VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

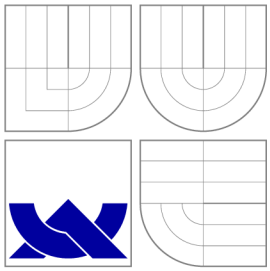
VIRTUÁLNÍ STROJ PRO OBJEKTOVĚ ORIENTOVANÉ PETRIHO SÍTĚ

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

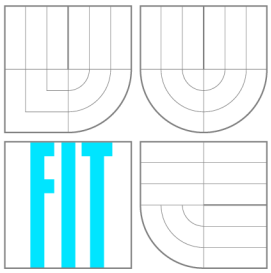
AUTOR PRÁCE
AUTHOR

Bc. ĽUBOŠ SITARČÍK

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

VIRTUÁLNÍ STROJ PRO OBJEKTOVĚ ORIENTOVANÉ PETRIHO SÍTĚ

OBJECT ORIENTED PETRI NETS VIRTUAL MACHINE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. ĽUBOŠ SITARČÍK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. RADEK KOČÍ, Ph.D.

BRNO 2015

Abstrakt

Diplomová práce formálne definuje Objektovo orientované Petriho síte a představuje pojem virtuální stroj. Pak práce představuje koncept virtuálního stroje pro Objektovo orientované Petriho síte. Nakonec je v práci popsána implementace virtuálního stroje pro OOPN.

Abstract

This diploma thesis formally defines the Object Oriented Petri Nets and presents term a virtual machine. Then it introduces the concept of Object Oriented Petri Nets Virtual Machine. Finally, project describes a procedure for implementation of the OOPN Virtual Machine.

Klíčová slova

Petriho síte, Objektovo orientované Petriho síte, Virtuální stroj, PNTalk.

Keywords

Petri Nets, Object Oriented Petri Nets, Virtual Machine, PNTalk.

Citace

Ľuboš Sitarčík: Virtuální stroj pro Objektově orientované Petriho sítě, diplomová práce, Brno, FIT VUT v Brně, 2015

Virtuální stroj pro Objektově orientované Petriho sítě

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. R. Kočího, Ph.D.

.....
Ľuboš Sitarčík
25. mája 2015

Obsah

Kapitola 1

Úvod

Tento dokument slúži ako technická správa k diplomovej práci *Virtuální stroj pro Objektově orientované Petriho sítě* v akademickom roku 2014/2015 na Fakulte informačných technológií VUT v Brne.

Dokument predpokladá, že čitateľ je oboznámený s problematikou Petriho sietí a má aspoň základné znalosti o tomto populárnom formalizme slúžiacom k modelovaniu diskrétnych systémov. V prípade, že čitateľ nie je s danou problematikou oboznámený, tak sa odporúča študijná opora k predmetu PES, viz. [5], pre úplné pochopenie *C-E systémov*, *P/T Petriho sietí* a k získaniu základných znalostí o farbených Petriho sieťach.

Ďalej dizertačnú prácu [3] od Doc. Ing. Vladimíra Janouška, Ph.D., ktorá bola vybraná ako hlavný zdroj informácií pre túto diplomovú prácu, pre získanie znalostí o *Vysokoúrovňových Petriho sieťach*, ktoré môžu poslúžiť k lepšiemu pochopeniu *Objektovo orientovaných Petriho sietí* (OOPN).

Dokument najprv v kapitole 2 predstaví veľmi jednoduché základy Petriho sietí a predstaví ich formálny popis.

Potom v kapitole 3 budú formálne predstavené Objektovo orientované Petriho siete a budú definované jednotlivé úrovne OOPN. Kapitola predstaví výrazy použité v OOPN, predstaví ich štruktúru zavedením pojmov ako napríklad *sieť objektu*, *sieť metódy*, *trieda*. V rámci tejto kapitoly bude predstavený *systém objektov* ako množina *inštancie siete objektu* a *inštancií sietí metód*. Nakoniec kapitola ukáže dynamiku OOPN, pri čom bude zavedený pojem *prevedenie prechodu* a budú predstavené 4 druhy možných prevedení prechodu nazývané *udalosti*.

Neskôr v kapitole 4 dokument popíše pojem *Virtuálny stroj*. Kapitola uvedie čitateľa do problematiky virtuálnych strojov, predstaví rôzne typy virtuálnych strojov a zavedie pojem *garbage collection* alebo zber smetí. Pri tom predstaví niekoľko algoritmov, ktoré môže *garbage collector* využívať.

Ďalej v kapitole 5 dokument popíše vlastnú prácu. Bude predstavený implementovaný virtuálny stroj pre Objektovo orientované Petriho siete a jeho jednotlivé časti, a to:

- Statická reprezentácia OOPN.
- Systém objektov.
- Dynamika, hľadanie naviazaní premenných a prevádzanie udalostí.
- Rozhranie virtuálneho stroja.

Samotný virtuálny stroj vychádza z formálnych definícií popísaných v kapitole 3 a snaží sa implementovať jednotlivé časti v súlade s týmito definíciami a pri tom si kladie za cieľ, aby bol najmä rozšíriteľný a jeho implementácia prehľadná, aby mohol byť virtuálny stroj v budúcnosti optimalizovaný a prispôsobený systému, ktorý by mohol nad týmto virtuálnym strojom pracovať.

Kapitola 2

Petriho siete

Petriho siete predstavujú populárny formalizmus pre modelovanie diskrétnych systémov, ktorý spojuje výhody zrozumiteľného grafického zápisu a možnosti simulácie s dobrou formálnou analyzovateľnosťou. Samotný model je popísaný miestami (*places*), ktoré obsahujú stavovú informáciu vo forme značiek (*tokens*), ďalej prechody (*transitions*), ktoré vyjadrujú možné zmeny stavu a hranami (*arcs*), ktoré prepájajú miesta a prechody.

Existuje veľké množstvo typov Petriho sietí ako napríklad C-E siete (*Condition-Event nets*), ‚P/T siete a ich špeciálne podtriedy, či vysokoúrovňové siete (*High-Level Petri nets*) alebo **objektovo orientované Petriho siete**.

Existencia rôznych variant Petriho sietí súvisí so snahou zvyšovať modelovacie schopnosti a úroveň popisu modelu a pritom zachovať konceptuálnu jednoduchosť, ktorá je pre Petriho siete príznačná. Obecne platí, že vyššie typy Petriho sietí sú horšie analyzovateľné, ale poskytujú vyšší komfort modelovania.

2.1 Základná P/T Petriho sieť

Pôvodným konceptom teórie Petriho sietí tak, ako ich zaviedol C. A. Petri, sú P/T Petriho siete (*Place/Transition Petri Nets*).

Petriho sieť môžme chápať ako špeciálnu triedu bipartitných grafov a pri ich vytváraní používame nasledujúce elementy:

- **Miesta** (*places*) slúžia k vyjadreniu stavu modelovaného systému.
- **Prechody** (*transitions*) popisujú zmeny v systéme.
- **Hrany** (*arcs*) sú povinne orientované a spojujú vždy v niektorom smere miesto a prechod. Nesmú prepojovať dve miesta alebo dva prechody. Jedno miesto s jedným prechodom smie v každom smere prepojovať najviac jedna hrana.
- **Inhibičné hrany** (*inhibitory arcs*) sú špeciálne hrany, ktoré sú vždy orientované od miesta k prechodu. Túto prepojovanú dvojicu potom už nie je možné prepojiť inou hranou.
- **Značky** (*tokens*) sú v vzájomne nerozlíšiteľné a svojim umiestnením v miestach siete vyjadrujú aktuálny stav Petriho siete, tzv. značenie siete (*marking*).

Pomocou Petriho sietí môžme nie len zachytiť stav modelovaného systému, ale takisto dynamiku prechodu medzi jednotlivými stavmi. Táto dynamika sa znázorňuje prevádzaním

prechodov siete. Prechod smie by prevedený (je preveditеľný), ak sú splnené nasledujúce podmienky:

- Ak je niektorá zo vstupných hrán prechodu inhibičnou hranou a v mieste asociovanom s touto hranou je aspoň toľko značiek, aká je hodnota hranovej funkcie tejto inhibičnej hrany, tak prechod nemôže byť prevedený.
- V každom vstupnom mieste prechodu, ktoré nie je spojené inhibičnou hranou, je počet značiek prinajmenšom rovný hodnote hranovej funkcie príslušnej hrany.

Ak sú podmienky pre prevedenie prechodu splnené, smie byť prechod prevedený. Samotné prevedenie prechodu spočíva v odobraní určitého počtu značiek zo všetkých vstupných miest prechodu, ktoré nie sú spojené inhibičnou hranou. Tento počet značiek je rovný hodnote vstupnej podmienky prechodu. Potom sa do všetkých výstupných miest pridá počet značiek odpovedajúci výstupnej podmienke prechodu.

2.2 Formálny popis Petriho sietí

Petriho sieť je matematický stroj s presne definovanou syntaxou a sémantikou. V literatúre je možné sa stretnúť s rôznymi variantami definícií Petriho sietí. Štýl definície väčšinou odráža účel, ku ktorému má byť použitá.

Definícia 2.2.1. Petriho sieť je štvorica $N = (P_N, T_N, PI_N, TI_N)$, kde

1. P_N je konečná množina miest.
2. T_N je konečná množina prechodov, $P_N \cap T_N = \emptyset$
3. $PI_N : P_N \longrightarrow \mathbb{N}$ je inicializačná funkcia (*place initialization function*).
4. TI_N je popis prechodov (*transition inscription function*). Je to funkcia, definovaná na T_N taká, že $\forall t \in T_N$:

   $$TI_N(t) = (PRECOND_t^N, POSTCOND_t^N),$$

   kde

   (a) $PRECOND_t^N : P_N \longrightarrow \mathbb{N}$ sú vstupné podmienky prechodu t.
   (b) $POSTCOND_t^N : P_N \longrightarrow \mathbb{N}$ sú výstupné podmienky prechodu t.

Kapitola 3

Objektovo orientované Petriho siete

Definícia objektovo orientovanej Petriho siete (OOPN) ma niekoľko úrovní. Základnú úroveň tvorí definícia *systému mien a primitívnych objektov*. Tento systém poskytuje primitívnu sémantiku výrazom, ktoré sú súčasťou Petriho sietí. Potom je definovaný *systém sietí* a *inštancie sietí*. Siete sú súčasťou tried. OOPN je definovaná *systémom tried* a špecifikáciou *počiatočnej triedy*. Potom je definovaný *objekt* a *systém objektov*. Systém objektov modeluje okamžitý stav systému. Dynamika OOPN je vymedzená *počiatočným systémom objektov* a pravidlami pre generovanie následujúcich systémov objektov prevádzaním prechodov Petriho siete.

3.1 Systém mien a primitívnych objektov

Rozlišujeme primitívne a neprimitívne objekty. Neprimitívny (užívateľom definovaný) objekt je špecifikovaný triedou (je to inštancia triedy) a obsahuje stavovú informáciu, ktorá môže byť počas behu systému menená jednak vnútornou aktivitou objektu, či prevádzaním metód na základe akceptovania príchodzích správ.

Primitívne objekty sú konštanty, ktoré sú dostupné prostredníctvom svojich mien. Ide o celé čísla, čísla s desatinou čiarkou, booleovské hodnoty, symboly (znaky) a reťazce. Keďže ide o konštanty, je ich možné stotožniť s ich menami. Podobne aj triedy majú charakter konštánt.[1]

Ako východzí bod formálnej definície OOPN bude zavedené *univerzum*, ktoré zahrňuje množinu primitívnych objektov, množinu mien neprimitívnych objektov a množinu mien tried. Prvky tejto množiny nazývame *atómy*. Každému atómu je priradený *typ*. Množinu všetkých potencionálnych inštancií nejakého typu nazveme *doména*.

Ďalej bude zavedená množina *selektorov správ* a množina *selektorov špeciálnych správ*. Pre primitívne objekty je definovaná sémantika všetkých správ pomocou funkcií. Pre neprimitívne objekty je takto definovaná len sémantika špeciálnych správ, ktoré operujú len nad menami, avšak nie and stavom neprimitívnych objektov.

Definícia 3.1.1. Systém mien a primitívnych objektov je n-tica:

$$\Pi = (CONST, NAME, CLASS, Type, MSG, Mth_T, MSG_S, MTH_S, V))$$

kde

[1] Trieda je hodnota metatriedy.

1. $CONST$ je množina, ktorej prvky tvoria **primitívne objekty**. $\{0, 1, ...\} \subset CONST$

2. $NAME$ je množina, ktorej prvky tvoria **mená neprimitívnych objektov**.

3. $CLASS$ je konečná množina **mien tried**. $CONST, NAME, CLASS$ sú po dvoch disjunktné. Nech $U_0 = CONST \cup NAME \cup CLASS$. Prvky množiny U_0 sú **atómy**. Univerzum U je definované ako:

   (a) $U_0 \subseteq U$,

   (b) $n \in \mathbb{N} \wedge x_1 \in U \ \wedge ... \wedge x_n \in U \ \Rightarrow (x_1, ..., x_n) \in U$.

4. $Type$ je funkcia definovaná nad U_0 takto:

   - $\forall x \in CONST : Type(x) \notin CLASS$, $Type(x)$ je typ primitívneho objektu,
   - $\forall x \in NAME : Type(x) \in CLASS$,
   - $\forall x \in CLASS : Type(x) = \texttt{class}$.

   $TYPE = \{Type(x) | x \in CONST\}$ je množina primitívnych typov.
   `class` je špeciálna konštanta, $\texttt{class} \notin CLASS \cup TYPE$.
   Funkcia $Dom : TYPE \cup CLASS \cup \{\texttt{class}\} \longrightarrow 2^{U_0}$ je definovaná následovne:
   $Dom(t) = \{x | x \in U_0 \wedge Type(x) = t\}$.

5. MSG je konečná množina **selektorov správ**.
   $MSG \cap U \ = \emptyset, \texttt{new}^{(0)} \in MSG$.
   Ak $msg^{(n)} \in MSG$ je selektor správy, tak n je jeho arita, $n \geq 0$.

6. Mth_T je funkcia, definovaná nad $MSG \times TYPE$ tak, že:
   $Mth_T(msg^{(n)}, t) = f$, kde f je funkcia tvaru $f : Dom(t) \times CONST^n \longrightarrow CONST$.

7. MSG_S je množina **selektorov špeciálnych správ**,
   $MSG_S \cap MSG = \emptyset, MSG_S \cap U \ = \emptyset$.

8. Mth_S je funkcia definovaná nad $MSG_S \times (TYPE \cup CLASS \cup \{\texttt{class}\})$ tak, že:
   $Mth_S(msg^{(n)}, c) = g$, kde g je funkcia tvaru $g : Dom(c) \times U_0^n \longrightarrow U_0$.

9. V je konečná množina **premenných**.
   Symboly `self` a `super` sú **pseudo-premenné**, $\{\texttt{self}, \ \texttt{super}\} \cap (U \ \cup V) = \emptyset$.
   Naviazanie množiny premenných V' je akákoľvek funkcia $b : V' \longrightarrow U, V' \subseteq V$.

3.1.1 Multimnožiny a n-tice

Definujme si multimnožiny a n-tice potrebné pre definíciu sémantiky hranových výrazov.

Definícia 3.1.2. Majme ľubovoľnú neprázdnu množinu E. Multimnožina nad množinou E je funkcia $x : E \longrightarrow \mathbb{N}$. Hodnota $x(e)$ je počet výskytov (koeficient) prvku e v multimnožine x. Multimnožinu zapisujeme ako formálnu sumu

$$\sum_{e \in E} x(e)`e$$

Pre multimnožiny x, y nad E a prirodzené číslo n sú definované:

1. sčítanie:
$$x + y = \sum_{e \in E} (x(e) + y(e))`e$$

2. skalárne násobenie:
$$n`x = \sum_{e \in E} (n.x(e))`e$$

3. porovnanie:
$x \neq y \iff \exists e \in E : x(e) \neq y(e)$
$x \leq y \iff \forall e \in E : x(e) \leq y(e)$

4. odčítanie (pre $x \geq y$):
$$x - y = \sum_{e \in E} (x(e) - y(e))`e$$

5. veľkosť:
$$|x| = \sum_{e \in E} x(e)$$

Definícia 3.1.3. Nech E je ľubovoľná množina. Prvok množiny $E^* = \bigcup_{n \in \mathbb{N}} E^n$ je **n-tica** a zapisuje sa $x = (e_1, ..., e_n)$. () je jedinečný prvok E^0. Akékoľvek usporiadanie $\leq$ nad E definuje reláciu čiastočného usporiadania $\leq$ nad E^* tak, že:

$$(x_1, ..., x_m) \leq (y_1, ..., y_n) \Leftrightarrow m = n \wedge \forall i \in 1, ..., n : x_i \leq y_i$$

Teraz doplníme definíciu funkcie *supp*, ktorá vracia množinu prvkov multimnožiny alebo n-tice. Funkcia pracuje korektne aj pre vnorené n-tice.

Definícia 3.1.4. Nech E je ľubovoľná množina a Π je systém mien a primitívnych objektov, viz. definícia 3.1.1

1. - Pre ľubovoľnú multimnožinu $x : E \longrightarrow \mathbb{N}$ definujeme funkciu $supp_{MS}(x) = \{e \in E | x(e) \neq 0)\}$.
   - Pre ľubovoľnú n-ticu $l = (a_1, ..., a_n), l \in E^n, n \in \mathbb{N}$, definujeme funkciu $supp^*(l) = \bigcup_{i \in \{1,...,n\}} \{a_i\}$.

2. Nad systémom mien a primitívnych objektov Π definujeme funkciu *supp* takto:
   - $\forall x \in U_0$ definujeme $supp(x) = x$.
   - $\forall x U \ - U_0$ definujeme $supp(x) = \bigcup_{e \in supp^{MS}(x)} supp(e)$.

3. Funkciu *supp* rozšírime aj na množiny a multimnožiny prvkov univerza takto:
   - $\forall x \in 2^U$ definujeme $supp(x) = \bigcup_{e \in x} supp(e)$.
   - $\forall x \in U^{MS}$ definujeme $supp(x) = \bigcup_{e \in supp^{MS}(x)} supp(e)$.

3.1.2 Výrazy

Teraz definujeme syntax a tzv. primitívnu sémantiku výrazov, ktoré sa budú používať pre popis Petriho sietí. Ide o *zasielanie správ* a *hranový výraz*. Zasielanie správ sa bude vyskytovať v strážach a akciách prechodov, hranové výrazy na hranách Petriho sietí.

Definícia 3.1.5. Nad systémom mien a primitívnych objektov Π, viz. definícia 3.1.1, sú definované tieto druhy výrazov:

1. **Term** je prvok množiny $TERM(\Pi), TERM(\Pi) = U_0 \cup V \ \cup \{\texttt{self, super}\}$.
2. **Zaslanie správy** je prvok množiny $EXPR(\Pi)$ definovanej následovne:
   - $TERM(\Pi) \subseteq EXPR(\Pi)$.
   - Každý výraz tvaru $e_0.msg^{(m)}(e_1, e_2, ..., e_m)$, kde $\forall i \in 0, ..., m : e_i \in TERM(\Pi)$ a $msg^{(m)} \in (MSG \cup MSG_S)$, je prvkom množiny $EXPR(\Pi)$.
3. Definujeme pomocnú množinu $LISTEXPR(\Pi)$ takto:
   - $TERM(\Pi) \subseteq LISTEXPR(\Pi)$.
   - Každý výraz tvaru $(e_1, e_2, ..., e_m)$, kde $\forall i \in \{0, ..., m\} : e_i \in LISTEXPR(\Pi), m \in \mathbb{N}$, je prvkom množiny $LISTEXPR(\Pi)$.
4. **Hranový výraz** je prvok množiny $ARCEXPR(\Pi)$, definovanej takto:
   - Každý výraz tvaru e_1‘e_2, kde $e_1 \in TERM(\Pi), e_2 \in LISTEXPR(\Pi)$, je prvkom množiny $ARCEXPR(\Pi)$.
   - Každý výraz tvaru $e_1 + e_2$, kde $e_1, e_2 \in ARCEXPR(\Pi)$, je prvkom množiny $ARCEXPR(\Pi)$.

Nech $Var(e) \subseteq V$ je množina všetkých premenných vo výraze e.

