
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**
**BRNO UNIVERSITY OF TECHNOLOGY**

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**
**FACULTY OF INFORMATION TECHNOLOGY**

**ÚSTAV INTELIGENTNÍCH SYSTÉMŮ**
**DEPARTMENT OF INTELLIGENT SYSTEMS**


# GENEROVÁNÍ KÓDU Z MODELŮ PETRIHO SÍTÍ

**CODE GENERATION FROM OBJECT ORIENTED PETRI NETS**

**BAKALÁŘSKÁ PRÁCE**
**BACHELOR'S THESIS**


**AUTOR PRÁCE** — **MICHAL CIBÁK**
**AUTHOR**

**VEDOUCÍ PRÁCE** — **Ing. RADEK KOČÍ, Ph.D.**
**SUPERVISOR**



**BRNO 2023**

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

# Zadání bakalářské práce

149028

Ústav: Ústav inteligentních systémů (UITS)
Student: **Cibák Michal**
Program: Informační technologie
Specializace: Informační technologie

Název: **Generování kódu z modelů Petriho sítí**
Kategorie: Softwarové inženýrství
Akademický rok: 2022/23

Zadání:

1. Prostudujte problematiku generování zdrojových kódů z modelů softwarového systému. Prostudujte koncept formalismu Objektově orientovaných Petriho sítí (OOPN).
2. Seznamte se s aktuálním řešením generování kódu z OOPN do jazyka Java a proveďte analýzu tohoto řešení.
3. Navrhněte úpravy aktuálního řešení, příp. navrhněte nové řešení, transformace modelů popsaných formalismem OOPN do programovacího jazyka Java.
4. Implementujte nástroj pro generování zdrojových kódů Java z OOPN modelů, který bude respektovat navržené mechanismy transformace. Vytvořte sadu testovacích příkladů.
5. Analyzujte možné problémy a omezení spojená s transformací OOPN modelů do programovacích jazyků. Pro vybrané problémy formálně specifikujte jejich podstatu, důsledky a možná řešení.

Literatura:

- Krysztof Czarnecki, Ulrich Eisenecker. Generative Programming: Methods, Tools, and Applications. Addison-Wesley Professional, 2000. ISBN-13: 978-0201309775
- O. Ringert, A. Roth, B. Rumpe, A. Wortmann: Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. In: MORSE 2014 - 1st International Workshop on Model-Driven Robot Software Engineering

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Kočí Radek, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1.11.2022
Termín pro odevzdání: 10.5.2023
Datum schválení: 3.11.2022

# Abstrakt

Cieľom tejto práce je analyzovať aktuálne riešenie generátoru kódu z Objektovo orientovaných Petriho sietí zapísaných v jazyku PNtalk do jazyka Java, navrhnúť jeho úpravy a implementovať ich. Ako prvý bol analyzovaný celkový návrh a boli špecifikované chýbajúce časti, následne bol analyzovaný zdrojový kód prekladača a boli odhalené chyby. Po implementácií opráv týchto chýb boli navrhnuté zmeny pre prekladač a simulátor, aby podporovali chýbajúcu funkcionalitu. Tieto zmeny boli implementované.



# Abstract

The goal of this thesis is to analyze the current solution of a code generator from Object oriented Petri nets written in PNtalk language to Java language, propose changes and implement them. First the overall scheme was analyzed and missing parts were specified, then the source code of the compiler was analyzed and errors were found. After implementing the fixes for these errors, it was proposed to modify the compiler and simulator such that the missing functionality was supported. These changes were implemented.



# Kľúčové slová

OOPN, Objektovo orientované Petriho siete, PNtalk, Java, prekladač, parser, skener, generátor, simulátor

# Keywords

OOPN, Object oriented Petri nets, PNtalk, Java, compiler, parser, scanner, generator, simulator


# Citácia

CIBÁK, Michal. *Generování kódu z modelů Petriho sítí*. Brno, 2023. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Kočí, Ph.D.

# Generování kódu z modelů Petriho sítí

## Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením Ing. Radka Kočího, Ph.D. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Michal Cibák
10. mája 2023

## Poďakovanie

Ďakujem vedúcemu tejto práce, Ing. Radkovi Kočímu, Ph.D., za poskytnuté konzultácie a rady, ktoré mi pomohli pri jej vypracovaní.

# Obsah

# Kapitola 1

# Úvod

Petriho siete sú jednoduchým formalizmom využívaným na modelovanie diskrétnych systémov. Ich výhodou je zrozumiteľná grafická podoba a dobrá formálna analyzovateľnosť. Ich použitie však nie je vhodné pri rozsiahlejších či podrobných modeloch systémov. Na tento účel sú vhodnejšie obecné programovacie jazyky, ktorých vývoj viedol na zavedenie objektovej orientácie.

Vhodným spojením Petriho sietí a objektovej orientácie je možné získať pozitívne vlastnosti oboch prístupov. Takýchto pokusov bolo viacero, každý výsledok mal svoje výhody a nevýhody. Jedným z nich sú Objektovo orientované Petriho siete, skrátene označované OOPN, ktorých formálna definícia je inšpirovaná čistou objektovou orientáciou jazyka Smalltalk. Tento formalizmus, ako aj jeho konkrétna implementácia v Smalltalku v podobe jazyka a systému PNtalk, sú popísané v dizertačnej práci docenta Janouška [3].

Pokusu priniesť tento formalizmus do moderného programovacieho jazyka sa venoval Tomáš Fryč vo svojej bakalárskej práci [2]. V rámci nej bol navrhnutý spôsob konverzie modelu z jazyka PNtalk do tried v jazyku Java, implementovaný prekladač, ktorý tento prevod zabezpečuje, a simulátor, ktorý dokáže simulovať beh prevedeného modelu. Jedná sa však len o čiastočné riešenie, ktoré demonštruje spôsob prekladu a simulácie modelu na zjednodušenej úrovni. Niektoré syntaktické konštrukcie jazyka PNtalk sú vynechané a preto nie je možné toto riešenie použiť na zložitejšie modely.

Táto práca nadväzuje na spomínané riešenie prekladu modelov z jazyka PNTalk do jazyka Java. Jej cieľom je toto riešenie upraviť a rozšíriť doplnením chýbajúcich častí a dostať ho tým bližšie do podoby reálne využiteľného produktu.

Ako prvé sú v kapitole 2 predstavené koncepty Objektovo orientovaných Petriho sietí. Nasleduje základný popis jazyka PNTalk. V kapitole 3 je v jednoduchosti popísaný výsledok predošlej práce, na ktorú táto práca naväzuje. V štvrtej kapitole sú predstavené nedostatky predošlej práce, návrh riešenia a je popísaná jeho implementácia. Kapitola 5 popisuje, akým spôsobom bolo vykonávané testovanie. V kapitole 6 je záver.s

# Kapitola 2

# Objektovo orientované Petriho siete a PNtalk

Táto kapitola vychádza z dizertačnej práce Modelování objektů Petriho sítěmi [3]. Sú v nej popísané Petriho siete, skrátene označované ako PN (z anglického Petri nets), je predstavený formalizmus Objektovo orientovaných Petriho sietí, skrátene OOPN (z anglického Object oriented Petri nets), a jeho konkrétna implementácia nazývaná PNtalk.

## 2.1 Petriho siete

Populárnym prostriedkom na modelovanie diskrétnych systémov sú Petriho siete. Jedná sa o jednoduchý formalizmus, ktorý umožňuje ním popísaný model formálne analyzovať a taktiež previesť, teda simulovať jeho dynamické chovanie. Ich významnou vlastnosťou je aj jednoduchá a teda zrozumiteľná grafická podoba.

Pôvodná podoba Petriho sietí bola rôznymi spôsobmi rozšírená s cieľom zvýšenia úrovne popisu modelu a teda zvýšenia ich modelovacích schopností. Z tohto dôvodu existuje množstvo rôznych variant tohto formalizmu. Obecne je však ich štruktúra statická, pre rozsiahlejšie či nedostatočne abstraktné modely sa preto takáto sieť stáva obsiahlou a neprehľadnou.

Modelované systémy sú obecne dynamické, teda meniace sa v čase. Ich aktuálny stav, predstavujúci nejakú informáciu o systéme, sa vyjadruje stavovou premennou a zmena jej hodnoty v čase charakterizuje chovanie tohoto systému. Priebeh týchto hodnôt v čase sa nazýva proces. V diskrétnych systémoch sú tieto zmeny skokové a sú reakciou na výskyt nejakej udalosti. Čas v takomto systéme je často možné abstrahovať a vyjadriť ho v rámci sekvencie výskytu udalostí. Zložitejšie systémy sa často delia na menšie komponenty predstavujúce nejaký podsystém. Každý komponent má svoju stavovú premennú a prebieha v ňom proces. Stav celého systému je vyjadrený množinou stavov všetkých komponent, je teda rozdelený na parciálne stavy. Jednotlivé procesy prebiehajú paralelne, teda sú na sebe nezávislé. Pomocou Petriho sietí je možné vyjadriť ich vzájomnú interakciu.

Typ Petriho siete, ktorý bude ďalej predstavený, je označovaný ako PT-sieť (z anglického place-transition net), ktorá predstavuje zobecnenú CE-sieť (z anglického condition-event net). Ako napovedá názov, základom sú miesta, graficky znázornené ako kružnice, a prechody, graficky znázornené ako štvorec či obdĺžnik. Miesto modeluje parciálny stav systému a môže obsahovať značky, graficky znázornené ako bodky alebo číslo vyjadrujúce ich počet, ktorých umiestnenie v sieti predstavuje aktuálny stav systému. Prechody definujú vzory udalostí, ktoré môžu v systéme nastať, a sú spojené s miestami ohodnotenými orien-

tovanými hranami, graficky znázornené šípkami s číslom smerujúcimi od vstupného miesta k prechodu a od prechodu k výstupnému miestu, ktoré predstavujú vstupné a výstupné podmienky tohto prechodu. Vstupná podmienka je splnená vtedy, keď príslušné miesto obsahuje určený počet značiek podľa ohodnotenia danej hrany, pričom implicitná hodnota je 1. Prechody, ktorých všetky vstupné podmienky sú splnené, sú vykonateľné, teda môže nastať príslušná udalosť, ktorá ovplyvní určené parciálne stavy systému. Vykonanie prechodu je atomické a predstavuje odobratie určeného počtu značiek zo vstupných miest a pridanie určeného počtu značiek podľa ohodnotení hrán predstavujúcich výstupné podmienky do výstupných miest.

Viaceré prechody môžu byť medzi sebou konfliktné, pokiaľ sú súčasne vykonateľné a vykonanie jedného môže spôsobiť, že ďalšie vykonateľnými prestanú byť. Jedná sa o prechody, ktoré majú spoločné aspoň jedno vstupné miesto. Takéto prechody predstavujú nedeterminizmus v sieti, nakoľko nie je určené, ktorý prechod sa vykoná. Nedeterministické chovanie záleží aj na poradí vykonania nekonfliktných prechodov, nakoľko v aktuálnom stave systému ich môže byť vykonateľných viacero, následkom čoho sa môže systém dostať do niekoľkých rôznych stavov.

## 2.2 Objektovo orientované Petriho siete

Pre účely modelovania systému na nižšom stupni abstrakcie či rozsiahlejších systémov, ktoré by viedli na zložitú Petriho sieť, sú vhodnejšie obecné programovacie jazyky, tie však neposkytujú možnosť formálnej analýzy modelu. Vývoj v tejto oblasti viedol na definíciu objektovej orientácie, ktorá zavádza jazykové konštrukcie vhodné pre tvorbu rozsiahlejších systémov. Nejedná sa len o jej zavedenie do programovacích jazykov, ale aj do procesu tvorby programových systémov, teda do ich analýzy, návrhu a implementácie. O tom, že je tento prístup dobre použiteľný, svedčí popularita objektovo orientovaných programovacích jazykov, ako je napríklad C++ a Java. Takéto programovacie jazyky však objektovú orientáciu popisujú na veľmi nízkej úrovni, čo má za následok zlú formálnu analyzovateľnosť takto popísaných modelov. Objektovo orientované modely, ktoré sú výstupom analýzy a návrhu z procesu tvorby programových systémov, zase nie je možné simulovať.

