VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

GENEROVÁNÍ KÓDU Z MODELŮ PETRIHO SÍTÍ

CODE GENERATION FROM OBJECT ORIENTED PETRI NETS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **TOMÁŠ FRYČ**
AUTHOR

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

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Fryč Tomáš**
Program: Informační technologie
Název: **Generování kódu z modelů Petriho sítí**
Code Generation from Object Oriented Petri Nets
Kategorie: Softwarové inženýrství

Zadání:

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

Literatura:

- Krysztof Czarnecki, Ulrich Eisenecker. Generative Programming: Methods, Tools, and Applications. Addison-Wesley Professional, 2000. ISBN-13: 978-0201309775

Pro udělení zápočtu za první semestr je požadováno:

- První 3 body.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Kočí Radek, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 1. listopadu 2018

Abstrakt

Tato práce se zabývá modelováním systémů pomocí Objektově orientovaných Petriho sítí (OOPN) a navržením mechanismu, který tento formalismus transformuje do programovacího jazyka Java. V první části práce je popis Petriho sítí, rozšíření o objektovou orientaci a jazyka PNtalk, který je konkretní implementací OOPN. Další část se zabývá samotným mechanismem, který z modelu zapsaných v jazyku PNtalk vygeneruje ekvivalentní modely v Javě a popisuje způsob simulace takto vygenerovaných modelů. Generátor je koncipiován tak, aby byl snadno rozšiřitelný a vygenerované modely modifikovatelné.

Abstract

This thesis describes modeling systems using Object Oriented Petri Nets (OOPN) and design of mechanism which transforms this formalism into Java programming language. The first part of the thesis describes Petri Nets formalism, its extension to object orientaton and PNtalk language, which is specific implementation of OOPN. The next part deals with the mechanism that generates equivalent models in Java from models described by PNtalk, and describes how to simulate these generated models. The generator is designed to be easily expandable and the generated models can be easily modifiable.

Klíčová slova

generátor kódu, objektově orientové Petriho sítě, PNtalk, transformace, překladač jazyka, simulátor

Keywords

code generator, object oriented Petri nets, PNtalk, transformation, compiler, simulator

Citace

FRYČ, Tomáš. *Generování kódu z modelů Petriho sítí.* Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Kočí, Ph.D.

Generování kódu z modelů Petriho sítí

Prohlášení

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

.....
Tomáš Fryč
15. května 2019

Poděkování

Chtěl bych poděkovat vedoucímu práce panu Radku Kočímu za jeho rady, trpělivost a vedení při vypracování práce.

Obsah

Kapitola 1

Úvod

S rozvojem softwarového inženýrství stoupají požadavky na kvalitní návrhy softwarů, které jsou komplexní a náročné. Modelování systémů umožňuje vytvořit ekvivalentní model nad kterým lze provádět simulace. Tím lze zkoumat vlastnosti systému, které jsou důležité [5].

Petriho sítě umožňují modelovat paralelismus a nedeterminismus. Jedná se o matematický formalismus, který poskytuje teorii pro analýzu a verifikaci [7]. Samotné Petriho sítě však nestačí pro modelování složitějších systémů, jelikož nenabízí výhody strukturovaného zápisu. Vzniklo tedy rozšíření o objektovou orientaci, tzv. Objektově orientované Petriho sítě (zkráceně OOPN), které poskytují zapouzdření a paralelismus. Jedná se však pořád o matematický formalismus, který je potřeba implementovat. Jazyk PNtalk umožňuje zapisovat modely definované formalismem OOPN. Dosavadní implementace běhu simulace v jazyku Smalltalk je v dnešní době příliš náročná. Jako alternativa se nabízí implementace v jazyku Java či C++.

Cílem mé bakalářské práce je navrhnout mechanismus transformace modelů, které jsou popsány formalismem OOPN. Hlavní náplní je tedy vygenerovat zdrojové soubory v jazyce Java, které budou reprezentovat modely zapsané v jazyce PNtalk včetně doplnění simulačního prostředí pro provedení modelu. Takový generátor by měl být snadno rozšiřitelný v případě budoucího přidání nových funkcionalit a vygenerované soubory modifikovatelné. Jazyk Java nabízí efektivnější běh simulace než implementace pomocí jazyka Smalltalk, možné využití kombinace s objekty Java a výhodu nezávislosti na operačním systému.

Celkový mechanismus transformace a následné simulace je rozdělen do tří aplikací. První aplikace - překladač - zpracuje model zapsaný v jazyce PNtalk a vygeneruje odpovídající zdrojové soubory v jazyce Java. Druhá aplikace - simulátor - slouží jako knihovna pro provedení vygenerovaného modelu. Třetí aplikace - spouštěcí - využívá knihovnu simulátoru a zdrojových souborů z překladače a spouští průchod a provedení modelu.

V kapitole 2 bude popsán základní formalismus Petriho sítí, jeho rozšíření o objektovou orientaci a popis syntaxe a možnosti jazyka PNtalk, který tento formalismus implementuje. Poté v kapitole 3 bude popsán můj návrh mechanismu transformace modelu zapsaného v jazyce PNtalk do zdrojových souborů Java, model aplikace a popis struktury tříd, které uchovávají data o modelu. V kapitole 4 bude nastíněna teorie o překladačích, můj návrh překladu a generování zdrojových souborů včetně demonstračních příkladů implementace. Dále bude v kapitole 5 představen návrh simulátoru, který slouží jako knihovna pro provedení modelu včetně popisu mé implementace procesů. V další kapitole 6 je představena metodika testování aplikace a demonstrace implementace na příkladech. V poslední kapitole 7 je shrnutí výsledků, dosažení cílů, omezení a možná rozšíření aplikace.

Kapitola 2

Objektově orientované Petriho sítě

V této kapitole jsou popsány základní Petriho sítě a jejich formalismy, z kterých vychází Objektově orientované Petriho Sítě, zkráceně *OOPN*. Budou rozebrána jednotlivá pravidla a komponenty, ze kterých se OOPN skládá. Nakonec bude popsán jazyk PNtalk, který představuje implementaci formalismu OOPN. Při studiu Petriho sítí, OOPN a PNtalku jsem vycházel z disertační práce doc. Janouška [3].

2.1 Petriho síť

Petriho sítě (dále jen *PN*) se používají k modelování a simulování diskrétních systémů. To znamená, že stav systému se může změnit pouze po výskytu nějaké definované události. Stav systémů v PN je rozdělen do množiny *parciálních stavů*. Parciální stav je reprezentován *místem*, zatímco událost je znázorněna *přechodem*.

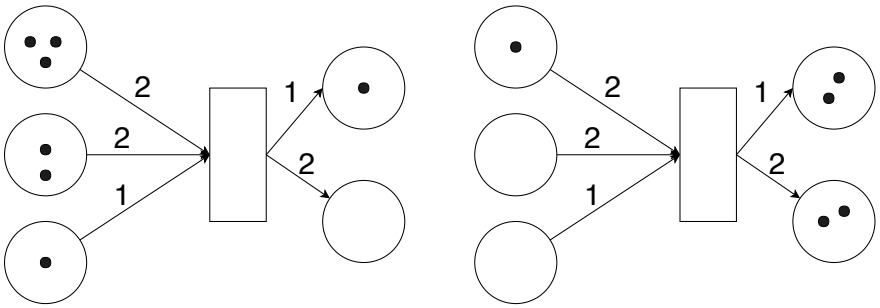
Místo může obsahovat značky, které jsou v grafických PN znázorněny tečkami a představuje pouze vlastnost celého systému.

Přechod obsahuje *hrany*, které představují vstupní a výstupní místa. V grafických PN jsou názorněny šipkami mezi místem a přechodem. Přechod je proveditelný v případě, že vstupní místa obsahují značky, které jsou definovány ve vstupní podmínce. Provedení přechodu probíhá ve dvou krocích. Prvně se odeberou značky ze vstupního místa dle vstupní podmínky a poté se do výstupního místa umístí značky definované ve výstupní hraně. Počet značek, které se mají odebrat/vložit se specifikují kardinalitou hrany. Tyto operace jsou atomické, což znamená, že se musí provést zároveň. Přechod se provádí pouze lokálně, takže v jednom kroku se změní pouze místa, která jsou spojená s daným přechodem.

Z toho vyplývá, že místa a přechody existují staticky a po celou dobu simulování, ale značky mohou vznikat a zanikat na základě provádění přechodů.

Z daného místa může být proveditelných více přechodů současně, což tvoří nedeterminismus. Tento problém se dá řešit pomocí různých rozšíření, mezi které patří například:

- Pravděpodobnost přechodu - přechodům, které jsou v konfliktu, se nastaví pravděpodobnost, podle které se mohou vyhodnotit a provést
- Priorita přechodu - přechodům, které jsou v konfliktu, se nastaví priorita pořadí, podle kterých se začnou vyhodnocovat a provádět.