Definícia 3.1.6. Majme systém mien a primitívnych objektov Π (def. 3.1.1), definujeme $e\langle b\rangle$ ako výsledok vyhodnotenia výrazu e pri naviazaní $b : V' \longrightarrow U, Var(e) \subseteq V' \subseteq V$, takto:

1. - Ak $e \in (CONST \cup CLASS$, potom $e\langle b\rangle = e$.
   - Ak $e \in V$, potom $e\langle b\rangle = b(e)$.
   - Ak $e \in \{\texttt{self, super}\}$, potom vyhodnotenie výrazu e závisí na kontexte a jeho sémantika bude definovaná neskôr v definícii 3.4.5.
2. Ak je výraz tvaru $e_0.msg(e_1, e_2, ..., e_m)$, kde $\forall i \in \{0, ..., m\} : e_i \in TERM(\Pi)$, potom:
   (a) Ak $msg^{(m)} \in MSG$ a $\forall i \in \{0, ..., m\} : e_i\langle b\rangle \in CONST$, potom $e\langle b\rangle = (Mth_T(msg^{(m)}, Type(e_0\langle b\rangle)))(e_1\langle b\rangle, e_2\langle b\rangle, ..., e_m\langle b\rangle)$.
   (b) Ak $msg^{(m($ $\in MSG_S$ a keď $\forall i \in \{0, ..., m\} : e_i\langle b\rangle \in U_0$, potom $e\langle b\rangle = (Mth_S(msg^{(m)}, Type(e_0\langle b\rangle)))(e_1\langle b\rangle, e_2\langle b\rangle, ..., e_m\langle b\rangle)$.
   (c) Ak $msg^{(m)} \in MSG$ a keď $e_0\langle b\rangle \in (NAME \cup CLASS)$, ide o **neprimitívne zaslanie správy**. Jeho sémantika bude popísaná v rámci definície dynamiky OOPN v sekcii 3.4

(d) V iných prípadoch nie je možné výraz e vyhodnotiť.[2]

3. (a) Ak $e = (e_1, e_2, ..., e_n)$, kde $\forall i \in \{1, ..., n\} : e_i \in LISTEXPR(\Pi)$, potom $e\langle b\rangle = (e_1\langle b\rangle, ..., e_n\langle b\rangle)$.

   (b) ak e je výraz tvaru e_1‘e_2, kde $e_1 \in TERM(\Pi), e_2 \in LISTEXPR(\Pi)$, potom $e\langle b\rangle = (e_1\langle b\rangle$‘$e_2\langle b\rangle)$.

   (c) Ak e je výraz tvaru $e_1 + e_2$, kde $e_1, e_2 \in ARCEXPR(\Pi)$, potom $e\langle b\rangle = e_1\langle b\rangle + e_2\langle b\rangle$.

3.2 Štruktúra OOPN

Neprimitívne objekty sú definované triedami, ktoré sú špecifikované prostredníctvom Petriho sietí. Siete sú zložené z miest a prechodov, prepojených hranami, ktoré vyjadrujú vstupné a výstupné podmienky prechodov. Miesto siete môže obsahovať značky, pri čom značka môže reprezentovať primitívny objekt, meno užívateľom definovaného, neprimitívneho objektu, meno triedy alebo n-ticu z nich zloženú. Hrany, reprezentujúce vstupné a výstupné podmienky prechodov, sú ohodnotené hranovými výrazmi, ktoré po naviazaní premenných reprezentujú multimnožinu prvkov univerza U.

Prechod obsahuje stráž a akciu. Stráž prechodu obsahuje výrazy a stráž prechodu je splnená práve vtedy, keď sú všetky tieto výrazy vyhodnotené ako `true`. Akcia je zaslanie správy s možnosťou priradiť výsledok do premennej. Zaslanie správy v akcii prechodu sa interpretuje za behu podľa typu adresáta a selektoru správy buď ako primitívne zaslanie správy alebo ako vytvorenie nového neprimitívneho objektu, prípadne ako prevedenie operácie objektu s čakaním na výsledok. Presný popis prevádzania prechodu bude popísaný v sekcii 3.4.

Špecifikácia triedy obsahuje sieť objektu, siete metód, synchrónne porty a mapovanie správ na metódy a synchrónne porty. Synchrónne porty sú určené k volaniu zo stráží prechodov. Slúžia k atomickému otestovaniu stavu neprimitívneho objektu a v prípade prevedenia prechodu i k jeho prípadnej zmene.

3.2.1 Siete

Definícia 3.2.1. Sieť objektu je pätica:

$$N = (\Pi, P_N, T_N, PI_N, TI_N),$$

kde

1. Π je systém primitívnych mien a objektov.
2. P_N je konečná množina **miest**.
3. T_N je konečná množina **prechodov**, $P_N \cap T_N = \emptyset$.
4. $PI_N : P_N \longrightarrow ARCEXPR(\Pi)$ je **inicializačná funkcia miest**,

$$\forall p \in P_N : Var(PI_N(p)) = \emptyset.$$

[2] Tento prípad môže nastať pri testovaní prevediteľnosti prechodu.

5. TI_N je popis prechodov. je to funkcia definovaná nad T_N taká, že $\forall t \in T_N$ platí:

$$TI_N(t) = (COND_t^N, PRECOND_t^N, GUARD_t^N, ACTION_t^N, POSTCOND_t^N)$$

kde

(a) $COND_t^N, PRECOND_t^N, POSTCOND_t^N : P_N \longrightarrow ARCEXPR(\Pi)$ sú funkcie **podmienok**, **vstupných podmienok** a **výstupných podmienok**.

(b) $GUARD_t^N = \{G_1, ..., G_m\}$, kde $\forall i \in \{1, 2, ..., m\} : G_i \in EXPR(\Pi)$, je **stráž prechodu**. Definujeme množinu

$$InVar_N(t) = \bigcup_{i=1}^{m} Var(G_i) \cup \bigcup_{p \in P_N} (Var(COND_t^N(p)) \cup Var(PRECOND_t^N(p)))$$

.

(c) $ACTION_t^N$ je **akcia prechodu**, výraz tvaru $y := expr$, kde $y \in V - InVar_N(t)$ a $expr \in EXPR(\Pi), Var(expr) \subseteq InVar_N(t)$. Definujeme množinu $Var_N(t) = InVar(t) \cup y$.

Definícia 3.2.2. Majme sieť objektu N. **Synchrónny port** nad sieťou N je štvorica:

$$R^N = (COND_R^N, PRECOND_R^N, GUARD_R^N, POSTCOND_R^N),$$

kde $COND_R^N, PRECOND_R^N, GUARD_R^N, POSTCOND_R^N$ sú definované rovnako ako v predchádzajúcej definícii 3.2.1. Definujeme $Var(R)$ ako množinu premenných vyskytujúcich sa vo výrazoch $COND_R^N, PRECOND_R^N, GUARD_R^N, POSTCOND_R^N$. Prvky $Var(R)$ sú **parametre** synchrónneho portu.

Synchrónny port je špecifikovaný v stráži prechodu a volá sa predaním správy. Vyhodnotenie jeho preveditelnosti má vplyv na preveditelnosť volajúceho prechodu. Pri prevedení prechodu sa synchrónne prevedie aj synchrónny port, ako keby to bol prechod, pre príslušné naviazanie premenných. Tieto skutočnosti budú definované v sekcii 3.4.

Definícia 3.2.3. Sieť metódy je štruktúra:

$$F = (\Pi, P_F, T_F, PI_F, TI_F, PP_F, RP_F),$$

kde

1. $\Pi, P_F, T_F, PI_F, TI_F$ majú rovnaký význam ako $\Pi, P_N, T_N, PI_N, TI_N$ v definícii siete objektu, viz. def. 3.2.1.
2. $PP_F \subset P_F$ je množina **parametrových miest**.
3. $RP_F \in P_F - PP_F$ je **výstupné miesto (*return place*)**. Toto miesto má prázdne počiatočné značenie.

V ďalšom texte bude sieť objektu aj sieť metódy nazývaná jednotne ako sieť všade tam, kde ich odlišnosti nebudú významné.

Definícia 3.2.4. Systém sietí NS je dvojica

$$NS = (NET, Inst),$$

kde

1. NET je množina sietí splňujúca podmienku $(\bigcup_{N \in NET} P_N) \cap (\bigcup_{N \in NET} T_N) = \emptyset$.
2. $Inst$ je funkcia definovaná nad NET taká, že
   - $\forall n \in NET : Inst(n)$ je množina, ktorej prvky nazveme **mená inštancií** siete n,
   - Pokiaľ $n \neq n'$ tak $Inst(n) \cap Inst(n') = \emptyset$.

   Pre systém sietí NS definujeme množinu:

$$INST_{NS} = \bigcup_{n \in NET} Inst(n)$$

   a funkciu
   $Net : INST_{NS} \longrightarrow NET$ takú, že:
   $\forall id \in INST_{NS} : Net(id) = n \iff id \in Inst(n)$.

Definícia 3.2.5. Majme systém sietí $NS = (NET, Inst)$. NS je **polodisjunktný systém sietí**, keď $\forall N_1, N_2 \in NET : N_1 \neq N_2 \Longrightarrow T_{N_1} \cap T_{N_2} = \emptyset$.

Systém sietí zaručuje, že prechod jednej siete nemôže vystupovať v roli miesta inej siete a naopak. Pripúšťa však, aby sa jeden uzol (miesto alebo prechod) mohol vyskytovať vo viacerých sieťach súčasne. Táto skutočnosť umožňuje definovať dedičnosť Petriho sietí. Polodisjunktný systém sietí umožňuje len zdieľanie miest rôznymi sieťami. Množiny prechodov jednotlivých sietí sú po dvoch disjunktné. Zdieľanie miest bude využité v definícii štruktúry inštancie triedy.

Definícia 3.2.6. Majme systém sietí $(\{N_1, N_2\}, Inst)$. Sieť N_1 **dedí štruktúru** siete N_2, čo sa označuje ako $N_1 \leq N_2$, práve vtedy, keď $P_{N_1} \supseteq P_{N_2} \wedge T_{N_1} \supseteq T_{N_2}$. Sieť N_1 **zdieľa miesta** siete N_2, čo sa označuje ako $N_1 \prec N_2$, práve vtedy, keď $P_{N_1} \supseteq P_{N_2} \wedge T_{N_1} \cap T_{N_2} = \emptyset$.

Dedičnosť sietí je definovaná ako zdieľanie uzlov rôznymi sieťami. Prepojenie uzlov tu nie je významné.

3.2.2 Triedy

Definícia 3.2.7. Systém tried je pätica

$$\Sigma = (\Pi, Spec_\Sigma, Inst_\Sigma, ROOT_\Sigma, Super_\Sigma),$$

kde

1. $\Pi = (CONST_\Sigma, NAME_\Sigma, CLASS\Sigma, Type_\Sigma, MSG_\Sigma, Mth^T_\Sigma, MSG^S_\Sigma, Mth^S_\Sigma, V_\Sigma)$ je systém mien a primitívnych objektov, viz. definícia 3.1.1.
2. $Spec_\Sigma$ je **špecifikácia štruktúry inštancií triedy** definovaná tak, že $\forall c \in CLASS_\Sigma$:

$$Spec_\Sigma(c) = (ONET_c, MNETS_c, SYNC_c, MSG_c, Method_c),$$

   kde

   (a) $ONET_c$ je sieť objektu,
   (b) $MNETS_c$ je konečná množina sietí metód,

(c) $SYNC_c$ je konečná množina synchrónnych portov nad sieťou $ONET_c$,

(d) MSG_c je konečná množina **selektorov správ**, $MSG_c \subseteq MSG_\Sigma$,

(e) $Method_c$ je bijekcia $Method_c : MSG_c \longleftrightarrow (MNETS_c \cup SYNC_c)$, taká, že:

- $\forall m^{(r)} \in MSG_c : Method_c(m^{(r)}) \in MNETS_c \Longrightarrow |PP_{Method_c(m^{(r)})}| = r$,
- $\forall m^{(r)} \in MSG_c : Method_c(m^{(r)}) \in SYNC_c \Longrightarrow |Var(method_c(m^{(r)}))| = r$.

3. $(NET_\Sigma, Inst_\Sigma)$, kde $NET_\Sigma = \bigcup_{c \in class_\Sigma}(\{ONET_c\} \cup MNETS_c)$, je systém sietí taký, že $\forall c \in CLASS_\Sigma$ sú súčasne splnené tieto podmienky:

   - $(\{ONET_c\} \cup MNETS_c, Inst_{\Sigma|\{ONET_c\} \cup MNETS_c})$ je polodisjunktný systém sietí, pre ktorí platí:

     $$\forall n \in MNETS_c : n \prec ONET_c \wedge PP_n \cap P_{ONET_c} = \emptyset \wedge RP_n \notin P_{ONET_c},$$

   - $Inst_\Sigma(ONET_c) = \bigcup_{c' \in H(c)} Dom(c')$[3], kde

     $$H(c) = \{c' | c' \in CLASS_\Sigma \wedge ONET_{c'} = ONET_c\},$$

   - $\forall n \in MNETS_c : Inst_\Sigma(n) \cap NAME_\Sigma = \emptyset$.

4. $ROOT_\Sigma \in CLASS_\Sigma$ je **koreň hierarchie dedičnosti tried**, pre ktorý platí:

   $$\forall c \in CLASS_\Sigma : ONET_c \leq ONET_{ROOT_\Sigma} \wedge MSG_c \supseteq MSG_{ROOT_\Sigma},$$

5. $Super_\Sigma - \{ROOT_\Sigma\} \longrightarrow CLASS_\Sigma$ je funkcia spĺňujúca súčasne tieto podmienky:

   - $\forall c \in CLASS_\Sigma - \{ROOT_\Sigma\} : ONET_c \leq ONET_{Super_\Sigma(c)} \wedge MSG_c \supseteq MSG_{Super_\Sigma(c)}$,
   - $\forall c_1, c_2 \in CLASS_\Sigma : MNETS_{c_1} \cap MNETS_{c_2} \neq \emptyset \vee SYNC_{c_1} \cap SYNC_{c_2} \neq \emptyset \vee ONET_{c_1} = ONET_{c_2} \Longrightarrow \exists c \in CLASS_\Sigma : c \in Super^*(c_1) \wedge c \in Super^*(c_2)$,
   - $\forall c_1, c_2 \in CLASS_\Sigma, \forall N_1, N_2 \in (MNETS_{c_1} \cup SYNC_{c_1}) :$ $N_1 = N_2 \Longrightarrow Method_{c_1}^{-1}(N_1) = Method_{c_2}^{-1}(N_2)$,
   - $\forall c_1, c_2, c_3 \in CLASS_\Sigma : c_1 \in Super^*(c_2) \wedge c2 \in Super^*(c_3) \Longrightarrow Method_{c_1} \cap Method_{c_3} \subseteq Method_{c_2}$,
   - $\forall c_1, c_2, c_3 \in CLASS_\Sigma : c_1 \in Super^*(c_2) \wedge c_2 \in Super^*(c_3) \wedge$ $ONET_{c_1} = ONET_{c_3} \Longrightarrow ONET_{c_1} = ONET_{c_2}$.

Definujeme množiny:

- $T_\Sigma = \bigcup_{n \in NET_\Sigma} T_n$,
- $P_\Sigma = \bigcup_{n \in NET_\Sigma} P_n$,
- $INST_\Sigma = \bigcup_{n \in NET_\Sigma} Inst_\Sigma(n)$,
- $SYNC_\Sigma = \bigcup_{c \in CLASS_\Sigma} SYNC_c$,

[3] $Dom(c)$ špecifikuje množinu mien potencionálnych inštancií triedy s menom c a špecifikáciou $Spec(c)$. Je to podmnožina množiny inštancií siete objektu $ONET_c$, pretože táto sieť sa môže vyskytovať v špecifikáciách viacerých tried a pri tom je potreba zaistiť, aby bolo možné objekt identifikovať rovnakým menom, ako má inštancia siete $ONET_c$.

- $MNET_\Sigma = \bigcup_{c \in CLASS_\Sigma} MNETS_c$,
- $ONET_\Sigma = \bigcup_{c \in CLASS_\Sigma} ONET_c$.

Teorém 3.2.1. Ak je Σ systém tried, potom:

$$\forall c_1, c_2 \in CLASS_\Sigma : c_1 \neq c_2 \Longrightarrow Dom(c_1) \cap Dom(c_2) = \emptyset.$$

Dôkaz. Dôkaz plynie priamo z bodu 4 v definícii 3.1.1. Funkcia $Type$ vytvára na množine U_0 rozklad.

Definícia 3.2.8. Majme systém tried Σ. Trieda $c_1 \in CLASS_\Sigma$ **dedí** od triedy $c_2 \in Class_\Sigma$, čo sa zapisuje ako $c_1 \leq c_2$, práve vtedy, keď $c_2 \in Super^*(c_1)$.

Dedičnosť umožňuje vytvárať triedy inkrementálne tak, že sa v definícii triedy c iba doplnia odlišnosti od triedy $Super_\Sigma(c)$. Funkcia $Super_\Sigma$ vracia meno najbližšej nadtriedy v hierarchii dedičnosti.

Ďalej definujeme funkciu Def_Σ, ktorá vracia meno triedy, v ktorej bola sieť alebo synchrónny port definovaný(á). Ostatné triedy, v ktorých sa daný synchrónny port vyskytuje, od tejto triedy dedí.

Definícia 3.2.9. Pre systém tried Σ definujeme funkciu:

$$Def_\Sigma : NET_\Sigma \cup SYNC_\Sigma \longrightarrow CLASS_\Sigma$$

tak, že $Def_\Sigma(n) = c$ práve vtedy, keď sú súčasne splnené tieto podmienky:

- $n \in \{NET_c\} \cup MNETS_c \cup SYNC_c$,
- $\forall c' \in CLASS_\Sigma : c \leq c' \wedge n \in (\{ONET_{c'}\} \cup MNETS_{c'} \cup SYNC_{c'}) \Longrightarrow c' = c$.

Teraz môžme definovať štruktúru OOPN.

Definícia 3.2.10. Objektovo orientovaná Petriho sieť je trojica:

$$OOPN = (\Sigma, c_0, oid_0),$$

kde Σ je systém tried, $c_0 \in CLASS_\Sigma$ je **počiatočná trieda** a $oid_0 \in Dom(c_0)$ je **meno počiatočného objektu**.

3.3 Systém objektov

OOPN modeluje dynamický systém. Aby bolo možné skúmať dynamiku OOPN, je potreba zaviesť inštancie prvkov OOPN. A to hlavne inštancie sietí, z ktorých sa skladajú inštancie tried (objekty). Inštanciu chápeme ako pomenovaný okamžitý stav príslušnej entity. Systém, resp. množina, objektov určuje stav systému modelovaného OOPN.

3.3.1 Inštancie sietí

Objekt (v danom stave) je množina inštancií sietí splňujúcich určité podmienky. Jedna z týchto inštancií je inštancia siete objektu, ostatné (ak vôbec nejaké) sú inštancie sietí práve rozpracovaných metód. Meno inštancie siete objektu je totožné s menom objektu (viz def. 3.2.7, bod 3). Inštancia siete je pomenovaný stav siete. Stav siete je určený jej značením – rozmiestnením značiek v miestach siete a stavom prechodov. Značka predstavuje prvok univerza.

Definícia 3.3.1. Majme systém tried Σ, triedu $c \in CLASS_\Sigma$, jej špecifikáciu $Spec_\Sigma(c)$, sieť objektu $ONET_c$ a sieť metódy $N \in MNETS_c$.

1. **Značenie miest siete objektu** $ONET_c$ je funkcia $PM : P_{ONET_c} \longrightarrow U^{MS}$.
2. **Značenie miest siete metódy** N je funkcia $M : P_N - P_{ONET_c} \longrightarrow U^{MS}$.

Prechody v OOPN sa môžu vykonávať neatomicky (narozdiel od tradičných Petriho sietí) a niesť stavovú informáciu, ktoré neukončené inštancie sietí (volanie metód) boli prechodom vytvorené. Stav prechodu je teda tvorený množinou invokovaných sietí.

Definícia 3.3.2. Majme systém sietí $NS = (NET, Inst)$ a siete $N_1, N_2 \in NET$.

1. **Invokácia siete** N_2 prechodom $t \in T_{N_1}$ je dvojica (id, b), kde $id \in Inst(N_2)$ je meno inštancie siete N_2 a b je naviazanie premenných $InVar_{N_1}(t)$. Pre každú invokáciu $i = (id, b)$ definujeme $Id(i) = id$. Túto funkciu rozšírime pre množiny invokácií: $Id(\{i_1, i_2, ..., id_n\}) = \{id_1, id_2, ..., id_n\}$.
   Množinu všetkých invokácií označíme INV.
2. Akúkoľvek funkciu $TM : T_{N_1} \longrightarrow INV$ nazveme **značenie prechodov** siete N_1.
3. Nech PM je značenie miest siete N_1. Funkciu $M = PM \cup TM$ nazveme **značenie siete** N_1.

Definícia 3.3.3. Majme systém sietí $NS = (NET, Inst)$

1. **Inštancia siete** $N \in NET$ je dvojica (id, m), kde $id \in Inst(N)$ je meno inštancie a m je značenie siete N.

   Pre každú inštanciu $i = (id, m)$ siete $N \in NET$ definujeme $Id(i) = id$. Túto funkciu rozšírime taktiež pre množinu inštancií sietí: $Id(\{i_1, ..., i_n\}) = \{id_1, ..., id_n\}$.
2. **Systém inštancií sietí** je množina inštancií sietí $Q = \{(id_1,), ..., (id_n, m_n)\}$ spĺňajúca tieto podmienky:
   (a) $\forall i \in \{1, ..., n\} : (id_i, m_i)$ je inštancia siete $Net(id_i) \in NET$,
   (b) $\forall i, j \in \{1, ..., n\} : i \neq j \Longrightarrow id_i \neq id_j$.
3. Pre systém inštancií sietí Q definujeme:
$$Marking_Q(id) = m \Longleftrightarrow (id, m) \in Q.$$
4. Definujeme funkcie $ref^P, ref^T : Q \longrightarrow 2^{INST_{NS}}$ takto:

- Nech pre každú množinu dvojíc $X = (a_1, b_1), ..., (a_n, b_n)$ je definované $Im(X) = \bigcup_{j \in \{1,...,n\}} \{b_j\}$. Potom pre $\forall (id, m)$ platí:

$$ref^P((id, m)) = \left(\bigcup_{p \in P_{Net(id)}} supp(m(p)) \right) \cap INST_{NS} \wedge$$

$$ref^T((id, m)) = \left(\bigcup_{t \in T_{Net(id)}} (Id(m(t)) \cup \bigcup_{b \in Im(m(t))} supp(Im(b))) \right) \cap INST_{NS}$$

5. **Uzatvorený systém inštancií sietí** je systém inštancií sietí Q, pre ktorý platí:

$$\forall i \in Q : (ref^P(i) \cup ref^T(i)) \subseteq Id(Q).$$

Funkcie ref^P a ref^T vracajú množiny mien inštancií sietí, ktoré sú referencované z miest, resp. prechodov danej inštancie siete. Uzatvorený systém inštancií sietí zaručuje platnosť všetkých referencií, tj. skutočnosť, že značenie miest a prechodov referencujú iba inštancie systému sietí.

