VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

INTERPRET PETRIHO SÍTÍ

INTERPRETER OF PETRI NETS FORMALISM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. TOMÁŠ BLAŽEK**
AUTHOR

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

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2019/2020

Zadání diplomové práce

Student: **Blažek Tomáš, Bc.**
Program: Informační technologie Obor: Informační systémy
Název: **Interpret Petriho sítí**
Interpreter of Petri Nets Formalism
Kategorie: Překladače
Zadání:

1. Seznamte se s konceptem Petriho sítí a jeho variantou Objektově orientované Petriho sítě (OOPN).
2. Navrhněte vnitřní reprezentaci modelů OOPN vhodnou pro efektivní interpretaci a implementujte překladač z jazyka PNtalk do vnitřní reprezentace.
3. Navrhněte a realizujte interpret (simulátor) OOPN využívající vnitřní reprezentaci modelů. Interpret implementujte v jazyce Java. Kromě objektů Petriho sítí musí interpret umět pracovat i s vybranou podmnožinou objektů z jazyka Java.
4. Navrhněte a proveďte sadu testů ověřující správnost simulátoru a demonstrující jeho vlastnosti.

Literatura:

- V. Janoušek: Modelování objektů Petriho sítěmi. Disertační práce. VUT v Brně, 1998.
- PNtalk 2.0. http://perchta.fit.vutbr.cz/pntalk2k/, 2019.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a 2.

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. listopadu 2019
Datum odevzdání: 20. května 2020
Datum schválení: 31. října 2019

Abstrakt

Tato práce se zabývá konceptem formalismu Petriho sítí, který umožňuje zkoumat dynamické chování paralelních a nedeterministických systémů. Dále se práce zabývá jeho variantou Objektově orientovaných Petriho sítí (OOPN), jejímž cílem je usnadnit proces modelování systémů objektově orientovaným přístupem. Cílem této diplomové práce je navrhnout vnitřní reprezentaci modelů OOPN vhodnou pro efektivní interpretaci a implementovat překladač z jazyka PNtalk do vnitřní reprezentace. Následně pak navrhnout a realizovat interpret modelů OOPN, který bude umožňovat provádění simulace běhu těchto modelů s tím, že výsledný interpret musí také kromě objektů Petriho sítí umět pracovat i s vybranou podmnožinou objektů z jazyka Java.

Abstract

This thesis deals with the concept of the formalism of Petri nets, which allows to investigate the dynamic behavior of parallel and nondeterministic systems. Furthermore, this deals with its variant of Object-Oriented Petri Nets (OOPN), which aims to facilitate the process of modeling systems with an object-oriented approach. The aim of this master thesis is to design an internal representation of OOPN models, which is suitable for efficient interpretation and implement compiler from PNtalk language into the internal representation. Subsequently, design and implement an OOPN model interpreter using this internal representation of models, which in addition to Petri net objects, must also be able to work with a selected subset of objects from the Java language.

Klíčová slova

Petriho sítě, Objektově orientované Petriho sítě, Překladač, Interpret, Simulátor

Keywords

Petri nets, Object oriented Petri nets, Compiler, Interpreter, Simulator

Citace

BLAŽEK, Tomáš. *Interpret Petriho sítí*. Brno, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Kočí, Ph.D.

Interpret Petriho sítí

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Radka Kočího Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

. . . .
Tomáš Blažek
1. června 2020

Poděkování

Zde bych rád poděkoval svému vedoucímu práce panu Ing. Radkovi Kočímu Ph.D. za odborné vedení mé diplomové práce a poskytnutí důležitých materiálů pro její vypracování.

Obsah

1 Úvod **3**

2 Petriho sítě **4**
2.1 Formalismus Petriho sití . . . 4
2.2 Modelování Petriho sítí . . . 6
2.3 Varianty Petriho sití . . . 8
2.3.1 Stochastické Petriho sítě . . . 8
2.3.2 Barevné Petriho sítě . . . 8
2.3.3 Hierarchické Petriho sítě . . . 8
2.3.4 Objektově orientované Petriho sítě . . . 9

3 Objektově orientované Petriho sítě **10**
3.1 Struktura OOPN . . . 10
3.1.1 Třídy a objekty . . . 10
3.1.2 Síť objektu . . . 11
3.1.3 Sítě metod . . . 11
3.1.4 Synchronní porty . . . 11
3.1.5 Místa, přechody a hrany . . . 12
3.2 Vlastnosti objektové orientace . . . 13
3.2.1 Zapouzdření . . . 13
3.2.2 Dědičnost . . . 14
3.2.3 Polymorfismus . . . 14
3.3 Existující nástroje . . . 14
3.3.1 Systém PNtalk . . . 15
3.3.2 Nástroj Renew . . . 15

4 Jazyk PNtalk **17**
4.1 Termy . . . 17
4.2 Základní stavební bloky . . . 18
4.2.1 Místo . . . 19
4.2.2 Přechod . . . 19
4.3 Zasílání zpráv . . . 20
4.4 Třídy . . . 22
4.4.1 Konstruktory . . . 22
4.4.2 Sítě . . . 23
4.4.3 Synchronní porty . . . 24
4.4.4 Dědičnost . . . 24

5 Překladač 25
- 5.1 Fáze překladu . . . 25
  - 5.1.1 Lexikální analýza . . . 26
  - 5.1.2 Syntaktická analýza . . . 27
  - 5.1.3 Generování OOPN . . . 28
- 5.2 Vnitřní reprezentace OOPN . . . 29
  - 5.2.1 Komponenty . . . 31
  - 5.2.2 Struktury . . . 33

6 Interpret 36
- 6.1 Struktura interpretu . . . 36
  - 6.1.1 Objekty tříd OOPN . . . 37
  - 6.1.2 Selektory . . . 37
  - 6.1.3 Události . . . 38
  - 6.1.4 Výjimky . . . 39
- 6.2 Běh intepretu . . . 40
  - 6.2.1 Inicializace interpretu . . . 40
  - 6.2.2 Analýza proveditelnosti přechodů . . . 41
  - 6.2.3 Provedení přechodu . . . 44
- 6.3 Vyhodnocování výrazů . . . 44
  - 6.3.1 Struktura výrazu . . . 45
  - 6.3.2 Operandy výrazu . . . 46
  - 6.3.3 Vyhodnocení výrazu . . . 48
- 6.4 Podpora Java objektů . . . 50
  - 6.4.1 Rozhraní . . . 50
  - 6.4.2 Použití . . . 51

7 Testování 52
- 7.1 Struktura testování . . . 52
  - 7.1.1 Překladač . . . 53
  - 7.1.2 Interpret . . . 56
- 7.2 Souhrn výsledků testování . . . 58

8 Závěr 59

Literatura 60

A Syntax jazyka PNtalk 62

B Diagramy 64

C Podporované Java objekty 66

D Manuál 71

E Obsah CD 72

Kapitola 1

Úvod

Softwarové inženýrství a zejména jeho odvětví návrhu systémů je postupem času čím dál komplikovanější oblastí z důvodu potřeb vývoje složitějších systémů. Z tohoto důvodu vznikají nové způsoby návrhů modelů systémů a procesy jejich validace.

Jedním z možných přístupů k této problematice je využití některého z již existujících matematických formalismů, jímž by byl modelovaný systém popsán. Matematický formalismus nabízí především jednoznačnost modelů, přičemž poskytuje možnost ověření validního chování systému, a to především za pomoci simulace. Simulace a analýza tohoto modelu by měla umožnit odhalení důležitých informací o modelovaném systému, a tím docílit lepšího návrhu modelovaného systému.

Formalismem, jímž se tato práce zabývá jsou Petriho sítě, které umožňují zkoumat dynamické chování paralelních a nedeterministických systémů. Ovšem modelování systémů v základní variantě Petriho sítí by bylo velmi náročné a neefektivní. Proto vnikla varianta Objektově orientovaných Petriho sítí (OOPN), jejímž cílem je přiblížit se k produkčnímu prostředí a tím usnadnit proces modelování. Pro efektivní zápis modelů definovaných formalismem OOPN byl vytvořen jazyk PNtalk. K analýze modelů OOPN je však nutné implementovat simulátor (interpret). Dosavadní implementace simulátoru modelů OOPN je však realizována v jazyce Smalltalk, který už v dnešní době nepatří k nejrozšířenějším programovacím jazykům. Tím pádem vznikají nové požadavky na implementaci, jejichž cílem je přiblížit se k produkčnímu prostředí vyvíjených systémů a tím docílit vyšší efektivity simulátoru. Pro implementaci simulátoru se tedy nabízejí programovací jazyky jako jsou C++ nebo Java, které jsou v dnešní době poměrně populární.

Cílem práce je navrhnout vnitřní reprezentaci modelů OOPN vhodnou pro efektivní interpretaci a implementovat překladač z jazyka PNtalk do vnitřní reprezentace v jazyku Java. Dále také navrhnout a realizovat interpret modelů OOPN s tím, že výsledný interpret musí také kromě objektů Petriho sítí umět pracovat i s vybranou podmnožinou objektů z jazyka Java.

V kapitolách 2 až 4 jsou popsány již existující koncepty v oblastech Petriho sítí, Objektově orientovaných Petriho sítí a popis jazyka PNtalk. V kapitole 5 je popis návrhu a implementace jednotlivých částí překladače z jazyka PNtalk do vnitřní reprezentace a popis vnitřní reprezentace modelů Objektově orientovaných Petriho sítí. Kapitola 6 obsahuje návrh interpretu a princip realizace jednotlivých jeho částí. Poslední kapitola 7 v textu práce je věnována procesu testování za účelem ověření správnosti výsledného simulátoru.

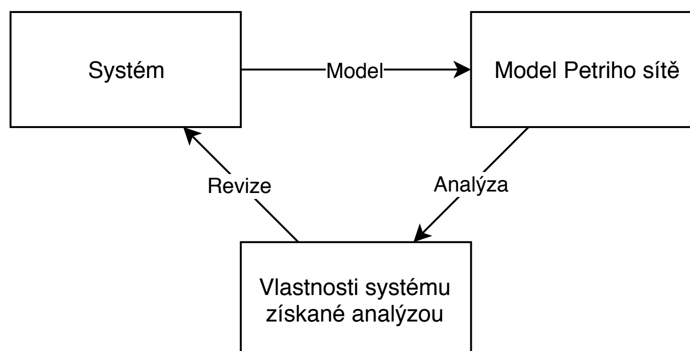
Kapitola 2

Petriho sítě

Co jsou to Petriho sítě? Jedná se o nástroj, který umožňuje formální zápis modelu systému pro následnou analýzu, která by měla dopomoci k vývoji lepšího systému. Velmi často je jako počátek Petriho sítí brána disertační práce [13], jejímž autorem je Carl Adam Petri. Tato disertační práce byla autorem publikována začátkem šedesátých let. Sám C. A. Petri nepojmenoval tuto třídu sítí po sobě, jak by se mohlo zdát. Ve své práci psal o formalismu, který později vedl ke vzniku „Petriho sítí“, jak je známe dnes, ale samotnou práci pojmenoval „Kommunikation mit Automaten“, což by se dalo přeložit jako „Komunikace skrze automaty“. Tato práce tedy neobsahuje přesnou definici sítí, které jsou nyní známy jako Petriho sítě. Ve skutečnosti byly později vytvořeny stovky definic a rozšíření Petriho sítí uváděných v literaturách. Velké množství autorů totiž nemělo v úmyslu vymýšlet něco úplně nového, a proto ve svých pracích používali výraz „Petriho sítě“, kterým vyjadřovali základní koncepty z disertační práce C. A. Petriho.[2]

2.1 Formalismus Petriho sítí

Petriho sítě jsou nástrojem pro modelování systémů v matematické reprezentaci, jejíž následná analýza by měla odhalit důležité informace o struktuře a dynamickém chování modelovaného systému. Systém je tedy nejprve modelován pomocí Petriho sítí, a poté je tento model analyzován. Porozumění systému z analýzy modelu by mělo vést k vytvoření lepšího výsledného systému.[12] Cyklus vývoje systému pomocí Petriho sítí je znázorněn na obrázku 2.1. To je důvod, proč se vyvíjí techniky modelování a analýza pomocí Petriho sítí.



Obrázek 2.1: Cyklus vývoje systému pomocí Petriho sítí

Definice 2.1.1. Petriho síť N je pětice $N = (P, T, F, W, M_0)$, kde
P je konečná množina míst,
T je konečná množina přechodů $P \cap T = \emptyset$,
F je incidenční relace $F \subseteq (P \times T) \cup (T \times P)$,
W je váhová funkce $F \to \mathbb{N}$,
M_0 je počáteční značení $P \to \mathbb{N}$ (M se nazývá značení Petriho sítě).

Průběh výpočtu Petriho sítě je dán posloupností vzniklých událostí, které mají za důsledek vznik parciálních stavů modelovaného systému, které se podílejí na celkém stavu modelovaného systému. Modely parciálních stavů rozumíme *místa* Petriho sítě, které mohou být označeny značkou, pokud se systém aktuálně nachází v daném parciálním stavu. Model události se v Petriho síti nazývá *přechod*, který po jeho provedení dostane systém ze stavu s_i do stavu s_j. Jinými slovy se Petriho síť dostane přechodem $t \in T$ ze značení M_i do značení M_j, což se zapisuje jako $M_i[t\rangle M_j$.[14]

Přechod může mít vstupní místa a výstupní místa, která jsou určena incidenčními relacemi. Z hlediska výpočtu před provedením přechodu musí být stav tzv. proveditelný. Přechod $t \in T$ je *proveditelný* tehdy, pokud v každém jeho vstupním místě je umístěno dostatečný počet značek. Formálně to lze zapsat takto:

$$\forall p \in P : M(p) \geq W(p, t)$$

Jestliže je přechod $t \in T$ proveditelný při značení M_i, pak může být proveden, čímž vnikne nové značení M_j, které popisuje následující výraz:

$$\forall p \in P : M_j(p) = M_i(p) - W(p, t) + W(t, p)$$

Výpočetní posloupností se rozumí posloupnost přechodu $\sigma = t_1,\ t_2,\ \dots,\ t_n$, kde $t_i \in T, i \in \mathbb{N} - \{0\}$, pokud existují značení $\{M_1,\ M_2, \dots,\ M_n\}$, které je možno docílit $M_0[t_1\rangle M_1,\ M_1[t_2\rangle M_2,\ \dots,\ M_{n-1}[t_n\rangle M_n$. Pokud chceme vyjádřit skutečnost, že značení M_n je dostupné ze znační M_0 provedením sekvencí přechodů σ, pak se použije značení:

$$M_0[\sigma\rangle M_n$$

V některých případech je vhodné využít tzv. kapacitu míst, čímž se omezí počet značek jednotlivých míst, jež mohou nabývat. Toto je vhodné například pro modelování skladiště, pamětí atd.[14] V tomto případě by se definice Petriho sítě rozšířila takto:

Definice 2.1.2. Petriho síť N je šestice $N = (P, T, F, W, M_0, K)$, kde
P je konečná množina míst,
T je konečná množina přechodů $P \cap T = \emptyset$,
F je incidenční relace $F \subseteq (P \times T) \cup (T \times P)$,
W je váhová funkce $F \to \mathbb{N}$,
M_0 je počáteční značení $P \to \mathbb{N}$ (M se nazývá značení Petriho sítě),
K je funkce kapacity míst $P \to \mathbb{N} \cup \{\infty\}$.

Funkce kapacity míst tedy přiřadí každému místu Petriho sítě přirozené číslo včetně ∞, které označuje maximální počet značek, které může dané místo nabývat. Pro symbol ∞ má dané místo neomezenou kapacitu. Tedy pro všechny místa v Petriho síti $p \in P$ musí

platit $M(p) \leq K(p)$. Úprava definice přidáním kapacity ovlivní i proveditelnost přechodu, pro kterou musí být splněna i následující podmínka:

$$\forall p \in P \; : \; W(t,p) > 0 \implies M(p) - W(p,t) + W(t,p) \leq K(p)$$

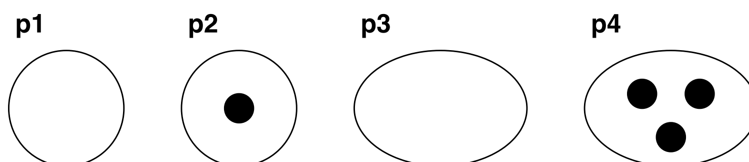
Tedy jinými slovy musí být ve všech výstupních místech přechodu dostatečná kapacita pro vkládané značky.[14]

2.2 Modelování Petriho sítí

V modelování Petriho sítí se často využívá grafické verze Petriho sítí, která je pro člověka přehlednější, což dopomáhá k efektivnějšímu modelování systému. V grafické reprezentaci se Petriho sítě zakreslují jako grafy se dvěma typy uzlů, propojené mezi sebou orientovanými hranami. Jak už bylo řečeno v kapitole 2.1, tak Petriho sítě se skládají z míst a přechodů, které mají mezi sebou vztahy určené incidenčními relacemi. Tedy prvním typem uzlů jsou místa a druhým typem jsou přechody. Orientovanými hranami jsou vyjádřeny vztahy mezi nimi.

Místa

Využívají se pro zachycení parciálních stavů modelovaného systému. V grafu jsou značeny většinou jako kružnice nebo případně elipsy. Ty v sobě mohou mít značky, někdy označovány jako tokeny, které se využívají k tomu, aby se zaznačilo, v jakém stavu se aktuálně modelovaný systém nachází. Token se v grafické reprezentaci označuje jako černá tečka. Každé místo musí být v Petriho síti být označeno unikátním identifikátorem.

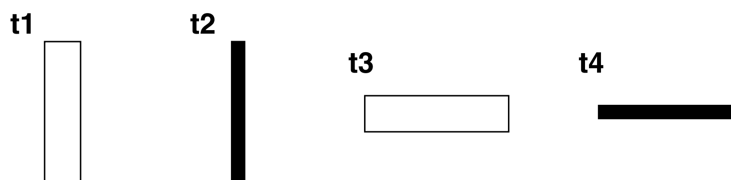


Obrázek 2.2: Příklady míst v grafové reprezentaci Petriho sítí

Na obrázku 2.2 jsou uvedeny příklady míst, přičemž v místě $p2$ je umístěn jeden token, v místě $p4$ jsou umístěny tři tokeny a v místech $p1$ a $p3$ nejsou umístěny žádné tokeny.

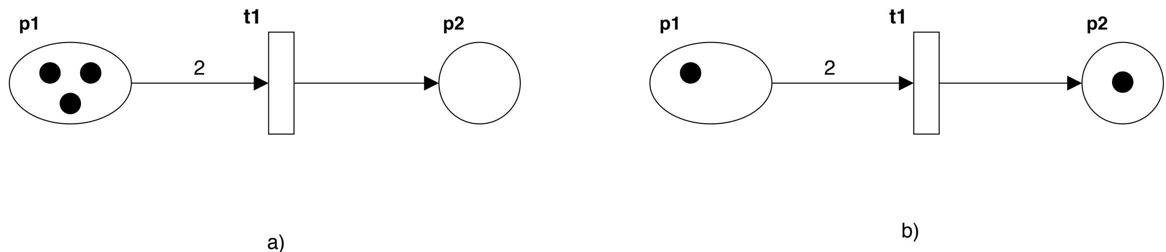
Přechody

Jedná se o druhý typ uzlu grafu Petriho sítě, kterým se modelují události v systému. V grafu se značí jako obdélník nebo tlustá čára. Stejně jako místa v Petriho síti, tak i přechody musí být označeny unikátním identifikátorem. Příklady přechodů jsou uvedeny na obrázku 2.3.



Obrázek 2.3: Příklady přechodů v grafové reprezentaci Petriho sítí

Přechody jsou propojovány v grafu orientovanými hranami podle vztahů na základě incidenčních relací Petriho sítě. K provedení přechodu je nutné splnit podmínku, a to ve formě existence značek ve vstupních místech přechodu. Každá incidenční relace je v grafu reprezentována jako orientovaná hrana propojující místo s přechodem nebo naopak. Dále má každá hrana určenou svoji váhu, která se uvádí nad danou hranu. Vyjadřuje podmínku, kolik je potřeba značek v daném vstupním místě přechodu pro jeho provedení. Pokud není nad hranou váha explicitně uvedena, uvažuje se hodnota 1.

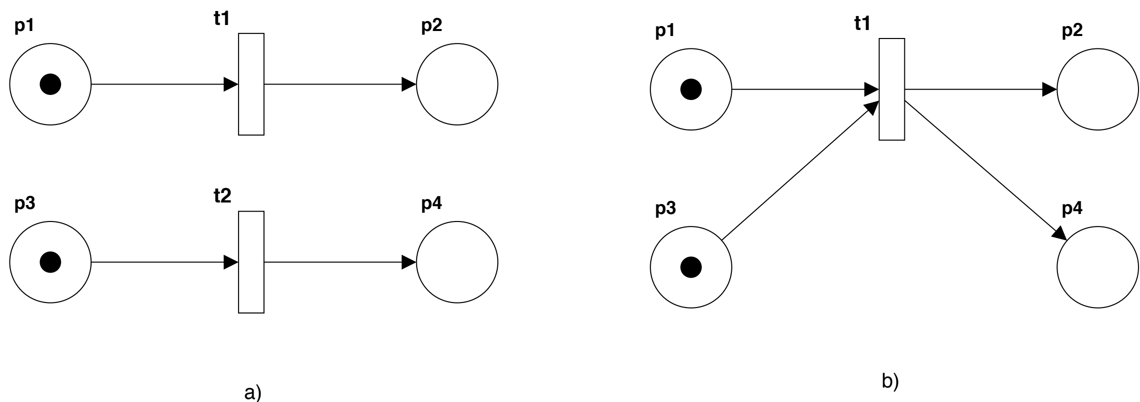


Obrázek 2.4: Příklad provedení přechodu v grafové reprezentaci Petriho sítí

Příklad, který je uveden na obrázku 2.4 vyobrazuje v části a) Petriho síť s počátečním značením. V části b) je zakreslena daná síť se značením po provedení přechodu $t1$. Z místa $p1$ byly odebrány dva tokeny podle vstupní podmínky přechodu $t1$ a po provedení přechodu byl vložen jeden token do jeho výstupního místa $p2$.

Synchronizace

Ve většině modelovaných systémech je velmi důležitá sychronizace procesů, které v systému figurují. Petriho sítě jsou navrženy tak, aby se tato situace dala vhodně modelovat.



Obrázek 2.5: Příklad synchronizace procesů v grafové reprezentaci Petriho sítí

Na obrázku 2.5 části a) je možné vidět příklad Petriho sítě ve které figurují dva nezávislé procesy. První proces je modelován vstupním místem $p1$ přechodu $t1$ a místo $p2$, které je jeho výstupním místem. Druhý proces je hned pod prvním modelován se stejnou strukturou, ale rozdílnými identifikátory uzlů, aby je bylo možno odlišit. Pokud bychom tyto procesy chtěli synchonizovat, aby se dané události modelované přechody $t1$ a $t2$ provedly současně, pak je možné tento systém modelovat tak, jak je znázorněno v části b). Obě vstupní místa

přechodů $t1$ a $t2$ je možné směřovat do jednoho přechodu a z daného přechodu (konkrétně $t1$) do původních výstupních míst $p3$ a $p4$. To zapříčiní, že bude proveditelný až tehdy, pokud budou značky umístěny v obou vstupních místech.

2.3 Varianty Petriho sítí

Petriho sítě jsou v klasické podobě, jak byly popsány v kapitole 2.1 někdy také označovány jako PTN (Place-Transition Nets) nebo PTPN (Place-Transition Petri Nets). Existuje však velké množství rozšíření této základní varianty. Byly navrhovány kvůli myšlence přiblížit se co nejvíce implementačním prostředkům systémů. Důvod je především takový, aby přechod z modelu systému Petriho sítí na implementaci samotného systému byl co možná nejjednoduší. V následující části jsou uvedeny některé příklady rozšíření Petriho sítí se stručným komentářem, které přinesly konceptu Petriho sítí poměrně důležité změny z hlediska formy návrhu.