(a) Před. (b) Po.

Obrázek 2.1: Provedení přechodu v Petriho síti.

2.2 Objektově orientované Petriho Sítě

Klasické Petriho sítě poskytují možnost modelovat diskrétní systém, ale v případě složitějších systémů se daná síť stane příliš komplikovanou a nečitelnou. Z toho důvodu vznikla myšlenka spojení Petriho sítí a objektové orientace. Každý objekt je definován svým stavem a operacemi, které poskytují ostatním objektům. Princip objektové orientace spočívá v *zapouzdření*, *dědičnosti* a *polymorfismu*. Z definice zapsané v opoře [9] vyplývá, že:

1. Zapouzdření představuje ukrytí struktury objektu. To znamená, že je viditelné pouze rozhraní daného objektu, které definuje způsob jeho použití. Změna vnitřního stavu objektu je možná pouze skrz operace z rozhraní.

2. Dědičnost umožňuje objektu dědit určité vlastnosti z jiného objektu. Takto se mohou sdílet chování a popřípadě je rozšiřovat. Zajištuje se tak rozšiřitelnost a udržovatelnost systémů.

3. Polymorfismus zaručuje, že objekty mohou využívat stejně definovaného rozhraní. Chování objektu se poté liší vlastní implementaci takového rozhraní. To znamená, že proměnná v programu může obsahovat různé objekty. Při zaslání stejné zprávy této proměnné v různých časech může vyvolat různé reakce.

Jedním řešením je formalismus OOPN, které je definované v disertační práci *Modelování objektů Petriho sítěmi* [3]. V objektově orientovaných Petriho sítích je systém definován jako množina objektů, které během života systému mohou dynamicky vznikat i zanikat. Každý objekt má stav, který se průbežně mění. Stav je možné změnit zasláním zprávy. Model celého systému je množina tříd, kdy jedna je označena za počáteční. Tato třída je instanciována okamžitě při spuštění modelu.

2.2.1 Objekt

Každý objekt má stav, který se v časech mění. Objekty spolu mohou komunikovat pomocí zasílání *zpráv*. V Petriho sítích se objekt rozlišuje na dva typy:

1. Primitivní objekt - představuje konstantu. Například číslo, booleovskou hodnotu, řetězec, znak aj.

2. Neprimitivní objekt - je uživatelem definovaný objekt, který je specifikovaný pomocí *třídy*. Obsahuje informaci o stavu, která se průběžně mění.

2.2.2 Třída a dědičnost

Třída obsahuje *objektovou síť* a množiny *sítí metod*, *konstruktorů* a *synchronních portů*. V rámci dědičnosti má definovaného právě jednoho předchůdce, kde na vrcholu je třída *PN*. Dědit musí právě třídu PN a nebo takovou, která ji na vrcholu hyerarchie dědičnosti má. Každá třída získává od svého předchůdce všechny vlastnosti, tedy objektovou síť, množinu sítí metod, konstruktory a sychronní porty. Všechny vlastnosti se dají přidat a nebo modifikovat pomocí nové definice pro stejný *selektor zprávy*. V rámci redefinice přechodu se musí předefinovat i *hrany*, které náleží přechodu.

Vytvořením objektu třídy, pomocí konstruktoru **new**, se automaticky aktivuje objektová síť dané třídy.

2.2.3 Sítě

Znázorňují Petriho sítě, které jsou tvořeny místy, přechody a hranami. Jedním typem je *síť objektu*. Každý třída ji může obsahovat pouze jednu. Místa takové sítě musí být globálně přístupné, jelikož k nim mají přístup metody i synchronní porty dané třídy. Druhým typem je *síť metody*. Ta vzniká při zavolání metody a zaniká při jejím ukončení. V rámci třídy může přistupovat k místům sítě objektu. Každá metoda má *selektor zprávy* na který reaguje. Dále obsahuje formální parametry, parametrová místa, které slouží pro uložení předávaných argumetů metody. Jako výstup metody je definované místo *return*, do kterého se ukládá výsledek invokované metody. Metoda může být invokovaná v akci přechodu jakékoliv třídy.

2.2.4 Místo

Jak již bylo zmíněno, místa představují *parciální stav* pomocí značek, které obsahují reference na objekty.

Značky

Značkám se mohou zasílat zprávy, které se liší dle typu objektu. Značka může být:

1. Primitivní - konstanty, jako např. čísla, řetězce, symboly
2. Třídy - během změny v čase se nemění
3. Neprimitivní - instance tříd Petriho sítí

Zpráva

Zpráva se skládá z adresáta zprávy, který na ni reaguje. Jménem operace, tzv. *selektor*, který specifikuje typ zprávy a parametry, které jsou pro vykonání zprávy nutné.

2.2.5 Přechod

Proveditelnost přechodu v OOPN je založená na stejném principu jako v klasických Petriho sítích. Aby byl proveditelný, musí platit vstupní a výstupní podmínka včetně

tzv. *stráže přechodu*. V přechodu lze pracovat s lokálními netypovanými proměnnými, které do přechodu vstupují pomocí hran.

Na rozdíl od klasických Petriho sítí se přechod nemusí provést automaticky. Přechod je rozšířen o *akci* a v případě, že se během provádění akce zavolá metoda Petriho sítě, čeká se na návrat z dané metody.

Hrany

Hrany, tedy podmínky přechodu obsahují nejen kardinalitu, ale i proměnnou, která bude reprezentovat referenci na objekt. Zde vzniká nedeterministické chování v případě, že je možné více navázání proměnné.

Stráž přechodu (guard)

Obsahuje posloupnost výrazů, které obsahuje posloupnost zaslání zpráv. Slouží k otestování stavu objektu. Stráž přechodu je vyhodnocena jako pravdivá pouze v případě, kdy každé zaslání zprávy je vyhodnoceno jako pravdivé. To znamená, že mezi jednotlivými výrazy je operace logického součinu, tzv. *konjunkce*. Stráž přechodu umožnuje i zaslání zprávy neprimitivnímu objektu. Sémantika zaslání zprávy se liší dle adresáta.

1. Primitivní objekt - výraz se vyhodnotí v rámci inskripčního jazyka, např. `10 < 2`.
2. Neprimitivní objekt - jedná se o synchronní komunikaci s další třídou pomocí tzv. *synchronního portu*. V tomto případě je zpráva vyhodnocena jako pravdivá pouze v případě, je-li synchronní port adresáta zprávy (třída) pro určité navázání proměnných proveditelný a zároveň není v konfliktu s přechodem či jiným synchronním portem.

Akce přechodu

Specifikuje zaslání zprávy objektu. Způsob provedení akce, podobně jako u stráže, závisí na adresátovi zprávy.

1. Primitivní objekt - představují konstanty, přechod se provede atomicky.
2. Třída - rozumí pouze zprávě **new**. Akce, tedy přechod, se provede atomicky s vytvořením nové instance třídy, která se uloží do případné proměnné dle výrazu.
3. Neprimitivní objekt - v případě třídy se postupuje takto:
   - Odeberou se značky dle vstupní podmínky přechodu.
   - Podle selektoru zprávy se nalezne odpovídající metoda, která se má provést.
   - Vytvoří se instance metody a do parametrových míst se vloží značky dle formálních parametrů.
   - Přechod čeká, až se vykoná metoda a dostane odpověď. V průběhu čekání se mohou provádět ostatní přechody.
   - Ve chvíli, kdy se do výstupního místa metody umístí značky reprezentující výsledek, metoda končí. Výsledek se uloží do proměnné dle výrazu a provede se výstupní podmínka přechodu, tedy umístění značek do míst.

2.2.6 Synchronní port

Synchronní port je součástí definice třídy. Má charakter přechodu i metody. To znamená, že obsahuje vzor zprávy, na který reaguje, ale provádí se stejným způsobem jako přechod. Port neobsahuje místa ani přechody, ale pouze pouze hrany a stráž přechodu.

Narozdíl od metody může být synchronní port volán ze stráže přechodu místo z akce. Může být volán buď s předem vyhodnocenými parametry, nebo nevyhodnocenými parametry. Pokud je argumentem předána navázaná proměnná, port je proveditelný, pokud lze najít navázání proměnné na daný token. V druhém případě získá referenci na objekt.

2.2.7 Události

Formalismus OOPN definuje 4 typy událostí, které mohou nastat.