Pre zachovanie vykonateľnosti modelu a jeho formálnej analyzovateľnosti je potrebný formalizmus objektovej orientácie na abstraktnejšej úrovni ako pri programovacích jazykoch, ktoré tento prístup využívajú. Vhodnou kombináciou Petriho sietí s princípmi objektovej orientácie môže viesť práve na takéto riešenie, ktoré zachováva výhody oboch prístupov. Takýchto pokusov bolo niekoľko, každý s iným výsledok. Jedným z nich je formalizmus OOPN, ktorý nadväzuje na vysokoúrovňové a hierarchické Petriho siete a je inšpirovaný čistou objektovou orientáciou v jazyku Smalltalk-80. Princíp tohto formalizmu bude ďalej bližšie predstavený.

### 2.2.1 Trieda a siete

Systém modelovaný prostredníctvom Objektovo orientovaných Petriho sietí je popísaný pomocou tried, ktoré sú počas celého behu systému konštantné, pričom jedna z nich je považovaná za počiatočnú. Trieda sa skladá z práve jednej siete objektu a z ľubovoľného počtu sietí metód a synchrónnych portov. Sieť objektu definuje reprezentáciu inštancie triedy, v ktorej môže prebiehať nezávislá aktivita ovplyvňujúca jej stav. Sieť metódy definuje reakciu na správu zasielanú z akcie prechodu, jej inštancia je vytvorená pri každom volaní metódy.

Synchrónny port definuje reakciu na synchrónnu správu zasielanú zo stráže prechodu, využívaný je na atomické otestovanie stavu siete objektu.

Tak ako Petriho siete spomínané v podkapitole 2.1, aj siete popísané v triede sa skladajú z miest a prechodov, ktoré sú spojené orientovanými hranami ohodnotenými hranovými výrazmi. Hrany opäť predstavujú vstupné a výstupné podmienky, navyše sú zavedené podmienky prechodu. Prechody boli rozšírené o stráž a akciu. Pretože metódy a synchrónne portu môžu byť spojené s miestami v sieti objektu, ich vykonanie môže ovplyvniť jej stav.

### 2.2.2 Dedičnosť

Dedičnosť umožňuje inkrementálne vytváranie tried, teda definíciu triedy doplnením odlišností od inej triedy, ktorá bude považovaná za jej rodičovskú triedu. OOPN poskytuje možnosť špecifikovať miesta a prechody spolu s ich príslušnými hranami v sieti objektu, a metódy, ktorými sa daná trieda líši od svojej rodičovskej triedy. Jedná sa však len o jednoduchú dedičnosť, teda nie je možné dediť od viacerých tried zároveň.

### 2.2.3 Objekt

V OOPN sú rozlišované tri druhy objektov. Samotné triedy sú objektom a umožňujú vytvárať svoje inštancie reakciou na správu new. Pre beh systému je inštanciovaná počiatočná trieda, takto vzniknutý objekt je označovaný ako prvotný. Aktuálny stav celého systému je vyjadrený množinou objektov v ňom.

Inštancia ľubovoľnej triedy sa označuje ako neprimitívny objekt a predstavuje množinu inštancií jej sietí. Jedna z nich je inštancia siete objektu a prípadné ďalšie sú inštancie sietí metód prislúchajúce rozpracovaným volaniam metód. Stav týchto sietí je vyjadrený rozmiestnením značiek v ich miestach a stavom ich prechodov. Pretože prechody v OOPN nemusia byť vykonané atomicky, aj tie môžu niesť stavovú informáciu predstavujúcu množinu inštancií sietí, ktoré boli vytvorené volaním metód v prechode a doposiaľ neukončené, a spôsob nadviazania premenných. Stavová informácia objektu teda môže byť menená v rámci reakcie na prichádzajúcu správu a vnútornou aktivitou objektu.

Primitívne objekty, ako sú napríklad čísla či reťazce, sú inštanciami primitívnych tried a predstavujú konštanty, teda je možné ich stotožniť s ich menami. Sú preto jazykovo dostupné ako literály, ich stav sa nemení, nevykonávajú žiadnu aktivitu a ich metódy predstavujú funkcie.

### 2.2.4 Miesta a značky

V miestach sa namiesto obecných značiek používaných v klasickej Petriho sieti vyskytujú značky reprezentujúce referencie na objekty, pričom sa môže jednať o primitívny objekt, identifikátor neprimitívneho objektu, názov triedy alebo n-ticu zloženú z predošlých typov značiek s možnosťou vnorených n-tíc. Jednotlivé značky sa teda od seba môžu líšiť.

Miesta v sieti objektu môžu byť prístupné aj mimo nej. Synchrónne porty sú s nimi spojené hranami, siete metód zas môžu niektoré z jej miest zdieľať. Vďaka tomuto prepojeniu môže vykonávanie metód a synchrónnych portov ovplyvniť stav siete objektu. Sieť metódy navyše obsahuje špeciálne miesto pomenované return, do ktorého je vložený výsledok vyhodnotenia správy, a pre správy s parametrami aj miesta, do ktorých sú pri jej volaní tieto parametre vložené.

### 2.2.5 Prechody

#### Hranový výraz

Rozdiel v definícii značky vedie aj na odlišný hranový výraz, ktorý predstavuje multimnožinu prvkov. Jedná sa o zložený výraz, ktorého časti vyjadrujú počet výskytov určitej značky. Môže obsahovať aj voľné premenné, na ktoré sa v rámci vyhodnocovania prechodu značky nadväzujú. Možností nadviazania môže byť viacero, čo má za následok nedeterministický vývoj systému.

#### Hrana

Hrany, ohodnotené hranovým výrazom, určujú, aké značky sa majú pri prevedení prechodu z miest odobrať v prípade vstupných podmienok a aké sa majú do miest pridať v prípade výstupných podmienok. Hrana predstavujúca podmienku prechodu, značená obojstrannou šípkou, je tiež ohodnotená hranovým výrazom a predstavuje kombináciu takto ohodnotenej vstupnej a výstupnej hrany, značky ale z miest neodoberá a ani ich do miest nepridáva, využíva sa teda ako testovacia hrana.

#### Stráž prechodu

Samotný hranový výraz na vstupnej hrane predstavuje podmienku kladenú na miesto v podobe požadovaného počtu určitých značiek. V prípade, že sa v tomto výraze vyskytujú premenné, môžu byť kladené dodatočné podmienky na to, aké značky sa na ne nadviažu, aby bol prechod považovaný za vykonateľný. Tieto podmienky sa špecifikujú v stráži prechodu, čo je sekvencia výrazov predstavujúcich zasielanie správ. Stráž je splnená práve vtedy, keď sú všetky jej výrazy vyhodnotené ako pravdivé. Pokiaľ je adresátom správy primitívny objekt, výraz je vyhodnotený v rámci inskripčného jazyka, v prípade neprimitívneho objektu s ním dochádza k synchrónnej komunikácii. V takom prípade musí byť príslušný synchrónny port vykonateľný a nebyť v konflikte s prechodom, z ktorého je volaný, a ani s prípadnými ďalšími synchrónnymi portami, ktoré sa zúčastňujú synchrónnej komunikácie.

#### Akcia prechodu

Akcia prechodu je výraz, ktorý predstavuje zaslanie správy pri vykonávaní prechodu. Môže sa jednať o primitívne zaslanie správy, vytvorenie nového neprimitívneho objektu alebo o invokáciu metódy objektu, pri ktorej sa čaká na výsledok a teda prechod nie je vykonaný atomicky. Na rozdiel od stráže prechodu sa výsledok zaslania správy nekontroluje, ale je možné ho priradiť do premennej, ktorá je ďalej využitá v hranovom výraze na výstupnej hrane.

#### Synchrónny port

Synchrónny port má podobu prechodu spojeného hranami s miestami v sieti objektu, ktorý je možné volať zaslaním správy zo stráže prechodu. Môže obsahovať stráž prechodu, na rozdiel od obyčajného prechodu však nemôže obsahovať akciu prechodu, pretože musí byť vykonateľný atomicky, čo pri akcii nie je zaručené. Pokiaľ je volajúci prechod vykonaný, synchrónne sa vykoná aj volaný synchrónny port v sieti objektu adresáta správy, v žiadnom inom prípade sa nevykonáva.

#### Vykonateľnosť a vykonanie prechodu, udalosti

Vykonateľnosť prechodu je podmienená vstupnými podmienkami a strážou prechodu. Vstupné miesta teda musia obsahovať požadované značky a stráž musí byť vyhodnotená ako pravdivá. Pre prípadné voľné premenné v týchto výrazoch to znamená, že musí existovať také nadviazanie premenných, aby vstupné podmienky aj stráž boli splnené, pričom pre každé takéto nadviazanie je prechod vykonateľný.

Vykonanie prechodu predstavuje odobratie značiek zo vstupných miest, vyhodnotenie akcie prechodu a vloženie značiek do výstupných miest. Pokiaľ je v stráži prechodu špecifikovaná synchrónna komunikácia, je súčasne vykonaný aj príslušný synchrónny port. Zmeny v stave systému sú teda spôsobené vykonávaním prechodov a odvíjajú sa od akcie prechodu, pričom je možné rozlíšiť štyri typy udalostí, ktoré sa vykonávajú atomicky.

Udalosť typu A (atomic) predstavuje atomické vykonanie prechodu. Dochádza k nej pri zaslaní ľubovoľnej správy primitívnemu objektu. Udalosť typu N (new) predstavuje vytvorenie nového objektu vykonaním prechodu, taktiež atomickým. Dochádza k nej pri zaslaní správy new triede. Pri zasielaní správy neprimitívnemu objektu dochádza najskôr k udalosti typu F (fork), predstavujúcej predanie správy, a následne udalosti typu J (join), predstavujúcej prijatie odpovedi. Prvá z nich zahŕňa uplatnenie vstupných podmienok prechodu, vytvorenie inštancie siete metódy prislúchajúcej uvedenému selektoru správy a umiestenie príslušných značiek podľa parametrov správy do jej parametrových miest. Jedná sa o neúplné vykonanie prechodu, pričom sa očakáva, že ho druhá udalosť dokončí. Tá môže nastať, keď sa v mieste return v inštancii invokovanej siete objaví značka, a zahŕňa predanie výsledku dokončenej metódy prechodu, z ktorého bola volaná, zrušenie inštancie siete príslušnej metódy vrátane prípadných vnorených inštancií invokovaných sietí a uplatnenie výstupných podmienok.

Súčasťou udalostí typu A, N a F môže byť aj synchrónna komunikácia, pokiaľ bola špecifikovaná v stráži prechodu. Čakanie medzi udalosťou typu F a J má za následok, že príslušný prechod nemôže byť vykonaný atomicky. Počas tohto čakania sa môžu vykonávať ostatné prechody, alebo sa môže pokračovať vo vykonávaní už rozpracovaných prechodov.

#### Garbage-collector

Rušenie objektov, ktoré už naďalej nie sú priamo ani nepriamo dostupné z prvotného objektu, je riešené implicitne. Stará sa o to takzvaný garbage-collector, ktorý predstavuje funkciu volanú synchrónne s vyššie spomínanými udalosťami, teda pri každom prevedení prechodu. K zrušeniu objektu patrí aj rekurzívne zrušenie prípadných jemu prislúchajúcich rozpracovaných metód.