2.3.1 Stochastické Petriho sítě

Koncept SPN (Stochastic Petri Nets) přidává k PTN sítím stochastické vlastnosti. To spočívá v tom, že z každého přechodu se může stát tzv. časový přechod, který má definovaný časovač. Tomuto časovači je vhodně přiřazena hodnota podle pravděpodobnostního rozdělení, která se v průběhu provádění výpočtu SPN sítě dekrementuje v případě, že je přechod proveditelný. Přechod je proveden až v případě, že jeho časovač dosáhl hodnoty nula. Existují tři základní přístupy zacházení s hodnotou časovače. Těmi jsou resampling (časovače přechodů jsou restartovány v případě každého provedení přechodu), enabling memory (pouze vypnuté časovače přechodu jsou restartovány) a age memory (časovače přechodů jsou restartovány pouze pokud dosáhnou nuly). Důvodem zavedení těchto časovačů je zamezení nežádoucím konfliktům mezi přechody. Mezi další způsoby řešení těchto konfliktů patří například zavedení priorit přechodů.[17]

2.3.2 Barevné Petriho sítě

Jedná se o rozšíření PTN sítě, které se označuje jako CPN (Colored Petri Nets) a spočívá v tom, že tokeny nyní mohou být spojeny hodnotou reprezentující data. Místa v CPN síti jsou označovány datovými typy, podle toho jakých hodnot tokenů mohou nabývat. Analogicky hrany indikují hodnoty tokenů, které jsou odebírány, anebo vkládány do míst. Případně je možné využití datových proměnných. Co se týče přechodů, tak mohou mít kromě podmínky ze vstupních míst definovanou tzv. stráž, která funguje jako binární podmínka, která musí být také splněna, aby byl přechod proveditelný.[4]

2.3.3 Hierarchické Petriho sítě

Představení CPN sítí přineslo značnou redukci velikosti modelů, což vedlo i ke zvýšení čitelnosti těchto modelů. Nicméně to stále nebylo dostatečné, a proto vznikly různé druhy strukturovacích technik Petriho sítí. Substituce přechodů, substituce míst, fúze a invokace přechodů jsou nejběžnější z nich.[17] Tyto techniky příspívají k další možné redukci velikosti tvořených modelů, zejména pokud se některé části sítě opakují. Jedním z konceptů zavedení hierarchie do Petriho sítí je například definice hierarchických CPN sítí (HCPN), kterou definoval Kurt Jensen ve své publikaci.[6]

2.3.4 Objektově orientované Petriho sítě

Objektově orientované Petriho sítě (OOPN) jsou založeny na CPN sítích, které jsou obohaceny o objektově orientované strukturování. Jinými slovy řečeno, zavedení tříd, objektů, metod a podobně. Toto obohacení poskytuje možnost zapouzdřit různé vlastnosti a chování do objektů, které mohou být navázány na tokeny, jejichž prostřednictvím se mohou přesunovat do jiných míst v síti. Díky tomu je možné přistupovat k metodám daného objektu z více míst. Metody objektů mohou být volány jako akce přechodu v případě, že je přechod proveditelný a nastane provedení přechodu. Jedna z hlavních výhod tohoto typu sítí je možnost využití dědičnosti, která nabízí možnost inkrementální definice tříd a sítí s nimiž spojených.[17] V kapitole 3 budou OOPN popsány podrobněji.

Kapitola 3

Objektově orientované Petriho sítě

Hlavním smyslem formalismu Objektově orientovaných Petriho sítí (OOPN), jenž definoval doc. Vladimír Janoušek ve své práci Modelování objektů Petriho sítěmi [5], je docílit kompletní integraci objektové orientace do formalismu Petriho sítí, a tím využít výhod obou těchto paradigmat. Objektově orientovaný přístup nabyl popularity vzhledem k jeho dobrým vlastnostem, co se týče schopnosti strukturování, což zdůrazňuje zapouzdření a podporuje znovupoužití již existujícího softwaru.[4]

3.1 Struktura OOPN

Modely popsané pomocí Objektově orientovaných Petriho sítí se skládají z Petriho sítí organizovaných do tříd. Každá třída obsahuje síť objektu a množinu dynamicky instanciovatelných metod sítě. Objektová síť podobně jako metody sítě mohou být zděděny z jiné třídy. Značky (tokeny) v Objektově orientované Petriho síti jsou reprezentovány jako *primitivní objekty*, to je například číslo a řetězec nebo *neprimitivní objekty*, jimiž se rozumí instance některé z tříd popisující OOPN. Každý neprimitivní objekt se skládá z jedné objektové sítě a případně z několika souběžně běžících instancí invokovaných metod sítí.[9]

3.1.1 Třídy a objekty

Třídy jsou definovány jednou *sítí objektu*, množinou *sítí metod*, množinou *synchronních portů* a množinou selektorů zpráv odpovídající sítím metod a synchronním portům. Zároveň mohou být tyto třídy definovány inkrementálně za pomoci dědičnosti.[9]

Systém popsaný modelem OOPN se skládá z množiny objektů, které mohou v průběhu výpočtu dynamicky vznikat a zanikat. Každý objekt je v systému jednoznačně odlišitelný nezávisle na čase. Zároveň stav, ve kterém se objekt nachází, se v čase může měnit. Existují dva způsoby jak ovlivnit stav objektu.[5].

- **Implicitně** - Provedení vnitřní události objektu
- **Explicitně** - Vykonáním služby (metody) vyvolané zaslaním zprávy jiným objektem

Každý objekt poskytuje rozhraní služeb, které umožňují změnit stav objektu jiným objektem v systému. Realizace jednotlivých služeb je v režii objektu samotného v podobě metod objektu. Invokaci (vyvolání) metody je možné provést zasláním zprávy objektu, jehož metoda má být vyvolána. *Zpráva* obsahuje jméno objektu, jméno požadované služby a případné parametry. Objekt reaguje na příchozí zprávu tím, že vyhledá patřičnou metodu a zahájí její vykonávání.[5]

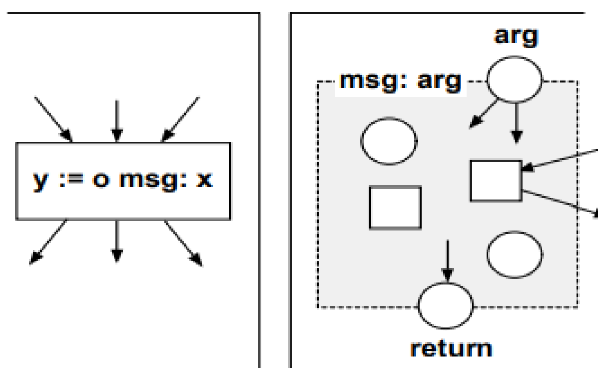
3.1.2 Síť objektu

Skládá se z míst a přechodů. Jejím účelem je uchovat aktuální stav objektu a provádění implicitních aktivit objektu. Jinými slovy by se dalo říct, že se jedná o analogii atributů, které jsou známé z klasického objektově orientovaného paradigmatu obohacené o možnost implicitních akcí objektu. Každý přechod v síti objektu může mít definované podmínky (testovací hrana, oboustraně orientovaná), vstupní podmínky (vstupní hrana), stráž, akci a výstupní podmínky (výstupní hrana).[9]

Místa v OOPN mohou obsahovat tokeny různých datových typů jako je tomu u CPN sítí, jelikož OOPN vychází z CPN sítí, což bylo zmíněno v podkapitole 2.3.4. Zároveň také mohou místa obsahovat určitý počet prvků několika různých datových typů. Tedy místa obsahují multimnožinu prvků (tokenů), které mohou být různých datových typů a v definovaném počtu, což je značeno pomocí multimnožinového operátoru. Např. $1‘a+2‘b+3‘c$ by znamenalo, že multimnožina obsahuje 1-krát znak a, 2-krát znak b a 3-krát znak c.

3.1.3 Sítě metod

Sítě metod jsou podobné síti objektu, ale každá z nich má definovanou množinu parametrů a návratové místo. Přechody ze sítí metod mohou přistupovat do míst sítě objektu příslušného objektu třídy OOPN, aby mohly upravovat jeho stav. Sítě metod jsou dynamicky instanciovány při zaslání zprávy specifikované v akci přechodu. Tato zpráva v sobě obsahuje selektor dané metody a její parametry.[9] Na obrázku 3.1 je možné vidět příklad sítě metody, kde na levé straně obrázku je uvedeno případné volání metody se selektorem zprávy `msg:` se vstupním parametrem `x` a na pravé straně obrázku je znázorněna struktura sítě metody se selektorem `msg:`, vstupním místem `arg` a výstupním místem `return`.



Obrázek 3.1: Příklad sítě metody.[9]

3.1.4 Synchronní porty

Jejich záměrem je zavést schopnost synchronní interakce s jinými objekty. Jedná se o hybridní ztvárnění metody a přechodu, který dovoluje již zmíněnou synchronní interakci. Co se týče provedení synchronního portu, tak musí být nejprve zavolán (jako je tomu u metod) a zároveň musí být ve stavu proveditelný, podobně jako je tomu u přechodu. Další podmínkou je to, že synchronní port může být volán pouze ze stráže přechodu. Při volání metody je nutné definovat její parametry (navázat je na hodnoty), což je rozdíl oproti synchronnímu portu, jelikož synchronní port může být volán s navázanými i nenavázanými parametry.

V případě, že je zavolán s nenavázanými paramery, tak hledá potenciální možné navázání dynamicky a pokud takové existuje, tak jsou dané hodnoty navázány k parametrům synchronního portu. To znamená, že mohou být využity nejen k testování proveditelnosti přechodu, ale i k uložení objektů v místech jiného objektu. Speciální variantou synchronního portu je tzv. negativní predikát, jehož sémantika je obrácená. Tedy je možné provést přechod v případě, že negativní predikát je neproveditelný.[9]

3.1.5 Místa, přechody a hrany

Základními stavebními bloky sítí objektů jsou místa, přechody a hrany. Místa a jejich značení v podobě tokenů představují parciální stavy sítě. Stav sítě může být změněn provedením přechodu, který je možné provést, pokud je přechod proveditelný. Hrany v síti jsou brány jako podmínky ovlivňující proveditenost přechodu, přičemž každá podmínka je spjata s jedním místem a přechodem.[5]

Proveditelnost přechodu

Přechod se stane proveditelným tehdy, pokud jsou splněny vstupní a výstupní podmínky přechodu podobně, jako je tomu u klasických Petriho sítí, obohacené o stráž přechodu. Stráž přechodu se skládá ze sekvence výrazů, kde výrazem je myšleno zaslání zprávy. Tato sekvence má charakter konjunkce, což znamená, že nabude logické hodnoty `true` pouze tehdy, pokud výsledkem každého dílčího zaslání zprávy je logická hodnota `true`. Jednotlivé dílčí výrazy jsou vyhodnocovány různě na základě adresáta zprávy.[5]

Ve výrazu stráže může být adresátem zprávy:

- Primitivní objekt - Výraz se vyhodnotí v rámci inskripčního jazyka.
- Synchronní port - Výraz má hodnotu `true` tehdy, pokud je synchronní port proveditelný. Jinak má hodnotu `false`.

Provedení přechodu

Přechod může být rozšířen o akci, která speficikuje zaslání zprávy. Provedení přechodu se provede buď atomicky nebo neatomicky, což se vyhodnocuje až za běhu na základě adresáta zprávy.[5]

V akci přechodu může být adresátem zprávy:

- Primitivní objekt - Přechod a vyhodnocení akce se provede atomicky.
- Třída - Přechod se provede atomicky a při zprávě typu `new` dojde k vytvoření nové instance třídy, která je adresátem zprávy.
- Neprimitivní objekt - Přechod se provede neatomicky. Provedení probíhá postupně následujícím způsobem:
    1. Nejdříve se provede vstupní část přechodu, tj. odebrání vstupních značek ze vstupních míst přechodu. Dále proběhne vyhledání metody odpovídající selektoru zprávy. Vytvoření nové instance sítě metody a umístění značek do vstupních míst sítě metody z parametrů zprávy.

2. Dále přechod čeká, než se volaná metoda dokončí. Mezi tím se mohou nezávisle provádět další přechody, včetně tohoto přechodu.
3. Jako poslední krok je dokončení provádění přechodu, které nastane tehdy, když se do výstupního místa sítě metody umístí nějaká značka. Dokončení přechodu provede zrušení instance dané sítě metody a objekt nacházející se ve výstupním místě sítě je brán jako výsledek po zaslání zprávy, který může být přiřazen do proměnné. Na závěr proběhne výstupní část přechodu, tj. umístění značek do výstupních míst přechodu.

Události

Systém OOPN se prováděním událostí dostává do nových stavů, tak jako je tomu u klasických Petriho sítí. V OOPN existují čtyři typy možných událostí. Všechny tyto události jsou provedeny atomicky.[5]

- Událost typu A - Jedná se o provedení přechodu uvnitř jednoho objektu, přičemž synchronní komunikace může ovlivit stav jiných objektů.
- Událost typu N - Znamená provedení přechodu a vytvoření nového objektu pomocí zprávy `new`.
- Událost typu F - Předání zprávy neprimitivnímu objektu, přičemž se odebrou značky ze vstupních míst přechodu a vytvoří se nová instance sítě metody podle selektoru zprávy.
- Událost typu J - Představuje dokončení události typu F, a to akceptováním odpovědi zprávy ve formě předání výsledku ze sítě metody, zrušení instance sítě metody a umístění značek do výstupních míst přechodu.

3.2 Vlastnosti objektové orientace

Tato podkapitola obsahuje popis základních vlastností objektově orientovaných systémů, které jsou jedny z hlavních důvodů k využívání objektově orientovaného paradigmatu. Mezi tyto vlastnosti patří zapouzdření, dědičnost a polymorfismus.

3.2.1 Zapouzdření

Tento koncept zajišťuje na úrovni definice sémantiky jazyka to, že uživatel nemůže měnit interní stav objektu přímo, což by mohlo vést k neočekávanému chování objektu. Stav objektu lze měnit pouze skrze rozhraní, které bylo předem definované. Pokud s daným objektem chtějí komunikovat i ostatní objekty, tak mohou pomocí operací, které nabízí rozhraní daného objektu v podobě jeho metod. Díky tomu, že lze s objektem komunikovat pouze skrze toto rozhraní, nabízí výhodu toho, že skutečná implementace metod, které poskytuje, je dokonale skryta. V tom spočívá výhoda konceptu zapouzdřední, jelikož je možné zachovat zpětnou kompatibilitu při dodržení původního rozhraní, i když se změní implementace původních metod.[10]

3.2.2 Dědičnost

Jedná se o vlastnost, která umožňuje implementovat sdílené chování. Nové objekty mohou převzít chování jiných bez nutnosti reimplementace a s případnou možností redefinovat část chování objektu nebo ho rozšířit o nové. V objektově orientovaných jazycích existují různé formy dědičnosti. Mezi ty patří například statická nebo dynamická dědičnost. Dále také jednoduchá nebo násobná dědičnost. S jednoduchou dědičností je možné se setkat u jazyku Smalltalk a s násobnou dědičností například u jazyku C++.[5] Existují dva hlavní důvody využití dědičnosti v praxi, a těmi jsou:

- Rozšíření chování některého objektu
- Sdílení kódu obecných metod

Příkladem rozšíření chování některého objektu by mohla být dvojice objektů *uživatel systému* a *administrátor systému*. *Uživatel systému* obsahuje implementaci základního chování pro práci se systémem a objekt *administrátor systému* rozšiřuje chování objektu *uživatele systému* například o funkce správy systému apod. Jako příklad případu sdílení kódu obecných metod lze uvést třeba třídu `Objekt` v jazyce Java, která implementuje metody, jimž by měli rozumět všechny existující objekty. Vlastnost dědičnosti velmi přispívá k udržitelnosti a rozšiřitelnosti (robustnosti) objektově orientovaných systémů.[10]

3.2.3 Polymorfismus

Tento pojem znamená jinými slovy mnohotvárnost. V objektově orientovaném programování se využívá s mechanismem zasílání zpráv, který se v objektově orientovaných jazycích používá místo klasického volání podprogramů (procedur a funkcí). Polymorfismus je vlastně koncept, když proměnná při běhu programu může odkazovat na různé objekty, a ty reagují na stejnou zprávu odlišným chováním. Tím se rozumí vyvolání metody patřící konkrétnímu specializovému objektu v dané proměnné.[10]

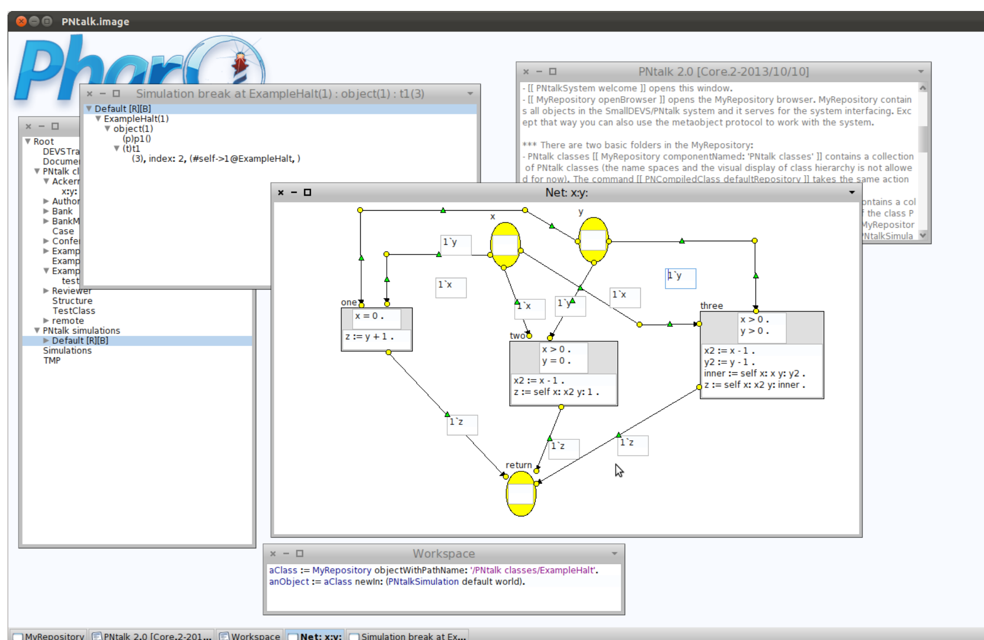
Příkladem polymorfismu by mohl být například tento. Mějme třídu *zvíře*, která by byla braná jako obecná třída. Dále by existovaly dvě různé třídy *pes* a *kočka*, které rozšiřují třídu *zvíře* (využita dědičnost 3.2.2) a všechny tyto tři třídy obsahují metodu *vydejZvuk*. Dále můžeme uvažovat následující případy běhu programu. V případě, že by se objekt typu *pes* uložil do proměnné typu *zvíře* a byla mu zaslána zpráva *vydejZvuk*, tak by výstup měl reprezentovat „štěkání“. Naopak, kdyby se do proměnné typu *zvíře* uložil objekt typu *kočka*, tak potom by se výsledek zaslání zprávy *vydejZvuk* reprezentoval „mňoukání“. V obou případech byla zaslána stejná zpráva proměnné typu *zvíře* s různou reakcí na tuto zprávu, čímž bylo docíleno polymorfismu.

3.3 Existující nástroje

V této sekci budou zmíněny dva existující nástroje pro vykonávání simulace Objektově orientovaných Petriho sítí. Prvním z těchto nástrojů je systém PNtalk, který je vyvíjen a spravován na fakultě informačních technologií na univerzitě Vysoké učení technické v Brně. Jako druhý simulační nástroj je zde zmíněn software Renew. Ten je vyvíjen a spravován skupinou Theoretical Foundation Group, Department for Informatics, University of Hamburg.

3.3.1 Systém PNtalk

Systém PNtalk [7] implementuje formalismus Objektově orientovaných Petriho sítí, jehož autorem je doc. Vladimír Janoušek. Systém PNtalk je experimentální software určený k výzkumným a studijním účelům. Systém je implementován v jazyku Squeak Smalltalk a vývojářem toho softwaru je dr. Radek Kočí. Nejnovější verzí toho systému je PNtalk 2.0 [8], který obsahuje novou implementaci a rozšíření původní verze. Byl změněn taktéž programovací jazyk z již zmíněného Squeak Smalltalk na Pharo 2.0. Nástroj PNtalk 2.0 je frameworkem, který umožňuje modelování a simulaci formalismu Objektově orientovaných sítí. Podobně jako první verze systému PNtalku, tak je tento framework určen k výzkumným a studijním účelům.

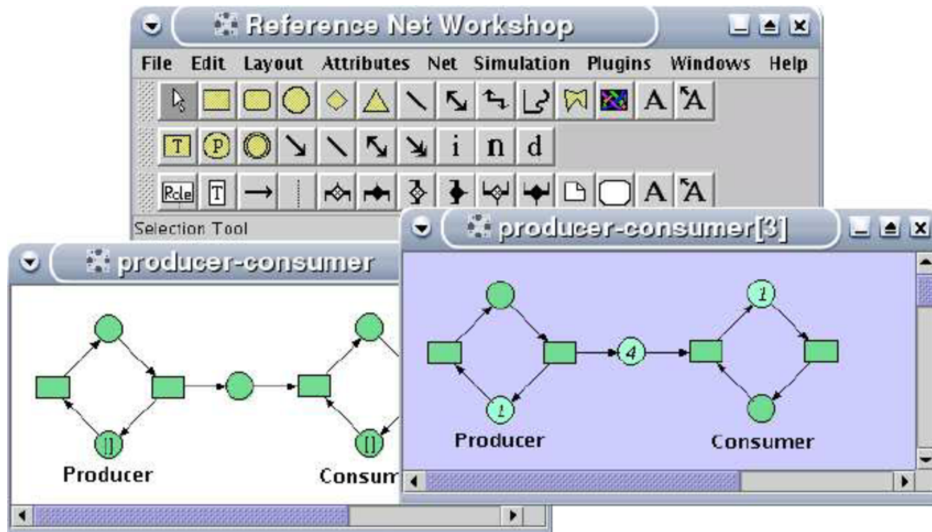


Obrázek 3.2: PNtalk 2.0 [8]

3.3.2 Nástroj Renew

Renew je editor a simulátor založený na jazyku Java, který poskytuje flexibilní přístup k modelování tzv. sítí s referencemi. Formalismus popisující tyto sítě s referencemi je založený na principu objektově orientovaného přístupu, kde tokeny předávané v síti mohou být reference na libovolný objekt. Těmito objekty jsou především myšleny objekty reprezentující jiné sítě, což umožňuje vnořování sítí.[15]

Na obrázku 3.3 je zobrazeno grafické prostředí editoru nástroje Renew, ve kterém je namodelovaná jednoduchá Petriho síť v pozadí a instance této sítě v popředí. Uživatelské rozhraní poskytuje panel s menu tlačítky, dvě palety nabízející nástroje prvky k modelování sítí a stavovou lištu.[1]



Obrázek 3.3: Renew - Petri net and net instance [1]

Kapitola 4

Jazyk PNtalk

Tato kapitola se bude věnovat jazyku PNtalk, jehož název vznikl ze spojení „Petri Nets and Smaltalk“. Při studiu jazyka PNtalk bylo čerpáno z práce Modelování objektů Petriho sítěmi [5], jejímž autorem je doc. Vladimír Janoušek.

Jazyk vznikl za účelem možnosti zápisu Objektově orientovaných Petriho sítí. Vychází tedy přímo z definice Objektově orientovaných Petriho sítí, ale má jisté syntaktické odlišnosti inspirované jazykem Smalltalk. V podstatě tyto odlišnosti zpočívájí v rozšíření, která umožňují snazší zápis složitějších konstrukcí.

4.1 Termy

Termy jsou základní výrazy v jazyce PNtalk a definují se nimi objekty. Termy se dají rozdělit do několika skupin:

1. **Literály** - Fungují jako identifikátory prvků v PNtalku. Dále je lze rozdělit na:

   (a) *Čísla* - Reprezentují číselné hodnoty.

   `1, -2, 456, 0.004, 1.34e10, 1.35e-15, ...`

   (b) *Znaky* - Označují jednotlivé symboly vstupní abecedy a je nutno je uvozovat symbolem dolar.

   `$a, $B, $1, $+, $$, ...`

   (c) *Řetězce* - Jsou sestaveny ze sekvence znaků vstupní abecedy a jsou umístěny do apostrofů. V případě nutnosti použití apostrofu v řetězci, je potřeba jej zdvojit.