1. Událost typu A (atomic) - atomické provedení přechodu, kdy synchronní komunikace pomocí portů může ovlivnit stav jiných objektů.
2. Událost typu N (new) - vytvoření nové instance třídy pomocí zprávy **new**. Přechod se provede stejně jako událost typu A, atomicky.
3. Událost typu F (fork) - předání zprávy neprimitivnímu objektu (třídě). Odebere značky ze vstupního místa, vytvoří odpovídající metodu dle selektoru zprávy a následně se vytvoří nová instance. Všechny operace atomicky. Spoléhá na vytvoření posledního typu události, typu J.
4. Událost typu J (join) - přijetí výsledku a zrušení instance invokované metody. Uloží se značky do výstupních míst přechodu. Obě operace atomicky.

2.3 PNtalk

V této části bude popsán programovací jazyk PNtalk, který vychází z imperativního jazyka Smalltalk. Jedná se o příklad implementace formalismu OOPN. Jazyk poskytuje různá rozšíření, která řeší zjednodušení zápisů oproti této definici, např. složené zprávy nebo seznamy. Bude znázorněna grafická i textová syntaxe jazyka. Model zapsaný tímto jazykem bude sloužit jako vstup pro aplikaci. Při popisu jednotlivých komponent bude využíván zápis:

- [...] znamená, že obsah v hranatých závorkách je nepovinný.
- [...]* znamená nepovinný výskyt 0 ... N.
- ... | ... [| ...]* znamená výběr jedné z variant.

Model je v jazyku PNtalk reprezentován množinou tříd, které spolu mohou komunikovat. Zápis takové množiny je:

```
[třída]* main jméno_hlavní_třídy [třída]*
```

Jedna třída je označená jako počáteční třída, jenž je implicitně instanciována při spuštění (simulaci) modelu. Tato třída je označená pomocí klíčového slova "main", za kterým následuje název dané třídy.

2.3.1 Třída

Jak již bylo zmíněno v předchozí podkapitole, třída je složená ze sítě objektu a množin sítí metod, konstruktorů a synchronních portů.

```
class hlavička_třídy [objektová_síť] [metoda|konstruktor|synchronní_port]*
```

Nová třída se označuje klíčových slovem `class`, po kterém následuje zápis hlavičky.

```
jméno_třídy is_a jméno_nadřazené_třídy
```

Kde jméno nadřazené třídy musí být taková třída, která buď dědí nebo přímo je třídou *PN*. Každá třída může mít maximálně jednu objektovou síť a poté množiny sítí metod, konstruktorů a synchronní porty. Jednotlivé elementy budou popsány v následujících podkapitolách.

2.3.2 Sítě

Sítě jsou tvořené místami a přechody, které jsou propojené hranami. Hrany jsou vázané na přechody, z toho důvodu jsou schované v samotném přechodu. Jak už bylo řečeno, sítě se dělí na objektovou a síť metod, proto se musí i zápis lišit.

```
object [místo|přechod]*
```

Z toho vyplývá, že objektová síť obsahuje pouze klíčové slovo `object` a poté samotnou síť. Metoda se liší v tom, že navíc obsahuje vzor zprávy, na který musí reagovat.

```
method vzor_zprávy [místo|přechod]*
```

Po klíčovém slovu `method` následuje vzor zprávy, po jejímž přijetí se dynamicky vytvoří instance dané sítě. Zpráva se skládá ze selektoru zprávy a parametrů. Metoda má přístup k místům objektové sítě skrze hrany. Může obsahovat vstupní parametrová místa definovaná ve zprávě jako argumenty. Avšak výstupní místo *return* musí obsahovat každá metoda.
Posledním typem sítě je tzv. *konstruktor*.

```
constructor vzor_zprávy [místo|přechod]*
```

Zápis konstruktoru je stejný jako u metody s rozdílným klíčovým slovem. Konstruktory slouží k vytvoření nové instance třídy, tedy objektu. Tato operace se provádí posláním zprávy *new*. K inicializaci s parametry slouží tzv. *neimlicitní konstruktor*. Sémantika takové instancializace je následující:

- Vytvoření instance pomocí zprávy *new*.
- Zaslání parametrizované zprávy vytvořenému objektu.
- Konstruktor vrací hodnotu *self*, jenž reprezentuje jméno daného objektu.

Funkcionalita konstruktoru lze nahradit metodou, která bude vkládat značky do míst.

2.3.3 Multimnožina

Multimnožina se vyskytuje na hranách přechodu jako tzv. *hranový výraz* nebo v místech jako tzv. *počáteční značení*. Je to zápis množiny, kde má každý prvek kvantifikátor, který reprezentuje počet výskytů prvku.

Multimnožina má tvar:

```
n1‘c1, n2‘c2, ..., nm‘cm
```

Prvky `n1, n2, n3 ... nm` značí kvantifikátory, které mohou být:

- celé kladné číslo
- proměnná, která je vyhodnotitelá jako celé kladné číslo. Jinak je automaticky vyhodnocená jako nula.

Prvky `c1, c2, c3 ... cm` značí prvky množiny, které mohou být:

- primitivní datový typ - číslo, znak, řetězec ...
- proměnná
- seznam

Příklad multimnožin je:

```
4‘#e, x‘3, x‘y, 4‘(x, y, z)
```

Tento zápis představuje multimnožinu, která obsahuje `4` výskyty symbolu `e`, `x` výskytů čísla `3`, `x` výskytů proměnné `y` a `4` výskyty seznamu, který má prvky `x, y, z`.

2.3.4 Výraz

Výraz, tzv. zaslání zprávy, se může vyskytnout v akci a stráži přechodu nebo počáteční akci v místě. Speciálním případem výrazu je *term*.

Term

Nejjednodušším výrazem v jazyce PNtalk je term. Reprezentuje všechny objekty, mezi které patří:

1. Literál
   - Čísla - objekty, které reprezentují číselnou hodnotu. Je to sekvence číslic. Číslo může být decimální, hexadecimální, záporné, desetinné a exponenciální. Reagují matematické operace. Příklad: 42, 2A, -31, 0.019, 1.53e7.
   - Znaky - znaky abecedy včetně speciálních znaků, které jsou uvozeny `$`. Příklad: `$3, $Z, $+, $:, $-`. Všechny povolené možnosti jsou v příloze syntaxe Pntalk.
   - Symboly - objekty, které se primárně používají jako jména. Je to sekvence znaků, které jsou na začátku uvozeny symbolem `#`. Příklad: `#a, #abc, #’symbol’`.

- Řetězce - objekty, které stejně jako symboly reprezentují sekvenci znaků. Místo uvození znakem `#` je uzavřen v apostrofech. Narozdíl od symbolů není u řetězců záruka, že dva stejně zapsané řetězce odpovídají stejnému objektu. Příklad: `'retezec', 'abc', 'Hello world', 'Bakalářská práce'`.
- Booleovské hodnoty - objekty, které reprezentují `true` a `false`. Reagují na logické operace.
- Nedefinovaný - objekt, který reprezentuje neinicalizované proměnné. Zapisován je jako `nil`.
- Pole, blok - je rozšíření jazyka Smalltalk.

2. Proměnná - v průběhu simulace mohou reprezentovat různé objekty - literály. Reprezentovány jsou identifikátory, což je sekvence znaků, které musí začínat malým písmenem. Příklad: `a, abc, varX`.
3. Pseudoproměnná - hodnota objektu závisí na kontextu, kde je použitá. Existují pseudoproměnné `self` a `super`. Příklad: Konstruktor třídy po vytvoření instance vrací hodnotu `self`, která reprenzetuje jméno daného objektu.
4. Jména tříd - objekt, který reprezentuje název třídy. Z toho důvodu je konstantou, která se nemění. Stejně jako proměnná je identifikátorem, ale musí začínat velkým písmenem. Příklad: `C1, ClassPN1, MediumInputClass`.

Zaslání zprávy

Zaslání zprávy má syntax:

```
<adresát> <zpráva>
```

Adresátem může být:

- Primitivní typ, tedy konstanta, které se pošle odpovídající zpráva.
- Třída, které se může poslat pouze zpráva typu **new**.
- Neprimitivní typ, tedy Petriho síť, které se může poslat zpráva, která invokuje metodu dané třídy.

Zpráva má obecný zápis:

```
<selektor> <argumenty>
```

Selektor udává, kterým typem zpráva je. Možnosti jsou:

- Unární zpráva - v tomto případě je selektor *identifikátor*, po kterém nenásledují žádné argumenty. Příkladem selektoru je **`abs, negated, truncate, rounded`**.
- Binární zpráva - selektor se liší dle typu adresáta, příkladem je:
  1. Číslo - `>, >=, <, <=, +, -, *, /, //` (celočíselné dělění)
  2. Boolean - `&` (logický součin), `|` (logický součet), `not` (negace)

3. Obecné - zprávy, kterým by měl rozumět každý objekt: `==` (rovnost identity), `==` (nerovnost identity), `=` (rovnost), `=` (nerovnost)

- Zpráva s klíčovými slovy - zpráva obsahuje 1 a více klíčů zapsané za sebou s rozšířením, že za selectorem následuje dvojtečka. Příkladem může být: `at:1 put:#e`.