## 2.3 PNtalk

PNtalk je názov používaný ako pre jazyk, ktorý predstavuje konkrétnu implementáciu formalizmu OOPN, tak aj pre systém, ktorý umožňuje s modelmi popísanými týmto jazykom pracovať. Jazyk PNtalk pre popisy Objektovo orientovaných Petriho sietí využíva jazyk Smalltalk (z čoho vychádza jeho názov). Voči definícii OOPN má niekoľko syntaktických odlišností, vylepšenia uľahčujúce zápis niektorých zložitejších konštrukcií a konkretizuje niektoré časti formalizmu, v ktorých bola ponechaná voľnosť. Siete umožňuje zapísať ako v grafickej, tak aj v textovej podobe. Bližšie popísaný bude v nasledujúcich podkapitolách, celá syntax jeho textovej podoby je popísaná v prílohe A. Systém PNtalk bol implemento-

vaný v dvoch verziách. Pôvodná implementácia mala za účel demonštrovať princípy OOPN, avšak na praktické použitie nebola príliš vhodná [1]. V rámci ďalšieho vývoja bola implementovaná nová verzia, ktorá mala za cieľ odstrániť nedostatky, aby bol produkt reálne použiteľný [2]. Aktuálne riešenie [3] je dostupné pod systémom Pharo 2.0, ktorý predstavuje vývojové prostredie pre jazyk Smalltalk [4].

### 2.3.1 Trieda

Tak ako v definícií OOPN, model v jazyku PNtalk je tvorený triedami, z ktorých jedna je označená ako počiatočná, a zapisuje sa nasledovne:

```
[trieda]* ‘‘main’’ ID_pociatocnej_triedy [trieda]*
```

Pri zahájení simulácie modelu je počiatočná trieda implicitne inštanciovaná. Jednotlivé triedy sú tvorené sieťou objektu a ľubovoľným počtom sietí metód, synchrónnych portov a navyše od OOPN aj konštruktorov. Každá trieda nesie aj označenie triedy, z ktorej priamo dedí – jej jednotlivé časti predstavujú buď novú definíciu, alebo predefinovanie zdedenej časti. Trieda sa zapisuje ako:

```
‘‘class’’ ID_triedy ‘‘is a’’ ID_nadradenej_triedy
[siet_objektu] [metoda|konstruktor|synchronny_port]*
```

#### Sieť objektu

Je zložená z miest a prechodov:

```
‘‘object’’ [miesto|prechod]*
```

#### Metóda

Predstavuje sieť, navyše má uvedený vzor správy, na ktorú reaguje inštanciáciou tejto siete. Túto správu možno poslať iba objektu.

```
‘‘method’’ vzor_spravy [miesto|prechod]*
```

#### Konštruktor

Navyše od definície OOPN prináša PNTalk možnosť mať mimo implicitného konštruktora new aj neimplicitné, užívateľom definované konštruktory. Definujú sa rovnakým spôsobom ako metódy triedy, ale používajú iné kľúčové slovo. Ich význam je rovnaký, ako obyčajná metóda, rozdiel je v tom, že takúto správu možno zaslať nie len už vytvorenému objektu, ale aj triede. V takom prípade sa najskôr vytvorí nový objekt zaslaním správy new a následne sa tomuto objektu pošle správa odpovedajúca tomuto konštruktoru, ako by sa jednalo o obyčajnú metódu. Jedná sa teda len o syntaktické vylepšenie jazyka, ktoré pre užívateľom definované triedy umožňuje po vytvorení objektu automatické zaslanie požadovanej správy.

```
‘‘constructor’’ vzor_spravy [miesto|prechod]*
```

[1] http://perchta.fit.vutbr.cz/projekty/2
[2] http://perchta.fit.vutbr.cz /projekty/12
[3] http://perchta.fit.vutbr.cz/pntalk2k/1
[4] https://pharo.org/about

**Synchrónny port**

Má podobu prechodu, má však zadaný aj vzor správy, na ktorú reaguje.

```
‘‘sync’’ vzor_spravy [podmienka_prechodu] [vstupna_podmienka]
[straz_prechodu] [vystupna_podmienka]
```

### 2.3.2 Dedičnosť

V hierarchii dedičnosti tried PNtalku má každá trieda práve jedného predchodcu. Jej vrchol je tvorený abstraktnou triedou Object, ktorá definuje reakcie na správy, ktorým musia rozumieť všetky objekty. Jedná sa napríklad o správy == a ∼==, ktoré slúžia na porovnanie identity objektov. Priamymi následníkmi triedy Object sú triedy primitívnych objektov (konštánt) PNtalku, z ktorých niektoré vybrané sú v PNtalku dostupné ako literály, a abstraktná trieda PN. Tá tvorí vrchol hierarchie tried popísaných Petriho sieťami, sama nimi však popísaná nie je. Nadtriedou každej užívateľom definovanej triedy je teda buď trieda PN, alebo iná ním už definovaná trieda. Triedy od svojej nadtriedy dedia celú jej štruktúru, teda sieť objektu, metódy, konštruktory a synchrónne porty, pričom všetky, s výnimkou metód zdedených z triedy Object, možno predefinovať.

Predefinovanie siete objektu predstavuje predefinovanie jej jednotlivých častí, teda uvedenie nových definícií zdedených miest a prechodov, pre ktoré je požadovaná zmena, a definovanie nových miest a prechodov. Miesta a prechody sa môžu predefinovať nezávisle na sebe, pri prechode sú však zahrnuté aj jeho hrany. Zdedené metódy, konštruktory a synchrónne porty sa nepredefinujú po častiach, ale vždy celé uvedením novej definície pre príslušný selektor správy.

### 2.3.3 Miesta a značky

Každé miesto má svoje meno a môže obsahovať počiatočné značenie predstavujúce značky, ktoré sú doň umiestnené pri zahájení simulácie modelu. Navyše od definície OOPN môže toto značenie obsahovať aj premenné, pre ktoré bola za účelom vyčíslenia ich hodnôt zavedená počiatočná akcia. Miesto sa zapisuje nasledovne:

```
‘‘place’’ ID_miesta‘‘(’’ [pociatocne_znacenie] ‘‘)’’
[‘‘init’’ ‘‘{’’ pociatocna_akcia ‘‘}’’]
```

V grafickej podobe sa značí elipsou s počiatočným značením a akciou zapísanými vo vnútri, jeho meno mimo nej. Pokiaľ miesto obsahuje aj počiatočnú akciu, táto je zapísaná pod jeho značením, pričom sú od seba oddelené vodorovnou čiarou. Meno slúži na referovanie miesta z prechodov a synchrónnych portov, konkrétne z ich podmienok (predstavujúcich hrany a ich výrazy), preto môže byť v grafickej podobe vynechané. Toto meno by malo byť unikátne v rámci príslušnej triedy.

Značka je vyjadrená buď termom, ktorý reprezentuje objekt PNtalku, alebo zoznamom. Zoznam predstavuje n-ticu, ktorá môže obsahovať ako termy, tak vnorený zoznam, pričom PNtalk ho umožňuje zapísať ako v Prologu, teda s oddeleným koncom zoznamu za zvislou čiarou:

```
‘‘(’’ [znacka [‘‘,’’ znacka]* [‘‘|’’ [premenna | zoznam] ]] ‘‘)’’
```

Term je najjednoduchším výrazom PNtalku.
Termy:

- Literál
  - Číslo
  - Znak
  - Reťazec uzavretý v apostrofoch s možnosťou zdvojeného apostrofu vo vnútri (kvôli rozlíšeniu znaku a koncu reťazca)
  - Symbol
  - Booleovská konštanta true alebo false
  - Nedefinovaný objekt nil
- Premenná, začína malým písmenom
- Pseudopremenná self a super
- Meno triedy, začína veľkým písmenom

Počiatočné značenie predstavuje multimnožinu značiek zapísanú ako:

```
[pocet "‘"] znacka [‘‘,’’[pocet "‘"] znacka]*
```

Počet značiek je vyjadrený termom, ktorý by mal byť vyhodnotiteľný za nezáporné celé číslo, v opačnom prípade sa berie ako 0. Ako oddeľovač je použitá spätná čiarka (takzvaný backtick).

Počiatočná akcia je výraz obdobný akcii prechodu. Táto akcia sa dá využiť na priradenie objektu do miesta.

### 2.3.4 Prechody

Prechod sa zapisuje nasledovne:

```
‘‘trans’’ ID_prechodu [podmienka_prechodu] [vstupna_podmienka]
[straz_prechodu] [akcia_prechodu] [vystupna_podmienka]
```

Hrana predstavuje podmienky prechodu, vyskytuje sa na nej hranový výraz, teda multimnožina.

Vstupná podmienka:

```
‘‘precond’’ ID_miesta‘‘(’’ hranovy_vyraz ‘‘)’’
[‘‘,’’ ID_miesta‘‘(’’ hranovy_vyraz ‘‘)’’]*
```

Výstupná podmienka:

```
‘‘postcond’’ ID_miesta‘‘(’’ hranovy_vyraz ‘‘)’’
[‘‘,’’ ID_miesta‘‘(’’ hranovy_vyraz ‘‘)’’]*
```

Podmienka prechodu:

```
‘‘cond’’ ID_miesta‘‘(’’ hranovy_vyraz ‘‘)’’
[‘‘," ID_miesta‘‘(’’ hranovy_vyraz ‘‘)’’]*
```

Stráž prechodu

```
‘‘guard’’ ‘‘{’’ vyraz_straze ‘‘}’’
```

Akcia prechodu:

```
‘‘action’’ ‘‘{’’ vyraz_akcie ‘‘}’’
```

Výraz akcie prechodu môže obsahovať dočasné premenné, výrazy priradenia do premenných, zasielanie správ a kaskádované správy.

# Kapitola 3

# Aktuálne riešenie

V rámci bakalárskej práce Generování kódu z modelů Petriho sítí [2] bolo navrhnuté a implementované riešenie konverzie modelu systému z jazyka PNtalk do jazyka Java a jeho následnej simulácie. Jedná sa o prekladač, ktorý generuje zdrojové súbory predstavujúce model ekvivalentný k Objektovo orientovanej Petriho sieti zo vstupu, a simulátor, ktorý dokáže simulovať beh takto vygenerovaného modelu, oboje implementované taktiež v jazyku Java. Toto riešenie demonštruje navrhnutý princíp prekladu a simulácie, jeho využitie v praxi je však do určitej miery obmedzené, pretože nepokrýva niektoré dôležité možnosti jazyka PNtalk. Poskytuje však základ, na ktorý možno nadviazať. Táto kapitola vychádza zo spomínanej práce, a to ako z písomnej časti, tak zo samotnej implementácie, pričom v prípadoch nezhody medzi nimi je uprednostnená implementácia. Sú v nej predstavené časti riešenia, ktoré sú významné pre ďalšiu prácu s ním.

## 3.1 Štruktúra aplikácie a jej použitie

Aplikácia je rozdelená do 3 logických častí:

- Prekladač zdrojových súborov písaných v jazyku PNtalk, ktoré popisujú Objektovo orientované Petriho siete [3], do jazyka Java. Skladá sa z parsera a generátora, ktoré sú tvorené niekoľkými triedami, z toho najpodstatnejšie sú:
  - PNParser – riadi preklad zdrojového textu
  - PNScanner – získava požadované časti zdrojového textu
  - PNSyntaxValidator – kontroluje syntaktickú správnosť vybraných častí zdrojového textu
  - PNGenerator – generuje z vnútornej štruktúry Java súbory
- Knižnica simulátoru, ktorá zabezpečuje beh preloženého modelu.
- Spúšťacia aplikácia, ktorá vzniká spojením preloženého modelu a simulátora.