   `'string', 'I''m', ...`

   (d) *Symboly* - Používají se typicky jako jména a jsou reprezentovány sekvencí symbolů vstupní abecedy uvozenou mřížkou. Oproti řetězcům se liší tím, že pokud jsou zapsané dva stejné symboly, pak označují stejný objekt.

   `#abc, #123, #'a5', ...`

   (e) *Pole* - Jedná se o rozšíření jazyka PNtalk. Pole je z obou stran ohraničené závorkami a může obsahovat seznam termů oddělených bílými znaky. Jelikož je pole term, tak může být obsaženo i jako prvek jiného pole. Pole na nejvyšší úrovni musí být uvozeno mřížkou.

   `#(1 'string'), #(($a $b) 5), ...`

(f) *Boolovské konstanty* - Existují pouze dvě, a to jako klíčová slova označují logické pravdivostní hodnoty.

`true, false`

(g) *Nedefinovaný objekt* - Označuje nedefinovaný prvek a značí se klíčovým slovem `nil`. Všechny neinicializované proměnné nabývají hodnoty `nil`.

2. **Proměnné** - Sekvence znaků začínající malým písmenem. V průběhu výpočtu se může měnit hodnota, kterou proměnná nabývá.

`a, b, isNum, ...`

3. **Pseudoproměnné** - Existují pouze dvě, a to jako klíčová slova `super` a `self`. Klíčové slovo `super` vyjadřuje objekt rodičovské třídy a `self` objekt aktuální třídy. Tedy hodnota, kterou nabývají záleží na kontextu v průběhu výpočtu.

`super, self`

4. **Jména tříd** - Jedná se o identifikátory, které začínají velkým písmenem a v průběhu výpočtu je s nimi zacházeno jako s konstantami.

`A, B, Num, ...`

4.2 Základní stavební bloky

Sítě se skládají z míst a přechodů, které jsou propojovovány orientovanými hranami. Propojovací hrany kromě toho, že propojují místa s přechody, tak navíc obsahují hranový výraz, který dále v průběhu výpočtu určuje, jaké značky odebírat nebo vkládat do míst, k němuž je daná hrana připojena.

Hranové výrazy

Používají se při specifikaci počátečního značení míst a na hranách sítě. Hranové výrazy představují multimnožiny a mají následující tvar

$$n_1'c_1, n_2'c_2, \ldots, n_m'c_m,$$

kde n_i je term[1], a c_i je term nebo seznam. Seznam má tvar

$$(e_1, e_2, \ldots, e_m),$$

kde e_1 je term, anebo seznam. V jazyce PNtalk je možné zapsat seznam i Prologovskou syntaxí

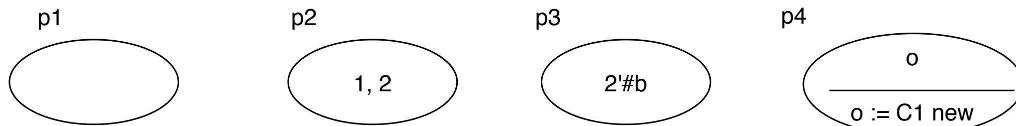
$$(h_1, h_2|z),$$

kde h_i jsou termy a z je zbytek seznamu. Jako příklad hranového výrazu můžeme zvolit `2‘#a, x‘b, 1, 3‘(x,y)`, který reprezentuje multimnožinu obsahující dvakrát symbol `#a`, `x`-krát `b`, číselný literál `1` a třikrát dvojici `(x,y)`.

[1] Očekávaná hodnota je kladné celé číslo, jinak je považován za nulový.

4.2.1 Místo

Místo se v PNtalk grafické podobě značí jako kružnice nebo elipsa. Každé místo může mít specifikováno počáteční značení. To je ve tvaru hranového výrazu. V systému PNtalk se sítě většinou specifikují graficky a následně převádí do strojové (textové) podoby. Gramatika popisující jazyk PNtalk je uvedena v příloze A. Na obrázku 4.1 jsou uvedeny příklady míst a jejich značení.



Obrázek 4.1: Příklad sítě modelované v jazyce PNtalk

Jelikož pro strojové zpracování je výhodnější textová forma zápisu oproti grafické, takže dále v textu budou podstatné konstrukty jazyka PNtalk uvedeny převážně v textové variantě zápisu. V textové podobě jazyka PNtalk by místa z 4.1 byly zapsány následovně:

p1	`place p1()`
p2	`place p2(1,2)`
p3	`place p3(2‘#b)`
p4	`place p4(o) init {o := C1 new}`

4.2.2 Přechod

Přechod se v PNtalk grafické podobně značí jako čtverec nebo obdelník. Z hlediska přechodu existují tři typy hran:

1. **Vstupní hrana** - Orientovaná směrem k přechodu. Specifikuje značky odebírané z místa při provedení přechodu. Značí se klíčovým slovem `precond`. Příklad hrany v jazyku PNtalk odebírající z místa `p1` číslo 1 při provedení přechodu:

   ```
   precond p1(1)
   ```

2. **Výstupní hrana** - Orientovaná směrem k místu. Specifikuje značky vkládané do místa při provedení přechodu. Značí se klíčovým slovem `postcond`. Příklad hrany v jazyku PNtalk vkládající do místa `p1` číslo 1 po provedení přechodu:

   ```
   postcond p1(1)
   ```

3. **Testovací hrana** - Orientovaná oběma směry. Pouze testuje přítomnost konkrétního značení, přičemž když nastane shoda, tak se daný přechod vykoná. Značí se klíčovým slovem `cond`. Příklad hrany v jazyku PNtalk testující existenci čísla 1 v místě `p1` pro provedení přechodu:

   ```
   cond p1(1)
   ```

Přechod může mít v sobě zabudovanou akci, případně stráž přechodu.

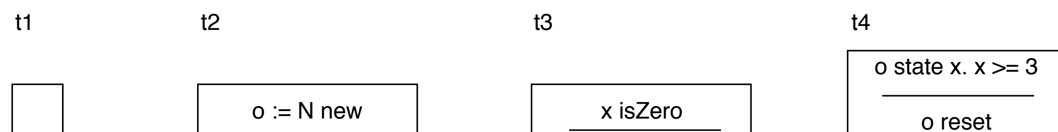
1. **Stráž přechodu** - Skládá se ze sekvence výrazů oddělených od sebe tečkou. Každý výraz v sekvenci se rozumí zasláním zprávy. V grafické podobě je stráž přechodu podtržena.

   `<zaslání zprávy>`$_1$. `<zaslání zprávy>`$_2$ ... `<zaslání zprávy>`$_n$

2. **Akce přechodu** - Obsahuje buď zaslání zprávy, anebo výraz přiřazení ve tvaru proměnná následovaná symbolem přiřazení a zasláním zprávy.

   `<proměnná> := <zaslání zprávy>`

Na obrázku 4.2 jsou uvedeny různé varianty přechodu a jejich značení.



Obrázek 4.2: Příklad sítě modelované v jazyce PNtalk

Rozsah platnosti proměnných zahrnuje akci a stráž přechodu s výrazy z okolních hran. U míst rozsah platnosti zahrnuje počáteční značení a počáteční akci. V textové podobě jazyka PNtalk by přechody z 4.2 byly zapsány následovně:

```
t1    trans t1
t2    trans t2
        action {o := C1 new.}
t3    trans t3
        guard {x isZero}
t4    trans t4
        guard {o state: x. x >= 3}
        action {o reset.}
```

4.3 Zasílání zpráv

Jedná se o určitý typ výrazu, který se může vyskytnout jako akce přechodu případně v akci stráže. Zprávy se skládájí ze selektoru (operace) a argumentů (operandy dané operace). Zaslání zprávy má tvar:

`<adresát> <zpráva>`

Existují tři typy zpráv, které se dělí podle selektorů.

1. **Unární zpráva** - Zprávy tohoto typu nemají žádné argumenty a selektor nesmí ve svém identifikátoru obsahovat znak dvojtečka. Příklady unárních zpráv jazyka PNtalk:

   Pro číselné hodnoty:
   - *abs* — Absolutní hodnota
   - *negated* — Opačná hodnota
   - *truncate* — Odseknutí čísla za desetinou čárkou
   - *rounded* — Zaokrouhlení

   Pro boolovské hodnoty:
   - *not* — Negace

   Pro třídy popisující Petriho sítě:
   - *new* — Zavolání konstruktoru třídy

   Například u zaslání zprávy ve tvaru `1 negated`, by bylo číslo `1` považováno jako adresát zpávy a `negated` jako selektor zprávy.

2. **Binární zpráva** - Zprávy tohoto typu mají právě jeden argument a mohou být použity pouze následující selektory:

   + − * / = < > ~= <= >= & | // \\ , == ~==

   Pro číselné hodnoty:
   - + — Sčítání
   - − — Odčítání
   - * — Násobení
   - / — Dělení
   - < — Menší než
   - > — Větší než
   - // — Celočíselné dělení
   - \\ — Zbytek po celočíselném dělení
   - <= — Menší nebo rovno
   - >= — Větší nebo rovno

   Pro boolovské hodnoty:
   - & — Logický AND
   - | — Logické OR

   Pro všechny objekty:
   - = — Rovnost
   - ~= — Nerovnost
   - == — Rovnost identity
   - ~== — Nerovnost identity

   Detailnější popis sémantiky selektorů je možno nalézt v popisu jazyka Smalltalk.[3] Například u zaslání zprávy tvaru `1+2` by se dala tato sekvence rozložit na adresáta `1`, selektor `+` a jeho argument `2`. (`1+2, 10-1, 4==4`)

3. **Zpráva s klíčovými slovy** - Zpráva tohoto typu obsahuje jeden, anebo více klíčů, které jsou zakončeny dvojtečkou, za kterou následuje argument daného klíče. Například u zaslání zprávy tvaru `netAdd x: 1 y: 2` by se dala tato sekvence rozložit na adresáta `netAdd`, selektor `x:y:` a jeho argumenty `1` a `2`.

V jazyce PNtalk existují dva druhy zaslání zpráv. Jednoduché zaslání zprávy a složené zaslání zprávy. Pokud adresát a argumenty zprávy jsou termy, potom se jedná o jednoduché zaslání zprávy. Jinak se jedná o složené zaslání zprávy, přičemž priorita vyhodnocování výrazu je brána v pořadí zleva doprava. Tedy vyhodnocování při zaslání zprávy `1+2*3` by probíhalo následovně `1+2*3 -> 3*3 -> 9`. Pro upřednostnění násobení před sčítáním by se muselo provést následující uzávorkování `1+(2*3) -> 1+6 -> 7`.

4.4 Třídy

Model je v jazyku PNtalk reprezentován množinou tříd, které si mezi sebou navzájem zasílají zprávy, čímž spolu komunikují. Model obsahuje vždy jednu výchozí třídu (je instanciována na začátku výpočtu), která je v textové formě PNtalk uvedena klíčovým slovem `main`. Tedy nastavení výchozí třídy s názvem „Class0“ by vypadalo následovně:

```
main Class0
```

Každá třída se skládá z jedné sítě objektu, množiny sítí, konstruktorů a synchronních portů.

4.4.1 Konstruktory

Jedná se o speciální případ metody, která se volá ve chvíli vytváření nové instance třídy. V jazyku PNtalk se konstruktor volá stejně tak jako u většiny ostatních objektově orientovaných jazyků pomocí klíčového slova `new`. Konstruktor slouží k parametrizované inicializaci stavu míst sítě objektu. V případě, že není explicitně definován, tak je využit implicitní konstruktor, který je definován pro každou třídu. Konstruktor vždy vrací hodnotu `self`, čímž se rozumí název vytvořeného objektu. V jazyku PNtalk se značí klíčovým slovem `constructor`. Příklad konstruktoru v textové podobě jazyka PNtalk:

```
class Class0 is_a PN
  object
    place p1 ()
  constructor init: x
    trans t
      precond x()
      postcond p1 (x)
      postcond return (self)
    place x ()
    place return ()
```

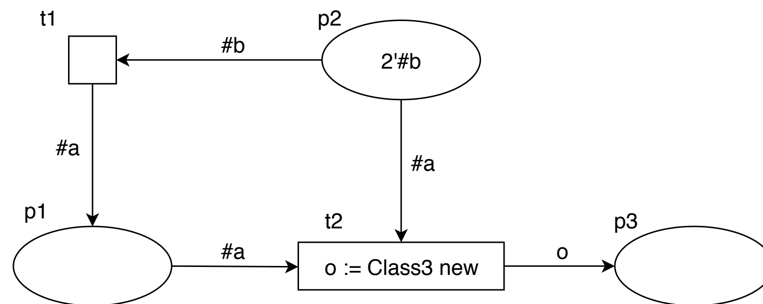
Pro vytvoření instance objektu třídy `Class0` se použije klíčové slovo `new` následováno zprávou `init: x`. To způsobí zaslání zprávy implicitnímu konstruktoru, který vytvoří samotnou instanci třídy, a hned poté se zavolá metoda explicitního konstruktoru, který vloží hodnotu z parametru proměnné x do místa sítě objektu `p1` dané třídy `Class0`.

4.4.2 Sítě

Každá třída může obsahovat sítě, přičemž existují dva druhy sítí, a to síť objektu a síť metod. Sítě vznikají propojením míst za pomoci přechodů.

1. **Síť objektu** - Může existovat pouze jedna a definuje stav objektu za pomoci míst a jejich obsahu. Také definuje jeho implicitní aktivitu definovanou přechody.

2. **Síť metody** - Ve třídě může být definován libovolný počet sítí metod, kde každá síť metod má vlastní vzor zprávy. Tento vzor obsahuje selektor zprávy a formální parametry. Každý z těchto parametrů je namapován do tzv. parametrických míst sítě metody. Při přijetí zprávy objektem s odpovídajícím vzorem zprávy se vytvoří nová instance sítě metody a vložení operandů do odpovídajících parametrických míst. Každá síť metody musí obsahovat jedno místo nazvané `return`, které slouží pro předání výsledku volajícímu objektu. Síť metody může obsahovat přechody, které jsou napojeny na místa sítě objektu.



Obrázek 4.3: Příklad sítě objektu modelované v grafické podobě jazyka PNtalk

Příklad sítě objektu třídy „Class1“ je uveden na obrázku 4.3. V textové podobě jazyka PNtalk by daná třída se sítí objektu byla definována následovně:

```
class Class1 is_a PN
  object
    place p1 ()
    place p2 (2‘#b)
    place p3 ()
    trans t1
      precond p2(#b)
      postcond p1(#a)
    trans t2
      precond p2(#a)
      precond p1(#a)
      action {o := Class3 new.}
      postcond p3(o)
```

4.4.3 Synchronní porty

Podobně jako sítě metod a konstruktory může třída obsahovat ještě tzv. synchronní porty. Jsou syntakticky podobné jako sítě metod, a tedy obsahují vzor zprávy a dále mohou obsahovat definice hran (vstupní, výstupní, testovací) s případnou stráží, ale neobsahuje žádné místa a přechody. Synchronní port slouží jako kanál pro komunikaci mezi jednotlivými objekty tříd a může být volán zasláním zprávy ze stráže přechodu. Akce, které s nimi mohou být provedeny jsou například testování stavu míst sítě jiného objektu, případně ovlivnění tohoto stavu. Speciálním případem synchronního portu je predikát, který obsahuje pouze testovací hranu, tudíž není schopný ovlivnit stav daného objektu. Příklad synchronního portu v textové podobě jazyka PNtalk:

```
class Class3 is_a PN
  object
    place p1 ()
  sync state: x
    cond p1 (x)
```

4.4.4 Dědičnost

Každá třída v jazyku PNtalk, co se týče dědičnosti má právě jednoho přímého předka. Rodičovská třída všech tříd definovaných Petriho sítěmi je třída `PN`, jejíž síť objektu je prázdná. Předkem abstraktní třídy `PN` už je jenom třída `Object`, která je na vrcholu této hierarchie a vychází z ní kromě třídy `PN` i primitivní objekty jazyka PNtalk (tj. konstanty) popsané v podkapitole 4.1. Třída `Object` je taktéž abstraktní a definuje reakci na zprávy, kterým by měly rozumět všechny objekty. To jsou například zprávy typu porovnání identity jako je rovnost nebo nerovnost. Třída popisující Petriho síť dědí od třídy `PN`, ale může dědit i od již definované třídy popisující nějakou Petriho síť. Taková třída potom obsahuje všechno, co její předci, tj. síť objektu, sítě metod, konstruktory a sychronní porty. Také je možné vše, co třída zdědila od svých předků, předefinovat uvedením nových definic se stejnými jmény. To platí například u synchornních portů, dále také míst a přechodů objektové sítě, ale v případě redefinice sítě metody se nahradí celá, a tedy použije nově definovaná síť metody.

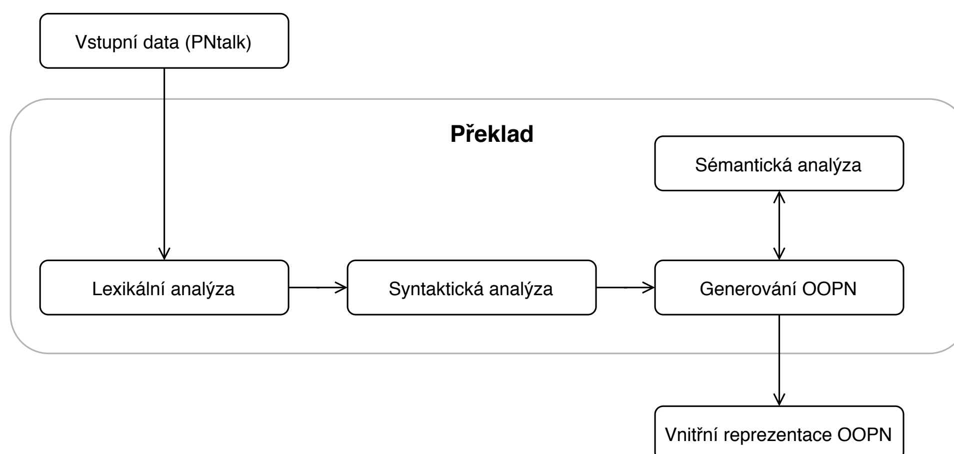
Kapitola 5

Překladač

V této kapitole bude popsán návrh a implementace překladače, který převádí vstup v jazyku PNtalk z textové podoby do vnitřní reprezentace pro další zpracování. Kapitola je rozdělena do dvou částí. V první části jsou uvedeny jednotlivé fáze překladu a jejich popis. Druhá část kapitoly je věnována návrhu a implementaci vnitřní reprezentace OOPN, která je brána jako výsledek překladu. V práci se vyskytují segmenty třídního diagramu popisující implementaci jednotlivých částí aplikace a z důvodu prostorových nároků v nich budou uvedeny pouze podstatné atributy a metody potřebné k výkladu, případně budou vynechány některé metody pro nastavování nebo získávání hodnot atributů objektů (angl. getter and setter methods). Celý třídní diagram popisující výsledný systém včetně programové dokumentace je k dispozici v elektronické příloze práce.

5.1 Fáze překladu

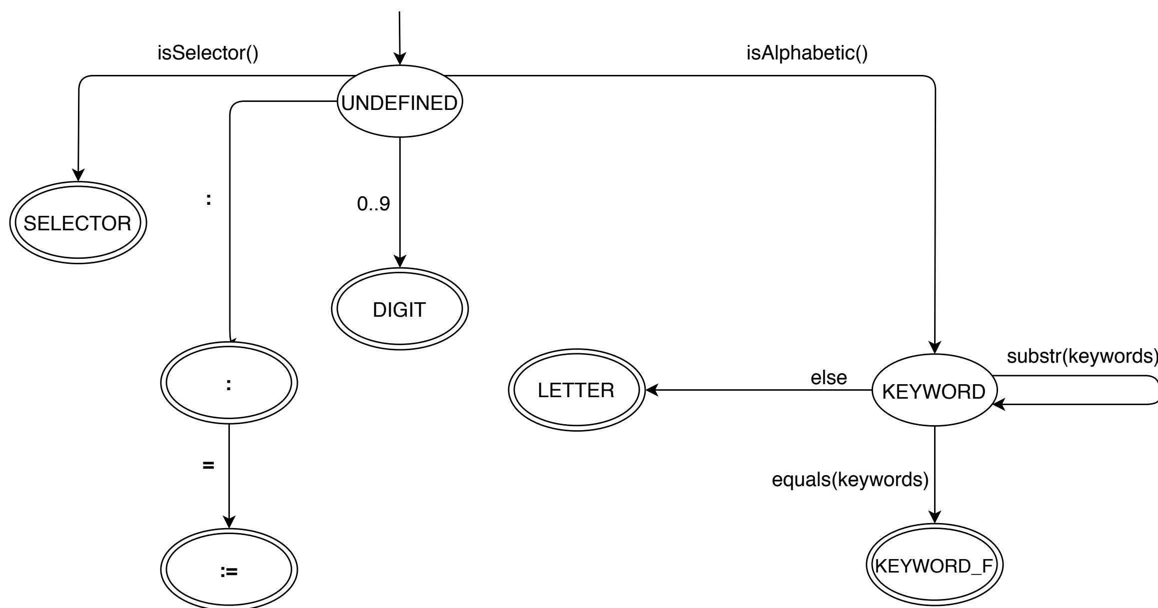
Proces překladu probíhá v několika fázích. První z nich se nazývá lexikální analýza, jejímž úkolem je ze vstupního souboru načítat a validovat lexémy daného jazyka. Druhou fází je syntaktický průchod vstupním řetězcem, čímž je provedena syntaktická analýza, podle předem definované gramatiky. A jako poslední fáze je generování dat do vnitřní reprezentace získaných z těchto analýz. Na diagramu 5.1 je vyobrazen chronologický postup fázemi překladu.



Obrázek 5.1: Fáze překladu

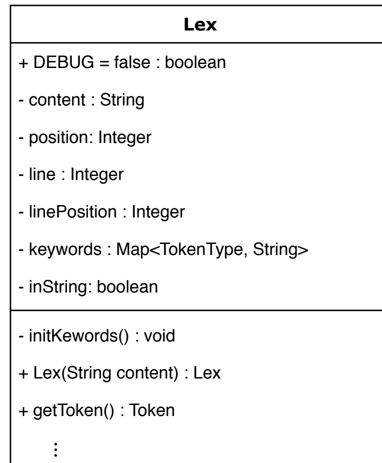
5.1.1 Lexikální analýza

Na vstupu aplikace je soubor obsahující zdrojový kód v jazyku PNtalk, který je potřeba zpracovat. Zpracování vstupu provádí lexikální analyzátor. Ten načítá jednotlivé lexikální symboly (lexémy) ze vstupního řetězce a vrací je ve formě tzv. tokenů. Na obrázku 5.2 je znázorněn konečný automat [11], podle kterého je implementován lexikální analyzátor.



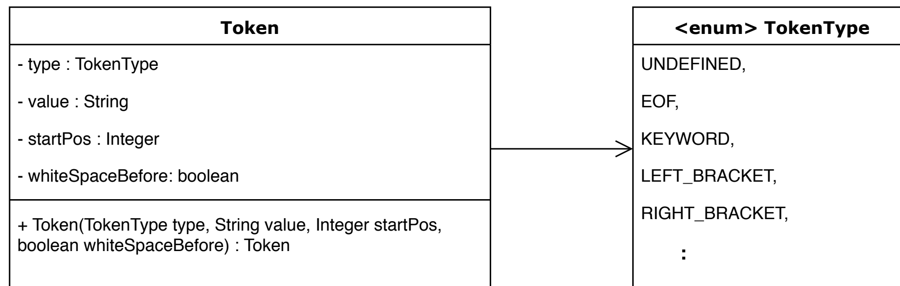
Obrázek 5.2: Konečný automat lexikální analýzy

Počáteční stav automatu je označen jako `UNDEFINED`. Při načítání vstupu je prováděno testování na různé hodnoty vstupních znaků, což vede k přesunu do dalších stavů automatu. První možností je, že vstupní znak patří do množiny binárních selektorů. V diagramu je tato část zjednodušeně reprezentována funkcí `isSelector` a následně koncovým stavem `SELECTOR`. Pokud znak nepatří do množiny binárních selektorů, tak je testován na hodnotu ‘ : ‘, případně dále na hodnotu ‘ = ‘ přiřazovacího symbolu ” := ”, přičemž jsou oba tyto stavy koncové. Další možností je, že znak reprezentuje číslici od 0 do 9, což by vedlo do koncového stavu `DIGIT`. Pokud znak není číslice, tak už zbývají jen dvě možnosti a to, že znak může být písmeno, což značí koncový stav `LETTER`, anebo začíná písmenem některého z klíčových slov, což značí koncový stav `KEYWORD`. Při načítání dalších znaků je testováno, jestli načtený řetězec je podřetězcem nějakého klíčového slova. V diagramu je tato část zaznačena jako `substr(keyword)`. Pokud je načtený řetězec podřetězcem některého z klíčových slov, tak automat setrvá ve stavu `KEYWORD`, případně pokud se danému klíčovému slovu rovná, tak dojde k přesunu do koncového stavu `KEYWORD_F`. V opačném případě, kdy načtený řetězec není podřetězcem žádného klíčového slova, tak nastane přesun do koncového stavu `LETTER` a token obsahuje pouze první znak z načteného řetězce. Nevyužitá část řetězce se vrátí zpět ke zpracování vstupu pro načtení dalšího tokenu.