Záslání zprávy se ještě dělí na dva podtypy:

- Jednoduché zaslání zprávy - v případě, že adresát i argument zprávy je *term*.
- Složité zaslání zprávy - adresát či argument zprávy je další zaslání zprávy.

Vyhodnocování samotných zpráv probíha **zleva doprava**. Prioritní vyhodnocování však lze ovlivnit závorkami

- `a + b + 42` - sémantika uvádí, že se nejdříve sečte hodnoty proměnných `a, b` a následně přičte číslo 42.
- `a + (b + 42)` - sémantika nyní uvádí, že se sečte hodnota proměnné `b` s číslem 42 a poté se přičte hodnota proměnné `a`.

2.3.5 Místo

Každé místo má unikátní jméno a dle definice může obsahovat značky, tzv. *tokeny*. Mimo to může obsahovat i *počáteční značení*.

Počáteční značení může v PNtalk rozšíření obsahovat i proměnnou. Počáteční vyhodnocení proměnné řeší tzv. *počáteční akce*. Každé místo má svůj seznam proměnných. Tyto proměnné jsou dostupné pro počáteční značení a akci.

Syntaktický zápis textové reprezentace je následující:

```
place jmémo_místa ( [multimnožina] ) [počáteční_akce]
```

2.3.6 Přechod

Stejně jako místo, tak i každá přechod má své unikátní jméno. Dle definice obsahuje hrany, stráž a akci přechodu. Každý přechod má svůj seznam proměnných, které se navazují pomocí hran. Rozsah platnosti těchto proměnných je pouze lokální. Tedy jsou dostupné pro stráž, akce a výrazy daného přechodu. Přechod má zápis:

```
trans jméno_přechodu [testovací_podmínka] [vstupní_podmínka] [stráž]
                     [akce] [výstupní_podmínka]
```

Hrany

Každá hrana obsahuje hranový výraz, tzv. multimnožinu popsanou v sekci 2.3.3.
Hrany se v jazyce PNtalk dělí na 3 typy:

1. Vstupní hrana - v grafické reprezentaci znázorněna šipkou směřující z místa do přechodu, v textové reprezentaci zapisovaná pomocí klíčového slova `precond`. Popisuje značky, které se mají odebrat z příslušných míst. Získává referenci na objekty, které může ukládat do proměnných.

2. Výstupní hrana - znázorněna šipkou směřující z přechodu do místa a textově je zapsána pomocí klíčového slova `textpostcond`. Specifikuje značky, které se mají umístit do příslušných míst.

3. Testovací hrana - v grafické reprezentaci je znázorněna oboustranou šipkou, v textové reprezentaci je zapsána pomocí klíčového slova `cond`. Testuje přítomnost značek v místě a získává referenci na objekt, ale neodebírá značky z místa.

Stráž přechodu

Stráž přechodu je tvořena sekvencemi výrazů, které jsou od sebe odděleny tečkou.

```
výraz1. výraz2. výraz3 ... výrazn
```

Jednotlivé výrazy mají, dle definice, mezi sebou logickou operaci konjunkce. To znamená, že aby byla celá sekvence platná, tak musí platit každy výraz z dané sekvence.

Příklad:

```
objekt stav: x. x > 42
```

Tuto stráž tvoří 2 výrazy. První výraz `objekt stav: x` obsahuje zprávu s klíčovými slovy, jejíž adresátem je objekt. Tento výraz je platný v případě, že objekt má definovaný synchronní port `stav` a provedení je úspěšné. Druhý výraz obsahuje binární zprávu, jejímž adresátem je `x`.

Akce přechodu

Rozšíření PNtalk oproti definici umožňuje, aby stejně jako ve stráži přechodu podporovala akce sekvenci výrazů. Navíc oproti stráži rozšiřuje výraz o možnost *přiřazení*. Takto rozšířený výraz má tvar:

```
proměnná := výraz
```

Příkladem může být:

```
y := 42
```

```
y := x + 3 + 2 + 1
```

2.3.7 Synchronní port

Z definice vychází, že synchronní porty slouží jako přechod mezi objekty. Synchronní port se zapisuje:

```
sync vzor_zpravy [testovaci_podminka] [vstupni_podminka] [straz]
```

Podobně jako *metoda* nebo *konstruktor* obsahuje vzor zprávy na kterou reaguje. Následují podmínky a stráž, jenž byly popsány v přechodu viz 2.3.6.

Synchronní porty tedy slouží k otestování stavu jiného objektu, navázání na proměnné či změně stavu míst. Speciálním synchronním portem je tzv. *predikát*, který obsahuje pouze testovací hranu, takže nemůže ovlivnit stav.

2.3.8 Příklad OOPN

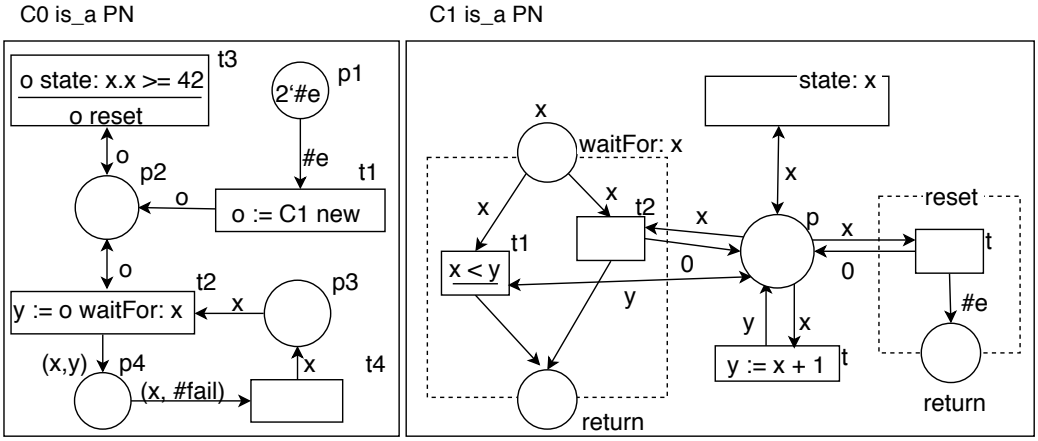
Jako příklad OOPN uvedeme příklad inspirovaný z disertační práce [3] na kterém budou demonstrovány jednotlivé komponenty OOPN. Grafický zápis modelu je na obr. 2.2. Textový zápis pomocí jazyka PNtalk je znázorněn na obr. 2.3.

Model obsahuje 2 třídy. První třída `C0` je *počáteční třídou*, tudíž se provádění začíná u ní. Přechod `t1` se provede pouze splní-li se *vstupní podmínka*. V případě, že místo `p1` obsahuje značku `#e`, vytvoří se v *akci* nová instance třídy `C1`, aktivuje se její *objektová síť* a uloží pomocí *výstupní podmínky* do místa `p2`. Tato objektová síť postupně inkrementuje hodnotu značky v místě `p`.

V druhém přechodu `t2` se po kontrole proveditelnosti přechodu pomocí vstupní a testovací hrany zavolá *metoda s parametrem* třídy `C1 - waitFor: x`. Metoda obsahuje, dle definice povinné, místo `return` a místo `x`. *Testovací podmínka* kontroluje přítomnost značky v místě objektové sítě `p`, které je pro *metody* a *synchronní porty* přístupné. Vstupní podmínkou se do proměnné x naváže hodnota, která byla zaslána jako argument pomocí parametrizovaného místa. Pokud je hodnota x menší než hodnota místa `p`, tedy hodnota uložená pomocí testovací podmínky do proměnné y, tak se do výstupního místa `return` uloží symbol `#fail`, čímž se metoda ukončí. V opačném případě se přechod neprovede. Přechod `t2` načte hodnoty z míst a vrací symbol `#success` do výstupního míst a nulovou hodnotu do místa `p`.

Třetí přechod `t3` obsahuje i volání synchronního portu `state: x` třídy `C1`. Jedná se o *predikát*, tedy synchronní port obsahující pouze *testovací hranu*, kterému je zaslána nenavázaná proměnná. Synchronní port vrátí značku z místa `p`, která je následně porovnána `>= 42`. Jsou-li výrazy ve stráži pravdivé, zavolá se metoda `reset` třídy `C1`, která do místa `p` vloží hodnotu 0.

Poslední přechod kontroluje vstupní podmínkou dvojici značek v místě `p4`. Výstupní podmínkou vkládá značku uloženou v proměnné `x` z místa `p3` do místa `p4`.