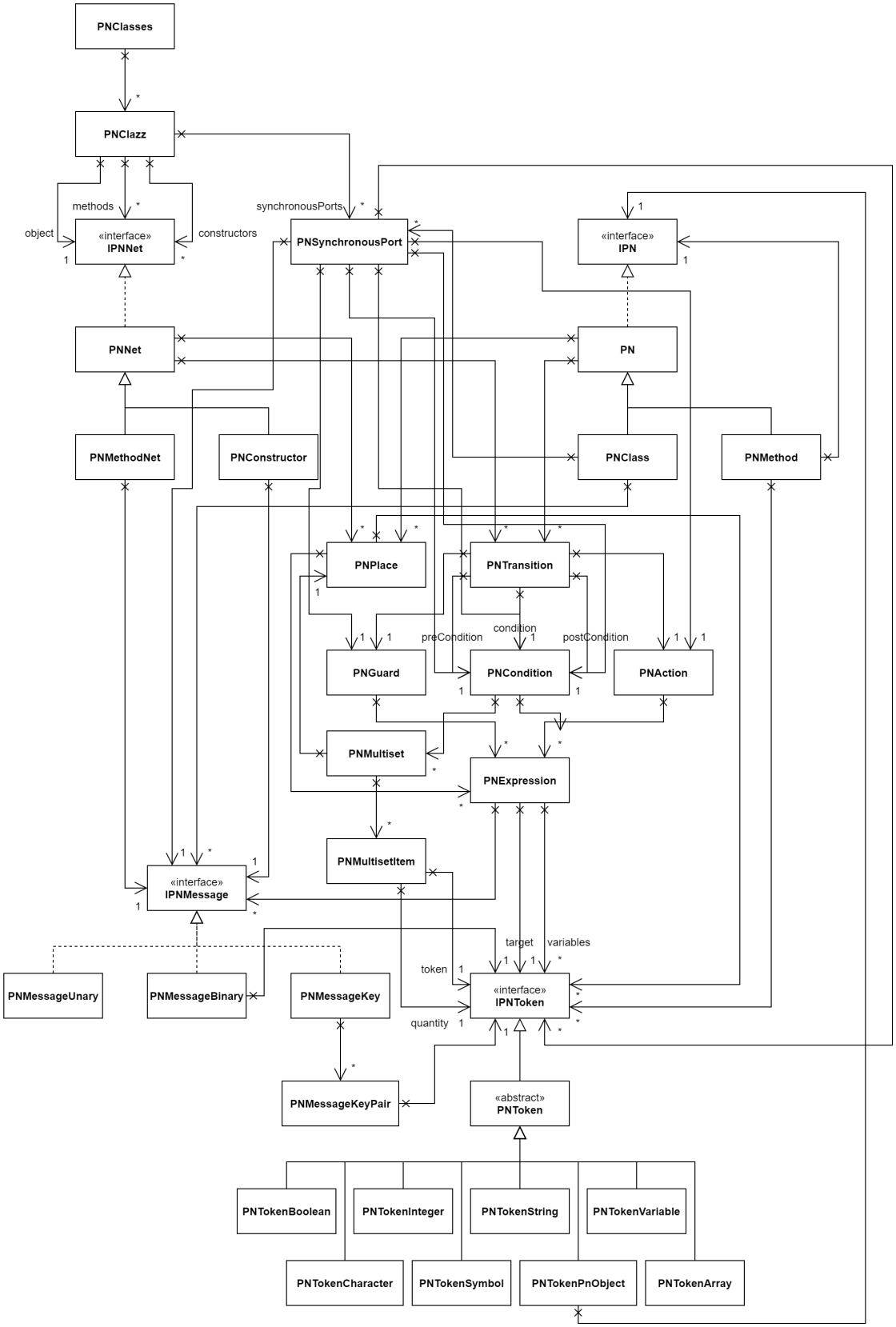
Na preklad a spustenie aplikácie sa využíva nástroj Ant [1]. Jedná sa o nástroj spoločnosti Apache podobný nástroju Make, ktorý sa využíva na zostavovanie aplikácií. Za týmto účelom využíva konfiguračný súbor vo formáte XML. Tento nástroj je napísaný v Jave, vďaka čomu je platformovo nezávislý.

Preklad aplikácie kompilátora do zložky build prebieha pomocou príkazu ant compile volanému zo zložky PNCompiler. Následné vytvorenie jar súboru, ktorý zároveň slúži ako

knižnica pre simulátor, prebieha pomocou príkazu **ant jar**, výsledok je PNCompiler.jar vložený do zložky dest. Preklad jednotlivých modelov OOPN potom prebieha pomocou **ant run -Darg0=cesta_k_súboru**. Príklady sa nachádzajú v zložke examples. Vygenerované súbory modelu, teda jednotlivé triedy a metódy, sa uložia do zložky **generated/model/názov_modelu**. Pre vytvorenie knižnice simulátora je najskôr potrebné presunúť vytvorenú knižnicu kompilátora PNCompiler.jar do zložky **PNSimulator/lib**. Následne sa v zložke PNSimulator zavolá ant jar, čím sa v zložke dest vytvorí súbor PNSimulator.jar. Príkaz ant compile nie je potrebné použiť, je totiž zahrnutý pri vytváraní knižnice. V zložke PNModelApp je následne možné spúšťať model. Najskôr je však potrebné vytvorenú knižnicu simulátora presunúť do zložky lib a vygenerované modely (celé zložky) presunúť do zložky src/model. Zdrojové súbory modelov a aplikácie sa následne preložia pomocou príkazu ant compile do zložky build/model. Simuláciu jednotlivých modelov je potom možné spustiť pomocou ant run **-Darg0=názov_modelu**.

## 3.2 Štruktúra tried modelu

Podľa definície jazyka PNtalk sú navrhnuté triedy, ktoré reprezentujú model OOPN. Väčšina z nich je využívaná ako pre vnútornú reprezentáciu modelu v prekladači, tak aj v rámci vygenerovaných zdrojových súborov, s ktorými pracuje simulátor, niektoré sú však určené výhradne pre jedno alebo druhé. V nasledujúcich podkapitolách je uvedený ich bližší popis.




Obr. 3.1: Diagram tried modelu.

### 3.2.1 Triedy pre prekladač

V tejto časti sú predstavené triedy a rozhranie, ktoré sú využívané iba v prekladači.

#### PNClasses

PNClasses predstavuje vrchol hierarchie vnútornej reprezentácie OOPN v prekladači. V jej inštancii je uchovaný celý model ako zoznam všetkých tried modelu a meno hlavnej triedy.

#### PNClazz

PNClazz predstavuje jednu triedu z modelu. V jej inštancii sú uložené dáta potrebné pre vygenerovanie zdrojových súborov v jazyku Java. Meno triedy sa využíva na začiatku názvu príslušných súborov, meno rodičovskej triedy je potrebné pre riešenie prípadnej dedičnosti, objektová sieť predstavuje buď celú sieť, alebo len zmeny voči sieti v rodičovskej triede v prípade dedenia, a nakoniec sú uchované zoznamy metód, konštruktorov a synchrónnych portov, ktoré sú buď nové, alebo nahrádzajú definíciu z rodičovskej triedy.

#### IPNNet, PNNet, PNMethodNet a PNConstructor

IPNNet je rozhranie pre akýkoľvek typ siete. PNNet predstavuje sieť zloženú z miest a prechodov, ktoré sú v jej inštancii uložené v zoznamoch, využívaná je pre objektovú sieť. PNMethodNet predstavuje sieť metódy, pričom dedí z PNNet a v inštancii má naviac správu, na ktorú reaguje. PNConstructor je obdobný PNMethodNet, ale predstavuje sieť konštruktora.

### 3.2.2 Triedy pre simulátor

V tejto časti sú predstavené triedy a rozhranie, ktoré sú využívané iba v simulátore a vygenerovaných súboroch, s ktorými pracuje. V týchto súboroch je zdrojový kód triedy, ktorá dedí z niektorej z týchto tried.

#### IPN, PN, PNClass, PNMethod

IPN je rozhranie pre sieť. PN je abstraktná trieda s atribútmi pre množinu miest a zoznam prechodov, slúži ako základ pre triedy PNClass a PNMethod, ktoré z nej dedia a ktoré sú použité ako rodičovské triedy vo vygenerovaných súboroch. PNClass predstavuje triedu OOPN a ako atribúty pridáva mapy pre synchrónne porty a metódy, ktoré jej prislúchajú. PNMethod predstavuje metódu v triede OOPN a ako atribúty pridáva triedu OOPN, ktorej patrí, a zoznam jej argumentov.

### 3.2.3 Spoločné triedy

V tejto časti sú predstavené triedy a rozhrania, ktoré sú využívané aj pre vnútornú reprezentáciu zdrojového textu v prekladači, aj vo vygenerovaných súboroch a v simulátore.

#### PNMultiset a PNMultisetItem

Trieda PNMultiset odpovedá v OOPN jednej hrane medzi miestom a prechodom spolu s jej hranovým výrazom. Vyjadruje, z akého miesta má odobrať značky alebo do akého miesta ich naopak pridať, čomu odpovedajú aj jej atribúty. Objekt tejto triedy v sebe má inštanciu

miesta, v ktorom vykonáva zmeny, a zoznam objektov typu PNMultisetItem, ktoré určujú, koľko akých značiek sa pridá alebo odstráni. V inštancii PNMultisetItem sú teda uložené dve značky, jedna vyjadruje množstvo a druhá typ.

### PNExpression

Táto trieda predstavuje jeden výraz z akcie alebo stráže prechodu. Jej inštancia obsahuje zoznam správ, značku, ktorej sú posielané a prípadne zoznam premenných, do ktorých je výsledok uložený.

### IPNMessage, PNMessageUnary, PNMessageBinary, PNMessageKey a PNMessageKeyPair

IPNMessage je rozhranie pre všetky typy správ. Správa vyjadruje, aká metóda alebo synchrónny port sa má zavolať a s akými argumentami. Trieda PNMessageUnary predstavuje unárnu správu, teda správu bez argumentov, v ktorej inštancii stačí uschovať selektor. Trieda PNMessageBinary predstavuje binárnu správu, teda správu s jedným argumentom, ktorá okrem selektora v inštancii má aj operand. Trieda PNMessageKey predstavuje kľúčovú správu, teda správu s potenciálne viacerými pomenovanými argumentami, ktorej inštancia obsahuje zoznam PNMessageKeyPair objektov. Tie obsahujú selektor a operand.

### IPNToken, PNToken, PNTokenBoolean, PNTokenInteger, PNTokenCharacter, PNTokenString, PNTokenSymbol, PNTokenVariable, PNTokenPnObject, PNTokenArray

IPNToken je rozhranie pre všetky typy značiek. PNToken je abstraktná trieda, ktorá predstavuje obecný typ značky, má teda aj obecný atribút typu Object pre uschovanie hodnoty danej značky. Triedy, ktoré predstavujú konkrétny typ značky, z tejto abstraktnej triedy dedia. PNTokenBoolean má v tomto atribúte objekt typu boolean, pre PNTokenInteger to je integer, PNTokenCharacter využíva String, tak ako aj PNTokenString, PNTokenSymbol a PNTokenVariable, PNTokenPnObject využíva tiež typ String, ktorý vyjadruje názov triedy z OOPN, ale pridáva ešte atribút pre uschovanie inštancie danej triedy. PNTokenArray predstavuje zoznam, tie ale nie sú podporované.

### PNPlace

Trieda PNPlace predstavuje miesto v sieti. V inštancii tejto triedy je uložené meno daného miesta, zoznam značiek, ktoré sa v ňom nachádzajú, a zoznam výrazov, ktoré slúžia ako inicializačná akcia, teda k počiatočnému značeniu ešte môžu pridať ďalšie značky.