Obrázek 5.3: Třída lexikální analýzy

Třída popisující lexikální analýzu, znázorněná na obrázku 5.3, uchovává data potřebná k načtení lexémů ze souboru tj. vstupní řetězec, pozice v souboru a klíčová slova jazyka PNtalk. Třída obsahuje metodu pro inicializaci seznamu klíčových slov. Dále obsahuje konstruktor, který má jako vstupní parametr vstupní řetězec se zdrojovým kódem v jazyku PNtalk. Hlavní metodou této třídy je metoda `getToken()`, která implementuje konečný automat z obrázku 5.2.

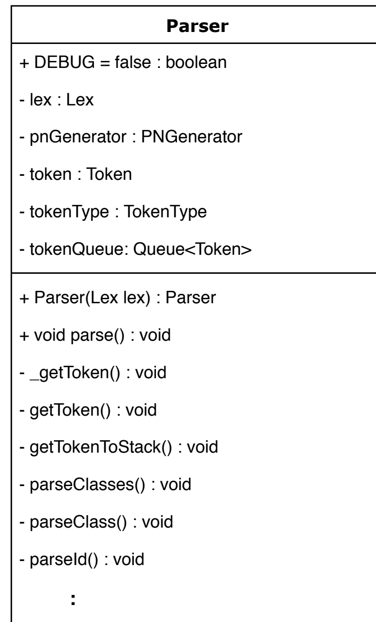


Obrázek 5.4: Třídy reprezentující Token

Na obrázku 5.4 jsou znázorněny třídy reprezentující token, který vrací lexikální analýza pomocí metody `getToken()`. Ten obsahuje informaci o typu tokenu, hodnotě, pozici v souboru, na níž se nachází jeho první znak a informaci, zda-li se před tímto tokenem nachází bílý znak. Třída `Token` obsahuje konstruktor, který vstupními parametry pokrývá všechny atributy objektu, a ty už se dále nemění.

5.1.2 Syntaktická analýza

Syntaktický analyzátor, někdy nazýván také jako „parser“, volá metodu `getToken()` z lexikální analýzy, čímž získává tokeny, které použije pro realizaci syntaktického průchodu celým vstupním řetězcem. Primárním účelem syntaktického průchodu je zkontrolovat, zda-li vstup odpovídá gramatice jazyka.



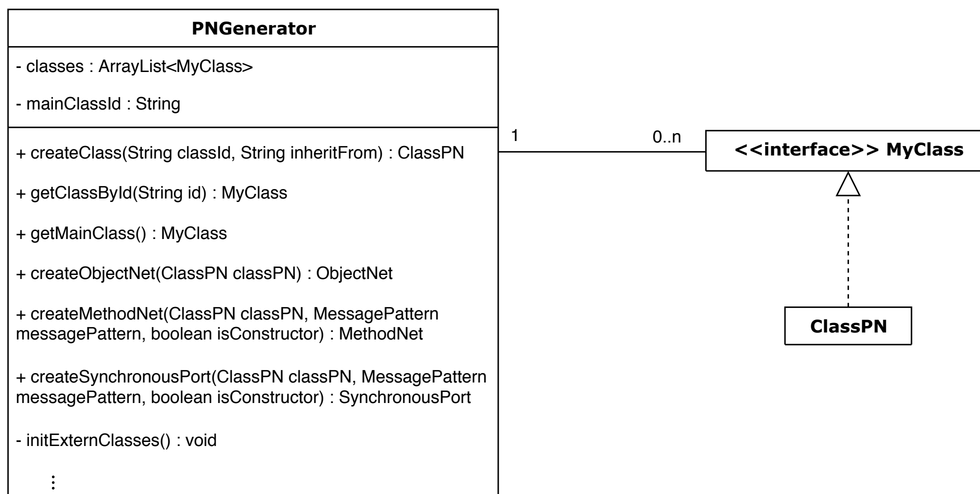
Obrázek 5.5: Třída syntaktické analýzy

Třída zobrazená na obrázku 5.5 implementuje již zmíněný syntaktický průchod. Objekt této třídy v sobě uchovává instanci lexikálního analyzátoru, aktuálně načtený token a jeho typ. Dále parser uchovává frontu tokenů, a to pro případ, kdy je potřeba nahlédnout při syntaktickém průchodu na více než jeden token pro určení správné derivační větve podle gramatiky jazyka PNtalk. V momentě, kdy už je možné s jistotou určit, do jaké větve se má výpočet směřovat, tak se začnou čerpat tokeny z této fronty. Dále je z této třídy možné vyčíst, že ještě obsahuje atributy `CHAR`, `CHAR2`, `SELCHAR` a `SELCHAR2`. Ty obsahují seznamy terminálních symbolů (lexému), spadající pod jeden neterminální symbol, k optimalizaci implementace.

Konstruktor k vytvoření instance třídy `Parser` má jako parametr lexikální analyzátor. Co se týče metod objektu této třídy, tak obsahuje pouze jednu veřejnou metodu a tou je metoda `parse()`, kterou zahájí syntaktický průchod. Ostatní metody jsou privátní a popisují průchod jednotlivými derivačními větvemi z gramatiky jazyka PNtalk (v příloze A). Průchod je prováděn rekurzivním způsobem. První metodou, jenž metoda `parse()` vyvolá, je metoda `parseClasses()`, která provede zpracování výchozího neterminálního symbolu *classes* z gramatiky PNtalk. Neterminální symbol *classes* je možno rozgenerovat na sekvenci symbolů skládající se z neterminálních symbolů *class*, *id* a terminálního symbolu *main*. Metoda `parseClasses()` proto volá metody `parseClass()` a `parseId()` pro zpracování těchto neterminální symbolů. Ostatní metody provádějící syntaktický průchod jsou implementovány analogicky.

5.1.3 Generování OOPN

Tato fáze je finální z procesu překladu a vykonává činnost převodu OOPN do vnitřní reprezentace, která bude popsaná v podkapitole 5.2. Metody pro generování jednotlivých tříd a jejich částí jsou volány během syntaktického průchodu popsaného v podkapitole 5.1.2.

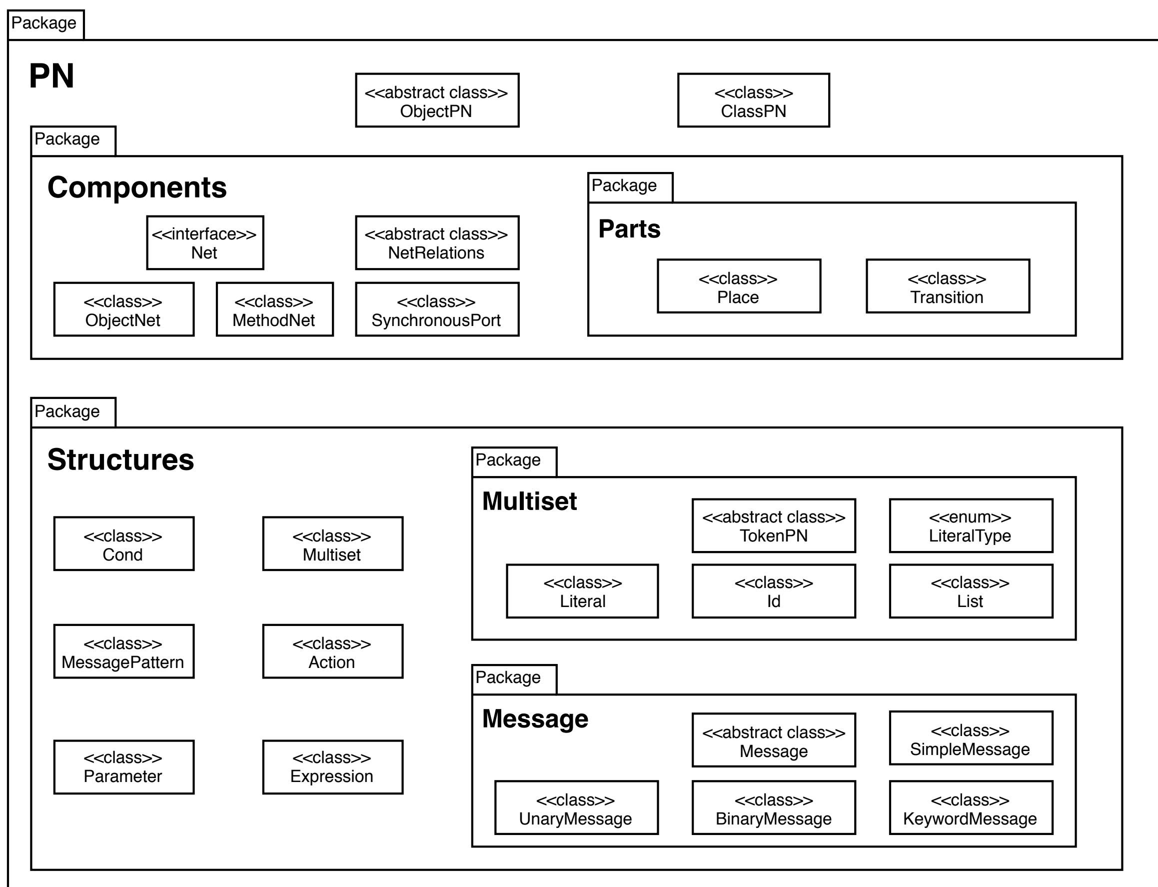


Obrázek 5.6: Třída generující třídy jazyka OOPN

Třída implementující generování OOPN do vnitřní reprezentace je zobrazena na obrázku 5.6. Obsahuje dva atributy, přičemž do prvního z nich ukládá všechny zpracované třídy a druhý atribut v sobě nese identifikátor výchozí třídy. Vygenerované třídy OOPN jsou reprezentovány třídou `ClassPN` implementující rozhraní `MyClass`, které je poskytováno modulem interpretu pro všechny objekty reprezentující třídy. Třída `ClassPN` bude popsána v následující podkapitole 5.2 a rozhraní `MyClass` v podkapitole 6.4.1. Třída generátoru dále obsahuje metody pro samotné generování jednotlivých částí OOPN. V diagramu generátoru na obrázku 5.6 je z prostorových důvodů uvedena pouze část metod. Zároveň je při generování prováděna sémantická analýza, která kontroluje, zda-li je vstup sémanticky správně (např. deklarace proměnných).

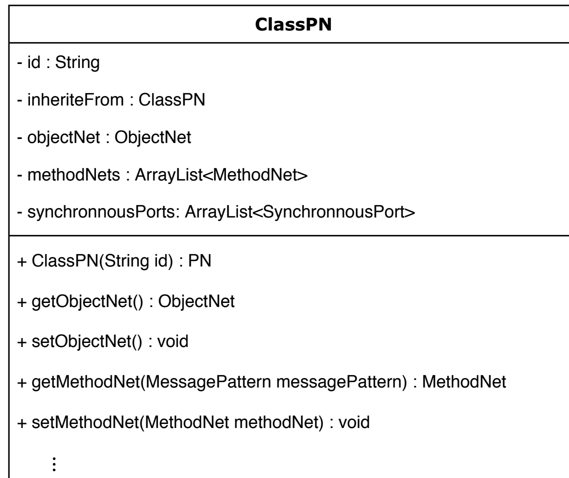
5.2 Vnitřní reprezentace OOPN

Fáze překladu, jenž se stará o generování OOPN, ukládá vygenerované celky do vnitřní reprezentace. Tato podoba dat by měla zajistit snazší a efektivnější zacházení při následném zpracovávání. V tomto případě se jedná o provádění simulace. Popis vnitřní reprezentace je uveden v této podkapitole strukturovaně podle logiky jednotlivých částí a celků. Částmi jsou v tomto případně myšleny implementační třídy, které jsou umístěny do balíčků. Uspořádání tříd implementující vnitřní reprezentaci OOPN je znázorněno v diagramu balíčků na obrázku 5.7.



Obrázek 5.7: Diagram balíčků znázorňující strukturování vnitřní reprezentace OOPN

V hlavním balíčku `PN` se nachází abstraktní třída `ObjectPN`, která je základem všech objektů (tokenů), které jsou součástí interpretace modelu OOPN. Dále balíček `PN` obsahuje třídu `ClassPN`, jejíž objekty reprezentují třídy Petriho sítí definované v jazyku PNtalk. Tato třída Petriho sítí může obsahovat komponenty z balíčku `Components`, který obsahuje třídy `ObjectNet`, `MethodNet` a `SynchronnousPort`. Objekty tříd `ObjectNet`, `MethodNet` mohou obsahovat uzly Petriho sítí v podobě instancí tříd `Place` a `Transition`. Jako dalším celkem nacházejícím se v balíčku `PN` je balíček `Structures`. Ten obsahuje třídy implementující pomocné struktury potřebné pro uložení speciálních atributů jednotlivých komponent. Mezi ty patří třída `Action`, jejíž objekty mohou představovat akci nebo stráž přechodu a jejich výrazy formou objektů třídy `Expression`. Objekt třídy `Expression` může obsahovat zprávy z balíčku `Message`, jejichž základem je abstraktní třída `Message`. Dalšími pomocnými strukturami jsou objekty pro podmínky přechodů `Cond`, vzorů zprávy `MessagePattern` a parametry vzorů zprávy `Parameter`. Poslední třídou v balíčku `Structures` je třída `Multiset`, jejíž objekty reprezentují hranové výrazy. Multimnožina může obsahovat prvky typu `TokenPN` reprezentované abstraktní třídou, ze které vychází tři možné druhy prvků v podobě tříd, a to `Literal`, `Id` a `List`. Prvek typu literál může mít svůj vlastní typ určen výčtovým typem `LiteralType`. V příloze B je uveden diagram vnitřní reprezentace se znázorněnými vazbami mezi jednotlivými třídami.

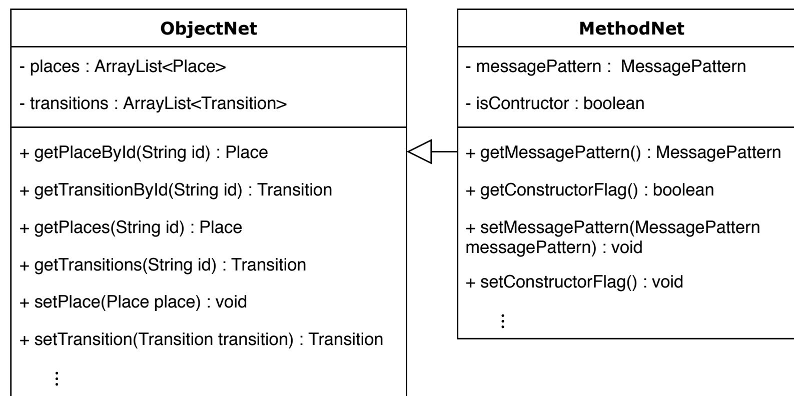


Obrázek 5.8: Třída reprezentující třídu v OOPN

Na obrázku B.2 je diagram popisující třídu reprezentující třídu v OOPN. Obsahuje atributy pro uložení názvu třídy, názvu nadřazené třídy, objektu sítě metody, pole pro uložení objektů sítí metod a synchronních portů. Třída dále implementuje metody pro nastavování jednotlivých atributů, případně pro získání jejich hodnot.

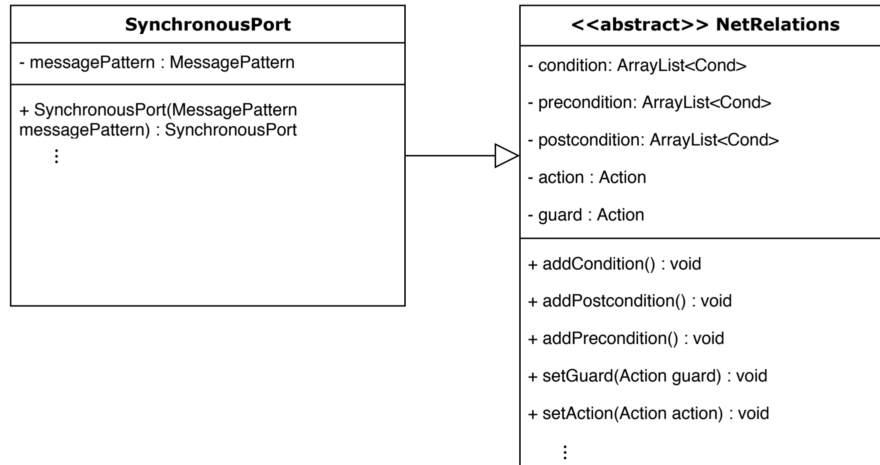
5.2.1 Komponenty

Tento balíček zastřešuje třídy popisující části, ze kterých se skládá třída v OOPN. Těmito částmi jsou sítě objektu, sítě metod a synchronní porty.



Obrázek 5.9: Třídy reprezentující sítě třídy v OOPN

Diagram tříd popisující síť objektu a síť metody je na obrázku 5.9 a ukazuje, že třída sítě metody rozšiřuje třídu sítě objektu o nové atributy a metody. Třída `ObjektNet` obsahuje dva atributy pro uložení míst a přechodů sítě objektu. Dále obsahuje metody pro přidávání jednotlivých částí sítě, případně pro získání hodnot jejich objektů. Třída `MethodNet` obsahuje navíc ještě vzor zprávy, na kterou daná síť metody reaguje a příznak, zda-li se jedná o konstruktor nebo obyčejnou metodu.

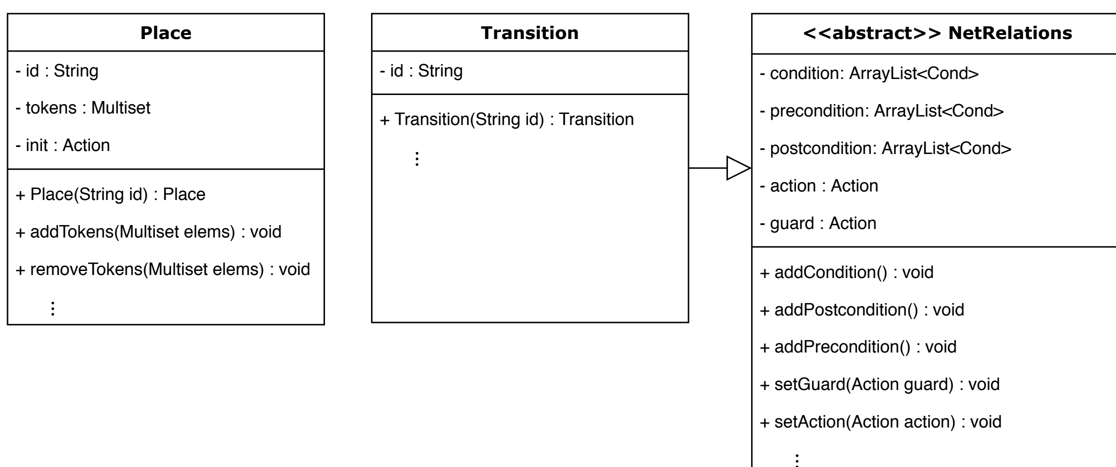


Obrázek 5.10: Třída reprezentující synchronní port třídy v OOPN

Synchorní port, jenž je implementován třídou `SynchronousPort` obsahuje atribut vzor zprávy podobně jako síť metody. Dále obsahuje atributy pro uložení podmínek k přesunu tokenů do míst, případně z míst sítě objektu při jeho provedení, které jsou zděděny z abstraktní třídy `NetRelations` s příslušnými metodami pro správu těchto dat. Abstraktní třída `NetRelations` ještě navíc poskytuje skruktury pro uložení akce a stráže. Třída reprezentující synchronní port je na obrázku 5.10.

Místa a přechody

Místa a přechody jsou základní stavební bloky sítí a jsou reprezentovány třídami z obrázku 5.11. Místa jsou vyjádřena objekty třídy `Place` a uchovávají v sobě identifikační řetězec (název místa), počáteční akci a strukturu multimnožiny reprezentující tokeny umístěné v daném místě sítě. Přechody jsou vyjádřeny objekty třídy `Transition` a podobně jako u místa obsahují název. Další důležité atributy a metody jsou jako u třídy synchronního portu z obrázku 5.10 zděděné z abstraktní třídy `NetRelations`.



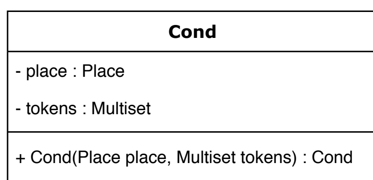
Obrázek 5.11: Třídy reprezentující místo a přechod sítě v OOPN

5.2.2 Struktury

Balíček struktur obsahuje pomocné třídy, jejichž objekty jsou využity v komponentách popsaných výše. Do tohoto balíčku pomocných struktur se řadí třídy popisující podmínky, vzory zpráv, akce přechodu, stráže přechodu a multimožiny.

Podmínky

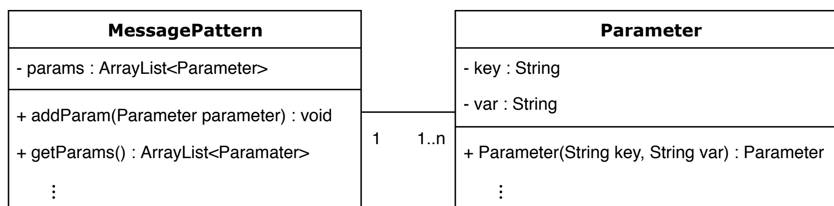
Třída `Cond` obsahuje atribut definující podmínku, která musí být splněna pro vykonání přechodu nebo případně synchronního portu. Druhým a zároveň posledním atributem je specifikace k jakému místu se podmínka vztahuje. V grafické reprezentaci by to určovalo místo, s nímž je přechod nebo synchronní port propojen hranou. Třída je na obrázku 5.12.



Obrázek 5.12: Třída popisující přechod a hranový výraz

Vzor zprávy

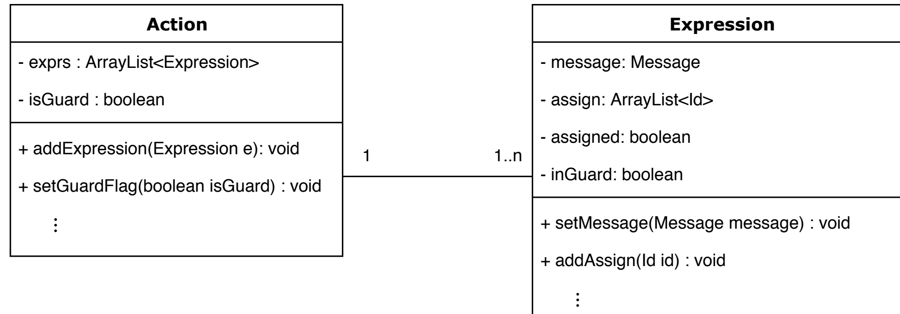
Vzor zprávy je využit pro identifikaci sítě metody, případně synchronního portu. Skládá se z řetězce parametrů, kde parametr obsahuje hodnotu selektoru a název vstupního místa sítě metody. Třídy `MessagePattern` a `Parameter` popisující vzor zprávy jsou uvedeny na obrázku 5.13.