3.3.2 Objekty

Definícia 3.3.4. Majme systém tried Σ.

1. **Inštancia (objekt) triedy** $c \in CLASS_\Sigma$ je systém inštancií o splňujúci súčasne tieto podmienky:
   (a) $\forall (id, m) \in o \ : Net(id) \in ONET_c \cup \bigcup_{c' \in Super^*(c)} MNETS_{c'}$,
   (b) $\forall i, j \in \{1, ..., n\} : i \neq j \Longrightarrow Id(o_i) \cap Id(o_j) = \emptyset$,
   (c) Nech $FLAT(S) = \bigcup_{o \in S} o$. $FLAT(S)$ je uzatvorený systém inštancií sietí.
2. Pre systém objektov S definujeme funkciu $Oid_S(id)$ tak, že $Oid(id) = oid$ práve vtedy, keď existujú značenia m_1, m_2 a objekt $o \in S$ také, že $(oid, m_1) \in o$ a $(id, m_2) \in o$ a $oid \in NAME_\Sigma$.
3. Pre systém objektov S definujeme funkciu $Marking_S = Marking_{FLATS(S)}$.
4. Pre systém objektov S definujeme funkciu $CMarking_S$ takto:
   - $\forall id \in INST_\Sigma : id = Oid_S(id) \Longrightarrow CMarking_S(id) = Marking_S(id)$,
   - $\forall id \in INST_\Sigma : id \neq Oid_S(id) \Longrightarrow$
     $CMarking(id) = Marking_S(id) \cup Marking_S(Oid_S(id))$.
5. Pre systém objektov S definujeme bijekciu $Object_S : Id(FLAT(S)) \cap NAME_\Sigma \longleftrightarrow S$ takto:
$$Object_S(oid) = o \iff \exists (oid, m) \in o$$
6. Množinu všetkých systémov objektov nad Σ označíme $SO(\Sigma)$.
7. Na množine $FLAT(S)$ definujeme **reláciu viditeľnosti** V_S takto:
$$\forall i_1, i_2 \in FLAT(S) : (i_1, i_2) \in V_S \iff Id(i_2) \in (ref^P(i_1) \cup ref^T(i_1))$$

Definícia 3.3.5. Počiatočný systém objektov pre $OOPN = (\Sigma, c_0, oid_0)$ je systém objektov $S_0 = \{o_0\}$, kde $o_0 = \{(oid_0, PI_{ONT_{c_0}}\langle\emptyset\rangle)\}$. Objekt $o0$ nazveme **prvotný objekt**.

Definícia 3.3.6. Majme $OOPN = (\Sigma, c_0, oid_0)$. **Garbage-collector** je funkcia:

$$GC : SO(\Sigma) \longrightarrow SO(\Sigma),$$

definovaná $\forall s \in SO(\Sigma)$ takto:

1. Nech $i_0 \in FLAT(S)$ je inštancia siete počiatočného objektu, nech $Id(i_0) = oid_0$.
2. $\forall o \in S : o \in S \wedge o \cap (V_S^*(i_0))) \neq \emptyset \iff (o \cap (V_S^*(i_0))) \in GC(S)$.

Funkcia GC reprezentuje garbage-collector, odstraňujúci tie inštancie sietí, ktoré nie sú (ani nepriamo) referencované z počiatočného objektu. Táto funkcia sa uplatní pri každom prevedení prechodu v OOPN, ako bude ukázané v ďalšej sekcii.

3.4 Dynamika OOPN

OOPN definuje počiatočný systém objektov, ktorý sa dynamicky vyvíja prevádzaním prechodov Petriho sietí. Hovoríme o tzv. evolúcií systému objektov. Prevádzanie prechodov je polymorfné, prechod sa môže previesť niekoľkými rôznymi spôsobmi na základe naviazania premenných. Rozlišujeme 4 druhy prevedenia prechodu a to:

- A-prevedenie
- N-prevedenie
- F-prevedenie
- J-prevedenie

3.4.1 Kontext

Výrazy, definované v sekcii 3.1.2 sa vyhodnocujú pri zisťovaní preveditelnosti, resp. pri prevádzaní prechodov vo vnútri inštancií sietí. Vyhodnotenie neprimitívneho zaslania správy a synchrónna komunikácia závisí na okamžitom stave systému. Tieto výrazy sa vyhodnocujú pri zisťovaní preveditelnosti a pri prevádzaní prechodov. Aby bolo možné tieto výrazy vyhodnotiť, je potrebné definovať *kontext* prevádzania prechodu.

Definícia 3.4.1. Majme systém tried Σ. **Kontext** je dvojica (S, id), kde $S \in SO(\Sigma)$ a $id \in Id(FLAT(S))$. V kontexte (S, id) pre ľubovoľné naviazanie b platí: $\texttt{self}\langle b\rangle = \texttt{super}\langle b\rangle = Oid_S(id)$.

3.4.2 Vyhodnocovanie stráži a synchrónna komunikácia

Stráž prechodu t v sieti N môže podľa definície 3.2.1 obsahovať množinu výrazov, z ktorých každý je zaslanie správy.Ak je možné pre nejaké naviazanie premenných $InVar_N(t)$ vyhodnotiť výrazy v stráži prechodu ako `true` a sú splnené ďalšie podmienky preveditelnosti prechodu, je dovolené prechod previesť. Inak hovoríme, že prechod je v danom stave nepreveditelný.

Ďalšie podmienky prevediteľnosti prechodu súvisia s prípadnou synchrónnou komunikáciou. Definujeme vyhodnotenie stráže prechodu a synchrónnu komunikáciu. Vyhodnotenie stráže prechodu prebieha následovne: Tá časť stráže stráže prechodu, ktorú je možné vyhodnotiť ako primitívne zaslanie správy, je takto vyhodnotená. Ďalej na základe neprimitívneho zasielania správ sa v stráži prechodu zistia všetky synchróne porty a objekty, ktoré sa zúčastňujú synchrónnej komunikácie. Vstupné podmienky všetkých týchto synchrónnych portov vrátane vstupných podmienok volajúceho prechodu sa sčítajú, ako multimnožiny, a otestuje sa ich splniteľnosť. Ak je každé primitívne zaslanie správy v strážach volajúceho prechodu aj vo volaných synchrónnych portoch vyhodnotené ako `true` a všetky synchrónne porty a volajúci prechod sú súčasne prevediteľné, tak je prechod *s-prevediteľný.*

Nasledujúca definícia zavádza pomocnú funkciu $Class$.

Definícia 3.4.2. Majme systém tried Σ. Pre ľubovoľnú sieť alebo synchrónny port $x \in NET_\Sigma \cup SYN_\Sigma$, literál e a naviazanie premenných $b : V' \longrightarrow U, Var(e) \subseteq V'$, definujeme $Class(x, e, b)$ takto: Ak je $e =$ `super`, potom $Class(x, e, b) = Super_\Sigma(Def_\Sigma(x))$, inak $Class(x, e, b) = Type(e\langle b\rangle)$.

Volanie funkcie $Class(x, e_0, b)$ sa bude používať pre zistenie triedy a adresáta správy, špecifikovanej výrazom $e_0.msg(e_1, ..., e_n)$ pri naviazaní b, pri čom uvedený výraz sa nachádza buď v stráži synchrónneho portu x alebo je súčasťou stráže alebo akcie prechodu x.

Nasledujúca definícia zavedie pomocnú funkciu $SATISFIED$ pre synchrónne porty. Táto funkcia bude využitá v definícii, ktorá zavádza pojem s-prevediteľnosti prechodu.

Definícia 3.4.3. majme systém tried Σ, kontext $(S, oid), oid \in NAME_\sigma$, synchrónny port $R \in SYNC_{Type(oid))}$ a naviazanie premenných $b : V' \longrightarrow U, Var(R) \subseteq V'$. Nech $GUARD_R^{Net(oid)} = \{G_1, ..., G_n\}$.

Ak je G_i, kde $i \in \{1, 2, ..., n\}$, pri naviazaní b vyhodnotiteľné primitívne zaslanie správy (podľa def. 3.1.6), pri čom $\forall i \in \{1, 2, ..., n\} : G_i\langle b\rangle =$ `true`, potom

$$SATISFIED(S, oid, R, B) = \texttt{true},$$

inak

$$SATISFIED(S, oid, R, b) = \texttt{false}$$

.

Definícia 3.4.4. Majme systém tried Σ, kontext (S, id), prechod $t \in Net(id)$ a naviazanie premenných b. Zavedieme tieto pomocné pojmy:

1. Nech $GUARD_t^{net(id)} = \{G_1, ..., G_m\}$, kde každý výraz $G_i, i \in \{1, 2, ..., m\}$, má tvar $e_0^i.msg^i(e_1^i, ..., e_m^i)$, pri čom bez straty na obecnosti predpokladáme, že $\exists k \in \{0, ..., m\}$ také, že $\forall j \in \{1, ..., k\}$ platí, že G_j je pri naviazaní b neprimitívne zaslanie správy. Súčasne $\forall j \in \{k + 1, ..., m\}$ platí, že G_j je pri naviazaní b primitívne zaslanie správy.

2. Označme $oid_i = e_0^i\langle b\rangle$ a $R_i = Method_{Class(net(id),e_0^i,b)}(msg^i), \forall i \in \{1, ..., k\}$.

3. Aby sme mohli ľahko definovať predávanie parametrov synchrónnym portom, predpokladajme, že pre každé volanie synchrónneho portu R_i zo stráže G_i, teda pre každú dvojicu (G_i, R_i), implicitne existuje mapovanie premenných tohoto synchrónneho portu na premenné, použité pri jeho volaní, $r_i : Var(R_i) \longrightarrow V_{(R_i,G_i)}$. Pri tom množina premenných $V_{(R_i,G_i)}$ rešpektuje predanie parametrov. Ak je parametrom e_j^i volanie

$e_0^i.msg^i(e_1^i, e_2^i, ..., e_n^i)$ synchrónneho portu R_i premenná z $Var(t)$, je táto premenná prvkom $V_{(R_i,G_i)}$. Ak je týmto parametrom konštanta $c \in CONST \cup CLASS$, príslušným prvok $Var_{(R_i,G_i)}$ je unikátna premenná v_c, reprezentujúca túto konštantu, ktorá je v b vždy na túto konštantu naviazaná, tj. $b(v_c) = c$.

4. Nech $IOIDS$ označuje množinu mien objektov, zúčastňujúcich sa synchrónnej komunikácie:
$$IOIDS = \bigcup_{j \in \{1,...,k\}} \{oid_j\}.$$

5. Nech $IPORTS$ je funkcia, ktorá každému menu objektu priraďuje množinu synchrónnych portov (spolu s premenovaním prechodov) zúčastnených v synchrónnej komunikácii, patriacich objektu uvedeného mena.
$$IPORT(oid) = \{(R_j, r_j) | oid_j = oid\}.$$

Definícia 3.4.5. Prechod t v kontexte (S, id) pre naviazanie b je **s-preveditelný**, ak sú súčasne splnené nasledujúce podmienky:

1. Každému neprimitívnemu zaslaniu správy v stráži prechodu t odpovedá volanie synchrónneho portu:
$$\forall j \in \{1, ..., k\} :$$
$$msg^j \in MSG_{Class(net(id),e_0^j,b)} \wedge Method_{Class(Net(id),e_0^j,b)}(msg^j) \in SYNC_{Class(Net(id),e_0^j,b)},$$

2. Všetky volané synchrónne porty majú splnenú stráž:
$$\forall j \in \{1, ..., k\} : SATISFIED(S, oid_j, R_j, r_j \circ b) = \texttt{true},$$

3. Každé primitívne zaslanie správy v stráži prechodu t sa vyhodnotí ako `true`:
$$\forall k \ \in \{k+1, ..., m\} : G_j\langle b \rangle = \texttt{true}$$
,

4. Všetky volané synchrónne porty sú súčasne a bezkonfliktne preveditelné:
$$\forall oid \in IOIDS, \forall p \in P_{Net(oid)} :$$
$$\sum_{(R,r) \in IPORTS(oid)} (COND_R^{Net(oid)}(p)\langle r \circ b \rangle + PRECOND_R^{Net(oid)}(p)\langle r \circ b \rangle) \leq Markings_S(oid)(p),$$

5. Všetky volané synchrónne porty, patriace objektu, v ktorom testujem s-preveditelnosť prechodu t, sú súčasne s prechodom t bezkonfliktne preveditelné.
$$\forall o \ \in P_{Net(Oid_S(id))} :$$
$$\sum_{(R,r) \in IPORTS(Oid+S(id))} (COND_R^{Net(oid_S(id))}(p)\langle r \circ b \rangle + PRECOND_R^{Net(Oid_S(id))}(p)\langle r \circ b \rangle) +$$
$$+ (COND_t^{Net(Oid_S(id))}(p)\langle b \rangle + PRECOND_t^{Net(Oid_S(id))}(p)\langle b \rangle \leq Markings_S(Oid_S(id))(p).$$

Definícia 3.4.6. Definujeme $SYNC(S, id, t, b)$ ako účinok synchrónnej komunikácie inicovanej prechodom t v kontexte (S, id) pri naviazaní b takto:

Nech $IOBJS = \{Object_S(oid) | oid \in IOIDS\}$. Potom

$$Sync(S, id, t, b) = (S \ - IOBJS) \cup IOBJS',$$

kde množina $IOBJS'$ je definovaná takto:

$$o \ \in IOBJS \iff o' \in IOBJS'm$$

kde objekt o' je definovaný na základe objektu o takto:

$$oid = Object_S^{-1}(o).$$

$$o' = (o \ - \{(oid, M)\} \cup \{(oid, M')\},$$

kde značenie M' je definované na základe M takto:

$$\forall tt \in T_{Net(oid)} : M'(tt) = M(tt), \forall p \in P_{Net(oid)} :$$

$$M'(p) = M(p) - \sum_{(R,r) \in IPORTS(oid)} PRECOND_R^{Net(oid)}(p)\langle r \circ b \rangle +$$

$$+ \sum_{(R,r) \in IPORTS(oid)} POSTCOND_R^{Net(oid)}(p)\langle r \circ b \rangle$$

3.4.3 Udalosť typu A

Udalosť typu A (atomic) je atomické prevedenie vo vnútri jedného objektu s tým, že synchrónna komunikácia môže ovplyvniť stav iných objektov. Odobranie značiek zo vstupných miest, prevedenie akcie a uloženie značiek do výstupných miest prechodu sa prevedie súčasne ako nedeliteľná operácia.

Definícia 3.4.7. Majme $OOPN = (\Sigma, c_0, oid_0)$ a kontext (S, id). Nech $N = Net(id), C = Type(Oid_S(id))$.

1. Prechod $t \in T_N$ je **a-prevediteľný** v kontexte (S, id) pre naviazanie b práve vtedy, keď sú súčasne splnené nasledujúce podmienky“

   - Prechod t je v kontexte (S, id) pre naviazanie b s-prevediteľný,
   - $\forall p \in P_N : (COND_t^N(p)\langle b \rangle + PRECOND_t^N(p)\langle b \rangle) \leq CMarking_S(id)(p)$,
   - $ACTION_t^N$ má tvar $y := expr$, kde $expr$ je pre naviazanie b vyhodnotiteľné primitívne zaslanie správy.

   Nech $b' = b \cup \{(y, r)\}$, kde $r = expr\langle b \rangle$.

   Nech $SY = SYNC(S, id, t, b), oid = Oid_{SY}(id), o \ = Object_{SY}(oid)$.

   Nech $M = CMarking_{SY}(id), MM = Marking_{SY}(id)$ a $OM = Marking_{SY}(Oid_SY(id))$.

2. Ak je prechod $t \in T_N$ a-prevediteľný v kontexte (S, id) pre naviazanie b, môže byť a-prevedený, čím sa systém S zmení na systém S', definovaný následovne:

   $$S' = GC((SY - \{o\}) \cup \} o' \}),$$

   kde objekt o' je definovaný takto:

(a) Ak $id = oid$, teda ak ide o sieť objektu, tak:

$$o' = (o - \{(oid, M)\}) \cup \{(oid, m')\},$$

kde M' je definované takto:

$$\forall tt \in T_N : M'(tt) = M(tt),$$

$$\forall p \in P_N : M'(p) = M(p) - PRECOND_t^N(p)\langle b\rangle + POSTCOND_t^N(p)\langle b\rangle.$$

(b) Ak $id \neq oid$, teda ak ide o sieť metódy, tak:

$$o' = (o\ - \{(id, MM), (oid, OM)\} \cup \{(id, MM'), (oidmOM')\},$$

kde $MM' = M'|_{P_N - P_{ONET_C}) \cup T_N}$ a $OM' = M'|_{P_{ONET_C} \cup T_{ONET_C}}$,
kde M' je definované rovnako ako v (a).

3. Ak môže byť prechod $t \in T_N$ v kontexte (S, id) pre naviazanie b a-prevedený, pri čom výsledným stavom je stav S', hovoríme, že S' je **priamo dostupný** z S **a-prevedením** t v inštancii id pre naviazanie b, čo zapisujeme:

$$S\ [A, id, t, b\rangle S'.$$

3.4.4 Udalosť typu N – vytvorenie nového objektu

Udalosť typu N (new) je vytvorenie nového objektu prevedením prechodu. Prechod sa prevedie atomicky, podobne ako v prípade udalosti A, ale vznikne pri tom nový objekt.

Definícia 3.4.8. Majme $OOPN = (\Sigma, c_0, oid_0)$ a kontext (S, id). Nech $N = Net(id)$, $C = Type(Oid_S(id))$.

1. Prechod $t \in T_N$ je **n-prevediteľný** v kontexte (S, id) pre naviazanie b práve vtedy, keď sú súčasne splnené tieto podmienky:

   - prechod t je v kontexte (S, id) pre naviazanie b s-prevediteľný,
   - $\forall p \in P_N : (COND_t^N(p)\langle b\rangle) \leq CMarking(id)(p)$,
   - $ACTION_t^N$ má tvar $y := e_0.\texttt{new}$,
     kde y je premenná a e_0 je term taký, že $e_0\langle b\rangle \in CLASS_\Sigma$. Označme $C_{new} = e_n\langle b\rangle$.
   - $\exists oid_{new} \in Dom(C_{new}) - Id(FLAT(S))$.

2. Ak je prechod $t \in T_N$ n-prevediteľný v inštancii id pre naviazanie b v systéme S, môže byť **n-prevedený**, čím sa systém S zmení na systém S', definovaný takto:

$$S' = GC((SY - \{o\}) \cup \{o', o_{new}\}),$$

kde objekty o' a o_{new} sú definované takto:

- Ak $id = oid$, tak:
$$o' = (o\ - \{(oid, M)\}) \cup \{(oid, M')\},$$
kde M' je definované takto:

$$\forall tt \in T_N : M'(tt) = M(tt),$$

$$\forall p \in P_N : M'(p) = M(p) - PRECOND_t^N(p)\langle b\rangle + POSTCOND_t^N(p)\langle b\rangle.$$

- Ak $id \neq oid$, teda ak ide o sieť metódy, tak:

$$o' = (o \ - \{(id, MM), (oid, OM)\} \cup \{(id, MM'), (oidmOM')\},$$

kde $MM' = M'|_{P_N - P_{ONET_C}) \cup T_N}$ a $OM' = M'|_{P_{ONET_C} \cup T_{ONET_C}}$,
kde M' je definované rovnako ako v (a).

- Objekt o_{new} je definovaný takto:

$$o_{new} = \{(oid_{new}, M_{new})\},$$

kde M_{new} je definované takto:

$$\forall p \in P_{ON} : M_{new}(p) = PI_{ON}(p)\langle \emptyset \rangle,$$

$$\forall t \in T_{ON} : M_{new}(t) = \emptyset.$$

3. Ak môže byť prechod $t \in T_N$ v kontexte (S, id) pre naviazanie b n-prevedený, pri čom výsledným stavom je stav S', hovoríme, že S' je **priamo dostupný** z S **n-prevedením** t v inštancii id pre naviazanie b, čo zapisujeme:

$$S \ [N, id, t, b\rangle S'.$$

3.4.5 Udalosť typu F – predanie správy

Udalosť typu F (fork) je predanie správy neprimitívnemu objektu. Ide vlastne o vytvorenie procesu pre odpovedajúcu metódu. Je to neúplné prevedenie prechodu. Odoberú sa značky zo vstupných miest a vytvorí sa nová inštancia siete odpovedajúcej metódy, čo dokopy tvorí atomickú operáciu. Očakáva sa, že niekedy neskôr dôjde k udalosti typu J (join), ktorá prechod dokončí, teda zruší inštanciu siete metódy a umiestni značky do výstupného miesta prechodu.

