m + n)$ [1, s.50]. Cesta je zaznamenávána jako seznam indexů uzlů a hran v odpovídajících seznamech.

6.7.3 Grafické rozhraní

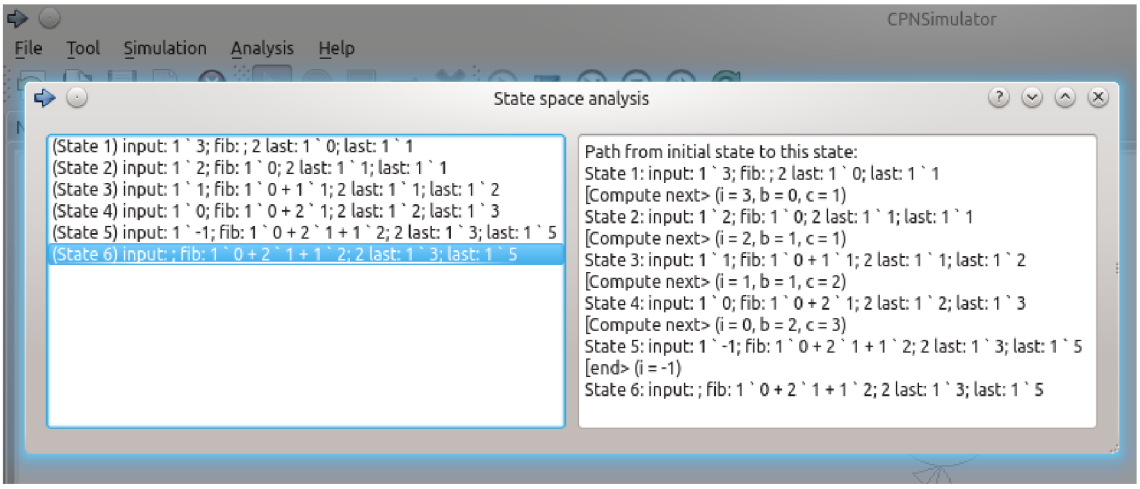
Po úspěšném vygenerování grafu se zobrazí samostatné okno s widgetem `StateSpaceExplorer`. Průzkumník dědí ze třídy `QDialog` a pracuje jako modální okno, při jeho zobrazení je tudíž potlačena jakákoli interakce se zbytkem aplikace, aby nevhodnou úpravou nevznikla nekonzistence mezi sítí a grafem.

Obsahem průzkumníka je instance `QListWidget` na levé straně obsahující instance `QListItemWidget` jako výpis jednotlivých stavů. Při označní některého ze stavů se vyhledá a v pravé časti reprezentované objektem třídy `QTextEdit` zobrazí nalezená nejkratší cesta grafem z původního značení do zvoleného uzlu.

Použití průzkumníka ilustruje obrázek 6.10.

6.8 Načítání a ukládání

Aplikace umožňuje vymodelované sítě ukládat do souborů .cpn a následně je z nich načítat. Obsahem souboru je XML dokument s výčtem jednotlivých míst, přechodů, hran a jejich



Obrázek 6.10: Modální okno průzkumníka grafu dosažitelných značení

vlastností.

Pro generování a zpracování XML dat je využito modulu QtXml [21], který poskytuje rozhraní pro práci s DOM strukturami a stará se o korektní uzavírání elementů a escapeování obsahu. Pro kontrolu načítaných souborů je použit modul Qt XML Patterns [20], který umožňuje validaci dokumentů proti předloženému schématu.

Schéma popisující formát uložení je definováno souborem `schema/savednet.xsd` a ověřuje následující vlastnosti dokumentu:

- správné pořadí a zanořování elementů
- datové typy atributů (výčet pro barevné množiny u míst atp.)
- odkazy napříč elementy pomocí klíčů a referencí (u hran se zkoumá existence incidentního místa a přechodu)

Tyto kontroly zajišťují srozumitelnost .cpn souboru a korektní načtení sítě.