Obrázek 5.13: Třídy popisující vzor zprávy

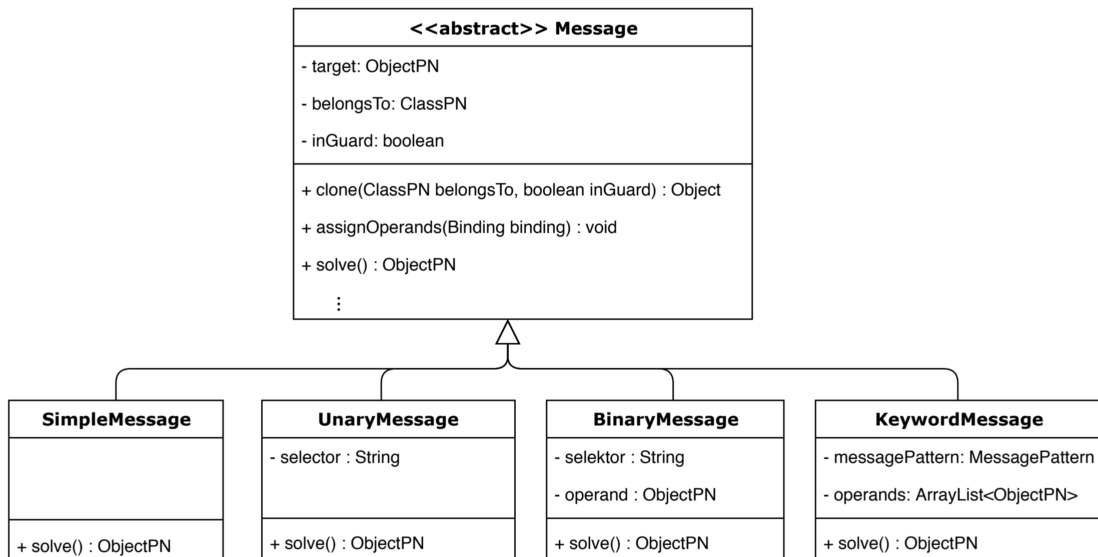
Akce a stráže

Třída `Action` vyjadřuje popis akce přechodu nebo synchronního portu, případně stráže přechodu. Akce se může skládat z výrazů a obsahovat výraz přiřazení. Stráž přechodu nemůže obsahovat výraz přiřazení, ale i přesto zde existuje velká podobnost, a proto třída `Action` obsahuje atribut k identifikaci, zda-li se jedná o akci nebo stráž. Výraz je reprezentován třídou `Experssion` a uchovává informaci o zprávě, kterou výraz nese a proměnné, do nichž má být přiřazen výsledek zaslání zprávy. Třídy `Action` a `Expression` popisující akci a stráž jsou uvedeny na obrázku 5.14.



Obrázek 5.14: Třídy popisující akci a stráž

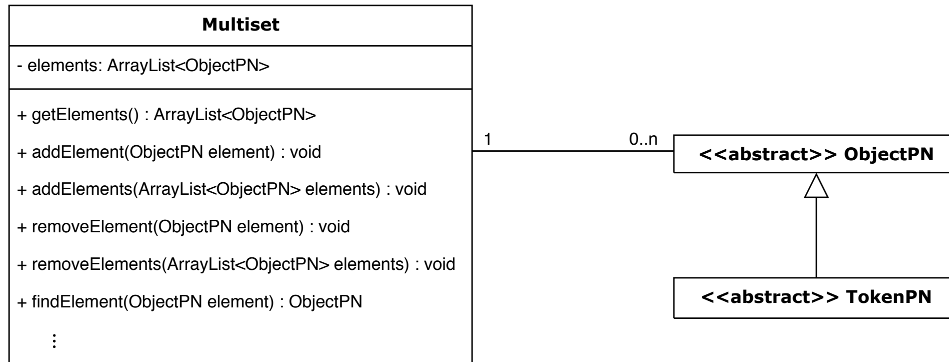
Zpráva může být různých typů, a to jednoduchá zpráva, unární zpráva, binární zpráva a nebo zpráva s klíčovými slovy. Tomu je přizpůsobena i implementace. Je definována abstraktní třída `Message`, kterou rozšiřují třídy `SimpleMessage`, `UnaryMessage`, `BinaryMessage` a `KeywordMessage`, reprezentující jednotlivé druhy zpráv. Třídy popisující jednotlivé zprávy jsou uvedeny na obrázku 5.15.



Obrázek 5.15: Diagram hierarchie tříd možných zpráv

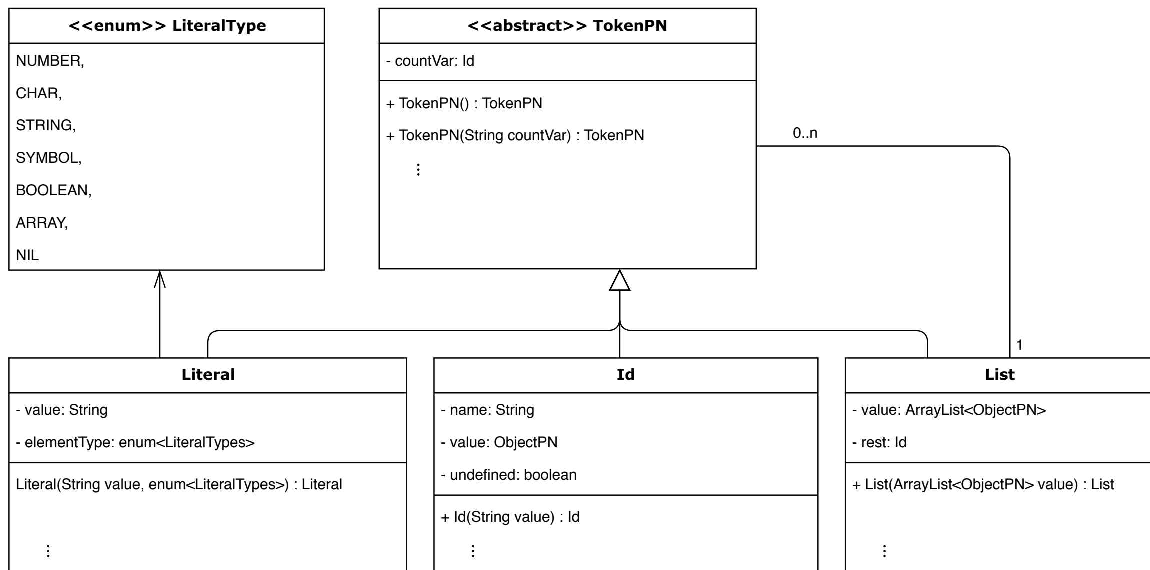
Multimnožiny

`Multiset` je třída, která reprezentuje sktukturu pro ukládání tokenů do místa v síti, případně k definování podmínky v hranovém výrazu. Obsahuje tedy atribut k uložení multimnožiny tokenů ve formě objektů abstraktní třídy `ObjectPN`, která tvoří vrchol v hierarchii tokenů a musí z ní vycházet všechny podporované typy tokenů. Třída `Multiset` popisující multimnožinu s patřičnou vazbou na třídu `ObjectPN` je uvedena na obrázku 5.16.



Obrázek 5.16: Třídy reprezentující multimnožinu

Abstraktní třída `TokenPN` reprezentuje základní typ tokenu, jenž může být definován přímo ve zdrojovém kódu jazyka PNtalk. Tuto abstraktní třídu rozšiřují tři třídy, které reprezentují druhy tokenů, jenž se mohou v síti vyskytovat. Mezi tyto druhy patří literály, identifikátory (proměnné) a složený druh nazývající se list, který může obsahovat rekurzivně tokeny všech tří druhů. Skupina tříd popisující základní tokeny jazyka PNtalk je uvedena na obrázku 5.17. Třída `TokenPN` ještě navíc poskytuje možnost definovat obecný atribut ve tvaru proměnné vyjadřující počet výskytů v multimnožině.



Obrázek 5.17: Třídy reprezentující prvky multimnožiny

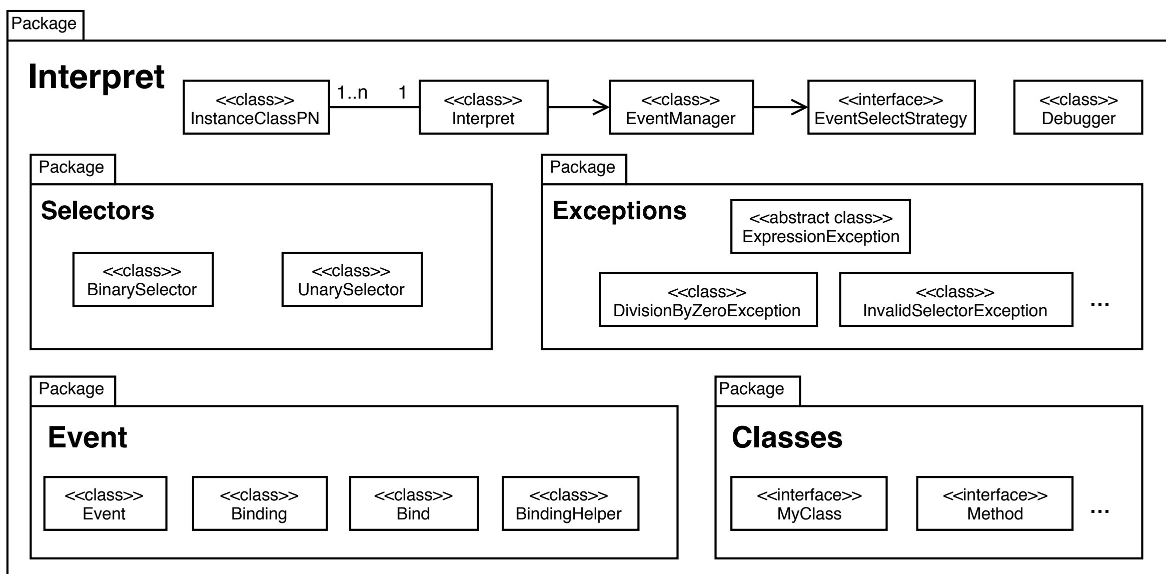
Kapitola 6

Interpret

V této kapitole bude popsán návrh a implementace interpretu, který slouží k provádění simulace běhu modelů OOPN. Kapitola je rozdělena na čtyři části. V první části je popsána samotná struktura interpretu. Druhá část kapitoly je věnována dekompozici běhu interpretu a popisu jednotlivých segmentů běhu. Ve třetí části této kapitoly bude vysvětlen princip vyhodnocování výrazů a systém zacházení s jednotlivými typy výrazů. Poslední část této kapitoly bude věnována podporovaným Java objektům, se kterými umí interpret pracovat. Jak již bylo zmíněno v úvodu kapitoly 5, tak se v práci vyskytují segmenty třídního diagramu popisující implementaci jednotlivých částí aplikace a z důvodu prostorových nároků v nich budou uvedeny pouze složky potřebné k výkladu.

6.1 Struktura interpretu

Interpret je navržen jako samostatný modul, jehož účelem je provádění simulace modelů OOPN. Třídy implementující interpret a jejich organizace je znázorněna za pomoci diagramu balíčků na obrázku 6.1.

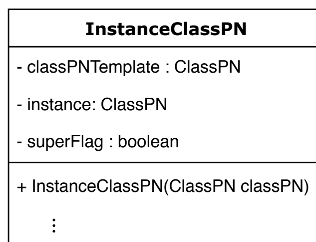


Obrázek 6.1: Diagram balíčků znázorňující strukturování tříd interpretu

Jádro intepretu je tvořeno třídami `Interpret` a `EventManager`. Objekt třídy `Interpret` je určen k řízení samotného procesu interpretace a obsahuje vazbu na množinu instancí tříd OOPN, jako objekty třídy `InstanceClassPN`, které jsou právě zpracovávány. Využívá při tom instanci třídy `EventManager` k získávání informací o proveditelnosti událostí v simulovaném systému. Ten poskytuje možnost zvolení strategie výběru událostí, které má interpret prioritně vykonávat, čímž je možné měnit vlastnosti chování interpretu pro různé modely. K tomuto rozšíření funkcionality slouží rozhraní `EventSelectStrategy` pro definici různých strategií volby událostí. Jako doplněková třída je ke třídě `Interpret` v balíčku obsažena ještě třída `Debugger`, která poskytuje nástroje pro ladění. Dále modul interpretu obsahuje balíček `Classes`, který poskytuje rozhraní pro definici tříd a vestavěných metod. V neposlední řadě modul interpretu obsahuje ještě balíčky `Selectors`, `Event` a `Exceptions`, jejichž podrobnější popis a účel bude vysvětlen v průběhu této kapitoly. V příloze B je uveden diagram modulu interpretu se znázorněnými vazbami mezi jednotlivými třídami.

6.1.1 Objekty tříd OOPN

Objekty, jakožto instance tříd nadefinovaných v jazyce PNtalk jsou reprezentovány objekty třídy `InstanceClassPN`, která je zobrazena na obrázku 6.2.



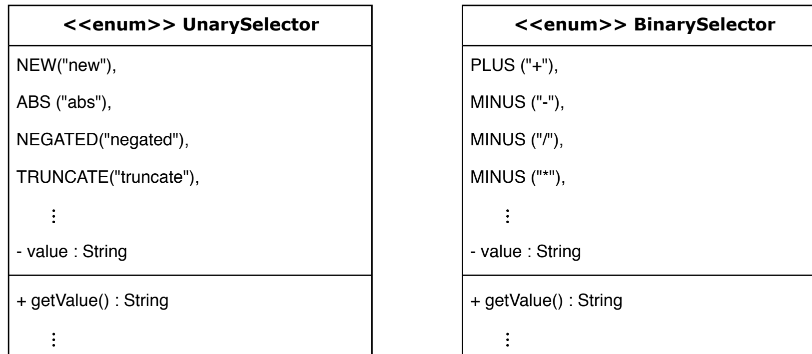
Obrázek 6.2: Třída reprezentující instanci třídy OOPN

Tato třída rozšiřuje třídu `ObjectPN` a poskytuje dva primární atributy, mezi které patří šablona třídy OOPN ve formě objektu třídy `ClassPN` sestavenou při překladu a objekt reprezentující samotnou instanci, taktéž ve formě objektu `ClassPN`. Během interpretace je primárním nástrojem pro sestavování jednotlivých instancí návrhový vzor „Prototype“, který slouží pro vytváření kopií objektů (klonování), a tedy je použit na třídu `ClassPN` a všechny její části.

Nový objekt třídy OOPN je tedy vytvářen pomocí naklonování šablony zvolené třídy OOPN, kterou má interpret k dispozici. Během tohoto procesu je vytvořen objekt třídy OOPN pouze s objektovou sítí, což je vlastně výchozí stav tohoto objektu. V průběhu interpretace do něj mohou být vkládány instance sítí metod, jenž byly nad daným objektem zavolány (invokovány).

6.1.2 Selektory

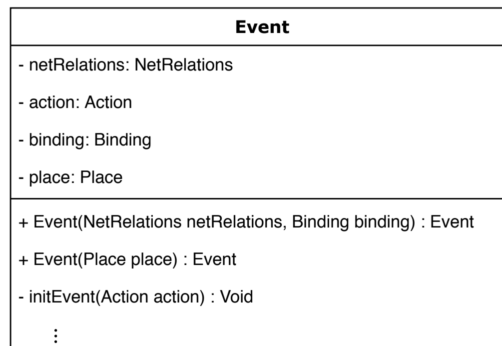
Balíček `Selectors` v sobě uchovává deklarace dvou typů selektorů v podobně enumerací, a to selektory unární a binární. Tyto enumerace obsahují veškeré podporované unární a binární selektory interpretu. Jednotlivé selektory mají definovaný klíč a k sobě přiřazenou hodnotu v řetězci, která odpovídá danému selektoru. Na obrázku 6.3 je vyobrazena zjednodušená podoba enumerací, již zmíněných selektorů.



Obrázek 6.3: Enumerace obsahující podporované unární a binární selektory

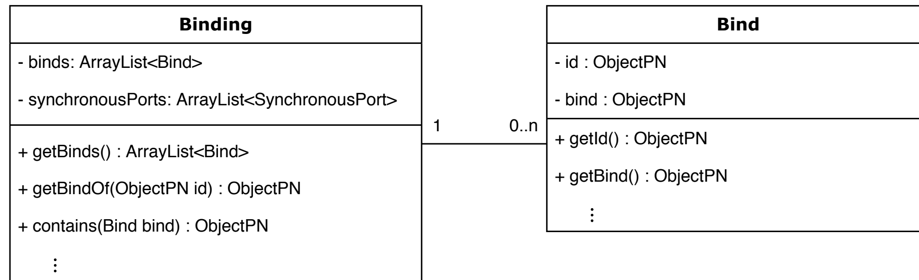
6.1.3 Události

Události, jsou z pohledu interpretu vnímány jako možné akce, které může interpret vykonat v čase jeho běhu. Události vznikají dynamicky na základě proveditelnosti jednotlivých přechodů v jednotlivých stavech interpretu. Identifikace a následné generování událostí probíhá ve fázi analýzy proveditelnosti přechodů, která bude popsána v podkapitole 6.2.2. Zpracovávané události mají podobu objektů třídy `Event`, jenž je zobrazena na obrázku 6.4.



Obrázek 6.4: Třída reprezentující událost

Událost obsahuje všechny potřebné údaje k jejímu vykonání. Prvním z těchto údajů je síťová relace, která je zvolena k provedení. Síťovou relací jsou myšleny objekty vycházející implementačně z abstraktní třídy `NetRelations`, mezi které patří například objekty vyjadřující přechody, případně synchronní porty, což bylo již dříve popsáno v podkapitole 5.2.1, která hovoří o komponentách vnitřní reprezentace OOPN. Speciálním případem události je vykonání počáteční akce místa při vytvoření nové instance sítě. V tomto případě není v události uložena síťová relace, ale je uloženo místo, jehož počáteční akce má být vykonána. Dalším údajem, jenž událost uchovává je kopie akce, která bude zpracovávána v průběhu provádění události. A jako posledním podstatným údajem k provedení události je navázání tokenů, které se vyskytují na vstupních, testovacích a následně i výstupních hranách. Na obrázku 6.5 jsou uvedeny třídy `Binding` a `Bind` popisující navázání tokenů.



Obrázek 6.5: Třídy popisující navázání tokenů

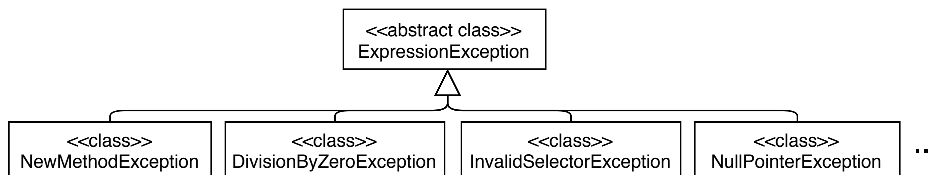
Třída `Bind` vyjadřuje navázání jednoho tokenu. To je možné vytvořit z proměnné na jakýkoliv token (např. $a \to$ '*str*'), anebo přímo tokenu na token se stejnou hodnotou (např. '*str*' $\to$ '*str*'). Jednotlivé navázání ve formě objektů třídy `Bind` jsou vloženy do kolekce objektu třídy `Binding` vyjadřující veškerá aktuální navázání pro danou událost. Objekt třídy `Binding` v sobě ještě nese navíc informaci o synchronních portech, které přísluší k danému navázání.

6.1.4 Výjimky

V průběhu vyhodnocování výrazů mohou nastat různé situace vyžadující přerušení běhu interpretu anebo vykonání příslušné akce odpovídající dané situaci. To zahrnuje například následující případy rozdělené podle typu:

- Vyžadující vykonání příslušné akce interpretem
  - Vytvoření nového objektu OOPN
  - Vytvoření nové instance sítě metody
- Vyžadující přerušení běhu interpretu
  - Dělení nulou
  - Neplatný selektor
  - Přistup k `null` objektu
  - Neplatná hodnota pro daný selektor
  - Neznámá operace

Tyto situace jsou zde řešeny pomocí systému výjimek. Každé z výše uvedených situací odpovídá jedna specifická výjimka, která v sobě nese zprávu s popisem vyskytnuté situace. Případně může výjimka obsahovat i informace potřebné pro vykonání příslušné akce interpretem. Například při vyvolání výjimky o vytvoření nové sítě metody jsou k výjimce připojeny ještě informace obsahující odkaz na nově vytvořenou instanci metody, výraz k uložení výsledku metody a objekt, jenž byl cílem volání metody. Je zavedena taktéž jednoduchá hierarchie výjimek, která je uvedena na obrázku 6.6.

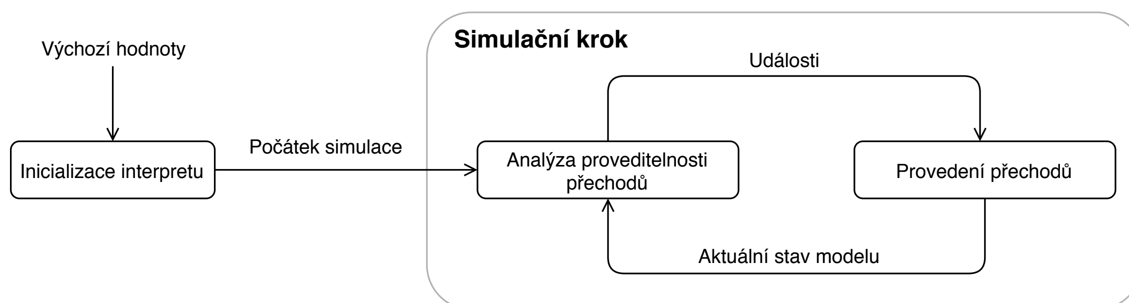


Obrázek 6.6: Hierarchie výjimek

Na vrcholů této hierarchie výjimek je abstraktní třída `ExpressionException` popisující obecnou výjimku vyvolanou při vyhodnocování výrazu. Vycházejí z ní všechny výjimky, které je možno vyvolat při vyhodnocování výrazu.

6.2 Běh intepretu

Na počátku, před samotnou interpretací modelu OOPN je nutné provést inicializaci interpretu, při které se nastaví všechny potřebné výchozí hodnoty. Následuje fáze samotné interpretace (provádění modelu), a to vykonáváním tzv. interpretačních/simulačních kroků. Každý krok se skládá ze dvou logických částí, jimiž jsou analýza proveditelnosti přechodů a provedení přechodů. Schéma běhu interpretu je možné vidět na obrázku 6.7.



Obrázek 6.7: Běh interpretu

Výstup analýzy proveditelnosti přechodů je množina událostí, které jsou určeny pro vykonání interpretem. Interpret v jednom simulačním kroku může provést více přechodů v případě, že je zajištěna jejich nekonfliktnost. Pokud je výstupní množina událostí prázdná, tak simulace končí.

6.2.1 Inicializace interpretu

Před samotnou interpretací je nutno interpretu dodat všechny potřebné výchozí hodnoty, ve formě parametrů, aby byl schopen vykonávat již zmíněný proces interpretace. Řídící jednotkou interpretu je objekt reprezentován třídou `Interpret` uvedenou na obrázku 6.8.

Interpret

- instancesClassPN: ArrayList<InstanceClassPN>
- PNGenerator: PNGenerator
- eventManager: EventManager
- steps : Intenger
- actualStep: Integer
- log: BufferedWriter

+ Interpret(PNGenerator png, EventSelectStrategy strategy)
+ run() : boolean
+ makeStep() : boolean
- execEvent(Event e) : void
- garbageCollection() : void
⋮

Obrázek 6.8: Třída představující řídící třídu interpretu

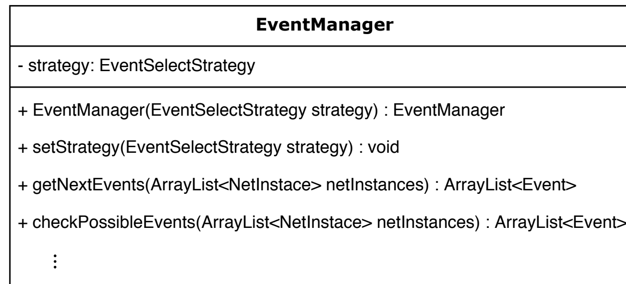
Povinným parametrem interpretu je instance třídy `PNGenerator`, kterou je možné získat po úspěšném překladu z instance třídy `Parser`. Ta obsahuje popis modelu ve formě (viz. podkapitola 5.2), která je vhodná k nadcházející interpretaci. Dalším povinným parametrem interpretu je instance třídy implementující rozhraní `EventSelectStrategy` k určení strategie volby událostí. Volitelným parametrem je pak možnost určení místa v podobě objektu `BuffferedWriter` k ukládání logovacích informací o průběhu simulace. Jako dalším volitelným parametrem a zároveň posledním je možnost určení maximálního počtu simulačních kroků. V případě zadání tohoto parametru může tedy interpret úspěšně skončit svůj běh dvěma možnými způsoby.