Definícia 3.4.9. Majme $OOPN = (\Sigma, c_0, oid_0)$ a kontext (S, id). Nech $N = Net(id)$, $C = Type(Oid_S(id))$.

1. Prechod $t \in T_N$ je **f-prevediteľný** v kontexte (S, id) pre naviazanie b práve vtedy, keď sú súčasne splnené tieto podmienky:

   - prechod t je v kontexte (S, id) pre naviazanie b s-prevediteľný,
   - $\forall p \in P_N : (COND_t^N(p)\langle b \rangle) \leq CMarking(id)(p)$,
   - $ACTION_t^N$ má tvar $y := expr$, kde y je premenná a $expr$ je pre naviazanie b neprimitívne zaslanie správy tvaru $e_0.msg(e_1, e_2, ..., e_m)$, kde e_i sú termy. Nech $oid_r = e_0\langle b \rangle$ a $C_r = Class(N, e_0, b)$.
   - $msg^{(m)} \in MSG_{C_r}$.[4] Nech $MN = Method_{C_r}(msg)$.
   - $MN \in MNETS_{C_r}$.
   - $\exists id' \in Inst(MN) - Id(FLAT(S))$.

[4] Ak objekt správe nerozumie, tak je prechod neprevediteľný.

- $PP_{MN} = \{pp_1, ..., pp_m\}$.
  Nech $SY = SYNC(S, id, t, b), oid = Oid_{SY}(id), o\ = Object_{SY}(oid)$.
  Nech $M = CMarking_{SY}(id)$,
  $MM = Marking_{SY}(id)$,
  $OM = Marking_SY(Oid_{SY}(id))$ a
  $o_r = Object_{SY}(oid_r)$.

2. Ak je prechod $t \in T_N$ f-preveditelný v kontexte (S, id) pre naviazanie b, môže byť **f-prevedený**, čím sa systém S zmení na systém S', definovaný takto:

   (a) Ak $o \neq o_r$, potom

   $$S' = GC((SY - \{o, o_r\}) \cup \{o', o'_r\}),$$

   kde o' a o'_r sú definované takto:

   - Objekt o' je definovaný takto:
     i. Ak $id = oid$, tak:

     $$o' = (o\ - \{(oid, M)\}) \cup \{(oid, M')\},$$

     kde M' je definované následovne:

     $$\forall p \in P_N : M'(p) = M(p) - PRECOND_t^N(p)\langle b\rangle,$$

     $$M'(t) = M(t) \cup \{(id', b\}, \forall tt \in T_N - \{t\} : M'(tt) = M(tt).$$

     ii. Ak $id \neq oid$ potom:

     $$o' = (o\ - \{(id, MM), (oid, OM)\}) \cup \{(id, MM'), (oid, OM')\},$$

     kde $MM' = M'|_{(P_N - P_{ONET_C}) \cup T_N}$ a $OM' = M'|_{P_{ONET_C} \cup T_{ONET_C}}$,
     kde M' je definované ako v (i).
   - Objekt o'_r je definovaný takto:

     $$o'_r = o_r \cup \{(id', M'')\},$$

     kde značenie M'' je definované následovne:

     $$\forall p \in P_{MN} - P_{ONET_{C_r}} - PPMN : M''(p) = PI_{MN}(p)\langle\emptyset\rangle,$$

     $$\forall i \in \{1, ..., m\} : M''(pp_i) = PI_{MN}(pp_i)\langle\emptyset\rangle + e_i\langle b\rangle.$$

   (b) Ak $o = o_r$, potom

   $$S' = GC((SY - \{o\}) \cup \{o'\}),$$

   kde o' je definované takto:

   i. Ak $id = oid$, tak:

   $$o' = (o\ - \{(oid, M)\}) \cup \{(oid, M'), (id', M'')\},$$

   kde M' a M'' sú definované ako v (a).

ii. Ak $id \neq oid$, tak:

$$o' = (o \ - \{(id, MM), (oid, OM)\}) \cup \{(id, MM'), (oid, OM'), (id', M'')\},$$

kde MM', OM' a M'' sú definované ako v (a).

3. Ak môže byť prechod $t \in T_N$ v kontexte (S, id) pre naviazanie b f-prevedený, pri čom výsledným stavom je stav S', hovoríme, že S' je **priamo dostupný** z S **f-prevedením** t v inštancii id pre naviazanie b, čo zapisujeme:

$$S\ [F, id, t, b\rangle S'.$$

3.4.6 Udalosť typu J – akceptovanie odpovede na správu

Udalosť typu J (join) je predanie výsledku dokončenej metódy volajúcemu prechodu a jeho dokončenie. Ide vlastne o ukončenie procesu, pred tým vytvoreného udalosťou typu F. Zruší si sa inštancia siete zodpovedajúcej metódy a uložia sa značky do výstupného miesta prechodu. A toto všetko je atomická operácia.

Definícia 3.4.10. Majme $OOPN = (\Sigma, c_0, oid_0)$ a kontext (S, id). Nech $N = Net(id)$, $C = Type(Oid_S(id))$.

1. Prechod $t \in T_N$ je **j-preveditelný** v kontexte (S, id) pre naviazanie $b' : Var_N(t) \longrightarrow U$ práve vtedy, keď

   $$\exists (pid, b) \in M(T), existsr \in U \ : Marking_S(pid)(RP_{Net(pid)})(r) > 0 \wedge b' = b \cup \{(y, r)\}.$$

   Nech $o_r = Object_S(Oid_S(pid)), oid = Oid_S(id), o \ = Object_S(oid)$.
   Nech $M = CMarking_S(id), MM = Marking_S(id)$ a $OM = Marking_S(Oid_S(id))$.

2. Ak je prechod $t \in T_N$ j-prevedený v kontexte (S, id) pre naviazanie b', môže byť **j-prevedený**, čím sa systém S zmení na systém S', definovaný následovne:

   (a) Ak $o \neq o_r$, potom
   $$S' = GC((S - \{o, o_r\}) \cup \{o', o'_r\}),$$
   kde o a o'_r sú definované takto:

   - Objekt o'_r je definovaný takto:
   $$o'_r = o_r - \{(pid, Marking_S(pid))\}.$$
   - Objekt o' je definovaný takto:
     i. Ak $id = oid$, tak:
     $$o\ ' = (o \ - \{(oid, M)\}) \cup \{(oid, M')\},$$
     kde M' je definované takto:
     $$\forall p \in P_N : M'(p) = M(p) + POSTCOND_T^N(p)\langle b\rangle,$$
     $$M'(t) = M(t) - \{(pid, b)\}, \forall tt \in T_N - \{t\} : M'(tt) = M(tt).$$

ii. Ak $id \neq oid$, tak:

$$o' = (o \ - \{(id, MM), (oid, OM)\}) \cup \{(id, MM'), (id, OM')\},$$

kde $MM' = M'|_{P_N - P_{ONET_C} \cup T_N}$ a $OM' = M'|_{P_{ONET_C} \cup T_{ONET_C}}$,
kde M' je definované ako v (i),

- Ak $o = o_r$, tak

$$S' = GC((S - \{o\}) \cup \{o'\}),$$

kde o' je definované takto:

i. Ak $id = oid$, tak:

$$o' = (o - \{(oid, M), (pid, Marking_S(pid))\}) \cup \{(oid, M')\},$$

kde M' je definované ako v (a).

ii. Ak $id \neq oid$, tak:
$o' = (o \ - \{(id, MM), (oid, OM), (pid, Marking_S(pid))\}) \cup$
$\{(id, MM'), (oid, OM')\}$,
kde MM' a OM' sú definované ako v (a).

3. Ak môže byť prechod $t \in T_N$ v kontexte (S, id) pre naviazanie b' j-prevedený, pri čom výsledným stavom je stav S', hovoríme, že S' je **priamo dostupný** z S **j-prevedením** t v inštancii id pre naviazanie b', čo zapisujeme:

$$S \ [J, id, t, b'\rangle S'.$$

3.4.7 Stavový priestor OOPN

Systém objektov reprezentuje okamžitý stav systému, popísaného OOPN. Stavový priestor OOPN je množiny všetkých stavov, ktoré môže OOPN potencionálne dosiahnuť z počiatočného stavu.

Definícia 3.4.11. Majme $OOPN = (\Sigma, c_0, oid_0)$, nech $S_0 = \{\{(oid_0, PI_{ONET_{c_0}}\langle\emptyset\rangle)\}\}$ je jej počiatočný systém objektov.

1. Množina všetkých udalostí $EV(\Sigma)$ je množina štvoríc takých, že:
   (a) $e \in \{A, N, F, J\}$ je typ udalosti,
   (b) $id \in INST_\Sigma$ je inštancia siete, v ktorej k udalosti došlo,
   (c) $t \in T_{Net(id)}$ je prechod, ktorý udalosť vyvolal,
   (d) b je naviazanie premenných prechodu t, pre ktoré k udalosti (e, id, t, b) došlo.
2. **Stavový priestor** X objektovo orientovanej Petriho siete OOPN s počiatočným systémom objektov S_0 je najmenšia množina systémov objektov taká, že
   (a) $S_0 \in X$,
   (b) $S \in X \wedge (e, id, t, b) \in EV(\Sigma) \wedge S[e, id, t, b\rangle S' \Longrightarrow S' \in X$.

Kapitola 4

Virtuálny stroj

Virtuálny stroj je v informatike softvér, ktorý vytvára virtualizované prostredie medzi platformou počítača a operačným systémom, v ktorom môže koncový užívateľ prevádzať softvér na abstraktnom stroji [1]. Samotný termín „virtuálny stroj“ má niekoľko odlišných významov.

4.1 Hardvérový virtuálny stroj

Pôvodný význam pre virtuálny stroj, tiež nazývaný hardvérový virtuálny stroj, označuje niekoľko totožných pracovných prostredí na jednom počítači, z nich na každom beží operačný systém. Vďaka tomu môže byť aplikácia písaná pre jeden operačný systém na stroji, na ktorom beží OS alebo zaisťuje vykonanie *sandboxu*[1], ktorý poskytuje väčšiu úroveň izolácie medzi procesmi než je dosiahnuté pri vykonávaní niekoľkých procesov naraz (*multitasking*) na tom istom operačnom systéme. Jedným využitím môže byť taktiež poskytnúť ilúziu viacerým užívateľom, že používajú celý počítač, ktorý je ich „súkromným”strojom, izolovaným od ostatných užívateľov aj napriek tomu, že všetci používajú jeden fyzický stroj. Ďalšou výhodou môže byť to, že *bootovanie* a reštart virtuálneho počítača môže by oveľa rýchlejší, než u fyzického stroja, pretože môžu byť preskočené mnohé úlohy, ako je napríklad inicializácia hardvéru.

Podobný softvér je často označovaný ako virtualizácia a virtuálne servery. Hostiteľský softvér, ktorý poskytuje túto schopnosť je označovaný ako *hypervisor* alebo monitor virtuálneho stroja (*virtual machine monitor*).

Softvérové virtualizácie môžu byť prevádzané na troch hlavných úrovniach:

- **Emulácia** je plná systémová simulácia alebo „plná virtualizácia“ s dynamickým prestavením (*recompilation*) – virtuálny stroj simuluje kompletný hardvér dovoľujúci prevádzku nemodifikovaného operačného systému na úplne inom procesore.
- **Paravirtualizácia** – virtuálny stroj nesimuluje hardvér, ale namiesto toho ponúkne špeciálne rozhranie API, ktoré vyžaduje modifikáciu operačného systému.
- **Natívna virtualizácia**[2] a „plná virtualizácia“ – virtuálny stroj je čiastočne simuluje

[1] *Sandbox* je označenie pre bezpečnostný mechanizmus v rámci počítačovej bezpečnosti, ktorý slúži k oddeľovaniu procesov bežiacich s rovnakým oprávnením

[2] Pojem *natívna virtualizácia* sa niekedy používa k zdôrazneniu, že je využitá hardvérová podpora pre virtualizáciu.

hardvér aby mohol nemodifikovaný operačný systém bežať samostatne, ale hostiteľský operačný systém musí byť určený pre rovnaký druh procesoru.

4.2 Virtuálne prostredie

Virtuálne prostredie (tiež uvádzané ako virtuálny súkromný server) je iný druh virtuálneho stroja.

V skutočnosti je to virtualizované prostredie pre beh programov na úrovni užívateľa (tj. nie jadra operačného systému a ovládačov, ale aplikácií). Virtuálne prostredie je vytvorené použitím softvéru zavádzajúceho virtualizáciu na úrovni operačného systému ako napríklad *Virtuozzo, FreeBSD Jails, Linux-VServer.*

4.3 Združovanie virtuálnych strojov

Taktiež menej bežný termín „počítačový cluster“, čo je mnoho počítačov združených do veľkého a výkonnejšieho virtuálneho stroja. V tomto prípade softvér vytvára jednotné prostredie fyzicky nachádzajúce sa na viacerých počítačoch tak, že sa koncovému užívateľovi javí, že používa jediný počítač.

4.4 Aplikačný virtuálny stroj

Ďalším významom termínu virtuálny stroj je počítačový softvér, ktorý izoluje aplikácie používané užívateľom na počítači od operačného systému. Pretože virtuálne stroje sú písané pre rôzne počítačové platformy, akákoľvek aplikácia napísaná pre virtuálny stroj môže byť prevádzaná na ktorejkoľvek z platforiem namiesto toho, aby sa museli vytvárať oddelené verzie aplikácii pre každý počítač a operačný systém. Aplikácia bežiaca na počítači používa interpreta alebo *Just in time* kompiláciu.

4.5 Garbage collector

Garbage collector je často súčasťou aplikačného virtuálneho stroja, ktorý ma za úlohu automaticky určiť, ktorá časť pamäte programu je už nepoužívaná a pripraviť ju pre ďalšie znovupoužitie.

4.5.1 Vznik garbage collectingu

Garbage collection je označenie pre metódu automatickej správy pamäte programu. Garbage collection vymyslel v roku 1959 John McCarthy pre riešenie problému manuálnej správy pamäte v Lispe[1]. Najčastejšie je popisovaná ako opak manuálnej správy pamäte, ktorá vyžaduje špecifikovať program tak, aby bolo zrejmé, ktoré objekty sa môžu uvoľniť a ktoré sa majú vrátiť späť do pamäte.

Miesta v pamäti, ktoré už program použil a ďalej ich program nepoužije sa nazývajú *memory leaks* a Garbage collector tieto miesta hľadá a odstraňuje ich. Ďalším problémom je tzv. *dangling pointer*. Je to ukazateľ na prázdnu pamäť, alebo pamäť, ktorá bola znovu alokovaná inde v programe a prepísaná inými dátami.

Tieto chyby sú ťažko odhaliteľné a zlá podmienka väčšinou spôsobí nepredvídateľné správanie programu. Vyvarovanie sa chýb spôsobených správou pamäte na halde bolo jedným z hlavných dôvodov vzniku automatickej správy pamäte.

4.5.2 Základný princíp garbage collectingu

1. Vyhľadajú sa v programe také dátové objekty, ktoré nebudú v budúcnosti použité
2. Uvoľnia sa pamäťové zdroje, kde sa vyskytovali nájdené objekty.

Uvoľňovanie pamäte pomocou garbage collectoru oslobodzuje programátora od uvoľňovania objektov, ktoré už ďalej nie sú potrebné, čo ho väčšinou stojí značné úsilie. Je to vlastne pomôcka pre stabilnejší program, pretože zabraňuje niektorým prevádzkovým chybám. Napríklad zabraňuje chybám ukazateľov, ktoré ukazujú na už nepoužívaný objekt alebo na objekt, ktorý je už zrušený.

4.5.3 Algoritmus počítania referencií

Vôbec prvý algoritmus pre garbage collector sa nazýval *reference counting* (počítanie referencií). Funguje tak, že ku každému objektu je priradený čítač referencií. Keď je objekt vytvorený, jeho čítač je nastavený na hodnotu 1. V okamžiku, keď si nejaký iný objekt alebo koreň programu (korene sú hľadané v programových registroch, v lokálnych premenných uložených na zásobníkoch jednotlivých vlákien a v statických premenných) uloží referenciu na tento objekt, hodnota čítača je inkrementovaná o 1. Vo chvíli, keď je referencia mimo rozsah platnosti (napr. po opustení funkcie, ktorá si referenciu uložila), alebo keď je referencii priradená nová hodnota, čítač je dekrementovaný o 1. Ak je hodnota čítača u niektorého objektu nulová, môže byť tento objekt uvoľnený z pamäte. Keď je uvoľňovaný z pamäte tak sa všetkým objektom, na ktoré má tento objekt referenciu, zníži hodnota čítača o 1, to znamená, že uvoľnenie jedného objektu môže viesť k uvoľneniu ďalších objektov.

Nevýhoda tejto metódy spočíva v tom, že nedokáže detekovať cykly. Cyklus nastáva v okamžiku, keď dva a viacej objektov ukazujú samy na seba, napríklad keď rodičovská trieda ukazuje na svojho potomka a ten má referenciu späť na rodiča. tieto dva objekty nebudú mať nikdy čítač rovný nule, hoci sú nedosiahnuteľné z koreňa programu. Ďalšia nevýhoda spočíva v réžii, ktorá je nutná pre inkrementáciu a dekrementáciu čítača u každého objektu.

4.5.4 Sledovacie algoritmy

Sledovacie algoritmy (*tracking algorithms*) zastavia svet (v tomto zmysle beh programu) a začnú vyhľadávať objekty, ktoré je možné uvoľniť z pamäte. Začínajú v koreňovej množine programu a pokračujú po referenciách, pokiaľ nepreskúmajú všetky dosiahnuteľné objekty. Algoritmy, založené na tomto princípe sa používajú takmer výlučne pre implementáciu garbage collectorov v dnešných programovacích jazykoch.

Mark & Sweep Algoritmus Mark & Sweep najprv nastaví všetkým objektom, ktoré sú v pamäti, špeciálny príznak *navštívený* na hodnotu *nie*. Potom prejde všetky objekty, ku ktorým sa je možné dostať, tým ktoré navštívil, nastaví príznak na hodnotu *áno*. V okamžiku, keď sa už nemôže dostať k žiadnemu ďalšiemu objektu, znamená to, že všetky objekty s príznakom *navštívený* nastaveným na hodnotu *nie* sú odpad.

Táto metóda má niekoľko nevýhod. Najväčšou je, že pri garbage collectingu je prerušený beh programu. To znamená, že sa program pravidelne zmrazí.

Kopírovací collector Algoritmus kopírovací collector (*Copying collector*) najprv rozdelí priestor na halde na dve časti, kde jedna je aktívna a s druhou sa nepracuje. Vždy môžeme alokovať objekty v celkovej veľkosti, ktorá je polovičná veľkosť haldy. Pokiaľ sa pri alokácii nevojde do miesta na časti haldy, je potreba previesť garbage collection. Ten spočíva v zámene aktívnej a neaktívnej časti. Do novo aktívnej časti sa prekopírujú živé objekty zo starej, už neaktívnej, časti. Mŕtve objekty sa nekopírujú, ale pri ďalšej zámene aktívnej a neaktívnej časti sa jednoducho prepíšu.

Kopírovací algoritmus prebieha zdĺhavo, pretože objekty sa musia presúvať. Kvôli náročnosti celého presunu môžu teda vzniknúť značné oneskorenia pri behu programu. Výhodou je, že nenastáva žiadna fragmentácia.

4.5.5 Generačný algoritmus

Pri použití garbage collectorov sa dajú empiricky vypozorovať dva dôležité fakty. Prvým faktom je, že mnoho objektov sa stane odpadom krátko po svojom vzniku. Tým druhým je skutočnosť, že len malé percento referencií v „starších“ objektoch ukazuje na objekty mladšie.

U sledovacieho collectoru 4.5.4, kde sa používa celá halda, musel collector pri každom čistení prechádzať medzi objektami a všetky živé objekty buďto prekopírovať do inej časti haldy alebo ich označiť a ďalej prejsť celou haldou a uvoľniť mŕtve objekty. A práve z dôvodu rýchleho úmrtia väčšiny objektov je táto metóda neefektívna.

Generačný garbage collector využíva týchto skutočností a rozdeľuje pamäť programu do niekoľkých častí, tzv. *generácií*. Objekty sú vytvárané v spodnej (najmladšej) generácii a po splnení určitej podmienky, obvykle na základe veku, sú preradené do generácie staršej. Pre každú generáciu môže byť garbage collection prevádzaný v rozdielnych časových intervaloch (obvykle najkratšie intervaly budú pre najmladšie generácie), a dokonca pre rozdielne generácie môžu byť použité rozdielne algoritmy. V okamžiku, keď sa priestor v spodnej generácii zaplní, všetky dosiahnuteľné objekty v najmladšej generácii sú skopírované do staršej generácie. Aj tak bude množstvo kopírovaných objektov iba zlomkom z celkového množstva mladších objektov, pretože väčšina z nich sa stane odpadom.