### PNTransition, PNCondition, PNGuard a PNAction

Táto trieda predstavuje prechod v sieti. V jej inštancii sa uchováva meno daného prechodu, vstupná, výstupná a obyčajná podmienka, akcia, stráž a mapa premenných, ktoré využíva simulátor. PNCondition má ako atribút zoznam PNMultiset objektov, teda vyjadruje, na ktoré miesta bude mať aký vplyv. PNGuard aj PNAction majú ako atribút zoznam výrazov a odpovedajú stráži a akcii prechodu.

#### PNSynchronousPort

PNSynchronouPort je trieda, ktorá predstavuje synchrónny port. Ako atribúty má správu, na ktorú reaguje, vstupnú, výstupnú a obyčajnú podmienku, stráž a zoznam argumentov, ktoré sú využívané v simulátori.

## 3.3 Implementácia

Správne spracovanie Objektovo orientovanej Petriho siete zapísanej v jazyku PNtalk a naplnenie navrhnutej štruktúry popísanej v časti 3.2 je základom pre to, aby mohol byť vygenerovaný kód v jazyku Java, ktorý odpovedá pôvodnej OOPN. Toto spracovanie zabezpečuje parser. Je dôležité vedieť, ako je implementovaný, aby bolo možné identifikovať príslušné miesta, kde je potrebné doplniť chýbajúcu funkcionalitu. Výslednú vnútornú reprezentáciu takto spracovanej OOPN využije generátor pre vygenerovanie súborov v jazyku Java, čím končí úloha prekladača.

### 3.3.1 Scanner

Scanner je implementovaný ako abstraktná trieda so statickými metódami a stará sa o delenie zdrojového textu v jazyku PNtalk na časti podľa jeho štruktúry. Na rozdiel od typického scanneru [4] neobsahuje metódu, ktorá by vracala token odpovedajúci vždy prvej lexikálnej jednotke na vstupe. Namiesto toho je kód členený do metód odpovedajúcich štruktúre OOPN. Týmto metódam je v argumente predaná nimi očakávaná časť zdrojového textu, v ktorej vyhľadajú požadovanú časť alebo viaceré časti, uložia ich do zoznamu a vrátia v objekte typu PNScannerData, ktorý okrem tohto zoznamu obsahuje aj pôvodný reťazec z argumentu metódy upravený tak, že nájdené podreťazce sú z neho odstránené. Parser teda volá metódy scanneru podľa toho, ktorú časť zdrojového textu práve spracúva. Prvá metóda z celého zdrojového textu vystrihne deklaráciu hlavnej triedy. Zvyšný text, predstavujúci triedy OOPN, očakáva na vstupe druhá metóda, ktorá ho rozdelí na jednotlivé definície tried. Ďalšie metódy z definície triedy získajú jej hlavičku, objektovú sieť, metódy a synchrónne porty, iné zo siete získajú miesta a prechody, obdobne sú z prechodu získané podmienky, akcia a stráž. Princíp je rovnaký aj pre ostatné časti. Vyhľadanie požadovanej časti pracuje tak, že sú nájdené kľúčové slová, ktoré ju ohraničujú. Napríklad deklarácia hlavnej triedy začína kľúčovým slovom main a končí pred definíciou triedy, ktorá začína kľúčovým slovom class. Tá zas končí pred začiatkom definície ďalšej triedy. Jej hlavička je na začiatku a končí pred telom, ktoré môže obsahovať objektovú sieť, metódy, konštruktory a synchrónne porty, uvádzané ich príslušnými kľúčovými slovami. Obdobne je to aj pre ďalšie časti zdrojového textu, viac o kľúčových slovách je možné nájsť v syntaxi jazyka PNtalk v prílohe A. Funkcionalita vyhľadania a vystrihnutia podreťazca podľa kľúčových slov je implementovaná v dvoch obecných metódach, a to getDataByDelimiters a getContentAsListByKeywords, výber časti pred kľúčovým slovom je riešený individuálne v príslušných metódach. Metóda getDataByDelimiters zo zadaného reťazca vystrihne každú časť medzi dvoma poskytnutými oddeľovačmi, či medzi prvým oddeľovačom a koncom reťazca. Tieto oddeľovače by mali mať podobu regulárneho výrazu odpovedajúcemu jednému kľúčovému slovu alebo zoznamu kľúčových slov oddelených logickou operáciou or. Predpokladá sa, že prvý oddeľovač nebude zoznam, preto toto kľúčové slovo nie je ponechané ani v pôvodnom, ani vo výslednom vystrihnutom reťazci – nemalo by byť potrebné, keďže sa vie, že ním musel začínať. Metóda getContentAsListByKeywords je podobná, ale oddeľovače jej nie sú

predávané v argumentoch, využíva zoznam kľúčových slov z abstraktnej triedy PNScanner-Constants, v ktorej sú definované konštanty predstavujúce kľúčové slová a niektoré dôležité znaky podľa syntaxe jazyka PNtalk, ako aj regulárne výrazy predstavujúce vybrané zoznamy kľúčových slov. Je volaná tam, kde ako počiatočný a koncový oddeľovač môže byť viac kľúčových slov, teda pri získaní častí prechodu a synchrónneho portu. V jej prípade však musí byť počiatočné kľúčové slovo vo výslednom vystrihnutom reťazci ponechané, aby bolo možné neskôr identifikovať, o akú časť z niekoľkých možností sa v skutočnosti jedná. Okrem nich sú ešte v scanneri metódy na získanie prvej sekvencie nebielych znakov z reťazca, odstránenie prvého výskytu zadaného reťazca v reťazci (keďže sú reťazce v Jave nemenné, táto metóda vracia nový, upravený reťazec), získanie reťazca pred zadaným oddeľovačom a rozdelenie reťazca podľa bielych znakov (na rozdelenie sa nerátajú biele znaky, ktoré sú súčasťou PNtalkového literálu predstavujúceho reťazec).

### 3.3.2 Parser

Parser je implementovaný podľa návrhového vzoru singleton. Jeho kód je členený do metód tak, aby odpovedali štruktúre zdrojového textu v jazyku PNtalk. Na najvyššej úrovni je metóda spracúvajúca celý text, z ktorej sú volané metódy spracúvajúce jeho časti, ktoré volajú ďalšie metódy spracúvajú ich časti, až kým sa nezavolá metóda na najnižšej úrovni, v ktorej sa text ďalej nedelí. Každá metóda teda očakáva v argumente predaný určitý úsek zdrojového textu, ktorý spracuje a buď vráti nový objekt niektorej triedy popísanej v časti 3.2, ktorá prislúcha spracovávanému reťazcu, alebo naplní objekt, ktorý jej bol predaný ako argument. Získanie požadovanej časti reťazca je zabezpečené volaním príslušných metód zo scanneru. Tento podreťazec je potom buď predaný ďalšej metóde, alebo je rovno overená jeho syntaktická správnosť, a to buď volaním príslušnej statickej metódy z abstraktnej triedy PNSyntaxValidator pre literály, selektory správ a výrazy, alebo je kontrola vykonaná priamo v danej metóde pre iné časti. Niektoré reťazce môže byť potrebné rozdeliť na jednotlivé časti podľa určitého znaku, čo je vyriešené volaním metódy split s príslušným znakom. Podľa bodky sú od seba oddelené výrazy, podľa čiarky hranové výrazy a jednotlivé časti multisetov a podľa apostrofu počty značiek od ich hodnoty. Správnosť väčšiny kľúčových slov nie je kontrolovaná, pretože pokiaľ podľa nich bol určitý reťazec rozdelený, museli sa na danej pozícii nachádzať a teda aj byť správne. V prípade nájdenia chyby je vyhodená výnimka s dodatočnými informáciami. Spracovanie zdrojového textu začína v počiatočnej metóde, ktorej je predaný celý zdrojový text a jej výsledkom je naplnený objekt typu PNClasses, ktorý predstavuje túto Objektovo orientovanú Petriho sieť. V tomto texte sa vyhľadá deklarácia hlavnej triedy OOPN, predá sa metóde, ktorá overí jej syntaktickú správnosť, a zvyšok textu je rozdelený na definície jednotlivých tried. Každá je predaná metóde, ktorá sa postará o jej spracovanie a vráti objekt typu PNClazz. V nej sa vyhľadá hlavička, predá sa príslušnej metóde na jej spracovanie, rovnako pre objektovú sieť, metódy a synchrónne porty. Obdobne je to aj v týchto metódach a postupuje sa až dovtedy, kým sa nespracuje najmenšia časť, ktorej prislúcha niektorá trieda z časti 3.2.

# Kapitola 4

# Návrh úprav a implementácia

## 4.1 Analýza návrhu

### 4.1.1 Generované súbory reprezentujúce model

Generované Java súbory sú na úpravu nepraktické. V prípade nutnosti zmeny je lepšie túto spraviť v PNtalk zdrojovom texte, z ktorého tieto súbory boli vygenerované, a preložiť ho znovu. Je preto dôležité, aby prekladač fungoval správne a vedel odhaliť všetky chyby, pretože nie je zaručené, že zdrojové texty v PNtalku budú generované nástrojom a teda správne, ale môžu byť ručne upravené.

### 4.1.2 Návrhový vzor singleton

Ako bolo spomenuté v časti 3.3.2, parser je implementovaný podľa tohto návrhového vzoru. Podľa literatúry [5] sa tento vzor okrem iného zvykne používať aj v prípadoch, keď by mohla byť daná trieda implementovaná ako abstraktná (nemožno vytvárať inštancie) a metódy ako statické, čo je tento prípad. Význam to ale nemá, pretože výhody tohto vzoru, ako je možnosť dynamicky vybrať jednu z možných implementácií alebo predísť problémom s nevhodným poradím inicializácie, ktoré môžu nastať, pokiaľ je daná trieda prepojená s inými, sa v tomto prípade užitia neprejavia.

## 4.2 Základné spracovanie zdrojového súboru

V parseri bolo mnoho metód, ktoré delili zdrojový súbor na časti podľa určitých kritérií. Nebrali však ohľad ma možnosť výskytu vybraných oddeľovačov v reťazcoch či v zátvorkách. Riešením sú metódy, sú napísané obecnejším spôsobom, takže sú používané naprieč triedov vo väčšine metód. Jedna slúži na delenie textu podľa kľúčových slov, druhá na delenie podľa znaku, tretia získa časť pred vybraným slovom. Všetky berú ohľad na reťazce, symboly a zátvorky.

Na rozdiel od syntaxe jazyka PNTalk, tak ako bola formálne definovaná rozšírenou Backus-Naurovou formou v práci Modelování objektů Petriho sítěmi [3] a rovnako spracúvaná aj v rámci práce Generování kódu z modelů Petriho sítí [2], je pri spracúvaní v rámci syntaktickej kontroly braný ohľad aj na používanie správnych identifikátorov (s malým alebo veľkým počiatočným písmenom) na miestach, kde je jednoznačné, či sa tam môže vyskytovať identifikátor triedy alebo nie.

Ďalší rozdiel oproti pôvodnej implementácii je so znakmi používanými v multimnožinách a na označenie reťazcov. Tá používala špeciálne znaky ľavej a pravej hornej jednočiarkovej úvodzovky. Tieto znaky sa nachádzajú aj v spomínanej formálnej definícii syntaxe PNTalku, zdrojové súbory v ňom písané ale využívajú spätnú čiarku (takzvaný backtick) a apostrof. Spracovanie je teda upravené tak, aby pracovalo s týmito znakmi, zároveň je doplnená aj podpora zdvojeného apostrofu v reťazci, ktorý po spracovaní predstavuje jeden apostrof vo výslednom reťazci. Ďalšia práca s reťazcami už zmeny nevyžaduje.