- Neexistuje žádný proveditelný přechod
- Dosaženo maximálního počtu simulačních kroků

Po zádání všech povinných parametrů se interpret uvede do stavu simulačního kroku 0, což se provede následujícím způsobem. Nejprve dojde k vytvoření výchozí instance třídy OOPN modelu, která vznikne za pomoci tzv. klonování objektu konkrétně ve formě hluboké kopie a vložení do kolekce aktivních instancí OOPN interpretu. Následuje vytvoření správce událostí reprezentovaného objektem třídy `EventManager` s patřičným parametrem strategie volby událostí. V tomto okamžiku je interpret připraven k samotné fázi simulace.

6.2.2 Analýza proveditelnosti přechodů

V každém simulačním kroku je provedena analýza proveditelnosti přechodů za pomoci správce událostí ve formě objektu třídy `EventManager`. Jeho úkolem je analyzovat přechody ve všech instancích sítí modelu, a vyhodnotit, které z nich jsou proveditelné. Výsledky poskytnuté správcem událostí, jsou ve formě kolekce událostí ve tvaru uvedeném v podkapitole 6.1.3. Třída reprezentující správce událostí je uvedena na obrázku 6.9



Obrázek 6.9: Třída reprezentující správce událostí

Správce událostí poskytuje dvě veřejné metody pro získání událostí. První z nich je metoda `checkPossibleEvents(...)`, která implementuje samotnou analýzu proveditelnosti přechodů a vrátí všechny dostupné události, které je v daném stavu modelu možné vykonat. Druhá metoda vrací taktéž množinu událostí, ale s tím rozdílem, že tyto události jsou již vyselektovány podle zvolené strategie volby událostí. Výsledkem je tedy množina nekonfliktních událostí, kterou je možné vykonat souběžně v jednom simulačním kroku interpretu.

Analýza proveditelnosti přechodů pro získání množiny dostupných událostí probíhá na množině instancí tříd OOPN, které vyjadřují aktuální stav modelu. Tyto instance tříd OOPN se dekomponují na sítě, z nichž se skládají (síť objektu, sítě metod). Následně se pro každý přechod v každé síti provede analýza proveditelnosti přechodu, jejíž výsledek je množina dostupných událostí pro daný přechod. Sjednocením událostí z analýz pro jednotlivé přechody získáme množinu všech dostupných událostí.

Analýza proveditelnosti přechodu

Jedná se o operaci, která se vykonává nezávisle pro každý přechod. Výsledkem analýzy je množina událostí, kde každá událost odpovídá určitému navázání tokenů ze vstupních podmínek přechodu. Analýza se skládá ze série činností, které probíhají v uvedeném pořadí.

1. Vytvoření množiny vstupních podmínek přechodu
2. Vygenerování všech možných navázání
3. Selekce validních navázání
4. Vyhodnocení stráže přechodu
5. Vygenerování událostí

Při vytváření vstupní množiny podmínek přechodu se sjednotí podmínky ze všech vstupních hran a testovacích hran. Následuje bod, kdy pro tuto množinu vstupních podmínek je nutné vyhledat všechny možné kombinace navázání z daných míst, s nimiž jsou propojeny. K tomu je využita pomocná třída `BindingHelper`, kterou je možné vidět na obrázku 6.10

BindingHelper
- vars : ArrayList<ObjectPN> - values: ArrayList<ObjectPN> - bindings: ArrayList<ArrayList<Bind>>
+ BindingHelper(ArrayList<ObjectPN> vars, ArrayList<ObjectPN> values) : BindingHelper + getAllBindings() : ArrayList<ArrayList<Bind>> ⋮

Obrázek 6.10: Pomocná třída BindingHelper

Při vytváření objektu třídy `BindingHelper` je vyžadováno vložit dvě multimnožiny objektů. První množina objektů reprezentovaná proměnnou `vars` označuje objekty, na které bude navazováno a druhá množina objektů `values` vyjadřuje ty objekty, které budou navazovány. Výslednou kolekci všech možných kombinací navázání je pak možné získat jedinou veřejnou metodou `getAllBindings()`, kterou třída `BindingHelper` poskytuje.

Získaná kolekce může obsahovat i kombinace navázání (Binding), které obsahují i nevalidní navázání (Bind), jelikož byly vygenerovány všechny možné kombinace. Takovým navázáním je například ‘*abc*‘ → ‘*str*‘, protože se nejedná o stejné objekty. Je tedy nutné z výsledné kolekce ještě vyselektovat všechny validní kombinace navázání (tj. obsahují pouze validní navázání). Při analyzování jednoho přechodu je nutné takto zpracovat podmínky ze všech vstupních míst a následně všechny získané validní kombinace navázání propojit. Proces propojení je možné si představit jako databázovou operaci „join“ adaptivně přes všechny prvky.

Dalším krokem analýzy proveditelnosti přechodu je vyhodnocení stráže přechodu. To proběhne jako atomická operace, která se skládá ze sekvence vyhodnocení všech výrazů ve stráži. Aby přechod byl proveditelný, tak musí všechny výrazy ve stráži přechodu odpovídat boolovské hodnotě „true“. Mezi výrazy ve stráži přechodu mohou být i volání synchronních portů, jejichž vyhodnocení probíhá podobně jako analýza proveditelnosti přechodu, přičemž pokud je synchronní port proveditelný, tak výsledek jeho vyhodnocení je roven boolovské hodnotě „true“. Důležité je také zabezpečit situaci, kdy ze stráže přechodu je voláno vícekrát stejný synchronní port, případně jiný synchronní port ovlivňující stejná místa. Z tohoto důvodu je nutné odebrat odpovídající tokeny ze vstupních míst podle vstupních podmínek daného synchronního portu, aby nedošlo k duplicitním navázáním již navázaných tokenů. Odebrané tokeny se po vykonání celé stráže přechodu vrátí zpět na původní místa, aby stav sítě zůstal po analýze beze změny. Další situací, která ještě může nastat u vyhodnocování synchronních portů, je možnost výskytu stráže v synchronním portu. V této situaci se postupuje stejně jako u vyhodnocování stráže přechodu s tím rozdílem, že všechny volané synchronní porty se ukládají do objektu navázání tokenů patřící synchronnímu portu, z jehož stráže byla tato volání uskutečněna. Tento proces vyhodnocení stráže přechodu proběhne pro všechny prvky z množiny validních kombinací navázání z předchozího kroku. Všechny validní kombinace navázání, jejichž vyhodnocení stráže splňuje podmínku proveditelnosti přechodu jsou vybrány k dalšímu zpracování.

Poslední fází je vygenerování událostí ve formě objektů třídy `Event` (viz. podkapitola 6.1.3) pro každou validní kombinaci navázání, která splňuje podmínku proveditelnosti na základě stráže přechodu. Mezi parametry pro vytvoření události patří tedy validní kombinace navázání a samotný přechod. Výsledkem analýzy proveditelnosti přechodu je množina dostupných událostí daného přechodu pro aktuální stav, ve kterém se model právě nachází.

6.2.3 Provedení přechodu

V situaci, kdy je z analýzy proveditelnosti přechodů získána alespoň jedna dostupná událost, tak je možné přejít k druhé logické části simulačního kroku, kterou je provedení přechodů. To, jaké přechody mají být provedeny, je určeno na základě zvolené strategie volby událostí ve správci událostí. Provedení jednoho přechodu se skládá z následující série činností, které probíhají v uvedeném pořadí.

1. Odebrání tokenů ze vstupních míst
2. Provedení synchronních portů
3. Vyhodnocení akce přechodu
4. Vložení tokenů do výstupních míst

V prvním kroku jsou odebrány všechny tokeny ze vstupních míst na základě vstupních podmínek podle odpovídajícího navázání tokenů. Všechny potřebné informace ohledně navázání tokenů a vstupních/výstupních podmínek přechodu pro tento krok jsou uloženy v události tj. objektu třídy `Event` (viz. podkapitola 6.1.3).

Následuje fáze provedení synchronních portů, které se vyskytovaly ve stráži přechodu. V případě, že by i ve stráži volaného synchronního portu bylo další volání synchronního portu, tak se rekurzivně vykonají všechny tyto synchronní porty. Postup provedení synchronního portu má stejný průběh, jako je tomu u přechodu, který je popisován právě v této podkapitole.

Vyhodnocení akce přechodu je krok zahrnující vyhodnocení všech výrazů, ze kterých je akce složena. Posloupnost vyhodnocování výrazů probíhá postupně zleva doprava, přičemž se mezivýsledky jednotlivých výrazů mohou propagovat skrze celou akci. Jako rozšíření oproti původní definici jazyka PNtalk, je přidána podpora možnosti definování akce i v synchronním portu.

Poslední krok při provádění přechodu je vložení tokenů z výstupních podmínek přechodu do příslušných míst. Před samotným úkonem vložení tokenů se ještě aktualizuje navázání proměnných podle výsledků získaných z akce přechodu, aby se projevilo vyhodnocení akce. Tímto úkonem je provedení přechodu u konce a je možné provést další událost, pokud je dostupná, případně započít nový simulační krok analýzou proveditelnosti přechodů.

6.3 Vyhodnocování výrazů

Tato podkapitola bude věnována popisu uložení struktury objektů reprezentující výraz a následně postup jeho vyhodnocování. Výraz jako takový je reprezentován objektem třídy `Expression`, která byla popsána v podkapitole 5.2.2 věnující se popisu akce a stráže. Tento objekt v sobě nese zprávu, která je hlavním předmětem vyhodnocování a kolekci proměnných, do kterých bude uložen výsledek operace zaslání zprávy. Zpráva v jazyku PNtalk, jenž je uložena ve výrazu může nabývat hodnoty jednoho ze čtyř typů zpráv, přičemž výraz odpovídá typu podle toho, jaký typ zprávy v sobě nese.

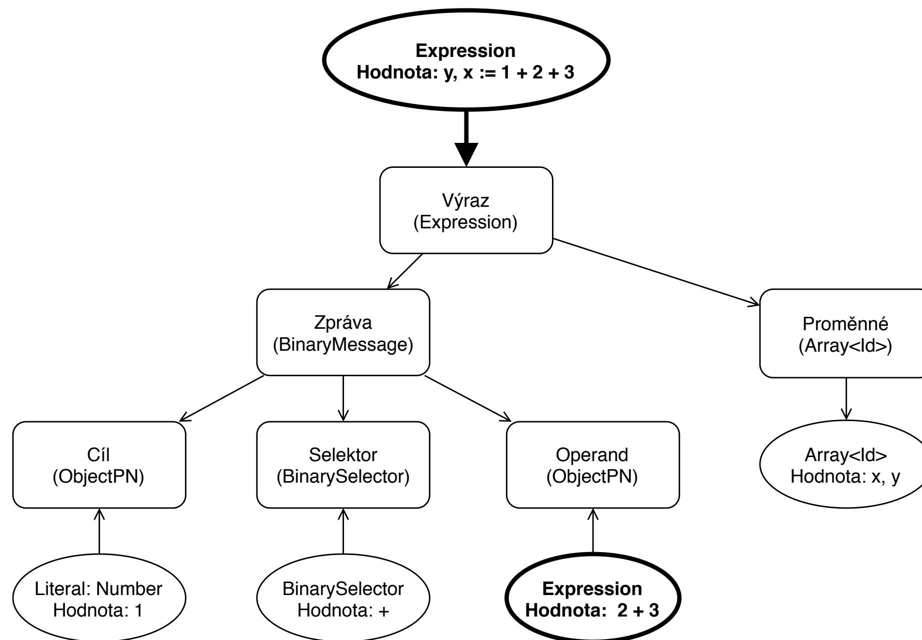
Typy zpráv/výrazů jsou následující:

- Jednoduchá zpráva (`SimpleMessage`)
- Unární zpráva (`UnaryMessage`)
- Binární zpráva (`BinaryMessage`)
- Zpráva z klíčových slov (`KeywordMessage`)

Zprávy se liší tím, že každý typ zprávy může obsahovat pouze specifický počet operandů až na zprávu s klíčovými slovy, která může obsahovat libovolný počet operandů. Každá zpráva, kromě jednoduché zprávy, obsahuje tzv. cíl zprávy (adresáta) a selektor zprávy. Jednoduchá zpráva obsahuje pouze cíl zprávy. Tento typ zprávy se využívá především k určení priority vyhodnocování při použití závorek, případně pro uložení výsledku do výrazu po vykonání operace spadající do skupiny vyjimek, které jsou popsány v podkapitole 6.1.4.

6.3.1 Struktura výrazu

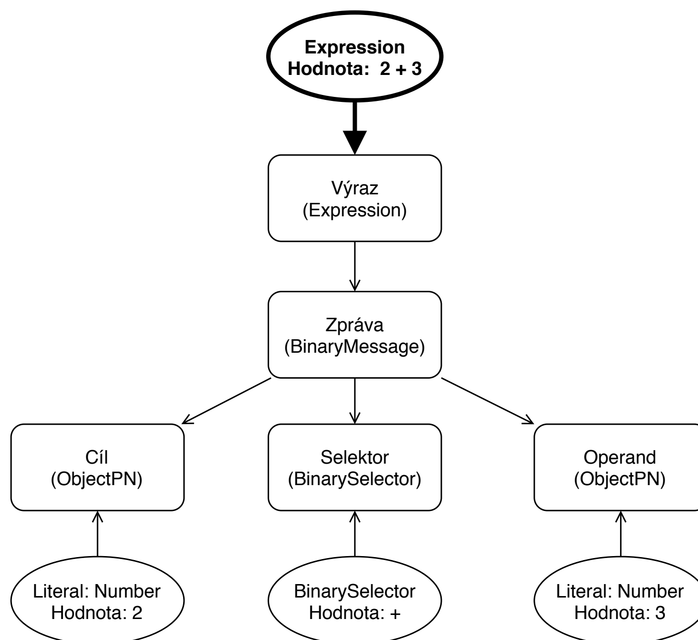
Všechny výrazy, které lze zapsat za pomoci jazyka PNtalk je možné převést do stromové struktury, díky které lze vhodně ukládat i složené výrazy obsahující více zaslání zpráv. Pro lepší znázornění struktury uložení objektů reprezentující výraz je na obrázku 6.11 uveden jednoduchý příklad složeného výrazu.



Obrázek 6.11: Příklad rozkladu výrazu $y, x := 1 + 2 + 3$

Uvedený příklad výrazu odpovídá součtu tří čísel s operátorem přiřazení výsledku do proměnných x a y. Příklad byl zvolen tak, aby na něm šlo jednoduše předvést princip vytváření stromové struktury výrazu, která pak dále slouží k určení pořadí vyhodnocování jednotlivých uzlů v případě složeného výrazu. Kořenem tohoto stromu je vždy objekt třídy `Expression`, který v sobě nese zprávu. Konkrétně výraz uvedený na obrázku 6.11 obsahuje

binární zprávu a kolekci o dvou proměnných x a y, do kterých bude přiřazen výsledek tohoto zaslání zprávy. Binární zpráva výrazu obsahuje cíl zprávy, kterým je číselný literální symbol obsahující hodnotu 1. Dále zpráva obsahuje selektor zprávy, kterým je binární selektor $+$. A jako posledním údajem binární zprávy je její operand, který nabývá hodnoty výrazu, jehož rozložení je uvedeno na obrázku 6.12. Jedná se tedy o složený výraz, který obsahuje dvě binární zprávy.

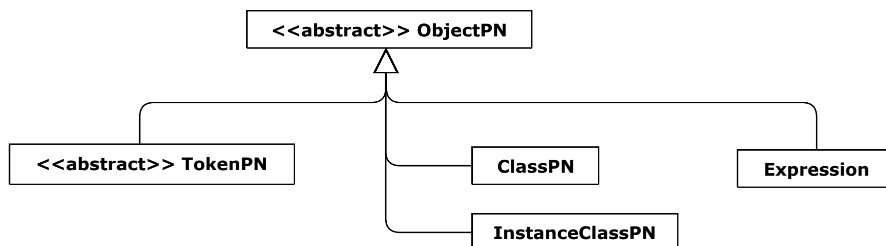


Obrázek 6.12: Rozklad výrazu $2 + 3$

Výraz na obrázku 6.12 obsahuje binární zprávu ve stejné formě, jako tomu bylo u výrazu na obrázku 6.11, ale neobsahuje žádné proměnné, do kterých by byl výsledek po vyhodnocení výrazu přiřazen. Liší se také obsah výrazu, a to tak, že obsahuje zprávu $2 + 3$, která se dá rozložit na cíl zprávy 2, selektor $+$ a operand zprávy 3. V tomto momentě je rozklad výrazu z obrázku 6.12 kompletní, jelikož v žádném z operandů se nevyskytují žádný další výraz. Ostatní typy výrazů mají podobnou strukturu jako je tomu u rozkladu z obrázků 6.11 a 6.12. Liší se pouze počet operandů a u zprávy s klíčovými slovy se místo selektoru zprávy použije vzor zprávy.

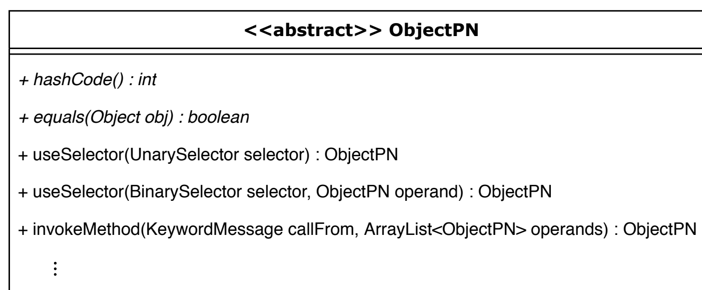
6.3.2 Operandy výrazu

Operandy výrazu mohou nabývat hodnot všech typů objektů (tokenů), které se mohou v sítích modelu OOPN vyskytovat. Nicméně všechny tyto typy objektů mají základ v abstraktní třídě `ObjectPN`, která poskytuje obecné metody, které by měly podporovat všechny objekty v síti.



Obrázek 6.13: Hierarchie tříd objektů (token) sítě

Na obrázku 6.13 je možné vidět hierarchii dědičnosti tříd, která zobrazuje všechny třídy, které dědí od abtraktní třídy `ObjectPN`. Mezi tyto třídy patří všechny primitivní typy objektů pokryté třídou `TokenPN`. Dále také uživatelské třídy OOPN a jejich instance vyjádřené třídami `ClassPN` a `InstanceClassPN`. Jako poslední uvedenou třídou, který vychází ze třídy `ObjectPN` je třída `Expression`, která sice nepředstavuje typ tokenu, ale tímto rozšířením se nabízí možnost vkládat výrazy přímo na pozici operandů zpráv, a tím docílit možnosti snadno ukládat složené výrazy. Abstraktní třída `ObjectPN` je uvedena na obrázku 6.14.



Obrázek 6.14: Abstraktní třída `ObjectPN`

Abstraktní třída `ObjectPN` poskytuje rozhraní v podobě abstraktních metod `hashCode()` a `equals(Object obj)`, které musí všechny objekty sítě implementovat, aby byla zajištěna integrita a podpora všech základních funkcionalit objektů. Dále abstraktní třída `ObjectPN` poskytuje sadu veřejných metod sloužící k přijetí zaslané zprávy adresovaným objektem. Sada obsahuje tři metody, tj. pro každý typ zprávy jedna metoda. Mezi tyto metody patří:

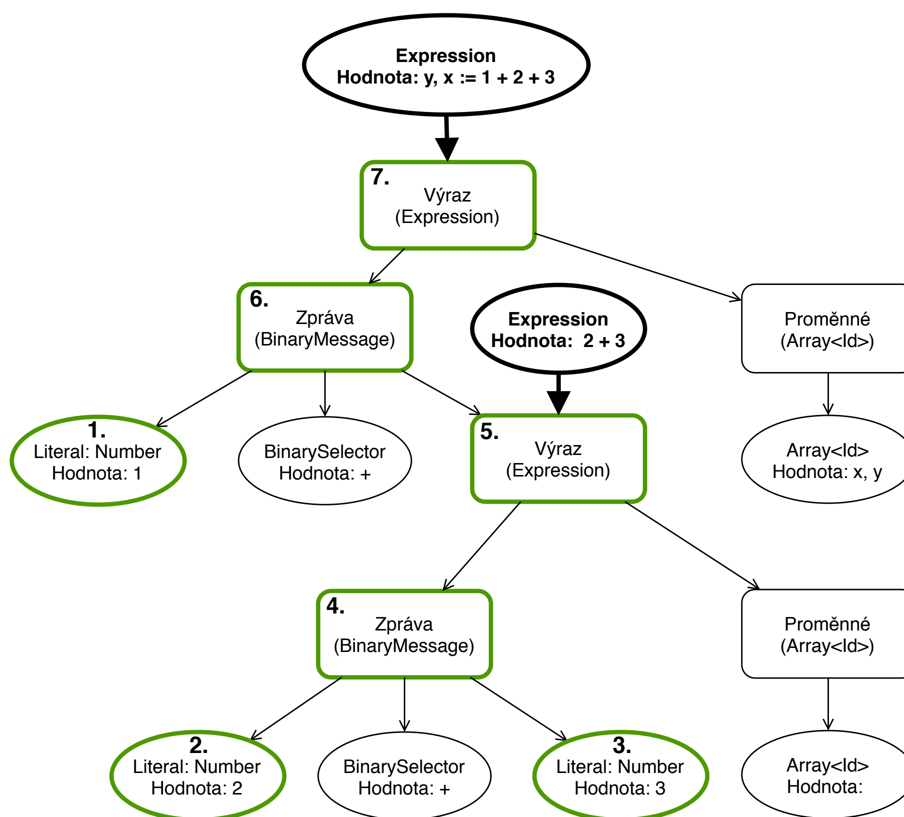
- Unární zpráva
  `useSelector(UnarySelector selector)`
- Binární zpráva
  `useSelector(BinarySelector selector, ObjectPN operand)`
- Zpráva z klíčových slov
  `invokeMethod(KeywordMessage callFrom, ArrayList<ObjectPN> operands)`

Tyto metody obsahují definici funkcionality základních selektorů, kterou by měli podporovat všechny objekty, mezi které patří selektory rovnosti, nerovnosti, rovnosti identity a nerovnosti identity. V případě potřeby implementování sémantiky jinému selektoru pro konkrétní typ objektu je nutné přepsat původní metodu (vytvořit `@Override` metody).

Metoda určená k přepsání musí odpovídat typu zprávy, na kterou by měl objekt umět reagovat. Na konci metody by mělo být přidáno ještě volání přepisované metody nadřazené třídy `ObjectPN` pro případ, že je danému objektu zaslána zpráva se selektorem, pro který nemá implementováno žádné chování.

6.3.3 Vyhodnocení výrazu

Vyhodnocení výrazu probíhá postupným vyhodnocováním listových uzlů stromové struktury výrazu postupně až ke kořenovému objektu výrazu. Pořadí zpracovávání jednotlivých listových uzlů probíhá postupným průchodem stromu za pomoci algoritmu „Post-Order traversal“ [16]. Tento algoritmus nejprve projde rekurzivně levý podstrom uzlu, pravý podstrom uzlu a následně vyhodnotí aktuální uzel, v němž se nachází. Příklad průchodu je předveden na obrázku 6.15 na zjednodušené reprezentaci příkladu z obrázku 6.11.



Obrázek 6.15: Znázornění pořadí zpracovávání členů výrazu průchodem „Post-Order“

Uzly stromu, které jsou zahrnuty v průchodu, jsou označeny zeleným okrajem. Jedná se o všechny uzly vyjadřující výrazy, zprávy a operandy zpráv (cíl zprávy je považován jako nultý operand). Každý uzel se zeleným okrajem má v sobě zároveň číslovku s pořadím, v jakém by byl v rekurzivním průchodu podle algoritmu „Post-Order traversal“ zpracován. Způsob zpracování jednotlivých uzlů je závislý na typu právě zpracovávaného uzlu.