Kapitola 5

Vlastná práca

Implementácia virtuálneho stroja vychádza priamo z formálnych definícií popísaných v kapitole 3. Samotný virtuálny stroj je tiež rozdelený do niekoľkých častí, a to:

- Statická reprezentácia OOPN.
- Systém objektov.
- Dynamika OOPN.
- Rozhranie.

Virtuálny stroj pracuje s jazykom PNTalk, ktorý je prebraný z [3], popísaný v prílohe D.

Objektovo orientovaná Petriho sieť zapísaná pomocou jazyka PNTalk je spracovaná pomocou prekladača, ktorý vytvorí medzikód pre virtuálny stroj popísaný v prílohe E. Medzikód sa inšpiruje bajtkódmi, používa ale len tlačiteľné znaky, čiže je možné daný kód prečítať. Medzikód je rozdelený na dve časti, statický popis Petriho siete a dynamická časť reprezentujúca aktuálny stav simulácie. Prekladač kontroluje iba syntax jazyka PNTalk a necháva sémantické kontroly na virtuálnom stroji.

Jazyk PNTalk rozširuje samotnú definíciu OOPN o niektoré konštrukcie, ako napríklad zložené správy, zoznamy, či konštruktory. Ide len o syntaktické vylepšenia a preto neboli v rámci virtuálneho stroja implementované.

Na základe tohto medzikódu je vytvorená statická reprezentácia OOPN v súlade s definíciou 3.2.10. Potom sú inicializované štruktúry podieľajúce sa na behu virtuálneho stroja.

Beh virtuálneho stroja je možné kedykoľvek zastaviť a aktuálny stav Petriho siete uložiť pomocou medzikódu a potom kedykoľvek spustiť virtuálny stroj načítaním tohto kódu uviesť Petriho sieť do uloženého stavu.

Po tom, ako je načítaná statická reprezentácia OOPN a prípadne jej stav alebo inicializovaný jej počiatočný stav, tak sa očakáva pokyn na spustenie simulácie v jednom z dvoch režimov:

- **Interaktívna simulácia**, pri ktorej virtuálny stroj poskytuje pred vykonaním nejakého prechodu zoznam preveditelných prechodov pre rôzne naviazania premenných a necháva na užívateľovi, aby udalosť, ktorá sa má vykonať, vybral.
- **Automatická simulácia**, pri ktorej virtuálny stroj prevádza udalosti paralelne pre náhodné naviazania premenných.

5.1 Statická reprezentácia OOPN

Objektovo orientovaná Petriho sieť je tvorená primitívnymi a neprimitívnymi objektami (inštanciami tried). Triedy sú definované množinou Petriho sietí skladajúcich sa z miest a prechodov, prepojených hranami. Miestam, prechodom a hranám môžu byť priradené popisy.

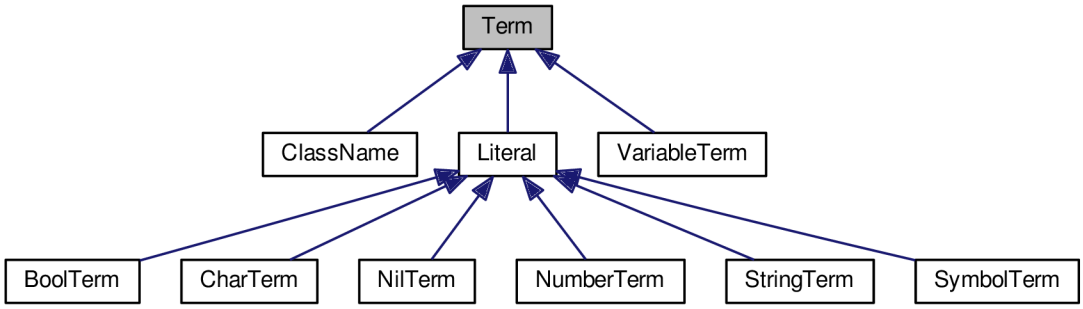
5.1.1 Term

Term je najjednoduchší výraz, ktorý jazykovo predstavuje objekt virtuálneho stroja. Je implementovaný pomocou abstraktnej triedy `Term`. Od tejto triedy dedia ostatné triedy virtuálneho stroja implementujúce konkrétne termy, čo nám umožňuje pracovať s termom bez znalosti o tom, o aký konkrétny term sa jedná. Dedičnosť jednotlivých tried je zobrazená na obrázku 5.1. Term môže byť:

1. *Literál* predstavujúci primitívny objekt.
   (a) *Číslo* je objekt reprezentujúci číselné hodnoty, reagujúci na správy požadujúce výsledky matematických operácií. Príklady čísiel sú `25, 13.677, -1.15e-10`.
   (b) *Znak* je objekt reprezentujúci znak ASCII tabuľky. Takýto literál je v jazyku PNTalk reprezentovaný ako dvojica znakov, kde prvý znak je vždy dolár a druhý znak je hodnota objektu, napr. `$$, $a, $+`.
   (c) *Reťazec* je objekt reprezentujúci sekvenciu znakov uzatvorených v apostrofoch (napr. `'abc'`).
   (d) *Symbol* je objekt používaný ako meno objektu. Symbol je reprezentovaný v jazyku PNTalk ako sekvencia znakov s prefixom `#`. Môže to napríklad byť symbol `#e, #failed, #notDoneYet`.
   (e) *Bool hodnota* je reprezentovaná kľúčovými slovami `true` a `false`.
   (f) *Nedefinovaný objekt* je reprezentovaný kľúčovým slovom `nil`. Napríklad neinicializovaná premenná obsahuje hodnotu nedefinovaného objektu. Pre pomenovanie nedefinovaného objektu sa často využíva symbol `#e`.
2. *Premenná* je identifikátor začínajúci malým písmenom. V priebehu výpočtu premenná reprezentuje rôzne objekty.
3. *Pseudopremenné* `self` a `super` reprezentujú premenné, ktorých hodnota závisí na kontexte, tzn. miestom v programe a stavom výpočtu.
4. *Meno triedy* je identifikátor s veľkým počiatočným písmenom. Mená tried sa v priebehu výpočtu nemenia.

5.1.2 Zaslanie správy

Zaslanie správy vychádza z druhého bodu definície 3.1.5, ktorá definuje množinu $EXPR(\Pi)$. Zaslanie správy je implementované pomocou triedy `Expr`, ktorá uchováva informácie o adresátovi správy, čo je Term, samotnú správu, reprezentovanú ako reťazec, a vektoru termov predstavujúcich parametre tejto správy. Implementácia zobecňuje všetky typy zaslania správy, popísané nižšie, na zaslanie správy s kľúčovými slovami.



Obr. 5.1: Dedičnosť triedy `Term`

Zaslanie správy môže byť Term, t.j. správa je prázdny reťazec alebo má zaslanie správy tvar `<adresát> <správa>`. Správa môže byť unárna, tzn. bez parametrov, binárna s použitím špeciálnych správ ako napríklad `+ - >= ==` a pod. alebo takzvaná správa s kľúčovými slovami, čo sú správy zakončené dvojbodkou, za ktorou nasledujú jednotlivé parametre správy. Príkladmi zaslania správy teda sú `C new`, `x - 4`, `'string' at: x`.

Samotné zaslanie správy a reakcie na správy sú implementované pomocou tried virtuálneho stroja popísaných v sekcii 5.2. Všetky objekty musia rozumieť správam `==` *rovnosť identity*, `~==` *nerovnosť identity*, `=` *rovnosť hodnoty*, `~=` *nerovnosť hodnoty*. Zoznam všetkých správ, na ktoré reagujú jednotlivé primitívne objekty je uvedený v prílohe C.

5.1.3 Stráž a akcia prechodu

Stráž prechodu je tvorená sekvenciou výrazov oddelených bodkou, pričom každý tento výraz je zaslanie správy. Sekvencia výrazov má význam konjunkcie výsledkov dielčich výrazov.

Akcia prechodu obsahuje výraz priradenia `:=`. Tento výraz má tvar `<premenná> := <zaslanie správy>`. PNTalk rozširuje definíciu OOPN o sekvenciu takýchto výrazov, čo si môžme predstaviť ako niekoľko prechodov za sebou prevádzajúcich dielčie akcie. Virtuálny stroj toto rozšírenie nepodporuje a teda každý prechod môže obsahovať iba jeden výraz priradenia, čo v súlade s definíciou OOPN. Sieť so sekvenciou výrazov v akcii prechodu je možné jednoducho previesť na sieť bez takejto sekvencie, prípadne je možné rozšíriť prekladač z jazyka PNTalk do medzikódu virtuálneho stroja tak, že automaticky zložené akcie prevedie na sériu prechodov.

Rozsah platnosti premenných v prechode zahrňuje stráž prechodu, akciu prechodu a výrazy na okolných hranách prechodu.

5.1.4 Hranové výrazy

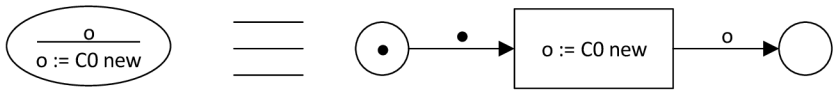
Implementácia hranových výrazov vychádza zo štvrtého bodu definície 3.1.5, ktorá definuje množinu $ARCEXPR(\Pi)$. Avšak narozdiel od definície sa v zápise multimnožiny používa čiarka namiesto symbolu `+`. Hranový výraz má teda tvar `n1'c1, n2'c2`, kde `n1, n2` sú termy reprezentované triedou `Term`. Tieto termy predstavujú počet výskytov `c1`, resp. `c2`,

kde `c1, c2` sú prvky množiny $LISTEXPR(\Pi)$. Ak je term `ni` číslo tak je počet výskytov jasný, takisto v prípade, že sa jedná o premennú referencujúcu číslo. Ak sa jedná o akýkoľvek iný term (znak, reťazec a pod.), tak sa jedná o syntaktickú chybu. Virtuálny stroj nedovoľuje také naviazanie premenných, aby premenná vyskytujúca sa v hranovom výraze ako term referencovala iný ako primitívny číselný objekt.

5.1.5 Miesta, prechody a hrany

Miesto je vo virtuálnom stroji reprezentované triedou `Place`. Každé miesto je špecifikované svojim názvom a *počiatočným značením*. Počiatočným značením sa zo syntaktického hľadiska rozumie hranový výraz, viz. sekcia 5.1.4, ktorý neobsahuje žiadne premenné. Implementácia značenia miesta vo virtuálnom stroji bude priblížená v sekcii 5.2.2, ktoré sa vytvára práve na základe hranového výrazu.

Jazyk PNTalk ďalej rozširuje miesto o možnosť špecifikovať tzv. *počiatočnú akciu*. Virtuálny stroj nepodporuje toto rozšírenie, ale podobne ako u zloženej akcie prechodu je možné sieť s počiatočnou akciou previesť na ekvivalentnú sieť bez počiatočnej akcie alebo by bolo možné rozšíriť prekladač z jazyka PNTalk, ktorý by počiatočnú akciu prevádzal na jej sémantický ekvivalent tak, ako to je ukázané na obrázku 5.2.



Obr. 5.2: Sémantika počiatočnej akcie

Prechod je vo virtuálnom stroji reprezentovaný triedou `Transition`. Každý prechod je špecifikovaný svojim názvom, strážou a akciou, ktorých syntax a sémantika boli vysvetlené v sekcii 5.1.3. Prechod je ďalej špecifikovaný tromi množinami hrán. Sú to konkrétne 3 množiny

- testovacie hrany
- vstupné hrany
- výstupné hrany

V statickej reprezentácii je hrana reprezentovaná dvojicou `<miesto, hranový_výraz>` a množina týchto hrán ako asociatívne pole, kde sa kľúčom rozumie názov miesta a hodnotou hranový výraz.

Rozsah platnosti mien miest a prechodov sa vzťahuje na sieť, v ktorej sa vyskytujú. Toto neplatí pre zdieľanie miest siete objektu sieťou metódy. Mená miest a prechodov sa uplatňujú pri využívaní dedičnosti, ako bude vysvetlené v sekcii 5.1.8.

5.1.6 Siete

Miesta a prechody, ktoré sú navzájom prepojené hranami, tvoria sieť. Pomocou sietí sú ďalej špecifikované triedy, pri čom rozlišujeme dva druhy sietí:

- **Sieť objektu** vychádzajúca z definície 3.2.1 a tvorí ju množina miest a prechodov.
- **Sieť metódy** špecifikuje reakciu objektu OOPN na príchodziu správu. Takisto, ako sieť objektu, je sieť metódy tvorená množinou miest, prechodov a hrán, preto trieda vo virtuálnom stroji reprezentujúca sieť metódy dedí od triedy reprezentujúcej sieť objektu a rozširuje ju v súlade s definíciou 3.2.3 o *vzor správy*, ktorý je zložený zo *selektoru správy* a parametrov správy. Ďalej o množinu *parametrových miest*, ktoré slúžia k predaniu parametrov pri volaní metódy. Ich mená zodpovedajú menám formálnych parametrov. Ak metóda nemá žiadne parametre, tak je táto množina prázdna. Nakoniec každá sieť metódy obsahuje jedno miesto pomenované `return` s prázdnym počiatočným značením, tzv. *výstupné miesto*, ktoré slúži k predaniu výsledku.

  Každá sieť metódy ďalej obsahuje referenciu na sieť objektu, pretože prechody siete metódy môžu byť prepojené s miestami siete objektu. Ak teda je prechod prepojený s miestom, ktoré sa nenachádza v sieti metódy, tak sa toto miesto vyhľadáva v sieti objektu.

5.1.7 Synchrónne porty

Ďalšou súčasťou špecifikácie tried sú *synchrónne porty*. Synchrónne porty majú charakter metód a zároveň prechodov. Synchrónny port je špecifikovaný vzorom správy, na ktorú reaguje a môže byť volaný zaslaním správy podobne ako metóda triedy. Neobsahuje ale žiadne miesta a podobne ako prechod, môže byť prepojený hranami s miestami siete objektu a môže obsahovať stráž.

Synchrónne porty slúžia k testovaniu a prípadnej zmene stavu objektu. Môžu byť volané s vyhodnotenými, ale i s nenaviazanými premennými. Synchrónny port môže byť buď prevediteľný alebo nie pre dané naviazanie premenných. Hľadanie vhodného naviazania premenných prebieha až pri volaní daného synchrónneho portu.

Volanie synchrónneho portu môže prebehnúť len zo stráže prechodu. Pri jeho volaní sa nájde vhodné naviazanie premenných. Pri vykonaní akcie prechodu, ktorý vo svojej stráži volá synchrónny port, sa prevedie pre náhodné naviazanie premenných aj synchrónny port ak neobsahuje len testovacie hrany.

5.1.8 Triedy a dedičnosť

Trieda je zložená zo siete objektu, sietí metód a synchrónnych portov. Statickú reprezentáciu triedy uchováva trieda virtuálneho stroja `ClassSpec`, ktorá popisuje špecifikáciu triedy v súlade s definíciou 3.2.7, bod 2.

Každá trieda má jedného rodiča v hierarchii dedičnosti, pri čom vrchol hierarchie tvorí trieda `PN`, ktorá nie je popísaná Petriho sieťami, ale je vytvorená v rámci inicializácie virtuálneho stroja ako trieda s prázdnou sieťou objektu, bez metód a bez synchrónnych portov.

V rámci dedičnosti získava každá trieda od svojho rodiča sieť objektu, všetky siete metód a všetky synchrónne porty. Jednotlivé metódy a synchrónne porty môžu byť redefinované uvedením nových definícií pre rovnaký selektor správy.

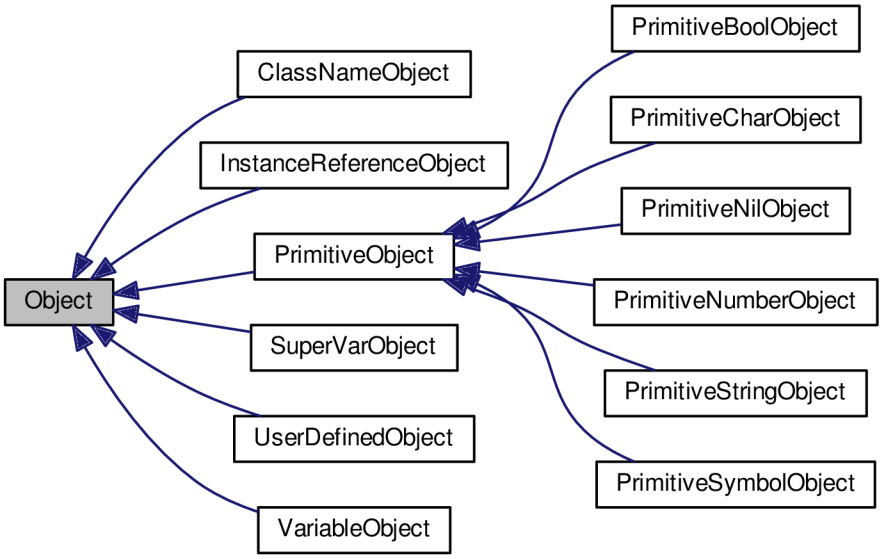
Sieť objektu môže byť predefinovaná po častiach, čiže je možné uviesť miesta alebo prechody s rovnakým názvom. Pri predefinovaní miesta sa zmení len jeho počiatočné značenie, pri predefinovaní prechodu sa zmenia okolné hrany prechodu, čo zodpovedá definícii 3.2.1, ktorá popisuje hrany ako súčasť prechodov.

Okrem predefinovania prvkov rodičovskej triedy je možné samozrejme pridávať nové metódy, synchrónne porty alebo doplňovať sieť objektu o nové miesta a prechody.

Dedičnosť je implementovaná tak, že pri vytváraní špecifikácie novej triedy sú najprv skopírované referencie na statickú reprezentáciu jednotlivých sietí metód, synchrónnych portov a je vytvorená nová sieť objektu ako hlboká kópia siete objektu rodičovskej triedy. V prípade, že dôjde k pridaniu nového prvku do triedy, tak je postup zrejmý. V prípade, že dôjde k predefinovaniu metódy, tak je vytvorená nová statická reprezentácia siete metódy, ktorá nahrádza referenciu na metódu rodičovskej triedy. Obdobne prebieha predefinovanie synchrónneho portu. Pri predefinovaní miest a prechodov siete objektu sú najprv staré prvky odstránené zo siete objektu a nahradené novými miestami, resp. prechodmi.

5.2 Systém objektov

Virtuálny stroj definuje abstraktnú triedu `Object`, ktorá reprezentuje objekt v systéme. Od tejto triedy dedia ďalšie triedy implementujúce konkrétne typy objektov, tak ako je to ukázané na obrázku 5.3.



Obr. 5.3: Dedičnosť triedy `Object`

Trieda poskytuje statickú metódu na vytvorenie objektu na základe abstraktnej triedy `Term` predstavenej v sekcii 5.1.1 pri čom nie je nutné poznať o aký term sa jedná. Ďalej trieda definuje čisto virtuálnu metódu `sendMessage`. Jednotlivé triedy dediace od triedy `Object` reimplementujú túto triedu tak, aby dokázali správne reagovať na príchodziu správu. Správy, na ktoré musia vedieť reagovať všetky objekty a správy, na ktoré reagujú primitívne objekty je možné nájsť v prílohe C.

Tento návrh umožňuje, aby bolo možné vytvárať objekty OOPN na základe termov a pracovať s týmito objektami vo virtuálnom stroji bez znalostí o typoch termov alebo objektov.

5.2.1 Primitívne objekty

Primitívne objekty vznikajú na základe literálov, sú to konštanty a po ich inicializácii už nemenia hodnoty, čo je v súlade s formálnou definíciou OOPN.

Triedy, ktoré reprezentujú primitívne objekty priamo definujú vo virtuálnej metóde `sendMessage` spracovanie správy a vracajú nový objekt ako výsledok prevedenia atomickej operácie[1].

5.2.2 Užívateľom definované objekty

Užívateľom definovaný objekt je inštancia triedy. Objekt je identifikovaný unikátnym `oid`, ktoré priraďuje trieda `Oid`. Táto implementuje oid ako hodnotu typu `int` a každému objektu priradí unikátne číslo pri čom čísla nerecykluje. Triedu je možné modifikovať, aby novým objektom priraďovala `oid` pred tým zmazaného objektu, ale keďže v rámci diplomovej práce nebol implementovaný žiadny grafický editor OOPN, ktorý by pracoval nad virtuálnym strojom, ale len jednoduchý *shell*, tak by recyklovanie `oid` mohlo viesť ku zhoršeniu prehľadnosti o vykonaných akciách virtuálneho stroja.

Inštancia siete. Podľa definície 3.3.4 objekt zapuzdruje inštanciu siete objektu a množinu inštancií sietí metód. Inštancia siete objektu je reprezentovaná triedou `NetInstance` a vychádza z definície 3.3.3. Teda reprezentuje inštanciu siete ako značenie miest a prechodov danej siete. Toto značenie je reprezentované pomocou asociatívneho poľa, ktorého kľúčom je identifikátor miesta (prechodu) a hodnota je značenie daného miesta (prechodu). Značenie miesta (prechodu) môžme chápať ako inštanciu daného miesta (prechodu), preto v ďalšom texte sú pojmy miesto a prechod myslené ako ich značenia (inštancie), nie ich statické reprezentácie. Každej inštancii siete je priradená unikátna hodnota `nid`, ktorá je unikátna v rámci objektu. Pre inštanciu siete objektu platí, že `nid=0`. Takto je možné špecifikovať ktorúkoľvek inštanciu siete v systéme ako dvojicu `oid:nid`.