6.9 Nápovědní systém

Aplikace obsahuje vestavěnou nápovědu implementovanou pomocí Qt Help frameworku [19]. Stránky nápovědy jsou sepsány ve formátu HTML a umístěny v adresáři `doc`. V souboru `doccpnsimulator.qhp` je definován projekt nápovědy a jeho obsah, třídění do sekcí pro vygenerování rejstříku a některá klíčová slova spolu se svými referencemi pro vyhledávač. Na základě tohoto projektu byla nástrojem `qhelpgenerator` vygenerována dokumentace ve formátu `.qch`.

Projekt kolekce nápovědy v souboru `doccpnsimulator.qhcp` dále deklaruje další vlastnosti nápovědy (titulek, odpovídající `qhp/qch` soubory, about-text atp) a nástrojem `qcollectiongenerator` z něj vtvoříme kolekci nápovědy `doc/cpnsimulator.qhc`

K samotnému zobrazení nápovědy slouží program `assistant`. Jeho volání z aplikace `CPNSimulator` zprostředkovává implementovaná třída `Assistant`. Její instance obsahuje ukazatel na proces `QProcess` představující běžící aplikaci nápovědy.

Program `assistant` je spuštěn tak, aby přijímal povely ze standardního vstupu (přepínač `-enableRemoteControl`), což umožňuje vzdáleně listovat nápovědou z aplikace `CPN-Simulator` (příkaz `SetSource`). Přepínačem `-collectionFile` je `assistantu` podsunuta vytvořená kolekce nápovědy namísto výchozí dokumentace Qt.

Kapitola 7

Závěr

V rámci projektu byly analyzovány základní syntaktické a sémantické prvky barvených Petriho sítí, stejně jako úskalí spojená s jejich modelováním a simulací. Tato práce poskytuje pevnou bázi pro vývoj požadované aplikace s důrazem kladeným na odstranění deklarovaných nedostatků aplikace `CPNTools`. Vyvinutý program má potenciál nahradit `CPNTools` při výuce tématiky a v porovnání s konkurencí studentům srozumitelněji tlumočit základní koncepty barvených Petriho sítí.

Současná verze stále poskytuje prostor pro další vývoj nástroje. Jedním z rozšíření diskutovaných v sekci 5.4 je import a export sítí z/do formátu PNML, což by usnadnilo komunikaci s jinými aplikacemi. Dále se nabízí možnost různých optimalizací, například hashování jednotlivých značení v grafu stavového prostoru, jak se o něm zmiňuje sekce 6.7.2. Aplikace sice nemá ambice nalézt využití v průmyslových projektech pracujících s výpočetně náročnými modely, detekce a zrychlení kritických úseků je však zajímavou akademickou výzvou. Z hlediska čitelnosti sítě by jistě byla přínosem též implementace hierarchických konceptů deklarovaných v kapitole 2.4 či variabilita zobrazování grafických elementů (např. větší volnost tvaru šipek, barvení elementů atp).

Nástroj `CPNSimulator` má perspektivu jak z hlediska použití, tak i možných rozšíření. Při budoucím rozvoji aplikace je však třeba dodržovat původní koncepty, které ji odlišují od konkurenčního `CPNTools` – jednoduchost, srozumitelnost a intuitivnost interakce s uživatelským rozhraním.

Literatura

[1] Demel, J.: *Grafy a jejich aplikace*. Academia, 2002, ISBN 80-200-0990-6.

[2] Janoušek, V.: *Modelování objektů Petriho sítěmi*. Dizertační práce, 1998. URL http://www.fit.vutbr.cz/research/view_pub.php?id=6001

[3] Jensen, K.: *Coloured Petri Nets: Basic Concepts*. Springer-Verlag, 1992, ISBN 0-387-55597-8.

[4] Jensen, K.; Kristensen, L. M.: *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer, 2009, ISBN 978-3-642-00283-0.

[5] Kolektiv autorů: CPN Tools for Editing, Simulating and Analysing Coloured Petri Nets. Technická zpráva, Department of Computer Science, University of Aarhus, 2007.

[6] Kršek, P.: Základy počítačové grafiky: studijní opora. VUT Brno, 2003.

[7] Levine, J.: *flex & bison*. O'Reilly, 2009, ISBN 978-0-596-15597-1.

[8] Meduna, A.; Lukáš, R.: Výstavba překladačů: studijní opora. VUT Brno, 2006.

[9] Mezinárodní standard: ISO/IEC 14882:1998 Programming languages – C++. http://www.iso.org/iso/catalogue_detail.htm?csnumber=25845.