V pôvodnej implementácii bolo spracovanie správ rovnaké ako pre ich definíciu, tak pre ich posielanie vo výraze. To je v poriadku, pokiaľ je vstupný zdrojový kód správny, definícia vzoru správy je totiž podkategória definície poslania správy vo výraze, akurát operandy môžu byť len premenné. Kvôli rozšíreniu syntaktickej kontroly však na tento rozdiel musí byť braný ohľad.

Kontrolu a spracovanie vzoru správy z definície metódy zabezpečujú metódy `parseUnaryMessageDef(String)`, `parseBinaryMessageDef(String)` a `parseKeyMessageDef(String)`, pre spracovanie správy vo výraze slúžia upravené pôvodné metódy `parseUnaryMessage(String)`, `parseBinaryMessage(String)` a `parseKeyMessage(String)`.

## 4.3 Chýbajúce typy tokenov

Rôzne typy značiek, ktoré sa môžu nachádzať v miestach, sú reprezentované rôznymi podtriedami triedy PNToken. V pôvodnej práci medzi nimi bola aj trieda PNTokenArray predstavujúca typ pole, pripravená pre budúce rozšírenie práce, v rámci nej však jeho spracovanie nebolo implementované. Okrem neho však zo základných typov značiek ešte chýbala podpora pre desatinné čísla, nil a zoznam.

Desatinné čísla sú reprezentované triedou PNTokenReal, ktorá je obdobná triede PNTokenInteger, iba typ uloženej hodnoty je iný. Po kontrole správnosti formátu reťazca, ktorý by mal predstavovať číslo, je využitá vstavaná funkcia jazyka Java Integer.parseInt(String) na pokus o prevod na celé číslo, pokiaľ toto zlyhá, nasleduje pokus o prevod na desatinné číslo pomocou Float.parseFloat(token). Z toho vyplýva, že nie sú podporované všetky formáty čísiel, a to aj vzhľadom k spôsobu, akým sa spracúva zdrojový súbor, teda že je rozdelený na postupne menšie časti pomocou rôznych oddeľovačov. Číslo by tak mohlo byť rozdelené na viac častí, napríklad pokiaľ by sa v ňom vyskytli medzery, a v takom prípade by bolo spracované nesprávne a spôsobilo by chybu. Generovanie prebieha pre oba typy rovnakým spôsobom. V simulátore je podpora numerických a booleovských operácií rozšírená z PNTokenInteger aj na PNTokenReal.

Literál nil je reprezentovaný triedou PNTokenNil, ktorá vďaka jeho jednoduchosti nenesie žiadnu hodnotu. Spracovanie v parseri spočíva len v porovnaní reťazca s hodnotou „nil“, pokiaľ reťazec odpovedal nejakému id, v generátore ide len o vytvorenie nového tokenu tohto typu. V simulátore je takýto token spracúvaný automaticky s ďalšími značkami.

Pseudopremenné self a super sú v parseri spracované ako obyčajná premenná, nie je pre ne samostatný typ tokenu. Keďže ich hodnota je závislá na kontexte, nie je potrebná ani úprava generátoru. Práca s nimi je teda vyriešená až v simulátore.

Pre prácu s typom pole je atribút value triedy PNTokenArray upravený rovnakým spôsobom, ako je to aj v iných triedach, teda je doplnený typ, v tomto prípade pole tokenov. Kvôli typu je upravená aj návratová hodnota metódy na získanie tohto zoznamu a sú pridané metódy na pridávanie a odstraňovanie tokenov, ktoré pracujú s týmto zoznamom. V generátore sa najskôr vytvorí nový PNTokenArray, následne sa originálnym zoznamom pre-

chádza token po tokene, každý sa vygeneruje a pridá do zoznamu nového poľa. V simulátore pre prácu s typom pole je potrebné doplniť metódy, ktorým rozumie.

Zoznamy sú taktiež reprezentované pomocou tokenu, konkrétne je pre ne pripravená trieda PNTokenList. Kvôli možnosti prologovského zápisu zoznamu je na rozdiel od PNTokenArray pre atribút value použitý typ Pair z balíčku javafx, ktorý drží dva zoznamy tokenov – jeden pre hlavičku, respektíve hlavnú časť zoznamu, druhý pre jeho chvost. Generácia prebieha podobne ako pri poli, teda najskôr sa vytvorí nový token, následne sa prechádzajú oba zoznamy a tokeny sa po jednom pridávajú. Podpora v simulátore nie je doplnená, vyžaduje úpravu nadväzovania premenných.

## 4.4 Výrazy

Funkcionalita výrazov bola značne obmedzená, najmä v akcii prechodu. Úplne chýbala podpora dočasných premenných, kaskádovania správ, zátvoriek vo výrazoch, kombinovania rôznych typov správ, reťazenia unárnych správ a posielania správ s viacerými kľúčovými slovami. Pri počiatočnej akcii prechodu navyše bol problém aj s priradením do premenných, respektíve so samotným vyhodnotením výrazu, k tomu viac v nasledujúcej podkapitole.

### 4.4.1 Pomocné premenné

Pomocné premenné majú význam pri sekvencii výrazov. Jednotlivé výrazy tak môžu pracovať s určitým medzivýsledkom, ktorý vznikol v niektorom predošlom výraze. V parseri sa kontroluje prítomnosť zoznamu pomocných premien medzi zvislými čiarami. Ten sa môže vyskytovať v rámci akcie prechodu a počiatočnej akcie miesta. K ich odpovedajúcim triedam je pridaný atribút so zoznamom objektov typu PNTokenVariable. Pri generovaní prechodov v rámci triedy či metódy sa akurát zabezpečí, aby došlo k naplneniu tohto zoznamu pri vytváraní inštancie triedy či metódy. Pri generovaní miesta s počiatočnou akciou sa už plnenie tohto zoznamu nerieši, pretože vzhľadom k významu počiatočnej akcie je k tomuto miestu pridaný prechod, ktorý túto akciu zabezpečuje, viac v nasledujúcej podkapitole. V rámci simulácie sú potom pri započatí vykonávania akcie prechodu tieto premenné doplnené do zoznamu premenných dostupných v rámci tohto prechodu, nie je teda nutné vytvárať špeciálny zoznam, ktorý by sa týkal len akcie prechodu. Na ich počiatočnú hodnotu je využitý PNTokenNil. Pri vyhodnocovaní výrazu sa nimi ďalej pracuje rovnako, ako s premennými, ktoré sú pre výraz dostupné z okolitých hrán predstavujúcich podmienky prechodu.

### 4.4.2 Reťazené unárne správy

Za účelom pridania podpory reťazenia unárnych správ stačí vzhľadom k implementácii výrazov len zmena v prekladači. Vďaka tomu, že trieda PNExpression už obsahovala atribút so zoznamom správ, ktorý bol využívaný pri zložených binárnych správach, v parseri sa časť výrazu predstavujúca zreťazené unárne správy rozdelí na jednotlivé správy, ktorá sa pridajú do tohto zoznamu. Generátor za týmto účelom úpravu nevyžadoval, zoznam správ má vo výsledku rovnakú podobu. Simulátor potom môže postupne posielať správy jednu za druhou vždy výsledku predošlého zaslania správy, rovnako ako pri binárnych správach.

### 4.4.3 Kaskáda správ

Posielanie niekoľkých správ rovnakému objektu je z pohľadu návrhu modelu možné pomocou sekvencie výrazov a využitiu pomocnej premennej. K tomuto účelu ale možno použiť aj

kaskádovanie správ. Nakoľko sa viaže vždy na jeden výraz, do triedy PNExpression, ktorá mu odpovedá, je pridaný ešte jeden atribút so zoznamom posielaných správ. V tomto prípade každá správa odpovedá jednej z kaskády správ. V parseri je spracovanie podobné ako spracovanie časti výrazu za priradením a pred samotnou kaskádou, ale kontroluje sa, či sa skutočne jedná len o jedno zaslanie unárnej, binárnej či kľúčovej správy, a nie o ich sekvenciu. V generátore je doplnené spracovanie tohto zoznamu rovnako, ako v prípade zoznamu správ hlavného výrazu, teda aby mal vygenerovaný súbor zoznam s rovnakým obsahom. Simulátor bol upravený tak, aby si ponechal cieľ poslania poslednej správy pred kaskádou, tento cieľ je potom zhodný aj pre správy kaskády. Tie sú následne jedna po druhej posielané tomuto istému cieľu.

### 4.4.4 Poradie posielania správ

Najväčšia úprava sa týka spracovania časti výrazu medzi priradením a kaskádovanými správami, ďalej označovaný ako jednoduchý výraz (cieľ a zoznam prípadných správ). Zatiaľ čo celý výraz je spracovávaný pomocou metódy `parseExpression(String)`, pre spracovanie jednoduchého výrazu je vytvorená samostatná metóda `parseSimpleExpression( List<String>, PNExpression)`, ktorá už spracúva jeho jednotlivé časti (vzniknuté jeho rozdelením podľa medzier) a vloží ich do PNExpression odpovedajúcemu celému hlavnému výrazu. Tieto časti môžu predstavovať aj kombináciu rôznych typov správ a podvýrazov v zátvorkácht

Pridaním zátvoriek sa môže upraviť poradie, v akom je vyhodnocované posielanie jednotlivých správ. Takýto podvýraz musí byť najskôr vyhodnotený rovnakým spôsobom, ako aj celý výraz, aby mohol byť jeho výsledok použitý ako cieľ či operand príslušnej správy. Vzhľadom k tomu, že výsledok je nejaký token, aj na celý výraz v zátvorkách je možné pozerať sa ako na token, ktorý je pred použitím potrebné vyhodnotiť. Preto je preň navrhnutý špeciálny typ tokenu PNTokenExpression s hodnotou atribútu value typu PNExpression. Jedná sa skutočne len o obalenie výrazu, aby bolo možné s ním pracovať rovnako, ako s iným tokenom. V parseri je pri spracovaní jednoduchého výrazu v prípade výskytu zátvoriek celý ich obsah spracovaný ako celý nový výraz, teda opäť pomocou `parseExpression(String)`. S takto získaným objektom typu PNExpression je vytvorený nový token (objekt typu PNTokenExpression), ktorý predstavuje cieľ alebo operand príslušnej správy. Posielať niekoľko unárnych či binárnych správ po sebe je potom jednoducho možné aj v prípade výskytu podvýrazu, stačí ho pred poslaním správy vyhodnotiť.

V prípade posielania rôznych typov správ za sebou ale poradie vyhodnocovania navyše záleží aj na type týchto správ. Každý má totiž určitú úroveň precedencie, na ktorú je potrebné brať ohľad. Pokiaľ sú jednotlivé správy v poradí od najvyššej precedencie k najnižšej, nie je s tým žiaden problém, môžu byť do zoznamu správ v PNExpression pridané v poradí, v akom sa vyskytujú, a rovnako aj vyhodnocované. V opačnom prípade by simulátor musel zabezpečiť, že dá prednosť vyhodnoteniu časti výrazu, ktorá má prednosť. To by ale nebolo potrebné, pokiaľ by tieto časti boli explicitne uzátvorkované. Už pri preklade je preto kontrolovaná precedencia správ a časti výrazu, ktoré by mali mať prednosť, sú spracované ako podvýraz. V takomto prípade sa ale určite jedná o jednoduchý výraz, preto je použitá metóda `parseSimpleExpression(List<String>)`, ktorá vráti nový objekt typu PNExpression, ten je následne použitý na vytvorenie odpovedajúceho tokenu.

Generátor zabezpečuje, že rovnaký výsledok bude aj vo vygenerovaných súboroch. Simulátor posiela správy stále jednu po druhej, jediná úprava je v tom, že pri výskyte tokenu s podvýrazom ho pred použitím vyhodnotí rovnakou metódou, ako celý výraz.