Značenie miesta reprezentované triedou `DynamicPlace` uchováva množinu *značiek*, teda objektov, v danom mieste a poskytuje rozhranie umožňujúce prechodu spojeného s daným miestom vstupnou, resp. testovacou hranou, nájsť objekty, ktoré sú buď zhodné s nejakým literálom, alebo objekty, ktoré je možné naviazať na premennú.

Značenie prechodu reprezentované triedou `DynamicTrans` uchováva informácie o hranách, ktoré existujú medzi daným prechodom a nejakým miestom v sieti. O hranách uchováva dve informácie, ktoré získa zo statickej reprezentácie hrany[2] a to literály z hranového výrazu, ktoré sa musia vyskytovať v prechode (alebo budú pridané po prevedení prechodu) a druhou informáciou sú premenné, ktoré je potreba naviazať.

Stav prechodu je reprezentovaný pomocou množiny špeciálnych objektov *referencia na sieť metódy*, viz. 5.2.6. Prechod tak uchováva informácie o všetkých inštanciách sietí metód, na ktorých dokončenie čaká.

Trieda `DynamicTrans` sa ďalej implementuje algoritmy a poskytuje rozhranie na hľadanie naviazania premenných, testovanie stráže prechodu, vykonávanie akcie prechodu a zber

[1] Atomická operácia je v tomto kontexte atomické prevedenie prechodu, čiže vykonanie udalosti typu A, tak ako je to popísané v sekcii 3.4.3

[2] t.j. od tried `Transition` a `ArcExpr`

smetí, *garbage collection*, ktoré využíva časť virtuálneho stroja starajúca sa o dynamiku OOPN, čo bude popísané v sekcii 5.3.

Metóda `sendMessage` je implementovaná tak, že na základe selektoru správy objekt vyhľadá v špecifikácii triedy, z ktorej bol tento objekt inštanciovaný, sieť metódy, na základe ktorej vytvorí inštanciu siete metódy a vráti špeciálny objekt *referencia na sieť metódy*, viz. 5.2.6.

5.2.3 Špeciálny objekt premenná

Premenná je špeciálny typ objektu, ktorý je reprezentovaný svojim *názvom* a *hodnotou*. Hodnota predstavuje referenciu na iný objekt. Hodnota premennej platnej v rámci prechodu sa počas výpočtu neustále mení.

Špeciálnym prípadom takéhoto objektu je pseudopremenná `self`, ktorá vzniká pri vytvorení nového neprimitívneho objektu, jej hodnota je inicializovaná ako referencia ne tento novovzniknutý objekt a potom už nemení svoju hodnotu nikdy počas výpočtu.

Premenná definuje virtuálnu metódu `sendMessage` tak, že volá metódu `sendMessage` svojej hodnoty, teda objektu, ktorý referencuje.

5.2.4 Špeciálny objekt `super`

Pseudopremenná `super` je implementovaná pomocou špeciálneho objektu, ktorý vzniká pri vytvorení nového, užívateľom definovaného objektu. Objekt si uloží referenciu na tento novovzniknutý objekt.

Objekt implementuje metódu `sendMessage` veľmi podobne ako užívateľom definovaný objekt s tým rozdielom, že na vytvorenie inštancie siete nevyužíva špecifikáciu triedy, z ktorej bol objekt inštanciovaný, ale špecifikáciu triedy, ktorá je rodičom danej triedy v rámci hierarchie dedičnosti tried.

5.2.5 Špeciálny objekt názov triedy

Objekt *názov triedy* uchováva informácie o názve triedy a uchováva referenciu na špecifikáciu danej triedy. Pre každú triedu v OOPN existuje práve jeden takýto objekt. Objekt v metóde `sendMessage` reaguje iba na správu `new` a po jej obdržaní vytvorí nový, užívateľom definovaný objekt na základe špecifikácie triedy.

5.2.6 Špeciálny objekt referencia na sieť metódy

Referencia na sieť metódy je špeciálny objekt, ktorý nie je užívateľovi prístupný. To znamená, že volanie metódy `sendMessage` vyvolá výnimku, pretože tento objekt by sa nikdy, počas simulácie OOPN, nemal objaviť v akomkoľvek značení miesta. Objekt vzniká pri prevedení udalosti typu `F` a slúži k uloženiu stavu prechodu. Objekt zaniká v momente ako je možné previesť udalosť typu `J`.

5.3 Dynamika OOPN

Po vytvorení statickej reprezentácie OOPN na základe statickej časti medzikódu sa OOPN uvedie do stavu, ktorý je popísaný dynamickou časťou medzikódu alebo sa Petriho sieť

inicializuje do počiatočného stavu na základe statického popisu OOPN. V obidvoch prípadoch prebehne statická inicializácia *systému mien*, čo obnáša vytvorenie objektov pre názvy tried.

Inicializácia OOPN[3] obnáša kontrolu sémantiky jednotlivých prechodov, testuje sa napríklad, či niektorý prechod nepoužíva vo svojej stráži, prípadne akcii premennú, ktorá nemôže byť nikdy naviazaná[4] alebo či špecifikácia prechodu neobsahuje prepojenie hranou s miestom, ktoré nebolo nikde definované. Nakoniec je vytvorený počiatočný objekt na základe špecifikácie počiatočnej triedy, čím sa virtuálny stroj uvedie do stavu, kedy je možné začať so simuláciou.

V prípade, že medzikód obsahuje dynamickú časť, tak sa predpokladá, že sieť už prešla týmito sémantickými kontrolami a sieť je uvedená do popísaného stavu.

Ďalej stačí len rozhodnúť, či chceme previesť interaktívnu simuláciu, automatickú simuláciu, nastaviť maximálny *simulačný čas* (tj. maximálny počet prevedení prechodov) alebo nastaviť niektorému prechodu *break-point*, čiže bod zastavenia, aby sme virtuálnemu stroju povedali, aby prerušil svoj beh po prevedení daného prechodu.

Prevádzanie prechodov súvisí s vyhľadaním preveditelných naviazaní premenných, výberu jedného z týchto naviazaní a vykonaní prechodu. Aby prechod zvládol vykonať tieto úlohy, tak musí mať informácie o všetkých premenných a primitívnych objektoch prístupných pre daný prechod a takisto o všetkých triedach vyskytujúcich sa v OOPN v prípade, že by mal vytvárať novú inštanciu danej triedy. K tomuto účelu slúži už spomenutý *systém mien*.

5.3.1 Systém mien

Systém mien vychádza z definície 3.1.1 a uchováva dve statické globálne dostupné informácie:

- množinu $CLASS$, čiže objekty názvov tried, ktoré sú prístupné na základe znalosti názvu triedy. Množina $CLASS$ je inicializovaná v rámci inicializácie OOPN.
- množinu $NAME$, čiže množinu všetkých užívateľom definovaných objektov, ktoré sú prístupné na základe znalosti `oid` objektu. Užívateľom definovaný objekt vo svojom konštruktore volá statickú metódu systému mien, čím sa pridá do množiny $NAME$. Naopak pri deštrukcii sa z tejto množiny odstráni.

Inštancia systému mien je súčasť značenia prechodu, viz. 5.2.2, je skonštruovaná na základe statickej reprezentácie prechodu a uchováva množinu $CONST$, čo je množina primitívnych objektov použitých v stráži a akcii prechodu.

Ďalej sa uchováva množina V, čo je množina špeciálnych objektov reprezentujúcich premenné, viz. 5.2.3.

Systém mien uchováva hodnoty pseudopremenných `self` a `super`, pri čom `self` je reprezentovaná ako premenná, viz. 5.2.3, ktorú nie je možné naviazať a nikdy nemení svoju hodnotu a `super` je implementovaná pomocou špeciálneho objektu tak, ako bolo popísané v sekcii 5.2.4.

Systém mien poskytuje rozhranie pre nastavenie naviazania premenných, tj. nastaviť jedno z nájdených preveditelných naviazaní prechodu.

[3] Uvedenie Objektovo orientovanej Petriho siete do počiatočného stavu.

[4] Pretože sa nevyskytuje nikde v rámci hranového výrazu žiadnej vstupnej alebo testovacej hrany.

Nakoniec systém mien preťažuje operátor `[]`, ktorý vyhľadá v systéme mien objekt na základe termu. Napríklad, ak ide o term reprezentujúci premennú, tak operátor `[]` vráti objekt z množiny V reprezentujúci premennú s rovnakým názvom, alebo ak sa jedná o literál, napríklad číslo 3, tak operátor `[]` vráti objekt z množiny $CONST$, čo je primitívny objekt reprezentujúci číslo 3.

5.3.2 Testovanie preveditelnosti prechodu

Prvým problémom pri simulácii Objektovo orientovanej Petriho siete, s ktorým sa stretávame je *hľadanie preveditelných naviazaní premenných*, na základe ktorých môžme určiť, aké udalosti je možné vykonať v rámci simulácie.

Testovanie preveditelnosti prechodu, resp. hľadanie preveditelných naviazaní prechodu, je inšpirované [4].[5] Hľadanie preveditelných naviazaní prechodu prebieha v troch základných krokoch.

1. Hľadanie čiastočných naviazaní premenných.
2. Zlučovanie čiastočných naviazaní premenných.
3. Vyhodnotenie stráže prechodu pre jednotlivé naviazania.

Čiastočné naviazanie premenných formálne definujeme na základe [4] ako zobrazenie $b : V(t) \longrightarrow O \cup \{\perp\}$, kde $t \in T$, $V(t)$ je množina premenných platných v rámci prechodu $t \in T$ a O je množina objektov. Ďalej definujeme špeciálnu hodnotu $\perp$, pre ktorú platí $b(v) = \perp$ v prípade, že premenná $v \in V(t)$ nie je naviazaná.

Prechod uchováva pre každé miesto spojené s prechodom testovacou alebo vstupnou hranou množinu čiastočných naviazaní. Hľadanie čiastočného naviazania prebieha v dvoch krokoch. Najprv sa v mieste hľadajú primitívne objekty zodpovedajúce literálom príslušného hranového výrazu. Pokiaľ toto hľadanie zlyhá, tak čiastočné naviazanie je vyhodnotené ako prázdne, čiže nie je možné naviazať žiadne premenné. Potom sa hľadajú všetky možné naviazania premenných vyskytujúcich sa v hranovom výraze príslušnej hrany, premenné nevyskytujúce sa v hranovom výraze príslušnej hrany ostávajú nenaviazané.

Zlučovanie čiastočných naviazaní premenných súvisí s konceptom dvoch kompatibilných naviazaní. Formálne sú dve čiastočné naviazania premenných b_1 a b_2 kompatibilné (zapisujeme $Compatible(b_1, b_2)$) ak platí:

$$Compatible(b_1, b_2) \iff \forall v \ \in V(t) : b_1(v) \neq \perp \wedge b_2(v) \neq \perp \implies b_1(v) = b_2(v).$$

Kombinované čiastočné naviazanie premenných (zapísané ako $b(v) = Combine(b_1, b_2)$) pre dve kompatibilné čiastočné naviazania premenných b_1 a b_2 spĺňa nasledujúce:

$$b(v) = \begin{cases} b_1(v) & \text{ak } b_1(v) \neq \perp \\ b_2(v) & \text{ak } b_2(v) \neq \perp \\ \perp & \text{inak.} \end{cases}$$

Zlučovanie dvoch množín čiastočných naviazaní premenných B_1 a B_2 definujeme na základe vyššie uvedeného ako:

$$Merge(B_1, B_2) = \{Combine(b_1, b_2) \mid \exists (b_1, b_2) \in B_1 \times B_2 : Compatible(b_1, b_2)\}.$$

[5] Článok síce hovorí o farbených Petriho sieťach, ale princípy popisované v tomto článku sa dajú uplatniť aj v OOPN.

Takže po tom, ako prechod vytvorí čiastočné naviazania pre všetky miesta spojené s prechodom testovacími alebo vstupnými hranami, tak všetky tieto čiastočné naviazania zlúči pomocou algoritmu vychádzajúceho z formálnej definície $Merge(B_1, B_2)$ uvedenej vyššie, čím vznikne množina možných naviazaní pre daný prechod. Ak je táto množina neprázdna, tak prechod pre všetky tieto naviazania postupne vyhodnotí stráž prechodu (viz. 5.3.3), ak je špecifikovaná. Pri tom odstráni všetky naviazania, ktoré nespĺňajú podmienky uvedené v stráži prechodu. Takto vznikne množina preveditelných naviazaní prechodu.

Hľadanie a ukladanie informácií o čiastočných naviazaniach premenných umožňuje, že pri zmene značenia siete, teda pri zmene stavu niektorého zo vstupných alebo testovacích miest, stačí nájsť nové čiastočné naviazania len pre dané, zmenené miesta, a znovu zlúčiť všetky čiastočné naviazania premenných namiesto toho, aby bolo pri zmene stavu jedného miesta nutné znovu prechádzať všetky vstupné a testovacie miesta prechodu.

5.3.3 Vyhodnotenie stráže prechodu

Pri vyhodnocovaní stráže prechodu sa vyhodnocujú jednotlivé výrazy. Pred tým je ale skontrolovaná sémantika jednotlivých nájdených naviazaní premenných, aby sa predišlo situácii, kedy sa premennej priradí objekt, ktorý nerozumie správe, ktorú by mal prijať v rámci stráže, či akcie prechodu. Takéto naviazania premenných sú vyhodnotené ako neprevediteľné a odstránené.

Podobne sa kontroluje, či naviazania premenných spĺňajú podmienku, aby v rámci stráže prechodu sa jednalo len o primitívne zaslanie správy a synchrónnu komunikáciu v súlade so sekciou 3.4.2. Teda aby premenné naväzovali buď primitívne objekty alebo aby správy v rámci stráže prechodu zodpovedali synchrónnym portom naviazaného objektu.

Sekcia 3.4.2 ďalej popisuje postup vyhodnotenia stráže ako vyhodnotenie primitívnych zaslaní správ a potom synchrónnej komunikácie. Virtuálny stroj napriek tomu vyhodnocuje stráž prechodu pre jednotlivé naviazania premenných postupne, tzn. vyhodnocuje jednotlivé podmienky, kým nenarazí na prvú neplatnú a odstráni naviazanie alebo ak sa mu podarí vyhodnotiť všetky podmienky, tak naviazania prehlási za platné, inými slovami nechá ho v zozname preveditelných udalostí. Testovanie prebieha volaním už spomenutej abstraktnej metódy `sendMessage`, ktorá vracia v tomto prípade objekt, ktorý je vždy primitívny.[6] Využíva sa ďalšia virtuálna metóda, ktorá vyhodnotí, o akú pravdivostnú hodnotu sa jedná a to podľa tabuľky 5.1.

5.3.4 Prevádzanie akcie prechodu

Prechod po nájdení naviazaní a po testovaní preveditelnosti jednotlivých nájdených naviazaní obsahuje zoznam všetkých preveditelných naviazaní premenných. Tento zoznam je prístupný okoliu prostredníctvom rozhrania triedy reprezentujúcej inštanciu prechodu. Stačí vybrať jedno z týchto naviazaní a prechod pre dané naviazanie previesť.

Abstraktné triedy `Term` a `Object` a preťaženie operátoru `[]` systému mien umožňuje, aby vykonanie akcie prechodu vyzeralo tak, ako je to ukázané v kóde 5.1.

V kóde 5.1 `paramTerms` je množina termov vyskytujúcich sa ako parametre správy, `AdressTerm` je term reprezentujúci adresáta správy a `msg` je selektor správy, pri čom nepotrebujeme poznať, či je adresát správy primitívny objekt, užívateľom definovaný objekt alebo sa jedná o vytvorenie nového objektu ako inštancie triedy. Toto umožňuje virtuálny

[6] Vzhľadom na fakt, že v rámci stráže prechodu sa nikdy nevolajú metódy objektu, tak toto tvrdenie platí.

Primitívny objekt	Hodnota	Pravdivostná hodnota
Číslo	0	`false`
	iné	`true`
Reťazec		`false`
	iné	`true`
Znak	akákoľvek hodnota	`true`
Symbol	akákoľvek hodnota	`true`
Bool	False	`false`
	True	`true`
Nedefinovaný objekt	nil	`false`

Tabuľka 5.1: Pravdivostné hodnoty primitívnych objektov

```
vector<Object*> argv;
for(auto it: paramTerms)
{
    argv.push_back(namesystem[it]);
}

Object *result=namesystem[AddressTerm]->sendMessage(msg, argv);
```

Kód 5.1: Vykonanie akcie

stroj rozšíriť o ďalšie primitívne typy objektov ako napríklad o niektorý z abstraktných dátových typov ako je vektor, asociatívne pole a podobne.

Po vykonaní akcie potrebujeme len otestovať výsledný objekt, či sa jedná o špeciálny objekt reprezentujúci inštanciu siete metódy a zmeniť stav prechodu. Vždy však sú odstránené objekty zo vstupných miest pre dané prevedené naviazania a ak sa nejednalo o prevedenie udalosti typu F, tak sú pridané objekty do výstupných miest prechodu a prevedený *garbage collection*.

5.3.5 Zmena značenia siete a garbage collection

Pred vykonaním akcie sa najprv zo vstupných miest odstránia primitívne objekty zodpovedajúce literálom v hranových výrazoch vstupných hrán. Tieto primitívne objekty sú ihneď uvoľnené z pamäte. Následne sa na základe vybraného naviazania premenných odstránia zo vstupných miest objekty naviazané na premenné vyskytujúce sa v hranových výrazoch vstupných hrán.

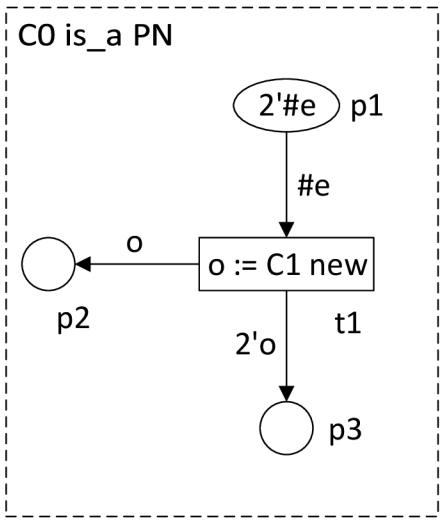
Pri prevedení udalosti typu F sa zmení stav prechodu, teda prechod si uloží referenciu na inštanciu siete metódy, ktorú vyvolal a naviazanie premenných, pre ktoré túto udalosť vykonal, tým prevádzanie prechodu skončí.

Pri prevedení udalostí typu A, N alebo J prechod po prevedení akcie vytvorí zoznam objektov, ktoré vybral zo vstupných miest na základe vybraného naviazania premenných a z tejto množiny odstráni tie objekty, ktoré sú naviazané na premenné vyskytujúce sa zároveň v hranových výrazoch vstupných aj výstupných hrán, čo indikuje, že sa jedná o presun objektu medzi miestami siete. V danej množine teda ostanú len objekty, ktoré je potrebné uvoľniť z pamäte.

Nakoniec sú vytvorené nové primitívne objekty na základe literálov nachádzajúcich sa v hranových výrazoch výstupných hrán a do výstupných miest pridané objekty zodpovedajúce naviazaniu premenných vyskytujúcich sa v hranových výrazoch výstupných hrán.

Miesta spojené s prechodom testovacími hranami ostávajú samozrejme nezmenené.

Garbage collector spracuje množinu objektov, ktoré boli vybrané zo vstupných miest a neboli poslané do miest výstupných. Pri tom ale potrebuje riešiť problém, že objekt sa môže vyskytovať v niekoľkých miestach naraz a v každom mieste v rôznych množstvách. Napríklad ako je tomu na obrázku 5.4, kde je prechodom `t1` vytvorený nový objekt a do miesta `p2` je priradená 1 referencia na tento objekt a do miesta `p3` 2 referencie. Keď neskôr vyberieme z jedného z týchto miest len 1 takýto objekt, tak nemôžme tento objekt uvoľniť z pamäte.



Obr. 5.4: Príklad vytvorenia niekoľkých referencií na objekt

Virtuálny stroj rieši tento problém tak, že objekty majú chránený deštruktor a teda nie je možné uvoľniť objekt z pamäte priamo, ale len pomocou na to určenej metódy.

Primitívne objekty sú podľa formálnej definície OOPN konštanty. To znamená, že ak majú dva primitívne objekty takú istú hodnotu, tak sa jedná o ten istý objekt. Viacnásobný výskyt primitívneho objektu je implementovaný ako niekoľko rôznych primitívnych objektov a teda pri požiadavke na uvoľnenie z pamäte je objekt hneď z pamäte uvoľnený, pretože je isté, že na daný objekt existuje vždy len jedna referencia.

Pri užívateľom definovaných objektoch sa virtuálny stroj inšpiruje algoritmom *reference counting*, viz. 4.5.3. Objekt si v rámci svojej štruktúry počíta koľko referencií na tento objekt existuje. Pri požiadavke na uvoľnenie objektu z pamäte sa tento počet referencií zníži a ak dosiahne hodnotu 0, tak je objekt z pamäte uvoľnený.