Obrázek 2.2: Grafický zápis OOPN.

```
main C0

class C0 is_a PN
  object
    place p1(2‘#e)
    place p2()
    place p3()
    place p4(1, 2)

    trans t1
      precond p1(#e)
      action {o := C1 new.}
      postcond p2(o)
    trans t2
      cond p2(o)
      precond p3(x)
      action {y := o waitFor: x}
      postcond p4((x, y))
    trans t3
      cond p2(o)
      guard {o state: x. x >= 42}
      action {o reset.}
    trans t4
      precond p4((x, #fail))
      postcond p3(x)
```

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

Obrázek 2.3: Zdrojový kód modelu.

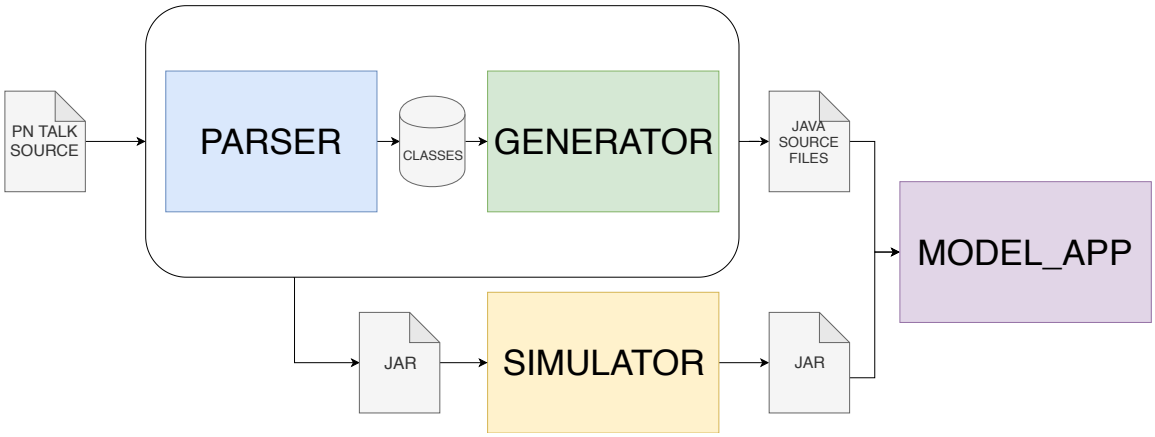
Kapitola 3

Návrh a datové struktry

Celková funkcionalita je rozdělená do 3 aplikací.

1. Překladač - přeloží model z jazyka PNtalk do zdrojových souborů Java.
2. Simulátor - knihovna ke spuštění modelu.
3. Spouštěcí aplikace - aplikace, která provede simulaci.

Struktura souborů a aplikací je popsaná v příloze C. Překladač vygeneruje zdrojové soubory modelu, simulátor vytvoří knihovnu a poté se obojí vloží do spouštěcí aplikace, která přeloží zdrojové soubory a spuštěním aplikace provede simulaci modelu. Jednotlivé kroky včetně vstupů a výstupů jednotlivých částí jsou znázorněny na obrázku 3.1. Pro modelování diagramů je používán jazyk UML.



Obrázek 3.1: Model mechanismu transformace a simulace.

Mechanismus transformace je navržen tak, že zdrojové texty v jazyce `PNtalk` slouží jako vstup analyzátoru, tzv. `Parser`. Parser daný vstup zpracuje a výstupem je struktura tříd, reprezentující model. Takto vytvořená struktura je vstupem pro `Generator`, který vytvoří odpovídající zdrojové soubory v jazyce Java. Tyto soubory tvoří model, který obsahuje potřebná data, ale neumí se provést. Z toho důvodu je implementovaný `Simulator`, který obsahuje logiku pro veškeré nutné operace. V následujících sekcích budou popsány návrhy, implementace, problémy, omezení a případné rozšíření jednotlivých bloků.

3.1 Struktura tříd

V této sekci bude popsán návrh a řešení struktury tříd, jenž slouží pro uložení dat o modelu. Při návrhu struktury tříd vycházím z definice jazyka PNtalk popsaného v kapitole 2.3. Třídy jsou navrženy jako tzv. `Value Object`, což znamená, že obsahují pouze hodnoty a operace spojené s jejich manipulací. Pro práci s touto strukturou slouží simulátor popsaný v sekci 5. Třídy *PNClazz* a *IPNNet*, tedy třída a síť jsou navržené výhradně pro generátor. Simulátor je nahrazuje vygenerovanými zdrojovými soubory dědící některé ze třídy *PNClass* nebo *PNMethod* popsáné dále.

3.1.1 Třída

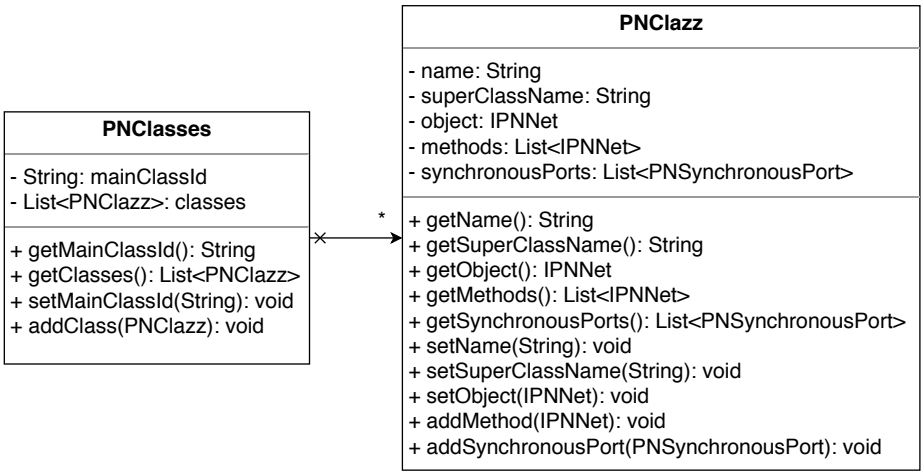
Třída slouží pro uložení dat o Petriho sítích na základě kterých se generují zdrojové soubory. Třída, dle definice v 2.3.1, může obsahovat *Objektovou síť*, *Síť metod*, *Konstruktory* a *Synchronní porty*. Konstruktory jsem v implementaci vynechal, jelikož je lze nahradit metodami s ekvivalentní funkcionalitou. Třída mimo jiné obsahuje atributy:

- *name* - obsahuje informaci o názvu třídy, jenž se bude generovat.
- *superClassName* - název rodičovské třídy z které bude třída dědit.

Z definice dále vyplývá, že takových tříd může být více. Z toho důvodu je implementována třída *PNClasses*, jenž obsahuje atribut typu list, v němž jsou uloženy všechny třídy, které se mají generovat. Mimo to obsahuje atribut jména počáteční třídy.

Třídy obsahují metody *get*, *set* a *add* pro práci s atributy a metodu *toString* pro ladící účely.

Schéma tříd je znázorněno na obrázku 3.2.



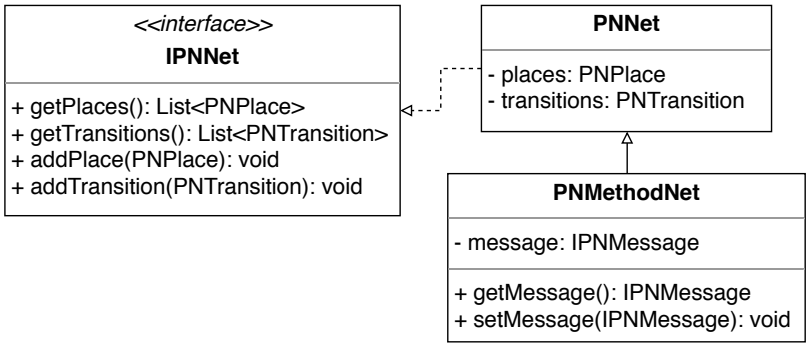
Obrázek 3.2: Model třídy.

3.1.2 Síť

V implementaci je síť rozdělena na dva typy. První je síť pro překladač, ve které jsou uloženy data z kterých se generuje soubor. Soubor musí dědit jinou síť, která definuje metody pro práci při simulování.