## 4.5 Počiatočná akcia miesta

V pôvodnej implementácii boli možnosti počiatočnej akcie miesta ešte viac obmedzené, ako možnosti akcie prechodu. Výrazy sa v parseri spracovali rovnako, v generátore ale bolo podporované len priradenie konkrétneho tokenu do miesta toľko krát, koľko premenných bolo vo výraze priradenia. Efektívne takáto počiatočná akcia nemá žiaden význam, rovnaký výsledok možno docieliť počiatočným značením. Hlavný význam počiatočnej akcie je, že podporuje posielanie správ a tým pádom umožňuje do miesta pridať aj nový objekt, nie len základné tokeny.

Keďže sa jedná o syntaktické vylepšenie jazyka, ktoré má určený význam, vhodným riešením je príslušným spôsobom upraviť samotnú sieť. Toto je vyriešené v generátore. Pokiaľ má miesto počiatočnú akciu, vygeneruje sa preň nové miesto a prechod. Počiatočné značenie tohto nového miesta je ľubovoľná jedna značku, použitý je literál nil. Názov miesta musí byť unikátny v rámci siete, aby ho pri simulácii bolo možné rozoznať. Za týmto účelom je použité meno spracúvaného miesta s podržítkom na začiatku, napríklad **__miesto**. Takéto meno určite iné miesto mať nebude, pretože sa jedná o nesprávny identifikátor, ktorý by pri spracovaní v parseri spôsobil chybu, pri simulácii to ale nevadí. Zároveň je zaručené, že ak iné miesto bude mať počiatočnú akciu, aj jeho nové miesto bude určite mať unikátny názov. Pre pridaný prechod je názov opäť vytvorený z názvu spracúvaného miesta, tentokrát pridaním dvoch podtržítok, napríklad **____miesto**. Jeho vstupná hrana je napojená na novovytvorené miesto, hranový výraz odpovedá tokenu v tomto mieste, teda nil. Jeho výstupná hrana je napojená na pôvodné miesto, hranový výraz odpovedá jeho počiatočnému značeniu. Keďže pre hranové výrazy sa používa typ PNMultiset a miesto v sebe má tokeny uložené ako zoznam, aby sa nemusel dodatočne riešiť prevod, do triedy PNPlace je navyše pridaný atribút multiset, ktorý taktiež obsahuje počiatočné značenie, ale ako objekt typu PNMultiset. Ako akcia prechodu sa použije počiatočná akcia pôvodného miesta. Samotné miesto sa vygeneruje bez akcie a počiatočného značenia.

V parseri je počiatočná akcia prechodu spracovaná rovnakým spôsobom, ako aj akcia prechodu, sú totiž zhodné. Počiatočné značenie je navyše okrem metódy

`parseInitMarking(String, PNPlace)` vracajúcej zoznam tokenov spracované aj pomocou metódy

`parseMultiset(PNMultise, String)` vracajúcej PNMultiset využívaný na hranách prechodu.

## 4.6 Podpora jednotlivých typov správ

Podľa pôvodného návrhu trieda PNMethod, ktorá je použitá ako rodičovská trieda pre vygenerované súbory odpovedajúce užívateľom definované metódy z PNTalku, nemá atribút, v ktorom by sa uchovával príslušný selektor správy. Ten je použitý v názve súboru (a teda aj danej triedy), navrhnutý spôsob práce so selektormi správ však neumožňoval rozoznať, o aký typ správy sa jedná. Zároveň tento spôsob nebol pre užívateľom definované metódy obecne použiteľný, v prípade unárnych správ a správ s kľúčovými slovami mohol byť pre príslušné metódy vygenerovaný rovnaký názov súboru. Pre binárne správy bol navrhnutý spôsob nepoužiteľný.

Selektory správ môžu obsahovať niektoré znaky, ktoré nemožno použiť v názve súboru alebo Java triedy. Tento názov preto musí byť generovaný z príslušného selektoru tak, aby boli rešpektované pravidlá pre názvy súborov príslušného operačného systému a názvy tried v jazyku Java. Zároveň je dôležité, aby sa nemohlo stať, že pre dve rôzne metódy definované

v jednej triede by bol použitý rovnaký názov. To platí aj pre dve rovnaké metódy definované v dvoch rôznych triedach, čo môže nastať najmä pri predefinovaní metódy z nadtriedy. Z toho dôvodu je navrhnutý nový mechanizmus prevodu selektoru na reťazec, ktorý je možné za týmto účelom použiť. Implementovaný je v metóde `methodFilename(IPNMessage message, String ownerClassName, boolean methodIsConstructor)`.

Je jej predaná správa, z ktorej sa zistí typ a získa sa selektor, reťazec s názvom triedy, v ktorej sa nachádza definícia s týmto vzorom správy, a pravdivostná hodnota, ktorá špecifikuje, či správa odpovedá konštruktoru, alebo metóde. Opačný proces je následne možné použiť na získanie selektoru z názvu súboru/triedy. Toto je implementované v metóde `selectorFromMethod(PNMethod)`.

Prvý krok je už v parseri, kde sa pri spracovaní vzoru správy ponechá selektor bez zmeny. Pre unárne správy je možné selektor použiť bez zmeny, je pre ne teda ponechaný pôvodný spôsob generovania názvu zložením názvu triedy, v rámci ktorej je príslušná metóda definovaná, a selektoru správy:

`NazovTriedy_selektor`

Pre binárne správy je prevod jednotlivých znakov selektoru nasledovný:

- '+' -> "PLUS"
- '-' -> "MINUS"
- '*' -> "ASTERISK"
- '/' -> "SLASH"
- '∼' -> "TILDE"
- '|' -> "PIPE"
- ',' -> "COMMA"
- '<' -> "LESS"
- '>' -> "GREATER"
- '=' -> "EQUAL"
- '&' -> "AMPERSAND"
- '\' -> "BACKSLASH"
- '@' -> "AT"
- '%' -> "PERCENT"
- '?' -> "QUESTIONMARK"
- '!' -> "EXCLAMATIONMARK"

Jednotlivé prevedené znaky sú od seba oddelené znakom "_". Takýto názov je vďaka použitiu veľkého počiatočného písmena dostatočný na zaručenie toho, že iný typ správy nebude mať rovnaký názov, pre jednoduchšie rozoznanie je však ešte zdvojený oddeľovač medzi názvom triedy a selektorom správy:

```
NazovTriedy__znakSelektoru
```

pre jednoznakový selektor a

```
NazovTriedy__znak1Selektoru_znak2Selektoru
```

pre dvojznakový selektor.

V správach s kľúčovými slovami samotné kľúčové slová nespôsobujú problém, stačí len nahradiť dvojbodky. Použitý je opäť znak "_", posledný je vynechaný, respektíve sa dá povedať, že sú ponechané len kľúčové slová a od seba oddelené týmto znakom. Kvôli rozlíšeniu od iných typov správy je oddeľovač medzi názvom triedy a selektorom použitý tri krát:

```
NazovTriedy___klucoveSlovo
```

pre selektor správy s jedným kľúčovým slovom a

```
NazovTriedy___klucoveSlovo1_klucoveSlovo2
```

pre selektor správy s dvoma kľúčovými slovami, obdobne potom pre ich ľubovoľný počet.

Objekty predstavujúce správy (inštancie niektorej triedy implementujúcej rozhranie IPNMessage) majú v sebe selektory správ uložené v pôvodnej podobe, teda tak, ako sú získané v parseri. Inštancia triedy (PNClass) má v sebe dva zoznamy. Jeden z nich je zoznam správ, ktorým odpovedajú užívateľom definované metódy či konštruktory tejto triedy. Ten má podobu mapy reťazcov (kľúč) a objektov triedy IPNMessage. Ako kľúč sa využíva selektor správy bez zmeny. Správam z tohto zoznamu rozumejú objekty tejto triedy, respektíve aj samotná trieda v prípade konštruktorov. Druhý zoznam slúži na uloženie synchrónnych portov. Má podobu mapy reťazcov (kľúč) a objektov triedy PNSynchronousPort. V tomto prípade ale ako kľúč už nemôže byť použitý samotný selektor správy. Tak, ako v prípade vygenerovaných súborov pre jednotlivé metódy, je potrebné k selektoru pridať aj názov triedy, v rámci ktorej bol daný synchrónny port definovaný. V tomto prípade je to iba z dôvodu dedičnosti. Selektor sa môže ponechať v pôvodnej podobe, tým pádom nemusí byť riešený ani rôzny počet oddeľovačov pre rôzne typy správ:

```
NazovTriedy_selektor
```

## 4.7 Konštruktory

Pre konštruktory bola už v pôvodnej implementácii pripravená trieda PNConstruktor s rovnakou štruktúrou ako PNMethodNet (taktiež dediaca z triedy PNNet) a v triede PNClazz pripravený príslušný zoznam na konštruktory a metódy na prácu s ním. Toto rozlíšenie je dôležité, pretože konštruktor možno použiť ako metódu, ale metódu nemožno použiť ako konštruktor. V oboch prípadoch sa ale jedná len o metódu, ktorá je invokovaná po zaslaní príslušnej správy. Rozdiel je len v tom, či túto správu možno zaslať aj triede, a že v takom prípade sa musí najskôr vytvoriť nový objekt, ktorý pôvodnú správu prijme. Táto trieda však nebola ďalej využitá, pretože neimplicitné konštruktory neboli podporované.

Za účelom spracovania definície konštruktoru je v parseri pridaná metóda parseConstructor(String), ktorá vstup spracuje rovnakým spôsobom, ako keby sa jednalo o definíciu metódy, rozdiel je len v úvodnom kľúčovom slove. Metóda vracia objekt triedy PNConstructor, ktorý je pridaný do zoznamu konštruktorov objektu PNClazz prislúchajúci spracovávanej triede.

Táto trieda pre konštruktor je však použitá iba v prekladači. Výstupom generátora je trieda PNmethod, tak ako v prípade obyčajnej metódy. Za účelom rozlíšenia, či sa jedná

o konštruktor alebo metódu, je do tejto triedy pridaný atribút isConstructor a príslušné metódy na priradenie hodnoty a jej získanie. Navyše je upravený aj názov tejto triedy, aby toto rozlíšenie bolo možné previesť jednoducho a rýchlo už podľa názvu súboru a nebolo tak potrebné triedu načítať a vytvárať objekt. Úprava spočíva v pridaní reťazca „_0“ tak, že názov má tvar NázovTriedy_0_selektorSprávy, kde selektor správy má podobu popísanú v predošlej podkapitole 4.6. Nakoľko selektor žiadnej správy nemôže začínať číslom, nestane sa, že pre inú metódu bude použitý rovnaký názov.

Simulátor pri poslaní inej správy ako new triede skontroluje, či bol príslušný konštruktor definovaný. Pokiaľ áno, triede je zaslaná správa new, teda sa vytvorí nový objekt. Následne dochádza k invokácii nájdenej metódy označenej ako konštruktor. Pokiaľ sa takýto konštruktor nenájde, existencia metódy odpovedajúca tomuto selektoru správy sa nekontroluje. Naopak, pokiaľ dôjde k predaniu takejto správy objektu, dôjde k vyhľadávaniu príslušnej metódy medzi dostupnými metódami a v prípade jej nenájdenia aj medzi konštruktormi.

## 4.8 Dedičnosť

Dedičnosť je jedna zo základných vlastností objektovej orientácie, umožňujúca inkrementálnu definíciu tried. V prípade PNTalku to predstavuje dedenie metód, konštruktorov, synchrónnych portov a objektovej siete. Metódy by malo byť možné predefinovať ako celok, rovnako aj synchrónne porty, objektovú sieť však možno predefinovať aj po častiach. Nakoniec by ešte malo byť možné vrátiť sa k pôvodnej definícii metódy či synchrónneho portu pomocou pseudopremennej super.