5.4 Prevádzanie udalostí v rámci virtuálneho stroja

Virtuálny stroj využíva triedu `Dynamics`, ktorá uchováva prevediteľné udalosti a stará sa o simuláciu Petriho siete, či sa už jedná o interaktívnu simuláciu, kedy je poskytnutý zoznam udalostí a očakáva sa výber jednej z týchto udalostí alebo o automatickú simuláciu, kedy sa spravuje simulačný čas. Je možné prevádzať simuláciu Petriho siete a pri tom vytvárať záznam o prevedených udalostiach a zmenách v stavoch miest a prechodov v textovom formáte.

Pri vytváraní, prevádzaní a odstraňovaní udalostí sa takisto aktualizuje zoznam závislostí medzi miestami a prechodmi a zoznam zmenených miest, aby bolo možné určiť, ktoré udalosti sa stali neplatnými alebo pre ktoré prechody je potrebné prepočítať naviazania premenných.

5.4.1 Udalosť

Udalosť vzniká na základe prechodu a jedného vybraného naviazania prevediteľného pre daný prechod. Udalosť si tieto dve informácie uloží a ďalej určí volanie metódy pre prevedenie prechodu pre dané naviazanie[7] a určí na základe naviazania o aký typ udalosti sa jedná.

Pokiaľ akcia prechodu obsahuje selektor správy `new`, tak sa jedná o udalosť typu N. Inak rozhoduje naviazanie premenných. Ak je adresátom správy primitívny objekt tak ide o udalosť typu A, v opačnom prípade o udalosť typu F.

Udalosti typu J vznikajú tak, že virtuálny stroj uchováva informácie o prechodoch, ktoré majú neprázdny stav a pravidelne testuje, či sú tieto prechody *dokončiteľné*. Ak inštancia siete metódy, spojená so stavom tohto prechodu, obsahuje nejaký objekt vo výstupnom mieste `return`, tak sa referencia na tento objekt uloží do naviazania premenných uloženého v rámci stavu prechodu a inštancia siete metódy sa zruší. Tým vznikne udalosť typu J, ktorá prevedie garbage collection a vloží objekty do výstupných miest daného prechodu.

Ďalšie informácie, ktoré si musí udalosť uchovať je `oid` objektu a `nid` inštancie siete, aby bolo možné určiť, o ktorú inštanciu prechodu sa jedná. Potom samozrejme je potrebné uchovať identifikátor prechodu v rámci statickej reprezentácie Petriho siete.

Udalosť je možné previesť do textového formátu, tak, aby bola človekom-čitateľná. Jej textový formát je následovný:

```
(typ_udalosti, oid:nid, trieda:metóda:prechod, naviazanie_premenných)
```

Pričom ak sa jedná o o prechod v sieti objektu, tak `nid` je 0 a `metóda` je prázdny reťazec. Uveďme si dva príklady textových reprezentácií udalostí:

```
(N, oid0:nid0, C0::t1, {})
(A, oid1:nid2, C1:waitFor:t2, {(x,1), (y,2)})
```

5.4.2 Interaktívna simulácia

Pokiaľ virtuálny stroj spustí interaktívnu simuláciu, tak poskytne zoznam všetkých prevediteľných udalostí a očakáva výber jednej z týchto udalostí a prevedie ju. Po výbere udalosti virtuálny stroj túto udalosť začne prevádzať.

[7] Prechod využíva pre uloženie naviazaní premenných vektor a teda asociuje naviazanie s indexom v danom vektore.

Najprv invalidujú všetky udalosti, ktoré súvisia s prechodmi, ktorých vstupné a testovacie hrany sú spojené s miestami prepojenými so vstupnými hranami prechodu, ktorého sa prevádzaná udalosť týka, pretože naviazania premenných týchto prechodov sa stávajú neplatné. Počas tohto procesu sa pridávajú prechody s neplatnými naviazaniami premenných do množiny prechodov, pre ktoré je potrebné prepočítať naviazanie premenných.

Následne sa vykoná samotná udalosť, čiže sa vyhodnotí akcia prechodu pre naviazanie premenných, ktoré popisovala udalosť. Pokiaľ sa jednalo o udalosť typu F, tak je prechod pridaný do množiny *nedokončených* prechodov. Ak sa jednalo o iný typ udalosti, tak všetky prechody, ktorých vstupné a testovacie hrany sú spojené s výstupnými miestami vykonaného prechodu, sú pridané do množiny prechodov, ktorým je potrebné prepočítať naviazanie premenných.

Nakoniec sa pri vykonaní udalosti prepočítajú znova naviazania všetkých prechodov, ktorých vstupné a testovacie miesta zmenili svoj stav a zistí sa, či niektorá inštancia siete metódy nedokončila svoje prevádzanie a je možné previesť udalosť typu J, ktorá je prípadne vygenerovaná.

Využitie vlákien. Vrámci tejto časti implementácie vznikol nápad, ktorý navrhoval využiť vlákna pre urýchlenie výpočtu. Idea bola vytvoriť jedno vlákno, ktoré by bežalo na pozadí a pristupovalo k množine prechodov, ktorým je potrebné prepočítať naviazania premenných a tieto naviazania prepočítavalo zatiaľ čo by vlákno na popredí vykonávalo zadanú udalosť.

Samozrejme pri interaktívnej simulácii sa očakáva komunikácia s užívateľom, takže urýchľovanie na prvý pohľad nemá zmysel, pretože ľudské reakcie sú príliš pomalé, aby si dané urýchlenie prejavilo, avšak užívateľ môže využiť interaktívnu simuláciu k tomu, aby použil svoju vlastnú automatickú stratégiu výberu udalostí.

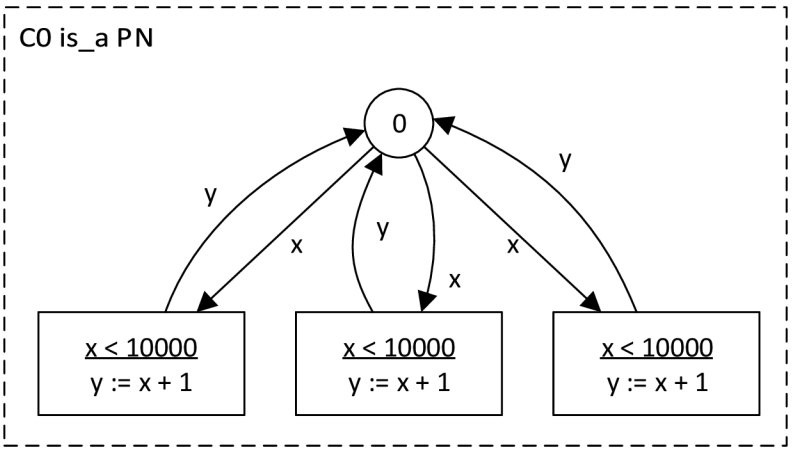
Návrh bol teda implementovaný a prebehol test, ktorý obsahoval sieť zobrazenú na obrázku 5.5, tzn. pri simulácii bolo vykonaných 10 000 udalostí a každá udalosť invalidovala tri prechody. Daná simulácia bola vykonaná nad implementáciou, ktorá využívala na prepočítanie naviazaní vlákno bežiace na pozadí a nad implementáciou, ktorá na konci prevádzania udalosti vždy prepočítala naviazania pre invalidované prechody. Pri výbere udalostí bola vybraná vždy prvá udalosť v zozname preveditelných udalostí. Meranie prebehlo pre každú implementáciu 5-krát a výsledky meraní ukazuje tabuľka 5.2.

Meranie	Sekvenčná implementácia [ms]	Implementácia s vláknom [ms]
1.	198	405
2.	209	406
3.	213	399
4.	205	412
5.	201	402

Tabuľka 5.2: Meranie výkonu pri využitý vlákna pracujúceho na pozadí.

Meranie ukázalo, že réžia spojená s výlučnými prístupom vlákien do kritických sekcií[8] a réžia zasielania správ medzi vláknami je príliš vysoká na to, aby sa oplatilo viacvláknový prístup využiť pri interaktívnej simulácii.

[8]Kritická sekcia je napríklad zoznam preveditelných udalostí a množina invalidovaných prechodov.



Obr. 5.5: Testovacia Objektovo orientovaná Petriho sieť.

5.4.3 Automatická simulácia

Automatická simulácia náhodne vyberá preveditelné udalosti Petriho siete a prevádza ich, pri tom postupuje podobne ako pri interaktívnej simulácii. Pri automatickej simulácii sa využívajú vlákna, aby túto simuláciu urýchlili. Pri návrhu virtuálneho stroja vznikli tri návrhy využitia vlákien:

- Využiť vlákno pre každý prechod.
- Využiť vlákno pre každú inštanciu siete.
- Využiť obmedzený počet vlákien, ktoré pristupujú k zoznamu preveditelných udalostí a náhodne vyberajú niektorú udalosť a prevádzajú ju.

Vlákno pre každý prechod. Tento návrh hovorí, že každý prechod v OOPN neustále prepočítava možné naviazania premenných a ak môže, tak ľubovoľné naviazanie prevedie. Ktorý prechod sa prevedie zaleží od toho, ktorý prechod sa získa prístup ku kritickým sekciám, čo sú miesta v sieti. Takto by bola zaručená nedeterministickosť OOPN. Problém nastáva práve pri prístupe k miestam v sieti, lebo nie je možné zaručiť poradie prístupu k jednotlivým miestam a predchádzanie uviaznutia vlákien by vyžadovalo implementáciu dodatočnej synchronizácie. Ďalším problémom je, že pri takomto návrhu nie je kontrolované, s koľkými vláknami virtuálny stroj pracuje.

Vlákno pre každú inštanciu siete. Tento návrh je najbližšie formálnej definícii OOPN. Pri prevedení udalosti typu F (*fork*) sa vytvorí nové vlákno a pri udalosti typu J (*join*) sa toto vlákno ukončí. Každé vlákno pri tom udržiava zoznam preveditelných udalostí v rámci prechodov vyskytujúcich sa v inštancii siete, ktorú prevádza. Tento návrh opäť nekontroluje počet vlákien bežiacich v rámci virtuálneho stroja, čo by bolo možné vyriešiť odoberaním vlákien inštanciám, ktoré nemajú preveditelný žiadny prechod a priraďovanie vlákien inštanciám napríklad podľa priority, kde priorita by bola priamo úmerná počtu preveditelných prechodov v sieti. To by mohlo viesť k problému, kedy sa donekonečna prevádza

niekoľko prechodov v niektorých sieťach namiesto toho, aby bol prevedený jediný prevediteľný prechod v inej sieti, ktorý by mohol tento nekonečný cyklus ukončiť. Ďalšou možnosťou by bolo prevádzať vždy niektorú z inštancií siete iba obmedzený čas, teda implementovať nepreemptívny plánovač úloh. Najvýhodnejšou alternatívou by pravdepodobne bolo využiť niektorí z existujúcich plánovačov implementovaných v operačných systémoch a implementovať ho vo virtuálnom stroji.

Náhodný výber udalostí vláknami. Tento návrh hovorí, že by každé vlákno malo pracovať v nekonečnom cykle, v ktorom náhodne vyberie jednu z vypočítaných prevediteľných udalostí, prevedie ju a potom prepočíta nové naviazania premenných pre prechody, ktorým sa zmenili vstupné a testovacie miesta na základe prevedenia danej udalosti. Tento spôsob bol implementovaný v rámci tejto diplomovej práce hlavne z dôvodu jednoduchosti.

5.5 Rozhranie

Podľa argumentov príkazového riadku je možné spustiť virtuálny stroj v niekoľkých rôznych režimoch. Pre spustenie je nutné špecifikovať vstupný súbor s medzikódom obsahujúcim špecifikáciu a stav načítanej Petriho siete alebo súbor so sieťou špecifikovanou v jazyku PNTalk.

Po načítaní Petriho siete je buď táto sieť automaticky prevedená a jej výsledný stav uložený pomocou medzikódu alebo je možné spustiť jednoduchý príkazový riadok, pomocou ktorého je možné špecifikovať typ simulácie, maximálny simulačný čas, je možné v ľubovoľnom okamžiku zastaviť simuláciu a pomocou pár jednoduchých príkazov nechať vypísať stav konkrétnych miest a prechodov existujúcich v systéme alebo zobraziť stav všetkých miest a prechodov v systéme.

Implementovaný príkazový riadok slúži ako kompenzácia za absenciu nejakého grafického editoru, ktorý by dokázal s virtuálnym strojom komunikovať. Príkazový riadok je implementovaný tak, aby demonštroval možné použitie rozhrania triedy, ktorá zapuzdruje celý virtuálny stroj a bolo možné jednoducho tento príkazový riadok nahradiť implementáciou, ktorá by napríklad umožňovala komunikáciu s grafickým editorom pomocou bajtkódu.

Argumenty príkazového riadku a takisto aj príkazy pre *shell*, ktorý je možné spustiť, sú uvedené v prílohe B.

Kapitola 6

Záver

Táto práca popísala implementáciu virtuálneho stroja pre Objektovo orientované Petriho siete alebo OOPN. OOPN boli predstavené a formálne popísané a z týchto formalizmov vychádza samotný virtuálny stroj. Virtuálny stroj je implementovaný v jazyku C++ s využitím štandardu `C++11`, pri tom neboli použité žiadne dodatočné knižnice, čo zaručuje jeho prenositeľnosť.

Vzhľadom na fakt, že je pomerne problematické samostatne vytvoriť plnohodnotný a optimalizovaný virtuálny stroj v priebehu jedného akademického roku, tak virtuálny stroj vytvorený v rámci tejto diplomovej práce je navrhnutý a implementovaný hlavne s dôrazom na jeho rozšíriteľnosť a čitateľnosť kódu, aby ho bolo možné jednoducho rozšíriť napríklad o nové primitívne objekty alebo upraviť medzikód virtuálneho stroja, pridať ďalší prekladač z iného jazyka ako PNTalk, implementovať rozšírenia jazyka PNTalk, upraviť rozhranie virtuálneho stroja, optimalizovať ľubovoľné časti virtuálneho stroja, či pozmeniť virtuálny stroj tak, aby dokázal komunikovať s nejakým vytvoreným grafickým editorom Objektovo orientovaných Petriho sietí, či dokonca umožňoval komunikovať s niekoľkými rôznymi grafickými editormi rôznymi spôsobmi. Každú takúto úpravu je možné previesť pridaním alebo úpravou jednej triedy virtuálneho stroja.

Súčasťou práce je aj vygenerovaná *doxygen* dokumentácia slúžiaca ku sprehľadneniu vytvoreného kódu, aby prípadné rozšírenia, či optimalizácie boli vykonané ešte jednoduchšie.

V implementovanom virtuálnom stroji je OOPN možné načítať za pomoci jazyka PNTalk, ktorý je popísaný v prílohe D. Popis OOPN v jazyku PNTalk je prevedený do medzikódu, viz. príloha E. Prekladač PNTalku by bolo možné doplniť o pokročilé kontroly a dosiahnuť tak lepšiu spätnú odozvu pri kontrole syntaxe tohto jazyka a ďalej rozšíriť implementáciu prekladača o spracovanie rozšírení zavedených do PNTalku, ako napríklad zložené akcie prechodov, či počiatočné akcie miest, ktoré momentálna implementácia nepodporuje.

Vytvorený medzikód sa skladá z dvoch častí, statickej a dynamickej. Statickou časťou rozumieme popis jednotlivých prvkov OOPN, tj. tried, sietí, miest, prechodov... Dynamická časť slúži pre uloženie aktuálneho stavu siete, čo umožňuje aby bol beh OOPN kedykoľvek pozastavený, stav siete uložený a inokedy opäť načítaný. Medzikód je človekom-čitateľný, čiže obsahuje len tlačiteľné znaky, čo prináša pár komplikácii najmä pri primitívnych objektoch typu *reťazec*. Medzikód môže byť zmenený na bajtkód jednoduchou úpravou jedného hlavičkového súboru.

Po vytvorení medzikódu je tento kód načítaný virtuálnym strojom a môže sa previesť

- **Automatická simulácia**, kedy sú náhodne vyberané a prevádzané udalosti OOPN.
- **Interaktívna simulácia**, kedy sa zobrazia všetky prevediteľné udalosti zapísané

v textovom formáte a očakáva sa výber jednej z týchto udalostí pomocou zadania indexu danej udalosti.

- **Spustenie príkazového riadku**, kedy je spustený veľmi jednoduchý *shell*, ktorý umožňuje virtuálny stroj ovládať a prevádzať na ňom základné operácie týkajúce sa OOPN.

Vyššie zmienený príkazový riadok slúži len ako kompenzácia za absenciu grafického editoru Petriho sietí, ktorý by dokázal s virtuálnym strojom komunikovať a pracovať. Implementácia ukazuje ako použiť rozhranie triedy zapuzdrujúcej virtuálny stroj a je možné tento príkazový riadok napríklad nahradiť implementáciou, ktorá by komunikovala s nejakým grafickým editorom na základe bajtkódu. Príkazy tohto príkazového riadku je možné nájsť v prílohe B.2.

Pre automatickú simuláciu využíva virtuálny stroj vlákna, čo je jeden z hlavných dôvodov využitia štandardu `C++11`. Bol implementovaný najjednoduchší návrh, kedy každé vlákno náhodne vyberie jednu z preveditelných udalostí a prevedie ju. Bolo by možné implementovať aj iný prístup, kedy je vláknam priradená celá inštancia nejakej siete, tak ako to bolo popísané v sekcii 5.4.3 a pomocou meraní zistiť, či by takáto implementácia bola rýchlejšia.

Literatúra

[1] Craig, I. D.: *Virtual Machines*. 2010, ISBN 1-85233-969-1.

[2] Goetz, B.: *Java theory and practice: A brief history of garbage collection*. [online], 2003, [cit. 2013-04-19].
URL http://www.ibm.com/developerworks/java/library/j-jtp10283/

[3] Janoušek, V.: *Modelování objektů Petriho sítěmi*. Dizertační práce, Brno, CZ, 1998.

[4] Liu, F.; Heiner, M.: *Computation of enabled transition instances for colored Petri nets*. [online], 2010.
URL http://ceur-ws.org/Vol-643/paper05.pdf

[5] Češka, M.; Marek, V.; Novosad, P.; aj.: *Petriho síte studijní opora*. 2009.

Príloha A

Obsah CD

↪ `src/` – Adresár so zdrojovými kódmi virtuálneho stroja

↪ `tex/` – Zdrojové kódy LaTeXpre túto technickú správu.

↪ `doc/` – Adresár s touto technickou správou.

↪ `doxy/` – Vygenerovaná doxygen dokumentácia.

↪ `examples/` – Niekoľko príkladov Objektovo orientovaných Petriho sietí zapísaných pomocou jazyka PNTalk a takisto uložených pomocou medzikódu virtuálneho stroja.

↪ `oopnvm` – Spustiteľný súbor virtuálneho serveru získaný prekladom na školskom serveri merlin.

↪ `Makefile` – Makefile pre preklad virtuálneho stroja na platforme Linux.

Príloha B

Manuál

Priložené CD obsahuje spustiteľný súbor vytvorený na školskom serveri merlin. V prípade, že je potrebné vytvoriť iný spustiteľný súbor, tak je to možné pomocou priloženého `Makefile`. Virtuálny stroj bol vytvorený len pomocou štandardnej knižnice C++ s využitím štandardu `C++11`. Pre preklad teda nie je nutné inštalovať žiadne dodatočné knižnice, jediná podmienka potrebná pre úspešný preklad je prekladač `gcc` verzie 4.7 a vyššie.

Virtuálny stroj je možné spustiť v niekoľkých režimoch pomocou argumentov príkazového riadku a potom prípadne virtuálny stroj ovládať pomocou jednoduchého príkazového riadku.

B.1 Parametre virtuálneho stroja

Virtuálny stroj je možné spustiť s nasledujúcim parametrami príkazového riadku:

`--pntalk <file>` `-p <file>`	Načítať OOPN zo súboru `file` zapísanej pomocou jazyka PNTalk.
`--bytecode <file>` `-b <file>`	Načítať OOPN zo súboru `file` zapísanej pomocou medzikódu.
`-s <steps>`	Nastaviť maximálny počet simulačných krokov, ak parameter nie je zadaný, tak je počet krokov nastavený na 100 000.
`-breakpoint <trans>`	Nastaviť prechodu `trans` bod zastavenia, tzn. automatická simulácia sa zastaví, keď sa daný prechod prevedie pre ľubovoľné naviazanie premenných v ľubovoľnej inštancii siete. Názov prechodu `trans` je tvorený dvojicou `class:name`, ako názov triedy a názov prechodu oddelené dvojbodkou.
`--interactive` `-i`	Spustiť interaktívnu simuláciu.
`--auto` `-a`	Spustiť automatickú simuláciu.
`--shell`	Spustiť jednoduchý shell.

`--output <file>` `-o <file>`	Uložiť stav OOPN pomocou medzikódu pri ukončení virtuálneho stroja do súboru `file`. Ak súbor nie je špecifikovaný, tak sa výsledný stav uloží do súboru `a.oopn`.
`--log <file>` `-l <file>`	Virtuálny stroj vytvára log o svojej činnosti. Ukladá informácie o tom akú udalosť vykonal a ako sa zmenil stav siete po vykonaní tejto udalosti. Tento log ukladá do súboru `file` alebo vypisuje na štandardný výstup, ak je súbor zadaný ako znak `-`.
`--threads X` `-t X`	Špecifikuje, koľko vlákien sa má použiť pre automatickú simuláciu. V prípade, že parameter nie je zadaný, tak virtuálny stroj použije hodnotu získanú pomocou volania statickej metódy `std::thread::hardware_concurrency`.