Síť pro překladač

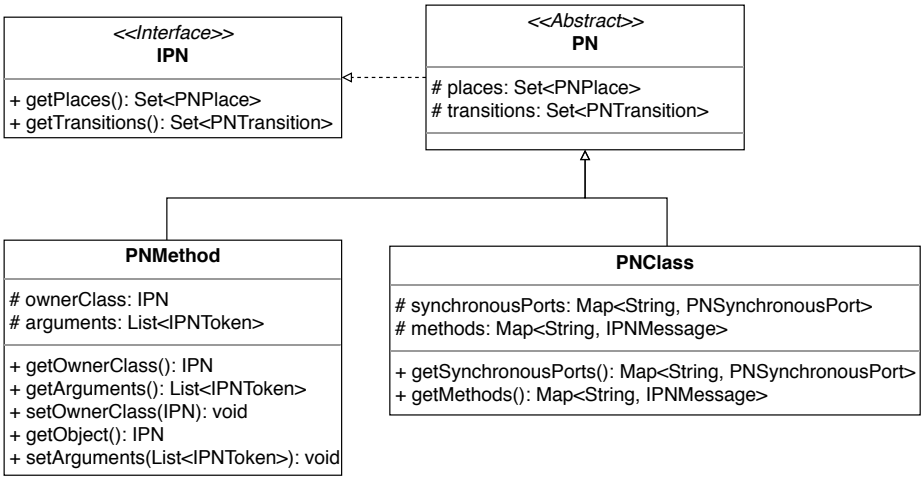
Definuje atributy pro místa, přechody a metody pro práci s nimi. Základní síť je implementovaná ve třídě `PNNet`. Pro síť metod je implementovaná další třída `PNMethodNet`, která dědí z třídy `PNNet` a přidává atribut zprávy, na kterou reaguje. Diagram je znázorněn na obrázku 3.3.



Obrázek 3.3: Model sítě pro překladač.

Síť pro simulátor

Mezi generované soubory patří buď Petriho síť, která dědí třídu `PNClass` a nebo jeho metody, dědící `PNMethod`. V případě Petriho sítě musí definovat mapy synchronních portů a metod. Synchronní port obsahuje v klíči selektor, podle kterého se vyhledává. Metoda má v klíči také selektor pro snažší vyhledání a v hodnotě obsahuje zprávu, na kterou má reagovat. Obě třídy jsou potomkem `PN`, která definuje atributy a metody pro místa a přechody. Při generování se tyto kolekce naplňují v konstruktoru. Diagram je znázorněn na obrázku 3.4.



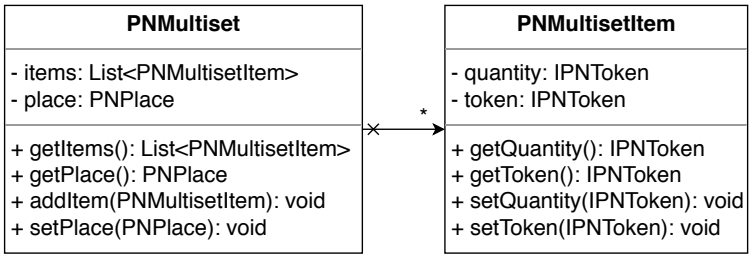
Obrázek 3.4: Model sítě pro simulátor.

3.1.3 Multimnožina

Multimnožiny definují, jak se stav místa změní. Změna se uskuteční přidáním či odebráním určitého počtu tokenů z místa. V implementovaná je reprezentována pomocí třídy *Multiset*, která uchovává atributy:

- Instanci místa, které ovlivňuje.
- List *MultisetItem*, tzv. položek, které obsahují kvantitu s prvkem, které se mají přidat nebo odebrat.

Jako atribut může být obsažena v místech pro inicializační akci nebo na hranách přechodu. Schéma diagramu tříd je zobrazeno na obrázku 3.5.

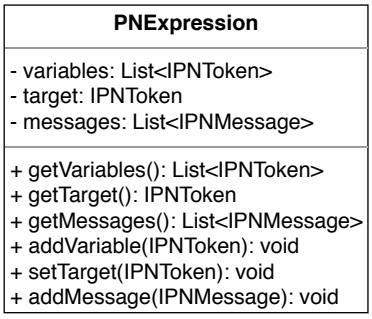


Obrázek 3.5: Model multimnožiny.

3.1.4 Výraz

V jazyce PNtalk (2.3) jsou definované výrazy, které jsou obsaženy v akcích a strážích přechodu. Výraz obsahuje cílový operand, čili *Token*, kterému se posílají zprávy.

Třídní diagram výrazu je na obrázku 3.6.



Obrázek 3.6: Model výrazu.

Cílem zaslání zpráv je *Token*, tedy *Term*. Zpráva může být unární, binární nebo klíčová (viz 2.3.4). Dle definice může výraz obsahovat složené zprávy. V rámci implementace jsem vyzkoušel tuto možnost implementovat. Tato varianta fungovala, ale ve výsledku byla příliš komplikovaná. Rozhodl jsem se implementovat variantu, kdy jsem takto složenou zprávu rozdělil na jednoduché zprávy. Provádění zjednodušených zpráv bude popsané v kapitole o simulování viz 5.

Term

Term, nebo-li značky slouží pro uchování informace, definovaný viz. 2.3.4. V implementaci je navržen jako abstraktní třída `PNToken`, která obsahuje atribut *value* typu *Object*. Rozhraní `IPNToken` definuje metody *getValue* a *setValue* pro práci s uchovanou hodnotou. Potomci dědí tyto metody a upravují datové typy *value* dle svých potřeb. Momentálně jsou implementovány tokeny pro práci s primitivními datovými typy, mezi které patří `PNTokenInteger`, `PNTokenBoolean`, `PNTokenCharacter`, `PNTokenSymbol`. Pro práci s proměnnou slouží `PNTokenVariable`, který uchovává pouze název proměnné, odkazující do tabulky proměnných daného přechodu, viz kapitola 3.1.6. Výjimkou je *TokenPnObject*, který v atributu ukládá název třídy a přidává atribut *object*, ve kterém si uchovává samotnou instanci vygenerované třídy typu *IPN*, jenž reprezentuje Petriho síť. Seznamy ze Smalltalku nejsou podporované.

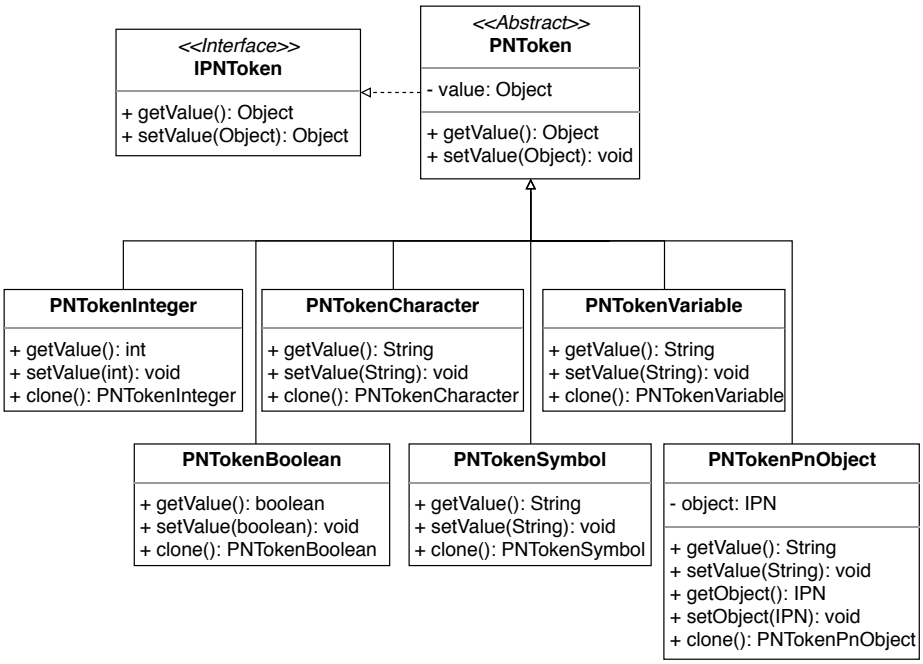
Tokeny implementují metody `toString`, `clone` a `equals`, definované v jazyce Java.

- toString - slouží pro snazší ladění aplikace a vypsání hodnot.
- clone - tokeny se musí umět kopírovat z důvodu návrhu multimnožin, viz kapitola 3.1.3. Vytvoří se nová instance stejného tokenu s atributem.
- equals - zkontroluje, zda nejsou objekty identické, typ třídy a atribut.

Z definice termu vyplývá, že každý musí umět reagovat na zprávy. Vzhledem k návrhu celé struktury tříd jako `Value Object` není tato funkcionalita implementována přímo v `PNToken`, ale v simulátoru popsaném v kapitole 5.

V implementaci je *Token* využíván pro uložení hodnoty nejen v místech, ale i multimnožině, tabulce proměnných přechodu a argumentech metod či synchronních portů.

Diagram tříd značek je znázorněn na obrázku 3.7.