[10] Mezinárodní standard: ISO/IEC 15909-1:2004 Systems and software engineering – High-level Petri nets – Part 1: Concepts, definitions and graphical notation. http://www.iso.org/iso/catalogue_detail.htm?csnumber=38225.

[11] Mezinárodní standard: ISO/IEC 15909-1:2004/Amd 1:2010 Symmetric Nets. http://www.iso.org/iso/catalogue_detail.htm?csnumber=52070.

[12] Mezinárodní standard: ISO/IEC 15909-2:2011 Systems and software engineering – High-level Petri nets – Part 2: Transfer format. http://www.iso.org/iso/catalogue_detail.htm?csnumber=43538.

[13] Milner, R.; Harper, R.; MacQueen, D.: *The definition of Standard ML: revised*. The MIT Press, 1997, ISBN 0-262-63181-4.

[14] Paxson, V.; Estes, W.; Millaway, J.: The flex Manual. http://flex.sourceforge.net/manual/index.html, [cit. 2013-05-19].

[15] Petri, C. A.: *Kommunikation mit Automaten*. Dizertační práce, Fakultät für Mathematik und Physik der Technischen Hochschule Darmstadt, červen 1962.

[16] WWW stránky: CPN Tools Homepage. http://cpntools.org/, [cit. 2013-05-19].

[17] WWW stránky: Graphics View Framework. http://qt-project.org/doc/qt-4.8/graphicsview.html, [cit. 2013-05-19].

[18] WWW stránky: Qt Documentation. http://qt-project.org/doc/, [cit. 2013-05-19].

[19] WWW stránky: The Qt Help Framework. http://qt-project.org/doc/qt-4.8/qthelp-framework.html, [cit. 2013-05-19].

[20] WWW stránky: Qt XML Patterns. http://qt-project.org/doc/qt-5.0/qtxmlpatterns/qtxmlpatterns-index.html, [cit. 2013-05-19].

[21] WWW stránky: QtXml module. http://qt-project.org/doc/qt-4.8/qtxml.html, [cit. 2013-05-19].

[22] Češka, M.: *Petriho sítě*. CERM, 1994, ISBN 80-85867-35-4.

[23] Češka, M.; Marek, V.: Petriho sítě: Studijní opora. VUT Brno, 2009.

Příloha A

Obsah CD

Přiložené CD obsahuje následující soubory a adresáře:

- `tex` – zdrojové `.tex` soubory a obrázky k tomuto dokumentu.
- `src` – zdrojové kódy aplikace včetně souborů nápovědy, XSD schémata a .qrc souboru s ikonami. Přeložení na spustitelnou aplikaci lze provést příkazem `qmake && make`.
- `examples` – sada ukázkových sítí pro program `CPNSimulator`. Slouží k demonstraci funkcí aplikace a základních modelovacích schopností barvených Petriho sítí.
- `projekt.pdf` – tento dokument.

Příloha B

Instalace a manuál

Aplikaci lze ze zdrojového adresáře přeložit pomocí `qmake && make`. Pro úspěch příkazu je nutné, aby na počítači byly nainstalovány knihovny Qt v aktuální verzi[1], dále nástroje `flex`[2] a `bison`[3]. Po překladu je k dispozici spustitelný binární soubor `CPNSimulator`. Zdrojové soubory jsou umístěny na CD, případně je možno nejnovější verzi stáhnout z repozitáře[4].

Typický postup instalace může vypadat takto:

```
#stahnuti aplikace z repozitare
git clone https://github.com/LordNavro/CPNSimulator
cd CPNSimulator

#prelozeni aplikace a vytvoreni spustitelneho binarniho souboru
qmake && make

#spusteni aplikace
./CPNSimulator
```

Manuál je součástí aplikace a je zprostředkováván programem `assistant`. Spustit nápovědu lze přes roletové menu (Help/Help).

Jednotlivé HTML dokumenty s obsahem nápovědy můžeme také nalézt v adresáři `src/doc` a spustit ve webovém prohlížeči.

[1] http://qt-project.org/downloads
[2] http://flex.sourceforge.net/
[3] http://www.gnu.org/software/bison/
[4] https://github.com/LordNavro/CPNSimulator