Pokud se jedná o operand, který není ve tvaru dalšího výrazu, tak je uzel považován jako zpracovaný a postupuje se dál v průchodu. V případě, že zpracovávaný uzel je typu zpráva, tak je v tomto momentu zahájen proces zaslání zprávy cílovému objektu. To je provedeno zavoláním metody `solve()`, kterou implementuje každý objekt zprávy (viz. ob-

razek 5.15). Metoda `solve()` využívá volání podle konkrétního typu zprávy jedné z funkcí typu `useSelector(...)`, případně `invokeMethod(...)` pro získání výsledku po aplikaci selektoru na cílový objekt. Návratová hodnota metody `solve()` je výsledkem zaslání zprávy a je ve formě tokenu. Posledním typem uzlu je výraz, jehož zprácování probíhá tak, že se převezme výsledek zaslání zprávy z metody `solve()` a pokud existují nějaké proměnné pro přiřazení, tak je do nich získaný výsledek uložen. Závěrem zpracování výrazu je předání výsledku výrazu rodičovskému uzlu, aby mohlo pokračovat vyhodnocování. Vyhodnocování výrazu končí v situaci, kdy je vyhodnocen výraz kořenového uzlu.

Při procesu vyhodnocování výrazu se mohou vyskytnout různé situace, při kterých je vyžadováno přerušit vyhodnocování výrazu a vykonat příslušné operace interpretem. Tyto situace jsou označovány jako výjimky a jsou popsány v podkapitole 6.1.4. Pokračování této kapitoly se bude věnovat výjimkám, které nevyžadují přerušení běhu interpretu. Mezi tyto výjimky patří vytvoření nové instance objektu a vytvoření nové instance metody.

Vytvoření nové instance OOPN

V případě, že se v průběhu vyhodnocování výrazu generuje vyjimka o vytvoření nové instance uživatelské třídy OOPN, tak je vyhodnocování výrazu pozastaveno a je vykonána následující série akcí interpretem.

- Nově vytvořená instance je uložena ve formě jednoduché zprávy do původního výrazu.
- Další akcí je vložení nově vytvořené instance do kolekce aktivních instancí interpretu.
- Poslední akcí je návrat k vyhodnocování právě rozpracovaného výrazu/události.

V tomto momentě je nová instance OOPN připravená v kolekci aktivních instancí interpretu k jakýmkoliv interakcím.

Vytvoření nové instance metody

Druhou ze situací patřící do skupiny výjimek, která může nastat při vyhodnocování výrazu je vytvoření nové instance metody. Podobně jako je tomu u výjimky vytvoření nové instance OOPN, je vyhodnocování výrazu pozastaveno, a je vykonána následující série akcí interpretem.

- První akce je vykonána pouze v případě, že se jedná o metodu konstruktoru uživatelské tříd OOPN. Spočívá ve vložení nově vzniklé instance OOPN obsahující instanci metody konstruktoru do kolekce aktivních instancí interpretu.
- Druhou akcí je přiřazení aktuálně zpracovávané události do instance metody.
- Poslední akcí je vložení objektu podvýrazu do instance metody, který obsahuje zaslání zprávy s klíčovými slovy invokující danou metodu.

Po vykonání všech těchto akcí interpret přeruší vykonávání právě vykonávané události a pokračuje ve vykonávání další v pořadí, případně zahájí nový simulační krok. Dokončení metody proběhne až v čase, kdy je do jejího místa s názvem „return“ uložen nějaký token. Tento token je považován za výsledek metody a je zahájena akce k jejímu dokončení. Dokončení metody je vykonáno následující sérií akcí interpretem.

- Nejprve je získán z instance metody objekt výrazu, do kterého se má uložit výsledek metody.
- Druhou akcí je vytvoření jednoduché zprávy, do níž je výsledek uložen.
- Navazující akcí je vložení jednoduché zprávy s výsledkem na pozici zprávy původního volání metody.
- Poslední činností je odebrání instance metody z instance třídy OOPN, na níž byla metoda zavolána.

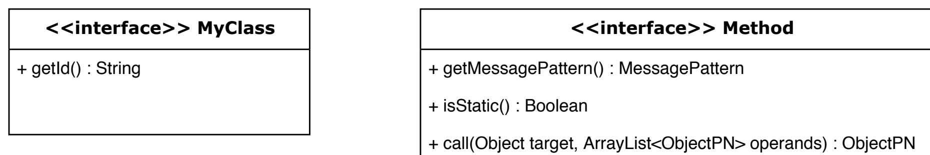
V tomto momentě přichází na řadu znovu zahájení vyhodnocování rozpracovaného události, která musela být pozastavena kvůli nutnosti čekat na dokončení metody instance uživatelské třídy OOPN.

6.4 Podpora Java objektů

Aplikace interpretu poskytuje podporu vybrané podmnožiny Java objektů s některými jejich metodami. V příloze C se nachází výčet všech podporovaných Java objektů s popisem metod, které poskytují. Tato podkapitola se tedy bude věnovat především principu jejich zakomponování do modulu interpretu. Nejprve bude popsáno rozhraní, které poskytuje prostředky k rozšiřování množiny podporovaných tříd objektů a jejich metod. A pro úplnost bude dále vysvětlen způsob jeho korektního použití s demonstračním příkladem.

6.4.1 Rozhraní

Za účelem možnosti definice nových typů objektů bylo navrženo jednoduché programové rozhraní. To umožňuje snadno zakomponovat podporu nových objektů z jazyka Java a metod jež poskytují. Rozhraní se skládá ze dvou částí, které jsou zobrazeny na obrázku 6.16



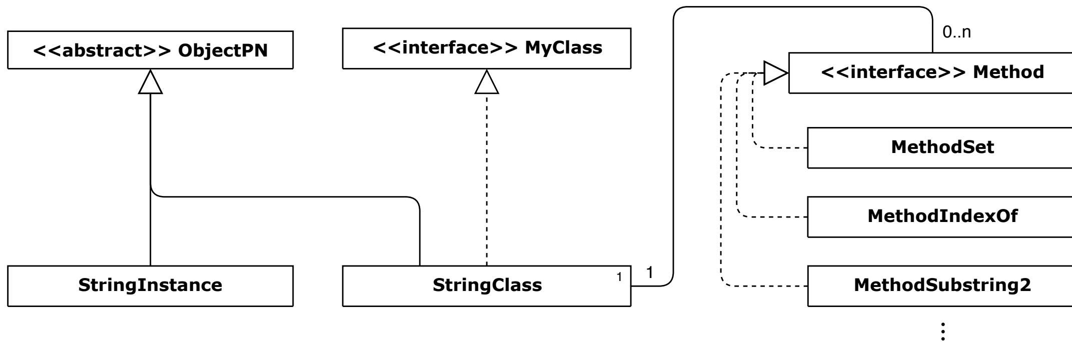
Obrázek 6.16: Rozhraní pro definici vlastních tříd a metod

Rozhraní `MyClass` slouží k zakomponování nových tříd do aplikace interpretu. Obsahuje pouze jedinou metodu, a to metodu `getId()` pro získání identifikace dané třídy. Nová třída by pak měla rozšiřovat také chování abstraktní třídy `ObjectPN`, která je základem všech tokenů vyskytujících se v sítích modelů OOPN. Rozhraní `Method` poskytuje výčet metod, které by měla poskytovat třída implementující určitou metodu dané třídy nebo její instance. První z metod, které by měla třída reprezentující metodu poskytovat je `getMessagePattern()`, kvůli nutnosti identifikování metody. Další poskytovanou metodou je `isStatic()`, která implementuje informaci o tom, zda-li má být metoda statická. Jelikož návratový typ této funkce je objekt typu `Boolean`, tak výsledkem funkce mohou být tři různé hodnoty, mezi které patří `Boolean.TRUE`, `Boolean.FALSE` a `null`. Hodnota `null` v tomto případě značí, že metoda může být volána v kontextu statickém i nestatickém. Poslední metodou, která je nutná implementovat ke splnění podmínek rozhraní `Method` je

metoda call(Object target, ArrayList<ObjectPN> operands). Tato metoda by měla implementovat funkcionalitu metody po jejím zavolání, kterou vyjadřuje daná třída implementující rozhraní Method.

6.4.2 Použití

Pro snazší pochopení principu použití rozhraní z podkapitoly 6.4.1 je zde uveden demonstrační příklad implementace objektu reprezentující třídu String z jazyka Java, její instanci a podporované metody. Příklad je uveden na obrázku 6.17.



Obrázek 6.17: Příklad implementace objektu String

Na příkladu je vidět, že definice třídy StringClass reprezentující třídu String z jazyka Java rozšiřuje chování abstraktní třídy ObjectPN a implementuje rozhraní MyClass. Implementaci třídy reprezentující třídu nového objektu (zde konkrétně StringClass) by bylo vhodné vždy realizovat návrhových vzorem „Singleton“, aby nedocházelo ke zbytečnému vytvoření více instancí reprezentující třídu. Dále je možné vidět definici instance třídy String ve formě třídy StringInstance, která musí taktéž rozšiřovat chování třídy ObjectPN, aby se instance nově přidané třídy mohly projevovat v sítích modelu jako tokeny. Zbývající část příkladu uvedená na pravé části obrázku 6.17 je věnována metodám nového objektu. Veškeré podporované metody nového objektu jsou vyjádřeny samostatnými třídami implementující rozhraní Method a jsou uloženy v kolekci objektu reprezentující třídu (zde konkrétně StringClass).

Po implementaci všech výše popsaných částí je nutné ještě vložit nově vytvořenou třídu do kolekce objektu třídy PNGenerator z podkapitoly 5.1.3. Vložení je možné provést pomocí připravené metody initExternClasses() implementovanou ve třídě PNGenerator. Tímto krokem by měl být nově přidaný typ objektu připravený k použití.

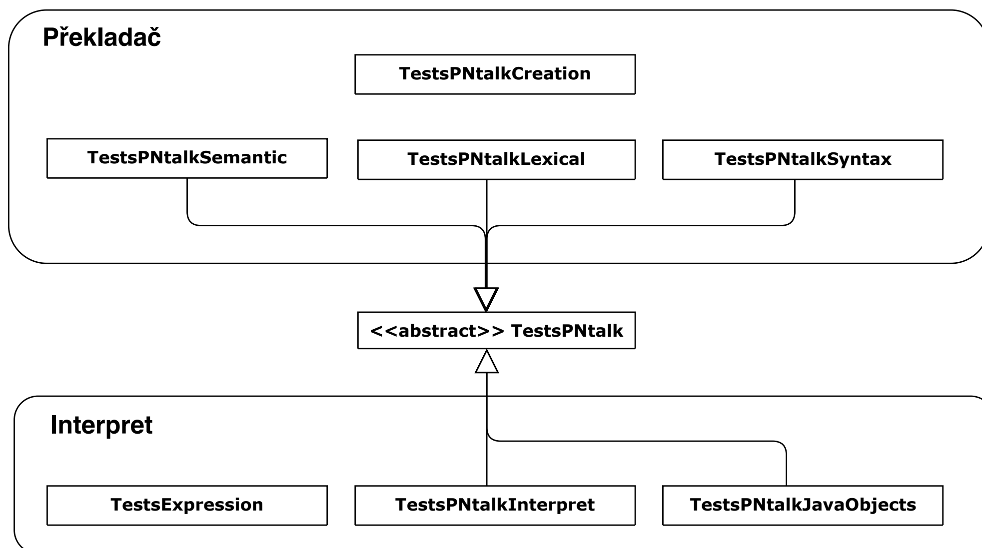
Kapitola 7

Testování

V této kapitole bude popsán způsob testování aplikace interpretu. Kapitola je rozdělena na část popisu struktury testování s bližším popisem jednotlivých segmentů a shrnutí výsledků testování. Testování jako takové je realizováno pomocí automatizovaných testů s použitím frameworku JUnit verze 5.4, který poskytuje možnost implementace jednotkových testů. Testy byly psány již v průběhu vývoje aplikace, především při dokončení nějakého logického úseku v aplikaci. Důvody, proč bylo zvoleno psaní testů už při vývoji aplikace, byly hned dva. První z nich byl kvůli ověření validity právě provedené implementace a druhým důvodem byla snaha o zamezení zanesení nových chyb do již existující implementace.

7.1 Struktura testování

Aplikaci interpretu, jakožto komplexní projekt, lze rozdělit na části a následně testovat jejich funkčnost v separátních oddílech. Jako hlavní dělení aplikace interpretu je dělení na dvě části, a to část překladače a část interpretu. Tyto dvě velké části lze však z pohledu testování ještě dále dělit podle sémantiky testů, kde každá sémantická skupina testů má svoji testovací třídu. Na obrázku 7.1 je uveden diagram struktury testovacích tříd a případných vztahů.



Obrázek 7.1: Struktura testovacích tříd

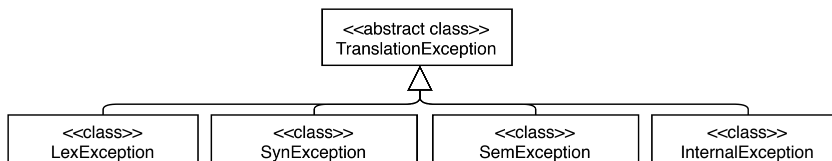
Na diagramu z obrázku 7.1 je možné vidět rozdělení na dvě skupiny testovacích tříd. Jedna skupina obsahuje třídy implementující testy překladače a druhá skupina je složena z tříd implementující testy interpretu. Dále balíček obsahuje abstraktní třídu `TestsPNtalk`, která nespadá ani do jedné z těchto skupin. Jedná se o pomocnou třídu, která obsahuje obecné metody využitelné v ostatní testovacích třídách. V následujících podkapitolách bude charakter jednotlivých tříd popsán blíže.

7.1.1 Překladač

Pro testování překladače byly navrženy testovací třídy odpovídající všem analýzám a procesům probíhajícím během překladu. Bližší informace o fázích překladu a analýzách s nimi spojenými se nachází v podkapitole 5.1. Testovanými fázemi jsou tedy následující:

- Lexikální analýza
- Syntaktická analýza
- Generování vnitřní reprezentance modelu
- Sémantická analýza

Každá z uvedených analýz byla testována na validních i nevalidních vstupech ve formě zdrojových souborů v jazyku PNtalk. K odlišení typů chyb vznikajících při překladu je využito systému výjimek. Hierarchie překladových výjimek je znázorněna na obrázku 7.2.



Obrázek 7.2: Hierarchie tříd překladových výjimek

Abstraktní třída `TraslationException` je na vrcholu hierarchie a je základem všech překladových výjimek. Ostatní uvedené třídy reprezentují konkrétní typy výjimek, které mohou nastat při procesu překladu. Třídy výjimek `LexException` a `SynException` označují chyby, které nastanou v průběhu lexikální nebo syntaktické analýzy. V průběhu překladové fáze generování komponent a struktur vnitřní reprezentace modelu, s níž je spjata sémantická analýza, se ještě může vyskytnout chyba sémantiky zápisu. Tento typ chyby je pokryt výjimkou třídy `SemException`. Poslední výjimka, která může být generována během překladu je reprezentována třídou `InternalException` značící chybu interního charakteru, jejíž příčinou není chyba ve vstupním souboru.

Lexikální testy

Lexikální testy ve třídě `TestsPNtalkLex` jsou zaměřeny dvěma směry, kde v prvním z nich je testována platnost jednotlivých načtených lexémů (tokenů) a druhý směr je spojen s lexikálním průchodem zdrojového textu, a případnou detekcí lexikálních nekonzistencí. Dále je uveden příklad ukázky kódu testu 7.1 na neplatný lexém (lexém nepatřící do jazyka).

```
@Test
public void testErrSymbolParagraph(){
    testLex("pntalk"+File.separator+"err-lex-symbol-paragraph.test", false);
}
```

Ukázka kódu 7.1: Test na neplatný lexém §

Ostatní lexikální testy jsou podobného tvaru, kde funkce `testLex(String file, boolean status)` provede načtení a analýzu všech lexémů ze zdrojového textu. Parametry funkce jsou název zdrojového souboru testu a požadovaný status testové operace. Pokud je status nastavený na `true`, tak je zdrojový soubor považován jako lexikálně validní.

Syntaktické testy

Gramatické konstrukce jsou validovány pomocí syntaktických testů ve třídě `TestsPNtalkSyntax`. Testy jsou tvořeny tak, aby byly pokryty všechny neterminální symboly gramatiky alespoň jedním testem. Zároveň jsou zdrojové texty testů v jazyku PNtalk stupňovány podle komplexnosti jazykových konstrukcí. Příklad syntaktického testu analyzující výraz v akci přechodu je uveden v ukázce kódu 7.2.

```
@Test
public void testOkMessageExpr6(){
    testSyntax("pntalk" + File.separator + "ok-message-expr6.test", true);
}
```

Ukázka kódu 7.2: Syntaktický test výrazu v akci přechodu

Ostatní syntaktické testy jsou podobného tvaru obdobně, jako tomu bylo u lexikálních testů. Pouze s tím rozdílem, že je zde využita funkce `testSyntax(String file, boolean status)`, která taktéž načítá tokeny jako tomu bylo u lexikálních testů, ale v tomto případě je využit objekt třídy `Parser` (viz. podkapitola 5.1.2) implementující realizaci syntaktického průchodu.

Testy generování vnitřní reprezetance modelu

Jakožto závěrečná fáze překladu, kdy by již všechny vstupy měly být lexikálně i syntakticky zvalidovány, jsou tyto testy uloženy separátně ve třídě `TestsPNtalkCreation`. Každý test obsahuje nějaký proces generování vnitřní reprezentace modelu OOPN (viz. podkapitola 5.2), který je poté testován, zda-li vygenerovaný model odpovídá předpokládanému tvaru. Do testování bylo zahrnuto generování nových uživatelských tříd OOPN. Dále také generování objektových sítí, sítí metod a částí, ze kterých se skládají. Velká pozornost byla věnována také generování multimnožin, které se mohou vyskytovat v modelech jako hranové výrazy, případně jako úložiště v místech, do nichž jsou vkládány tokeny. Jako příklad testu generování vnitřní reprezentace modelu je zde uvedeno přidání místa do sítě metody.

```
@Test
public void testAddPlaceToMethodNet() {
    PNGenerator png = new PNGenerator();
    ClassPN classPN;
    try {
        classPN = png.createClass("C0", "PN");
    } catch (SemException e) {
        throw new AssertionError(e);
    }
    MethodNet methodNet;
    MessagePattern m = new MessagePattern();
    Parameter param = new Parameter("par", "x");
    try {
        m.addParam(param);
        methodNet = png.createMethodNet(classPN, m, false);
    } catch (SemException e) {
        throw new AssertionError(e);
    }

    Place p = new Place("p1", null);
    methodNet.setPlace(p);

    assertEquals(classPN, png.getClassById("C0"));
    assertEquals(methodNet, ((ClassPN) png.getClassById("C0"))
                            .getMethodNet(m, false));
    assertEquals(p, ((ClassPN) png.getClassById("C0"))
                    .getMethodNet(m, false).getPlaceById("p1"));
}
```

Ukázka kódu 7.3: Test přidání místa do sítě metody

V ukázce kódu 7.3 je možné vidět vygenerování uživatelské třídy C0, sítě metody dané třídy se vzorem zprávy `par: x` a následné vložení nového místa `p1` do této sítě metody. Pokud se vyskytne nějaká sémantická chyba v průběhu generování modelu, tak je hned vyvolána výjimka o chybě v testu a test skončí neúspěšně. V opačném případě, kdy je vše vygenerováno bez sémantických chyb, tak je testováno, zda-li jsou opravdu jednotlivé komponenty správně zakomponovány do sebe. V tomto případě se jedná o test existence nově vytvořené třídy `C0` v objektu třídy `PNGenerator`. Dále také test prezence nově vytvořené metody ve třídě `C0`. A závěrem testu je ověření, zda-li se místo `p1` opravdu nachází v nově vytvořené síti metody ve třídě `C0`. Pokud všechny `assert` metody skončí bez vyvolání výjimky o chybě, tak test úspěšně skončí.

Sémantické testy

Sémantické testy ve třídě `TestsPNtalkSemantic` byly vytvořeny spíše za účelem integračního testování z hlediska zakomponování generátoru vnitřní reprezentace do syntaktického průchodu. Testy tedy byly navrženy tak, aby ověřovaly, zda-li jsou korektně detekovány sémantické chyby přímo ze zdrojového souboru modelu v jazyku PNtalk a následně korektně zpracovány.

7.1.2 Interpret

Při testování interpretu byla věnována velká pozornost výrazům, což bylo pojato jako jednotkové testování jednotlivých selektorů zpráv. Dále byly navrženy integrační testy pro ověřování funkce celého procesu interpretace. Posledními testy spadající pod testování interpretu jsou testy zaměřené na podporované Java objekty. V bodech se tedy testování interpretu zabývalo testováním:

- Vyhodnocování výrazů
- Proces interpretace
- Podpora Java objektů

Primární zaměření bylo věnováno na vyhodnocování validních vstupů získaných z překladových analýz a procesů interpretem, po kterém následuje testování výsledků na očekávané hodnoty.

Testy vyhodnocování výrazů

Nedílnou součástí interpretačního procesu je vyhodnocování výrazů, které probíhá v rámci jakéhokoliv vyhodnocování akce nebo stráže nacházející se v přechodu nebo synchronním portu. Testy na vyhodnocování výrazů jsou uloženy ve třídě `TestsExpression` a jsou zaměřeny především na testování výsledků získaných po aplikaci selektorů. Byla věnována velká pozornost, aby se dosáhlo, co nejvyššího možného pokrytí selektorů a různých kombinací vstupů. Příkladem testu je zde uvedena ukázka kódu 7.4 pro otestování binárního selektoru dělení se zápornými čísly.

```
@Test
public void testLiteralDivision3() {
    ObjectPN a = new Literal("-6", LiteralType.NUMBER);
    ObjectPN b = new Literal("-2", LiteralType.NUMBER);

    try {
        ObjectPN result = a.useSelector(null, BinarySelector.DIV, b);
        String resultStr = ((Literal) result).getValue();
        assertEquals("3", resultStr);
    } catch (ExpressionException e) {
        throw new AssertionError(e);
    }
}
```

Ukázka kódu 7.4: Test selektoru dělení se zápornými čísly

Během testování byl každý selektor podroben několika testům pro různé vstupní hodnoty. Taktéž byly testovány i vstupy, které vedly k neplatné operaci, jako je například dělení nulou, použití neplatného selektoru a podobně. V tomto případě je očekáváno vygenerování konkrétní výjimky při vyhodnocování výrazu (viz. podkapitola 6.1.4).

Interpretační testy

Proces samotné interpretace byl testován už jako celek, kde vstupem testu je vstupní soubor s modelem OOPN zapsaným v jazyku PNtalk. Během testu se provede vyhodnocení modelu pomocí interpretu bez omezení simulačních kroků a následně je testován očekávaný stav modelu. Testy jsou navrženy od jednoduchých interpretačních modelů, kde jsou testovány jednoduché přesuny tokenů v sítích pomocí provedení přechodů až po komplexnější modely obsahující vyhodnocování výrazů v akcích a strážích. Při testování je využívána strategie výběru událostí, která volí vždy první událost ze seznamu. V ukázce kódu 7.5 je jako příklad zobrazen test vytvoření nové instance třídy `C1` a zvalidování výsledku této operace.

```
@Test
public void testOkEventN1(){
    ArrayList<InstanceClassPN> instanceClassPNs;
    String testFilePath = "pntalk" + File.separator + "ok-event-n1.test";
    instanceClassPNs = testInterpret(testFilePath, true);
    Integer size = instanceClassPNs.size();
    assertEquals(2, size);

    InstanceClassPN netInstance1 = instanceClassPNs.get(1);
    assertEquals("C1", netInstance1.getClassID());
}
```

Ukázka kódu 7.5: Test vytvoření nové instance třídy `C1`

V testu se vyskytují dvě `assert` metody, kde první z nich ověřuje prezenci dvou instancí uživatelských tříd v seznamu aktivních instancí interpretu. Ověření je nastaveno na hodnotu dvě, jelikož ve výchozím stavu interpretu je vždy vytvořena jedna instance `main` třídy. Tudíž po provedení události vytvoření nové instance a její uložení do výstupního místa přechodu se v interpretu vyskytují právě dvě instance. Druhá metoda `assert` v zápětí ještě ověřuje, zda-li nově vytvořená instance odpovídá třídě, z níž byla vytvořena.