B.2 Jednoduchý shell

Vrámci virtuálneho stroja bol implementovaný jednoduchý príkazový riadok, ktorý slúži ako demonštrácia využitia rozhrania triedy zapuzdrujúcej virtuálny stroj. Na základe tejto demonštrácie môže byť napríklad vytvorené rozhranie, ktoré by dokázalo komunikovať s nejakým grafickým editorom napríklad pomocou nejakého bajtkódu.

Príkazový riadok pozná niekoľko jednoduchých príkazov:

`events`	Zobrazí prevediteľné udalosti.
`run X`	Prevedie udalosť s indexom `X`.
`save <file>`	Uloží aktuálny stav siete pomocou medzikódu do súboru `file`.
`interactive`	Spustí interaktívnu simuláciu.
`steps X`	Nastaví maximálny počet simulačných krokov na hodnotu `X`.
`breakpoint class:trans`	Nastaví prechodu `trans` v sieti `class` bod zastavenia.
`auto`	Spustí automatickú simuláciu.
`place oid:nid:name`	Zobrazí stav miesta.
`transition oid:nid:name`	Zobrazí stav prechodu.
`object oid`	Zobrazí informácie o objekte `oid`.
`all`	Zobrazí stav všetkých miest a prechodov v sieti.
`verbose`	Vypisuje sa všetko, čo virtuálny stroj prevádza.
`exit`	Ukončí shell.
`stop`	Zastaví prebiehajúcu automatickú simuláciu.

Príloha C

Primitívne objekty OOPN

Nasledujúca príloha ukazuje jednotlivé primitívne objekty virtuálneho stroja a popisuje všetky správy, na ktoré primitívne objekty reagujú v nasledujúcom formáte

`<výsledný_typ> := <typ_objektu> <správa> [parameter]*`,

kde `<výsledný_typ>` je primitívny typ objektu, ktorý je vrátený ako výsledok odoslania správy. `<typ_objektu>` je primitívny typ objektu, ktorému sa zasiela správa. `<správa>` je samotná správa a `parameter` predstavuje primitívny typ Objektu, ktorý musí byť dodaný ako parameter.

Všetky objekty musia reagovať na 4 správy, a to

`bool := Objekt == Objekt`	rovnosť identity objektov
`bool := Objekt ~== Objekt`	nerovnosť identity objektov
`bool := Objekt = Objekt`	rovnosť hodnoty objektov
`bool := Objekt ~= Objekt`	nerovnosť hodnoty objektov

Formálna definícia OOPN hovorí o primitívnych objektoch a ako o konštantách a preto porovnanie identity a porovnanie hodnoty primitívnych objektov vracia tie isté výsledky. U užívateľom definovaných objektoch platí, že dva objekty sú identické, ak referencujú ten istý objekt a dva objekty sú rovnaké, ak sú tieto dva objekty inštancie tej istej triedy.

Keďže jazyk PNTalk vychádza z jazyka Smalltalk, tak aj tieto jednotlivé správy vychádzajú z primitívnych objektov jazyka Smalltalk.

C.1 Číslo

`číslo := číslo + číslo`	sčítanie
`číslo := číslo - číslo`	odčítanie
`číslo := číslo * číslo`	násobenie
`číslo := číslo / číslo`	delenie
`číslo := číslo // číslo`	celočíselné delenie
`číslo := číslo \\ číslo`	zvyšok po celočíselnom delení
`číslo := číslo negated`	negácia čísla (násobenie -1)
`číslo := číslo abs`	absolútna hodnota čísla
`číslo := číslo sign`	znamienko, výsledok je z oboru hodnôt {-1, 0 , 1}
`číslo := číslo sqrt`	druhá odmocnina
`číslo := číslo truncated`	zaokrúhlenie smerom k 0 (odstránenie desatinej časti)
`číslo := číslo floor`	zaokrúhlenie smerom k negatívnemu nekonečnu
`číslo := číslo ceiling`	zaokrúhlenie smerom k pozitívnemu nekonečnu
`číslo := číslo rounded`	zaokrúhlenie
`číslo := číslo factorial`	faktoriál čísla
`číslo := integer1 gcd:integer2`	nájdenie najväčšieho spoločného deliteľa
`číslo := integer1 lcm:integer2`	nájdeme najmenšieho spoločného násobku
`číslo := číslo min:číslo`	Výber minima
`číslo := číslo max:číslo`	Výber maxima
`bool := číslo negative`	číslo je záporné
`bool := číslo even`	číslo je párne
`bool := číslo odd`	číslo je nepárne
`bool := číslo > číslo`	porovnanie čísiel
`bool := číslo >= číslo`	porovnanie čísiel
`bool := číslo < číslo`	porovnanie čísiel
`bool := číslo <= číslo`	porovnanie čísiel
`bool := číslo between:číslo1 číslo2`	číslo sa nachádza v intervale <`číslo1, číslo2`>

C.2 Znak

`bool := znak < znak`	porovnanie znakov (ascii hodnôt)
`bool := znak <= znak`	porovnanie znakov (ASCII hodnôt)
`bool := znak > znak`	porovnanie znakov (ASCII hodnôt)
`bool := znak >= znak`	porovnanie znakov (ASCII hodnôt)
`bool := znak isDigit`	test, či je znak číslo
`bool := znak isAlpha`	test, či je znak písmeno
`bool := znak isSpace`	test, či je znak znakom medzeri
`bool := znak isLower`	test, či je znak malé písmeno
`bool := znak isUpper`	test, či je znak veľké písmeno
`znak := znak asLower`	prevedenie znaku na malé písmeno
`znak := znak asUpper`	prevedenie znaku na veľké písmeno

C.3 Reťazec

`reťazec := reťazec , reťazec`	konkatenácia reťazcov
`reťazec := reťazec first: n`	vráti prefix reťazca po znak s indexom n
`reťazec := reťazec asLowercase`	prevedie všetky znaky v reťazci na malé
`reťazec := reťazec asUppercase`	prevedie všetky znaky v reťazci na veľké
`znak := reťazec at: číslo`	znak na pozícii `číslo`
`číslo := reťazec compare: reťazec`	porovnanie reťazcov
`číslo := reťazec indexOf: znak`	vráti prvú nájdenú pozíciu znaku
`číslo := reťazec size`	vráti veľkosť reťazca
`bool := reťazec includesSubstring: reťazec`	reťazec obsahuje reťazec
`bool := reťazec includes: znak`	reťazec obsahuje znak

C.4 Bool

`bool := bool not`	negácia
`bool := bool | bool`	logický súčet
`bool := bool & bool`	logický súčin
`bool := bool or: bool`	logický súčet
`bool := bool and: bool`	logický súčin

C.5 Symbol a nedefinovaný objekt

Primitívne objekty reprezentujúce symboly a nedefinovaný objekt nereagujú na žiadne správy okrem správ pre rovnosť identity a hodnoty.

Príloha D

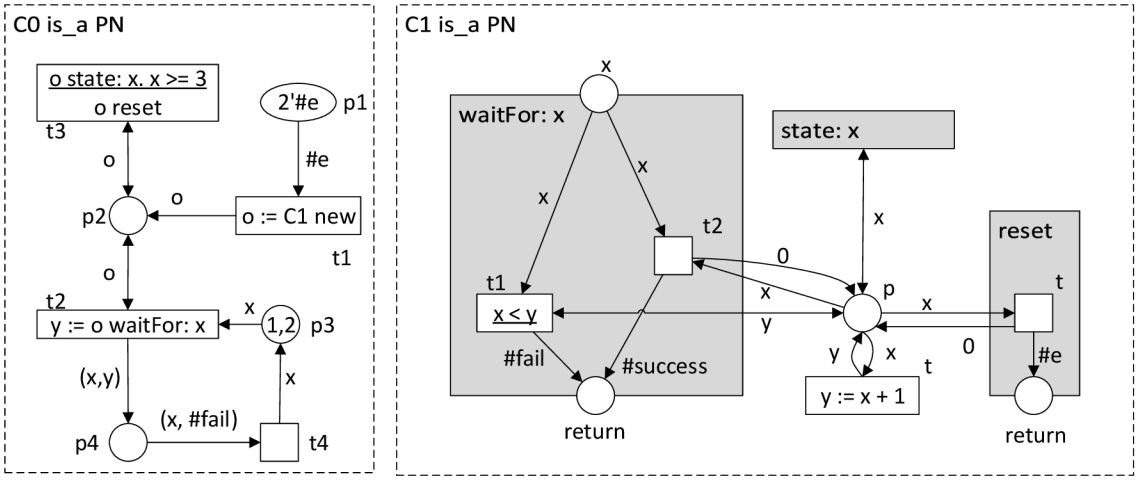
Jazyk PNtalk

Nasledujúca príloha je prebraná z [3].

Textová verzia jazyka PNtalk definuje syntax inskripčného jazyka pre stráž a akciu prechodu a taktiež popisuje štruktúru objektovo orientovanej Petriho siete. Textový popis štruktúry Petriho siete by obvykle mal byť automaticky generovaný nástrojom pre vizuálne programovanie OOPN. Syntax štruktúry sietí a celého systému tried formálne definujeme rozšírenou `Backus-Naurovou formou`. Zápis `[ ... ]` znamená, že „`...`“ sa môže vyskytnúť nepovinne, `[ ... ]*` znamená ľubovoľne násobný nepovinný výskyt „`...`“, `[ ... ]+` znamená výskyt „`...`“ raz alebo viackrát, `... | ... [ | ... ]*` znamená výber jednej z variant. Reťazce v úvodzovkách zodpovedajú lexikálnym symbolom. Prvotný symbol je `classes`.

```
classes: [class]* "main" id [class]*
class: "class" classhead [objectnet] [methodnet|constructor|sync]*
classhead: id "is a" id

objectnet: "object" net
methodnet: "method" message net
constructor: "constructor" message net
sync: "sync" message [cond] [precond] [guard] [postcond]
message: id | binsel id | [keysel id]+
net: [place|transition]*

place: "place" id"(" [initmarking] ")" [init]
init: "init" "{" initaction "}"
initmarking: multiset
initaction: [temps] exprs

transition: "trans" id [cond] [precond] [guard] [action] [postcond]
cond: "cond" id"(" arcexpr ")" ["," id"(" arcexpr ")"]*
precond: "precond" id"(" arcexpr ")" ["," id"(" arcexpr ")"]*
postcond: "postcond" id"(" arcexpr ")" ["," id"(" arcexpr ")"]*
guard: "guard" "{" expr3 "}"
action: "action" "{" [temps] exprs "}"
arcexpr: multiset
```

```
multiset: [n "?"] c ["," [n "?"] c]*
n: [dig]+ | id
c: literal | id | list
list: "(" [c ["," c]* ["|" [id | list] ]] ")"

temps: "|" [id]* "|"
unit: id | literal | "(" expr ")"
unaryexpr: unit [ id ]+
primary: unit | unaryexpr
exprs: [expr "."]* [expr]
expr: [id ":="]* expr2
expr2: primary | msgexpr [ ";" cascade ]*
expr3: primary | msgexpr
msgexpr: unaryexpr | binexpr | keyexpr
cascade: id | binmsg | keymsg
binexpr: primary binmsg [ binmsg ]*
binmsg: binsel primary
binsel: selchar[selchar]
keyexpr: keyexpr2 keymsg
keyexpr2: binexpr | primary
keymsg: [keysel expr2]+
keysel: id":"
literal: číslo | string | charconst | symconst | arrayconst
arrayconst: "#" array
array: "(" [číslo | string | symbol | array | charconst]* ")"
číslo: [-][[dig]+ "r"] [hexDig]+ ["." [hexDig]+] ["e"["-"][dig]+].
string: "?"[char]*"?"
charconst: "$"char
symconst: "#"symbol
symbol: id | binsel | keysel[keysel]* | string
id: letter[letter|dig]*
selchar: "+" | "-" | "*" | "/" | "~" | "|" | "," | "<" | ">" | selchar2
selchar2: "=" | "&" | "\" | "@" | "%" | "?" | "!"
hexDig: "0".."9" | "A".."F"
dig: "0".."9"
letter: "A".."Z" | "a".."z"
char: letter | dig | selchar | "[" | "]" | "{" | "}" | "(" | ")" | char2
char2: " " | "^" | ";" | "$" | "#" | ":" | "." | "-" | "?"
```



Obr. D.1: Príklad OOPN, počiatočná trieda je `C0`

Pre ilustráciu taktiež preberáme z [3] príklad OOPN špecifikovanej v jazyku PNtalk, a to graficky, viz. obr. D.1, a taktiež textovo:

```
main C0

class C0 is_a PN
  object
    trans t4
      precond p4((x, #fail))
      postcond p3(x)
    trans t2
      cond p2(o)
      precond p3(x)
      action {y := o waitFor: x}
      postcond p4((x, y))
    trans t1
      precond p1(#e)
      action {o := C1 new.}
      postcond p2(o)
    trans t3
      cond p2(o)
      guard {o state: x. x >= 3}
      action {o reset.}
    place p1(2'#e)
    place p2()
    place p4()
    place p3(1, 2)
```

```
class C1 is_a PN
  object
    place p(0)
    trans t
      precond p(x)
      action {y := x + 1.}
      postcond p(y)
  method waitFor: x
    place return()
    place x()
    trans t1
      cond p(y)
      precond x(x)
      guard {x < y}
      postcond return(#fail)
    trans t2
      precond x(x), p(x)
      postcond return(#success), p(0)
  method reset
    place return()
    trans t
      precond p(x)
      postcond return(#e), p(0)
  sync state: x
    cond p(x)
```

Príloha E

Medzikód virtuálneho stroja

Medzikód sa inšpiruje bajtkódmi, používa ale len tlačiteľné znaky, takže je možné, aby ho človek prečítal.

Medzikód je rozdelený na dve časti. V prvej časti popisuje statickú reprezentáciu OOPN a v druhej časti jej uložený stav. Stav siete nemusí byť popísaný, v takom prípade sa virtuálny stroj inicializuje a uvedie do počiatočného stavu.

Medzikód začína vždy siedmymi znakmi určujúcimi jeho verziu, pre prípad, že by došlo k vylepšeniu tohto medzikódu v budúcnosti. Tieto znaky tvoria reťazec `OOPNVM1.0`. Ďalej nasleduje statická časť medzikódu, ktorá pomocou prefixov definuje elementy OOPN, či už triedy, siete objektu, miesta a podobne. Za týmto znakom sú špecifikované atribúty tohto elementu a to tak, ako popisuje tabuľka E.1.

Tabuľka popisuje hranový výraz a výrazy pre akciu a stráž prechodu. O týchto výrazoch platí, že:

- **Hranový výraz** je v súlade s hranovými výrazmi jazyka PNTalk, viz. sekcia 5.1.4. Hranový výraz v medzikóde je avšak vždy upravený tak, aby neobsahoval žiadne prázdne znaky a násobnosti výskytov sú vždy uvedené. To znamená, že výraz v PNTalk zapísaný ako „`1, 2' #e`“ je prevedený pri preklade do medzikódu na tvar „`1'1,2'#e`“.
- **Výraz** má tvar `a:msg:p1:p2:`, kde `a` je adresát správy `msg` s parametrami `p1` a `p2`. Každý parameter je ukončený znakom `':'`.
- **Výraz stráže prechodu** sú výrazy oddelené znakom `'.'`.
- **Výraz akcie prechodu** má tvar `r:=a:msg:p1:p2:`, kde `r` je premenná špecifikujúca kam sa má výsledok volania správy uložiť alebo je to term `nil` v prípade, že sa má výsledok ignorovať.

Za statickou časťou nasleduje dynamická časť. Tieto dve časti sú od seba oddelené znakom nového riadku a dynamická časť je uvedená znakom `D`. Ďalej, podobne ako u statickej časti, nasledujú znaky, ktoré ako prefixy popisujú všetky užívateľom definované objekty a stavy všetkých miest a prechodov v OOPN. Tieto prefixy a ich syntax je popísaný pomocou tabuľky E.2.

Dynamická časť najprv uvedie zoznam užívateľom definovaných objektov, ktoré virtuálny stroj vytvorí, potom nasledujú jednotlivé inštancie sietí, či už objektu alebo metód spolu so stavmi miest nachádzajúcich sa v týchto inštanciách sietí. Nakoniec je uvedený stav jednotlivých prechodov v OOPN.

Stav miesta je zoznam textovej reprezentácie objektov v mieste oddelených znakom `','` uzatvorených do množinových zátvoriek.

Stav prechodu môže byť jedného z dvoch druhov. Buď je v stave typu `'s'`, kedy čaká na dokončenie nejakej inštancie siete metódy alebo je v stave `'j'`, kedy čaká na vykonanie udalosti typu J. V prvom prípade je uvedená identita inštancie siete metódy, na ktorej dokončenie prechod čaká, pomocou dvojice `oid:nid` a ďalej je uvedené naviazanie premenných, ktoré viedlo k invokácii tejto siete metódy. Naviazanie premenných je uzatvorené do množinových zátvoriek a tvoria ho dvojice `premenná, naviazaný objekt` oddelené znakom `'|'`.

V druhom prípade nie je uvedená identita vyvolanej inštancie siete metódy, pretože tá už neexistuje, bola dokončená. Je uvedené len naviazanie premenných, ktoré zároveň obsahuje aj výsledok vyvolanej metódy. Pre ilustráciu uvádzame príklad reprezentácie stavu prechodu.

```
Roid0:nid0:t2:s:oid1:nid1:{|o,oid1|};Roid0:nid0:t2:j:{|x,5.000000|o,oid2|};
```

Prechod `t2` v inštancii siete `oid0:nid0` čaká na dokončenie inštancie siete metódy `oid1:nid1`, ktorá bola invokovaná pre naviazanie premenných `o = oid1`. Prechod `t2` v inštancii siete `oid0:nid0` ďalej čaká na vykonanie udalosti typu J pre naviazanie premenných `x = 5` a `o = oid2`.

Prefix	Atribúty	Slovný popis
`C`	`Cname:topName;`	Trieda `name`, ktorá dedí od `topName`. Objektová sieť a všetky siete metód definované ďalej v medzikóde patria do tejto triedy.
`O`	`O`	Sieť objektu. Všetky miesta a prechody definované ďalej v medzikóde patria do tejto siete.
`M`	`Mmethod:x1:x2::`	Metoda `method` s parametrami `x1` a `x2`. Posledný parameter je ukončený dvomi znakmi `::`. Všetky miesta a prechody definované ďalej v medzikóde patria do siete tejto metódy.
`S`	`Sport:x1::`	Synchrónny port `port` s parametrom `x1`. Posledný parameter je ukončený dvomi znakmi `::`. Hrany a stráž definované ďalej v medzikóde patria do tohto synchrónneho portu.
`P`	`Pname:arcexpr;`	Miesto s názvom `name` a počiatočným značením `arcexpr`, ktoré môže byť prázdne. Počiatočné značenie `arcexpr` je hranový výraz.
`T`	`Tname:`	Prechod s názvom `name`. Hrany, stráž a akcia prechodu špecifikované ďalej v medzikóde patria do tohto prechodu.
`i`	`iplace:arcexpr;`	Vstupná hrana spajajúca naposledy definovaný prechod a miesto `place` a hranový výraz `arcexpr` tejto hrany.
`o`	`oplace:arcexpr;`	Výstupná hrana spajajúca naposledy definovaný prechod a miesto `place` a hranový výraz `arcexpr` tejto hrany.
`t`	`tplace:arcexpr;`	Testovacia hrana spajajúca naposledy definovaný prechod a miesto `place` a hranový výraz `arcexpr` tejto hrany.
`a`	`a:expr;`	Akcia naposledy definovaného prechodu a výraz `expr` špecifikovaný v tejto akcii.
`g`	`g:expr;`	Stráž prechodu a výraz `expr` tejto stráže.

Tabuľka E.1: Statická časť medzikódu

Prefix	Atribúty	Slovný popis
`A`	`Aoid:class;`	Úživateľom definovaný objekt s oid `oid` ako inštancia triedy `class`.
`N`	`Noid:nid:class:method;`	Inštancia siete metódy, nachádzajúca sa v objekte s oid `oid`, s nid `nid` definovaná v triede `class` so selektorom správy `method`. Všetky stavy miest popísané ďalej v medzikóde patria to tejto inštancie siete.
`L`	`Lname:objects;`	Stav miesta s názvom `name` je určený zoznamom objektov, `objects` nachádzajúcich sa v tomto mieste.
`R`	`Roid:nid:name:state`	Prefix pre stav prechodu s názvom `name` nachádzajúcom sa v inštancii siete `oid:nid`, kde `state` určuje ako aký stav prechodu (`s` alebo `j`) sa jedná (viz. nižšie).
`s`	`s:oid:nid:bindings;`	Pokračovanie pre stav prechodu, kde `s` je `state`, a ktorý hovorí, že prechod čaká na dokončenie inštancie metódy `oid:nid`, ktorá bola invokovaná pre naviazanie premenných `bindings`.
`j`	`j:bindings;`	Pokračovanie pre stav prechodu, kde `j` je `state`, a ktorý hovorí, že prechod čaká na vykonanie udalosti typu J pre naviazanie premenných `bindings`.

Tabuľka E.2: Dynamická časť medzikódu