Možností, ako dedičnosť implementovať, je viacero. Jedna z nich je, ako bolo aj v pôvodnej práci naznačené, využiť dedičnosť jazyka Java. Ak napríklad model obsahuje triedu C1 dediacu z triedy PN a C2 dediacu z C1, vo vygenerovaných súboroch by trieda C1 dedila z PNClass a C2 z C1. Vzhľadom k tomu, že naplnenie atribútov prebieha v konštruktore, C2 by zdedila všetky miesta, prechody, synchrónne porty a vzory správ pre dostupné metódy. Nová definícia v C2 predstavuje doplnenie do príslušného zoznamu, pre predefinovanie metódy je to ponechanie zoznamu bez zmeny a vygenerovanie nového súboru, pre synchrónny port jeho doplnenie do zoznamu a ponechanie pôvodného, pre prechod vyhľadanie v zozname a nahradenie pôvodného, pre miesto nahradenie pôvodného ako v zozname, tak v prechodoch a synchrónnych portoch.

Druhá možnosť je nevyužívať dedičnosť tried v Jave a pre všetky triedy modelu použiť PNClass. V takom prípade musí niekde byť dostupná aj informácia o predkoch. V konštruktore potom musí byť vyriešené naplnenie všetkých zoznamov obdobným spôsobom, ako keby sa postupne dedilo. Pre každú generovanú triedu sa teda musia prejsť aj jej predkovia, ale zas nemusí byť riešené nahrádzanie miest a prechodov, pokiaľ sa prechádza zoznam predkov správnym spôsobom. Simulácia by bola v oboch prípadoch podobná.

Implementovaný je druhý variant. Parser v tomto ohľade nevyžaduje ďalšie zmeny, dedičnosť sa začína riešiť až v generátore. Metóda **generateClasses(PNClasses)** prechádza zoznamom tried, pre každú jednu zostaví nový zoznam, v ktorom je na prvom mieste táto trieda a nasledovaná je svojimi predkami v poradí od najbližšieho k najvzdialenejšiemu. Pri tomto zostavení sa zároveň overí, či v dedičnosti nie je cyklus alebo či niektorý predok nechýba. Takto vytvorený zoznam sa predá metóde **buildClass(List<PNClazz>)**, ktorá začne s generovaním triedy. Atribút **parents** v triede PNClass predstavuje zoznam mien predkov danej triedy, a to od najbližšej, až po „PN“ vrátane. Najskôr sa prejde zoznam predkov danej triedy a naplní sa zoznam ich mien. Informácia o predkoch tak bude hneď na začiatku konštruktora. Následne sa postupne pre každú triedu vygenerujú jej miesta, pokiaľ

ešte s daným menom miesto nebolo generované. Tým sa zaručí, že trieda bude obsahovať iba tie, ktoré boli definované či predefinované v nej alebo najbližšie k nej v hierarchii dedičnosti. Ďalej sa rovnakým spôsobom vygenerujú jej prechody, opäť teda bude trieda obsahovať iba prechody odpovedajúce najnovším definíciám. Nasleduje generovanie súborov pre konštruktory a metódy danej triedy, v tomto prípade však pre predkov nie, pretože sa predpokladá, že pre nich už generácia prebehla, alebo prebehne. Na záver sa opäť prejde celý zoznam tried a naplnenia sa zoznamy správ, ktorým rozumie daná trieda (pre konštruktory) či jej inštancia (pre metódy), a zoznam synchrónnych portov. Týmto končí generácia danej triedy a pokračuje sa ďalšou.

V simulátore sa na dedičnosť berie ohľad pri zasielaní správ triedam a objektom, pretože odpovedajúci konštruktor, metóda či synchrónny port mohol byť definovaný v niektorom predkovi a je potrebné ho nájsť.

Keď je príjemcom správy trieda a nejedná sa o správu *new*, mal by jej prislúchať nejaký užívateľom definovaný konštruktor. Aby bolo možné skontrolovať, či vôbec trieda takejto správe môže rozumieť, musí sa vytvoriť jej inštancia, z ktorej je možné získať zoznam podporovaných správ. Zavolá sa metóda **getMethodFilename(IPNMessage message, PNClass ownerClass, boolean isConstructor, boolean searchFromParent)**, ktorá prítomnosť správy v tomto zozname skontroluje. Následne zabezpečí vyhľadanie súboru, ktorý odpovedá príslušnému konštruktoru. Vyhľadávanie začína od danej triedy a pokračuje jej predkami tak, že zo selektoru správy vytvorí názov súboru rovnako, ako to bolo pri generovaní súboru. Ak takýto súbor existuje, predstavuje požadovaný konštruktor, v opačnom prípade selektoru odpovedá metóda. Následne je možné triede poslať správu new a invokovať nájdenú metódu (konštruktor).

Ak je príjemcom správy objekt, nemusí sa vytvárať nová inštancia, možno hneď začať vyhľadávanie pomocou tej istej metódy. V prípade hľadania metódy je rozdiel v tom, že sa skontroluje existencia súboru ako pre metódu, tak aj pre konštruktor. Pri úspechu je možné danú metódu (klasickú alebo konštruktor) invokovať.

V prípade, že príjemcom správy bol objekt, ale správa bola zaslaná zo stráže prechodu, vyhľadáva sa synchrónny port pomocou metódy **getSynchronousPortKey(IPNMessage message, PNClass ownerClass, boolean searchFromParent)**. Vyhľadanie je podobné, vygeneruje sa názov, ktorý by bol použitý ako kľúč v mape so synchrónnymi portami, a postupne sa prechádza hierarchia tried.

Posielať takéto správy možno aj pseudopremenným self a super. Obe predstavujú objekt, v rámci ktorého je daná správa poslaná, ale super má navyše význam, že hľadanie príslušného konštruktoru, metódy či synchrónneho portu začína až od prvého rodiča. Aby sa dalo k tomuto objektu dostať, v triede PNTransition je zavedený atribút **owner** typu PNClass , ktorý sa odkazuje na inštanciu triedy, do ktorej tento prechod patrí. Pri vyhodnocovaní výrazu je potom vo volaných metódach vždy predávaný aj daný prechod, aby sa prostredníctvom neho dali self a super vyhodnotiť.

# Kapitola 5

# Testovanie

Počas práce na tomto projekte bolo všetko testovanie robené ručne. Na úvodné testovanie pôvodného riešenia za účelom oboznámenia sa s jeho použitím bola využitá sada príkladov, ktorá bola súčasťou tohto riešenia. Tieto slúžili na demonštrovanie úspešného prekladu a simulácie v aktuálnom stave daného riešenia. Počas analýzy zdrojového kódu boli vytvárané podobné jednoduché príklady modelov v jazyku PNtalk, ktoré mali poukázať na odhalené nedostatky riešenia. Vďaka nim bolo možné demonštrovať, že riešenie obsahovalo chyby dvojakého typu:

- chyba OOPN modelu nebola odhalená, preklad prebehol úspešne
- správny model OOPN spôsobil chybu pri jeho preklade

Po odstránení daných chýb v prekladači bolo upravené riešenie otestované rovnakým spôsobom. Pôvodná sada príkladov bola opätovne preložená a pri troch z nich, konkrétne basicInput5, intermediateInput3 a advancedInput6, bola odhalená chyba, kde definícia vstupnej podmienky predchádzala definíciu obyčajnej podmienky.

Počas implementácie chýbajúcich častí z pôvodnej práce boli využívané ďalšie jednoduché príklady, ktoré overovali novú funkcionalitu. Napríklad na overenie, že zdvojený apostrof v reťazci sa správne spracuje ako jeden apostrof vo výslednom reťazci, bol použitý nasledujúci model:

```
main StringWithApostrophe

class StringWithApostrophe is_a PN
       object
              place p1('Apostrophe: ''')

              trans t1
                     precond p1(x)
                     action {x print}
```

Výpis 5.1: Model OOPN s apostrofom v reťazci

# Kapitola 6

# Záver

V rámci tejto práce bolo analyzované predošlé riešenie z bakalárskej práce Generování kódu z modelů Petriho sítí [2], identifikované jeho nedostatky, navrhnuté a implementované úpravy. Účelom úprav v prekladači bolo doplniť syntaktickú kontrolu a zabezpečiť korektné spracovanie zdrojového textu napísaného v jazyku PNtalk do vnútornej štruktúry, a tiež rozšíriť spracovanie o chýbajúce časti. Rovnakým spôsobom bol doplnený aj generátor a pridaná podpora týchto častí aj do simulátora.

Konkrétne bolo v rámci tejto práce:

- Prerobené spracovanie reťazcov a multimnožín, aby pracovali so správnymi znakmi (apostrof a backtick).
- Pridaná podpora zvyšných tokenov aj do generátora. V simulátore by ale ešte boli potrebné metódy, ktoré by demonštrovali ich využitie.
- Pridaná podpora pomocných premenných, cascade, zložených výrazov a zátvoriek v nich.
- Upravené, ako sa ukladajú a spracúvajú selektory správ (kvôli konštruktorom a dedičnosti).
- Pridaná podpora konštruktorov.
- Pridaná podpora dedičnosti - metódy, konštruktory, synchrónne porty, pseudopremenné self a super.
- Prerobená/dokončená počiatočná akcia miesta, ktorá pôvodne nerobila prakticky to, čo bolo potrebné.
- Zoznamy existujú len ako token, podpora naväzovania premenných nie je implementovaná.

V prípade naväzujúcej práce by mohlo byť riešenie dôkladnejšie otestované, mohla by byť doplnená sémantická kontrola a rozšírená knižnica metód. Pre jednoduchšie oboznámenie sa so zdrojovým kódom je tento dôkladne okomentovaný, využívané sú javadoc komentáre[1] pre jednotlivé metódy. Ďalšie komentáre zahrnujú mnohé poznámky a vysvetlivky na miestach, kde to pre čitateľa môže byť dôležité.

[1] https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html

# Literatúra


[1] Apache. *Apache Ant* [online]. [cit. 2023-05-07]. Dostupné z: https://ant.apache.org/.

[2] Fryč, T. *Generování kódu z modelů Petriho sítí*. Brno, CZ, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/21446/.

[3] Janoušek, V. *Modelování objektů Petriho sítěmi*. Faculty of Information Technology BUT, 2008. 164 s. ISBN 978-80-214-3749-4. Dostupné z: https://www.fit.vut.cz/research/publication/8831.

[4] Meduna, A. *Elements of compiler design*. Boca Raton: Auerbach Publications, 2008. ISBN 1-4200-6323-5.

[5] Pecinovský, R. *Návrhové vzory*. Vyd. 1. Brno: Computer Press, 2007. ISBN 978-80-251-1582-4.

# Príloha A

# Syntax jazyka PNtalk

Popis syntaxe jazyka PNTalk Backus-Naurovou formou prebratý z dizertačnej práce "Modelování objektů Petriho sítěmi"[3]. Východiskový 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: number | string | charconst | symconst | arrayconst
arrayconst: "#" array
array: "(" [number | string | symbol | array | charconst]* ")"
number: [-][[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: " " | "\^" | ";" | "\$" | "#" | ":" | "." | "-" | "`"
```

# Príloha B

# Obsah pamäťového média

- **xcibak00.pdf** - táto práca vo formáte PDF
- **readme.txt** - informácie o obsahu média
- **xcibak00.zip** - archív s adresárovou štruktúrou obsahujúcou zdrojové kódy z pôvodnej práce, kde sú nahradené tie, ktoré boli v rámci tejto práce upravené
- **original.zip** - archív s adresárovou štruktúrou obsahujúcou pôvodné zdrojové kódy z predošlej práce bez akýchkoľvek zmien
- **latex.zip** - archív so zdrojovými kódmi tejto písomnej práce