Testy podporovaných Java objektů

Testování podporovaných Java objektů je realizováno kombinací přístupů z testů vyhodnocování výrazů a interpretačních testů, což spočívá v tom, že v každém testu je testován pouze jeden Java objekt s nějakou jeho metodou a zároveň je vstup testu ve formě zdrojového souboru s modelem OOPN zapsaným v jazyku PNtalk. Jako příklad testu podpory Java objektů je zde uvedena ukázka kódu 7.6, na níž se nachází test metody `isDigit` objektu Character, který patří do množiny podporovaných objektů z jazyka Java.

```
@Test
public void testOkJavaCharacterIsDigit1(){
    ArrayList<InstanceClassPN> instanceClassPNS;
    String testFileName = "ok-java-character-isDigit1.test";
    String testFilePath = "pntalk" + File.separator + testFileName;
    instanceClassPNS = testInterpret(testFilePath);
    InstanceClassPN instanceClassPN1 = instanceClassPNS.get(0);

    Place p1 = new Place("p1", null);
    Place p2 = new Place("p2", null);
    Multiset m = new Multiset();
    m.addElement(new Literal("true", LiteralType.BOOLEAN));
    p2.addTokens(m);

    assertEquals(p1.getTokens(), instanceClassPN1.getObjectNet()
                                 .getPlaceById("p1").getTokens());
    assertEquals(p2.getTokens(), instanceClassPN1.getObjectNet()
                                 .getPlaceById("p2").getTokens());
}
```

Ukázka kódu 7.6: Test metody `isDigit()` objektu Character

V ukázce kódu 7.6 je možné vidět, že jako vstup testu je využit vstupní soubor. Model zapsaný v testovaném souboru obsahuje volání metody `isDigit` a následné uložení výsledku do místa `p2`. Testovaná síť obsahuje pouze dvě interakční místa, které jsou obě podrobeny otestování na obsah tokenů. Pokud obsah tokenů v místech odpovídá očekávaným hodnotám, tak je test vyhodnocen jako úspěšný.

7.2 Souhrn výsledků testování

Informace poskytnuté z procesu testování potvrzují správnost implementace simulátoru a demonstrují jeho vlastnosti. Celkově bylo vytvořeno více než dvě stě testů ověřující správnost jednotlivých procesů vykonávaných aplikací interpretu. Byla ověřena správnost implementace analýz překladače a ostatních procesů probíhajících v rámci překladu. Dále byly také provedeny testy ověřující správnost výsledků jednotlivých akcí a postupů probíhajících při vyhodnocování simulačních kroků v modulu interpretu. Veškeré vytvořené testy jsou přiloženy k výsledné aplikaci, která se nachází na přiloženém paměťovém médiu v podobě CD.

Kapitola 8

Závěr

Cílem práce bylo navrhnout vnitřní reprezentaci modelů OOPN vhodnou pro efektivní interpretaci a vytvořit překladač, který by měl umožnit překlad modelů zapsaných v jazyku PNtalk do vnitřní reprezentace. Následně pak navrhnout a realizovat interpret OOPN využívající tuto vnitřní reprezentaci modelů, který kromě objektů Petriho sítí musí umět pracovat i s vybranou podmnožinou objektů z jazyka Java. Tento cíl byl splněn.

Výsledkem práce je překladač implementovaný v programovacím jazyku Java, který převádí modely OOPN zapsané v jazyku PNtalk do vnitřní reprezentace připravené pro následnou interpretaci. Překladač dokáže statickou kontrolou validovat vstup v podobě zdrojového souboru popisujícího model OOPN a v případě chyby informuje uživatele o dané chybě, jejím typu a pozici ve zdrojovém souboru. Dále byl navržen a realizován interpret využívající takto zpracované modely OOPN, který dokáže provádět příslušnou simulaci těchto modelů. Do aplikace interpretu byla také zavedena podpora vybrané podmnožiny objektů z jazyka Java, se kterými umí interpret pracovat. Během řešení této práce byla navržena a sestavena sada testů ověřující správnost simulátoru a demonstrující jeho vlastnosti.

Literatura

[1] Cabac, L., Duvigneau, M., Moldt, D. a Rölke, H. *Modeling Dynamic Architectures Using Nets-Within-Nets: Renew GUI - Petri net and net instance* [online]. 2005 [cit. 2020-01-03]. Dostupné z: https://www.researchgate.net/publication/220783996_Modeling_Dynamic_Architectures_Using_Nets-Within-Nets.

[2] Ehrig, H., Juhas, G., Padberg, J. a Rozenberg, G. *Unifying Petri Nets: Advances in Petri Nets.* Springer, 2001. ISBN 978-3540430674.

[3] Goldberg, A. a Robson, D. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983. ISBN 978-0201113716.

[4] Gul A. Agha, G. R. *Concurrent Object-Oriented Programming and Petri Nets.* Springer, 2001. ISBN 978-3-540-45397-0.

[5] Janoušek, V. *Modelování objektů Petriho sítěmi.* 1998. Disertační práce. Vysoké učení technické v Brně, Fakulta informačních technologií.

[6] Jensen, K. *Hierarchical Coloured Petri Nets.* Springer, 1992. ISBN 978-3-662-06289-0.

[7] Kočí, R. *PNtalk system* [online]. 2008 [cit. 2020-01-01]. Dostupné z: http://perchta.fit.vutbr.cz/pntalk2k/28.

[8] Kočí, R. *PNtalk 2.0* [online]. 2019 [cit. 2020-01-01]. Dostupné z: http://perchta.fit.vutbr.cz/pntalk2k/.

[9] Kočí, R., Janoušek, V. a František Zbořil, j. *Object Oriented Petri Nets — Modelling Techniques Case Study* [online]. 2009 [cit. 2019-12-21]. Dostupné z: http://ijssst.info/Vol-10/No-3/paper4.pdf.

[10] Křivka, Z. a Kolář, D. *Studijní opora: Principy programovacích jazyků a objektově orientovaného programování* [online]. 2003 [cit. 2019-12-25]. Dostupné z: https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FIPP-IT%2Ftexts%2FIPP-II-ESF-1_3b_hyperlinks.pdf&cid=11469.

[11] Meduna, A. *Automata and Languages: Theory and Applications.* Springer, 2000. ISBN 978-1-4471-0501-5.

[12] Peterson, J. L. *Petri Nets Theory and The Modeling of Systems.* Prentice Hall, 1981. ISBN 978-0136619833.

[13] Petri, C. A. *Kommunikation mit Automaten.* 1962. Disertační práce. Universität Hamburg.

[14] Rábová, Z., Češka, M., Zendulka, J., Peringer, P. a Janoušek, V. *Modelování a simulace* [online]. 2005 [cit. 2019-11-22]. Dostupné z: https://www.fit.vutbr.cz/study/courses/IMS/private/SkriptaMS.pdf.

[15] Theoretical Foundations Group. *Renew* [online]. Department for Informatics of the University of Hamburg. 2016 [cit. 2020-01-03]. Dostupné z: http://www.renew.de/.

[16] Valiente, G. *Algorithms on Trees and Graphs*. Springer, Berlin, Heidelberg, 2002. ISBN 978-3-642-07809-5.

[17] Vojnar, T. *Various Kinds of Petri Nets in Simulation and Modelling* [online]. 1997 [cit. 2019-12-15]. Dostupné z: https://www.fit.vutbr.cz/~vojnar/Publications/vojnar-mosis97.ps.gz.

Příloha A

Syntax jazyka PNtalk

V této příloze je uvedena gramatika jazyka PNtalk v rozšířené Backus-Naurové formě, kterou definoval doc. Vladimír Janoušek ve své práci Modelování objektů Petriho sítěmi.[5] Gramatika obsahuje drobné úpravy, které byly zavedeny po konzultaci s vedoucím práce za účelem rozšíření funkcionality, případně kvůli odebrání nepoužívaných konstrukcí jazyka. Všechny úpravy gramatiky oproti původní definici jsou uvedeny v poznámkách pod čarou.

```
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] [action][1] [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" "{" exprs[2] "}"
action: "action" "{" [temps] exprs "}"
arcexpr: multiset

multiset: [n "‘"] c ["," [n "‘"] c]*
```

[1] Rozšíření o neterminální symbol `action` oproti původní definici gramatiky jazyka PNtalk.

[2] Výměna neterminálního symbol `expr3` za neterminální symbol `exprs`, aby stráž mohla obsahovat více výrazů oproti původní definici gramatiky jazyka PNtalk, která podporovala pouze jeden výraz.

```
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 ":="]* expr3 [3]
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 primary[4]]+
keysel: id":"
literal: number | string | charconst | symconst | arrayconst | "true"
        "false" | "nil" [5]
arrayconst: "#" array
array: "(" [number | string | symbol | array | charconst]* ")"
number: [-] [dig]+ ["." [dig]+] ["e"["-"][dig]+] [6]
string: "’"[char]*"’"
charconst: "$"char
symconst: "#"symbol
symbol: id | binsel | keysel[keysel]* | string
id: letter[letter|dig]*
selchar: "+" | "-" | "*" | "/" | "~" | "|" | "," | "<" | ">" | selchar2
selchar2: "=" | "&" | "\" | "@" | "%" | "?" | "!"
hexDig: "0".."9" | "A".."F"
dig: "0".."9"
letter: "A".."Z" | "a".."z"
char: letter | dig | selchar | "[" | "]" | "{" | "}" | "(" | ")" | char2
char2: " " | "^" | ";" | "$" | "#" | ":" | "." | "-" | "‘"
```

[3] Výměna neterminálního symbol `expr3` za neterminální symbol `expr2` z důvodu odebrání jazykové konstrukce s neterminálním symbolem `cascade`, aby nedocházelo k nedeterminismům v případě použití přiřazovacího operátoru.

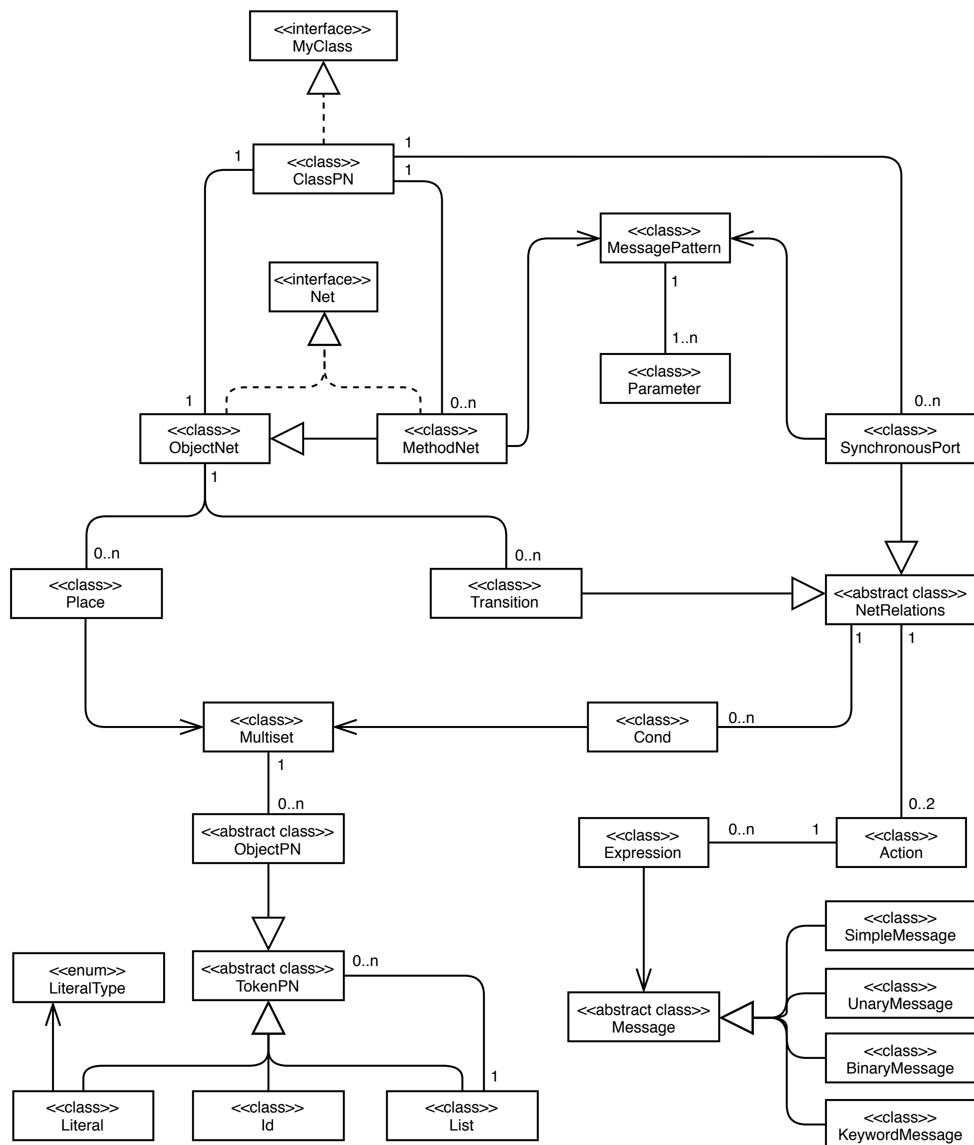
[4] Výměna neterminálního symbol `expr2` za neterminální symbol `primary`, aby nedocházelo k nedeterminismům při definici složených zaslání zpráv z klíčových slov vynucením použití závorek.

[5] Rozšíření neterminálního symbolu `literal` o terminální symboly "`true`", "`false`" a "`nil`" oproti původní definici gramatiky jazyka PNtalk.

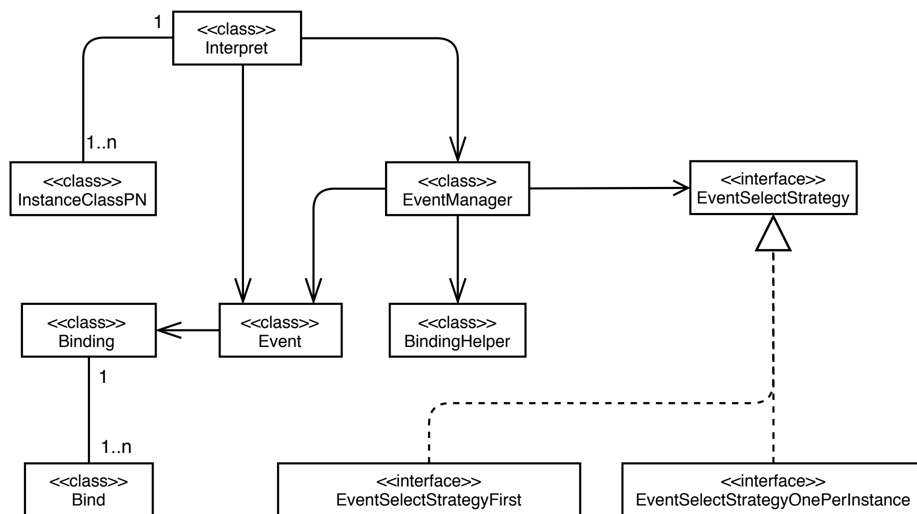
[6] Úprava složení neterminálního symbolu `number` oproti původní definici gramatiky jazyka PNtalk. Neterminální symbol `hexDig` změněn na `dig` a odebrána možnost volby radixu terminálním symbolem "`r`".

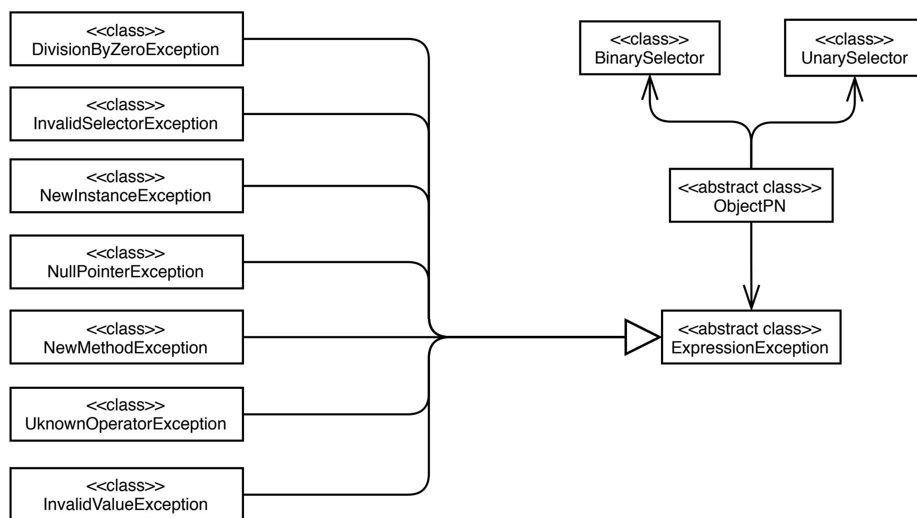
Příloha B

Diagramy



Obrázek B.1: Diagram vnitřní reprezentace modelu OOPN





Obrázek B.2: Diagram modulu interpretu

Příloha C

Podporované Java objekty

V této příloze jsou uvedeny třídy podporované interpretem, které zavádí podporu vybrané podmnožiny Java objektů s jejich konstruktory a metodami.

C.1 String

Třída reprezentující Java objekt String.

C.1.1 Konstruktory

- `new`
  Bezparametrický konstruktor.
- `set: str`
  Konstruktor s parametrem `str` přijímající řetězec nebo String objekt.

C.1.2 Metody

- `set: str`
  Metoda vrátí objekt typu String s nastavenou hodnotou podle parametru `str` přijímající řetězec nebo String objekt.
- `toLowercase`
  Metoda vrátí objekt typu String, který změní všechny znaky ve vstupním řetězci na malé.
- `indexOf: str`
  Metoda vrátí index, na kterém se nachází řetězec `str`.
- `substringBeginIndex: x endIndex: y`
  Metoda vrátí objekt string, který se nachází od indexu `x` do indexu `y`. Parametry `x` a `y` jsou očekávány ve tvaru primitivního objektu celého čísla jazyka PNtalk.
- `toLiteral`
  Metoda převede objekt typu String na primitivní objekt řetězce jazyka PNtalk.
- `instanceOf`
  Metoda vrátí třídu objektu.

C.2 Character

Třída reprezentující Java objekt Character.

C.2.1 Konstruktory

- `new`
  Bezparametrický konstruktor.

- `set: char`
  Konstruktor s parametrem `char` přijímající znak nebo Character objekt.

C.2.2 Metody

- `set: char`
  Metoda vrátí objekt typu Character s nastavenou hodnotou podle parametru `char` přijímající znak nebo Character objekt.

- `isLetter`
  Metoda vrátí primitivní boolovský objekt s informací, zda-li je znak písmeno.

- `isDigit`
  Metoda vrátí primitivní boolovský objekt s informací, zda-li je znak číslo.

- `isWhiteSpace`
  Metoda vrátí primitivní boolovský objekt s informací, zda-li je znak bílým znakem.

- `toLiteral`
  Metoda převede objekt typu Character na primitivní objekt znaku jazyka PNtalk.

- `instanceOf`
  Metoda vrátí třídu objektu.

C.3 Integer

Třída reprezentující Java objekt Integer.

C.3.1 Konstruktory

- `new`
  Bezparametrický konstruktor.

- `set: int`
  Konstruktor s parametrem `int` přijímající celé číslo nebo Integer objekt.

C.3.2 Metody

- `set: int`
  Metoda vrátí objekt typu Integer s nastavenou hodnotou podle parametru `int` přijímající celé číslo nebo Integer objekt.

- `doubleValue`
  Metoda vrátí objekt Double s hodnotou objektu Integer.

- `compareTo: int`
  Metoda vrátí primitivní objekt číslo s následující informací. Číslo je rovno 0, pokud jsou hodnoty obou Integer objektů stejné. Číslo je menší než 0, pokud je hodnota objektu Integer menší než je hodnota objektu Integer v parametru metody. Číslo je větší než 0, pokud je hodnota objektu Integer větší než je hodnota objektu Integer v parametru metody. Parametr `int` je očekáván ve tvaru primitivního objektu celého čísla jazyka PNtalk nebo objektu Integer.

- `toLiteral`
  Metoda převede objekt typu Integer na primitivní objekt čísla jazyka PNtalk.

- `instanceOf`
  Metoda vrátí třídu objektu.

C.4 Double

Třída reprezentující Java objekt Double.

C.4.1 Konstruktory

- `new`
  Bezparametrický konstruktor.

- `set: double`
  Konstruktor s parametrem `double` přijímající číslo nebo Double objekt.

C.4.2 Metody

- `set: double`
  Metoda vrátí objekt typu Double s nastavenou hodnotou podle parametru `str` přijímající číslo nebo Double objekt.

- `intValue`
  Metoda vrátí objekt Integer s hodnotou objektu Double bez desetinné části.

- `compareTo: double`
  Metoda vrátí primitivní objekt číslo s následující informací. Číslo je rovno 0, pokud jsou hodnoty obou Double objektů stejné. Číslo je menší než 0, pokud je hodnota objektu Double menší než je hodnota objektu Double v parametru metody. Číslo je větší než 0, pokud je hodnota objektu Double větší než je hodnota objektu Double v parametru metody. Parametr `double` je očekáván ve tvaru primitivního objektu čísla jazyka PNtalk nebo objektu Double.

- `toLiteral`
  Metoda převede objekt typu Double na primitivní objekt čísla jazyka PNtalk.

- `instanceOf`
  Metoda vrátí třídu objektu.

C.5 BigInteger

Třída reprezentující Java objekt BigInteger.

C.5.1 Konstruktory

- `new`
  Bezparametrický konstruktor.

- `set: str`
  Konstruktor s parametrem `str` přijímající řetězec nebo String objekt.

C.5.2 Metody

- `set: str`
  Metoda vrátí objekt typu BigInteger s nastavenou hodnotou podle parametru `str` přijímající řetězec nebo String objekt.

- `toLiteral`
  Metoda převede objekt typu BigInteger na primitivní objekt řetězce jazyka PNtalk.

- `instanceOf`
  Metoda vrátí třídu objektu.

C.6 BigDecimal

Třída reprezentující Java objekt BigDecimal.

C.6.1 Konstruktory

- `new`
  Bezparametrický konstruktor.

- `set: str`
  Konstruktor s parametrem `str` přijímající řetězec nebo String objekt.

C.6.2 Metody

- `set: str`
  Metoda vrátí objekt typu BigDecimal s nastavenou hodnotou podle parametru `str` přijímající řetězec nebo String objekt.

- `toLiteral`
  Metoda převede objekt typu BigDecimal na primitivní objekt řetězce jazyka PNtalk.

- `instanceOf`
  Metoda vrátí třídu objektu.

C.7 Transcript

Třída podporuje výstupní operace.

C.7.1 Statické metody

- `show: str`
  Vypíše řetězec `str` na standartní výstup. Parametr `str` přijimá řetězec nebo String objekt.

Příloha D

Manuál

Manuál k překladu a použití aplikace interpretu jazyka PNtalk

Překlad

V kořenovém adresáři je soubor `pom.xml`, který obsahuje informace potřebné k překladu pomocí nástroje Maven. Stačí tedy v tomto adresáři použít příkaz:

```
mvn package
```

Tímto se vytvoří spustitelná aplikace `pntalk-1.0.jar`.

Použití

Příklady použití aplikace `pntalk-1.0.jar`.

Aplikace se spustí pomocí příkazu:

```
java -jar pntalk-1.0.jar ../pntalk/demo-helloworld.pntalk
```

Spuštění interpretu v ladícím módu:

```
java -jar pntalk-1.0.jar ../pntalk/demo-guard.pntalk -d
```

Nápovědu pro další volby ke spuštění aplikace lze získat příkazem:

```
java -jar pntalk-1.0.jar -h
```

Programová dokumentace

Programová dokumentace je psána formátem JavaDoc a lze vygenerovat příkazy:

`mvn site` (Generuje kompletní dokumentaci projektu)

`mvn javadoc:javadoc` (Generuje dokumentaci k programu)

Vygenerovaná programová dokumentace je uložna do adresáře `target/site`.

Příloha E

Obsah CD

- `src/` - zdrojové soubory aplikace
- `pntalk/` - příklady modelů OOPN k demonstraci a testování
- `latex/` - zdrojové soubory k textu diplomové práce
- `xblaze31-Interpret-Petriho-siti.pdf` - text diplomové práce
- `class-diagram.pdf` - třídní diagram návrhu aplikace interpretu
- `pom.xml` - Maven konfigurační soubor
- `README.md` - manuál k aplikaci