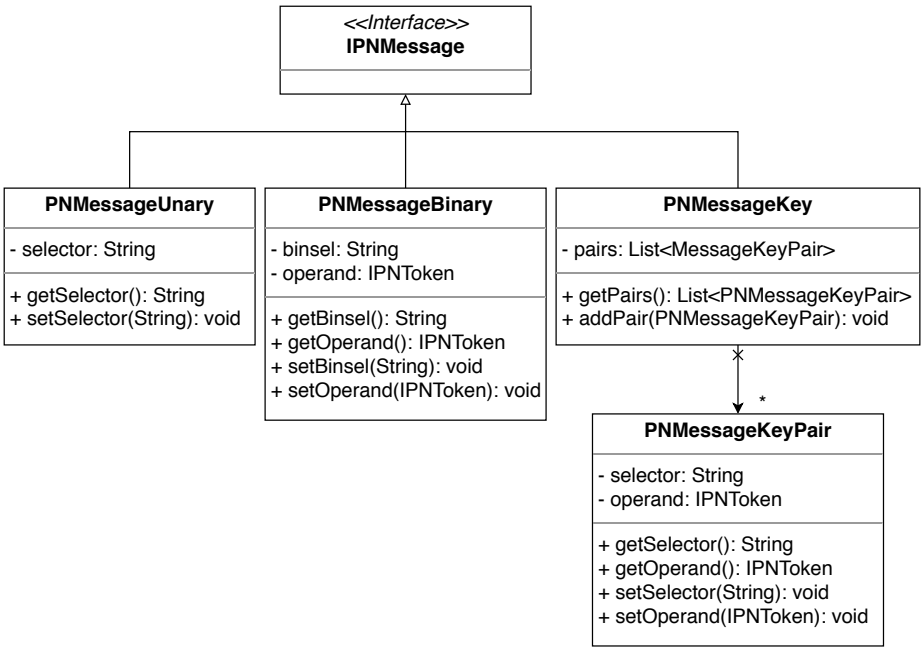
Obrázek 3.7: Model značek.

Zpráva

Zprávy jsou implementované pomocí rozhraní *IPNMessage*. Každý typ zprávy má vlastní třídu, která obsahuje atributy.

- Unární zpráva - pouze atribut *selector*. Selectorem může být například `print`, které rozumí každý token.
- Binární zpráva - atributy *binsel* a *operand*.
- Klíčová zpráva - list klíčových páru, tzv. PNMessageKeyPair. Každý pár obsahuje *selector* a *operand*.

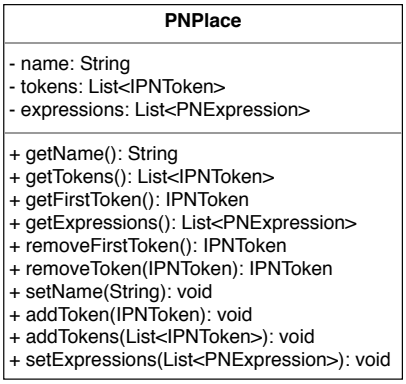
Každý typ zprávy obsahuje metody k manipulaci s atributy a metodu *toString* pro kvalitnější vypisování při ladění aplikace.



Obrázek 3.8: Model zpráv.

3.1.5 Místo

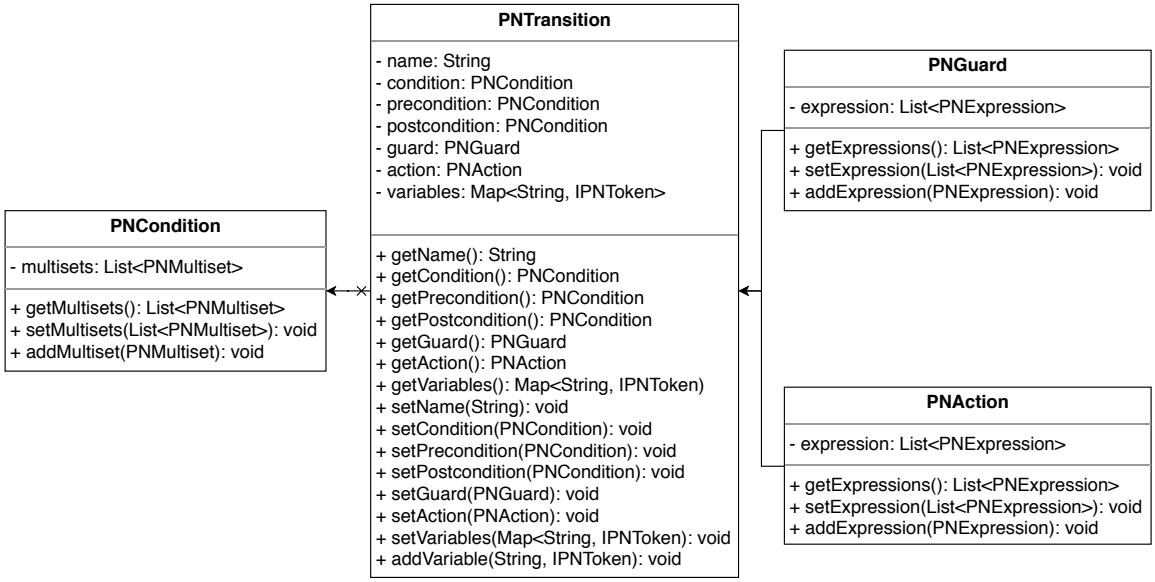
Místa, dle definice v kapitole 2.3.5, představují parciální stav. Kromě značek mohou také obsahovat inicializační akci, která se provede hned na začátku. Tím může být inicializace proměnné v místě. Implementace místa má tři atributy - jméno, list značek a list výrazů. V případě inicializační akce se při generování takového místa vytváří značky, které se do místa vkládají. Implementace tedy podporuje pouze inicializaci proměnné. Diagram místa je znázorněn na obrázku 3.9.



Obrázek 3.9: Model místa.

3.1.6 Přechod

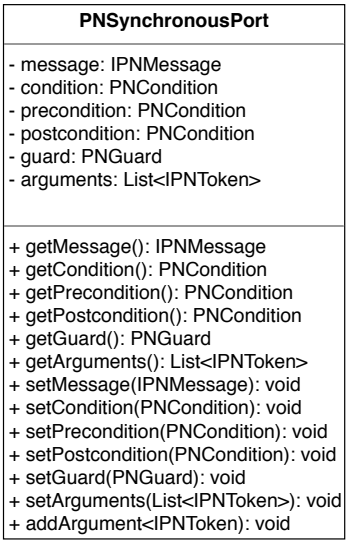
Z definice *PNtalk* v kapitole 2.3.6 vychází, že přechod může obsahovat hrany, stráž a akci. V implementaci je přechod navržen tak, že každá hrana je uložena v samostatném atributu. Hrany se liší pouze změnou stavu, kterou představují. Z toho důvodu používají všechny podmínky stejnou třídu `Condition` a odlišná práce s podmínkou se řeší až v simulátoru. Stráž i akce přechodu mohou obsahovat výrazy, které byly popsané v kapitole 3.1.4. Diagram tříd je znázorněn na obrázku 3.10.



Obrázek 3.10: Model přechodu.

3.1.7 Synchronní port

Synchronní port je něco mezi přechodem a metodou. Tomu odpovídá i návrh jeho třídy, který neobsahuje akci, ale zprávu na kterou reaguje.



Obrázek 3.11: Model synchronního portu.

3.1.8 Konstruktor

Vzhledem k implementaci, kdy se funkcionalita konstruktoru nahradila metodami, které je dokážou plně nahradit, nejsou konstruktory implementovány.

Kapitola 4

Překladač

V této části bude představena teorie a implementační část překladače, metodika překladu a styl generovaných tříd. První část překladače je implementována pomocí třídy *PNParser*, která spoluprace se třídou *PNScanner*. Výstupem je vytvoření struktury tříd reprezentující model, která slouží pro vstup druhé fáze překladu - generátoru, implementovaného ve třídě *PNGenerator*. Reprezentace dat při překladu, které se naplňují a generují využívají stejnou strukturu tříd. Výstupem jsou vygenerované zdrojové soubory v balíčku modelu.

4.1 Teorie

Při studiu překladačů jsem čerpal z opory předmětu *Formální jazyky a překladače* [1] a knihy *Automata and languages : theory and applications* [4]. Z opory předmětu, která čerpá z této knihy vyplývá, že:

Kompilátor je program, který na vstupu přijímá zdrojový program zapsaný ve zdrojovém jazyce, aby k němu na výstupu generoval funkčně ekvivalentní cílový program v cílovém jazyce. Transformace, kterou kompilátor provádí, se nazývá překlad. Z definice vyplývá, že kompilátor se skládá z:

1. Lexikální analýza - zpracovává zdrojový program a rozděluje jej na jednotlivé lexémy, tedy lexikální symboly, což jsou elementy jazyka. Tím může být identifikátor, klíčové slovo, čísla aj.

2. Syntaktická analýza - úkolem syntaktické analýzy je zjistit, zda je zdrojový program syntakticky správně zapsán, tzn. zda řetěz lexikálních symbolů patří do zdrojového jazyka. Pokud tomu tak skutečně je, pak výstupem syntaktického analyzátoru je tzv. derivační strom, který reprezentuje syntaktickou strukturu zdrojového programu.

3. Sémantická analýza - provádí kontrolu nejrůznějších sémantických aspektů programu, jako např. deklarovanost proměnných.

4. Generování vnitřní reprezentace programu - ze syntaktického stromu se nejprve vygeneruje vnitřní forma, která umožňuje provést optimalizaci.

5. Optimalizace

6. Generování cílového programu - vygenerování funkčního ekvivalentního programu v cílovém jazyce.

4.2 Lexikální a syntaktická analýza

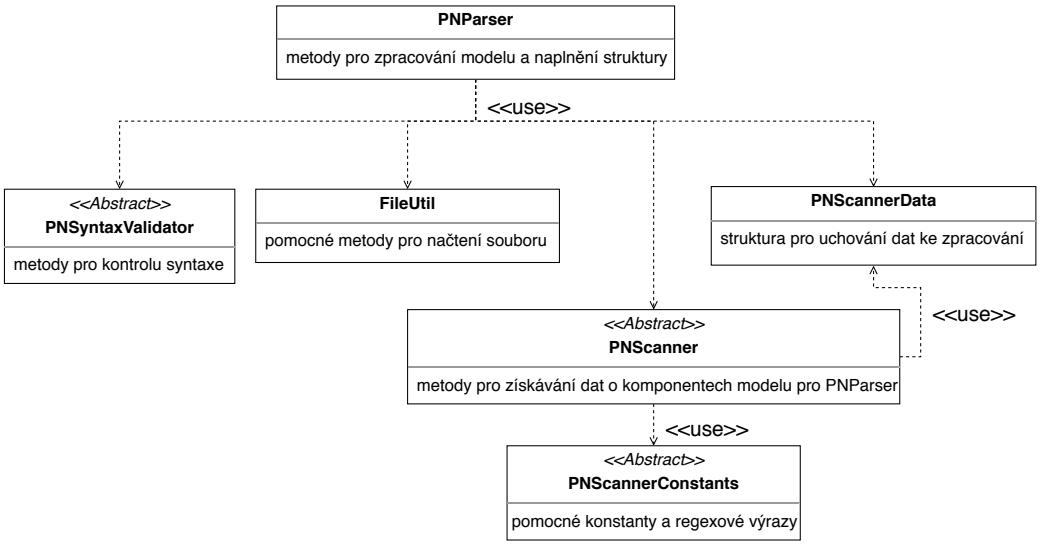
O přípravu dat pro generátor se stará lexikální analýza, dále *Scanner* a syntaktický, tzv. *Parser*. Celý překlad řídí Parser, implementovaný v třídě `PNParser` pomocí návrhového vzoru *Singleton* [2]. Spouštěcí metodou pro překlad je `run`, která očekává na vstupu cestu k souboru nebo případně metoda `parseClasses`, která už očekává uložený obsah modelu v řetězci. K načtení souboru do řetězci slouží pomocná singleton třída `FileUtil`, která implementuje metodu `getFileAsString`. Parser volá pomocné abstraktní třídy `PNScanner`, `PNScannerConstants` a `PNSyntaxValidator`. Postupným zanořováním metod se zpracovává model jazyka PNtalk do struktury tříd z kapitoly 3.1. Výstupem analyzátoru je hlavní třída `PNClasses` popsaná v kapitole 3.1.1.

`PNScanner` poskytuje metody pro získávání dat. Statické metody očekávájí na vstupu obsah modelu, který následně pomocí regexu a vyhledávání v řetězci zpracují na jednotlivé komponenty. Jako návratovou hodnotu využívá třídu `PNScannerData`, která obsahuje upravený vstupní model v atributu `content` a získaná data v seznamu `data`.

`PNScannerConstants` uchovává *regexy*, které se používají pro zpracování modelu v lexikálním analyzátoru a klíčové slova, definované v jazyku pntalk, viz příloha A.

`PNSyntaxValidator` slouží pro syntaktickou kontrolu elementů modelu. Pro zpracování využívá tzv. *regexy* a pomocné metody.

Pomocí metod ve třídě *PNScanner* získává aktuálně potřebná data ke zpracování. Diagram tříd je znázorněn na obr. 4.1. Detailní modely tříd jsou znázorněny v přílohách pro lexikální analyzátor B.1 a syntaktický analyzátor B.2.



Obrázek 4.1: Model tříd lexikálního a syntaktického analyzátoru.

4.2.1 Příklad analyzátoru

Příklad zpracování se syntaktickou kontrolou je znázorněn na zpracování tříd pomocí třídy `parseClasses`. Množina tříd, dle definice v kapitole 2.3.1 má definovanou jednu jako počáteční, tzv. *main*. Metoda očekává na vstupu celkový model, kde jsou třídy a počáteční označení třídy. Třída pro pomocné výpisy `LoggingUtil` je popsaná v kapitole 6.

Pro demonstraci překladu bude sloužit vstup zobrazený na obr. 4.2.

```
main TestovaciTrida1

class TestovaciTrida1 is_a PN
class TestovaciTrida2 is_a PN
```

Obrázek 4.2: Zdrojový kód modelu pro demonstraci překladu.

Řetězec s takto načteným modelem slouží jako vstup pro metodu *parseClasses* znázorněnou na obrázku 4.3.

```
public PNClasses parseClasses(String content)
{
        LoggingUtil.trace("[T]: PNParser: parseClasses()");

        PNScannerData mainData = PNScanner.getMainData(content);

        PNClasses classes = parseMain(mainData.getData().get(0));

        content = mainData.getContent();

        PNScannerData classesData = PNScanner.getClassesData(content);

        for (String clazzData : classesData.getData())
        {
                PNClazz clazz = parseClass(clazzData);

                classes.addClass(clazz);
        }

        LoggingUtil.debug("[D]: Classes: %s", classes.toString());

        return classes;
}
```

Obrázek 4.3: Kód zpracování tříd.

Metoda nejprve získá data o počáteční třídě. Výstupem metody `getMainData` je struktura, která má v atributech:

- content - model s odstraněnou počáteční třídou, tedy pouze `class TestovaciTrida1 is_a PN` a `class TestovaciTrida2 is_a PN`
- data - list výsledků dat. V případě počáteční třídy, která je pouze jedna obsahuje list pouze jeden záznam.

Získaná data jsou vstupem pro zpracování informace o počáteční třídě, která zároveň vytvoří instanci třídy `PNClasses`. Model v řětězci `content` se musí aktualizovat, jelikož už

nesmí obsahovat informaci o *main*. Stejným principem se získají data o třídách metodou `getClassesData`, která naplní strukturu `PNScannerData`:

- content - prázdný řetězec modelu, jelikož nic jiného v této fázi překladu nezbývá.
- data - list 2 řetězců, kde jsou uloženy nové vstupy modelů pro další metodu.

Takto získané modely jednotlivých tříd slouží jako vstup další metodě `parseClass`, která je implementována stejným principem.

Pro demonstraci `PNSyntaxValidator` představím zmíněnou metodu `parseMain` na obr. 4.4.

```
public PNClasses parseMain(String content)
{
        LoggingUtil.trace("[T]: PNParser: parseMain()");

        PNClasses classes = new PNClasses();

        String token = PNScanner.getTokenFromContent(content);

        if (!PNSyntaxValidator.isId(token))
        {
                throw new SyntaxException("Invalid main class id!");
        }

        LoggingUtil.debug("[D]: Main class is: %s", token);

        classes.setMainClassId(token);

        return classes;
}
```

Obrázek 4.4: Kód zpracování počáteční třídy.

Implementace metody `isID` třídy `PNSyntaxValidator` je znázorněna na obr. 4.5.

```
public static boolean isId(String token)
{
        LoggingUtil.trace("[T]: PNSyntaxValidator: isId(%s)", token);
        return token.matches(ID_REGEX);
}
```

Obrázek 4.5: Kód pro syntaktickou kontrolu identifikátoru.

Kde `ID_REGEX` odpovídá regexu:

`[a-zA-Z][a-zA-Z0-9]*`

Stejným principem je procházen, kontrolován a zpracován i zbytek modelu.

4.3 Generátor

V druhé části překladače, *Generátoru*, budou popsány možnosti generování, návrh implementovaných tříd, struktura vytvořených souborů a metodika generování znázorněná na příkladu.
Pro druhou část překladače - generování cílového programu - existují dvě možnosti.

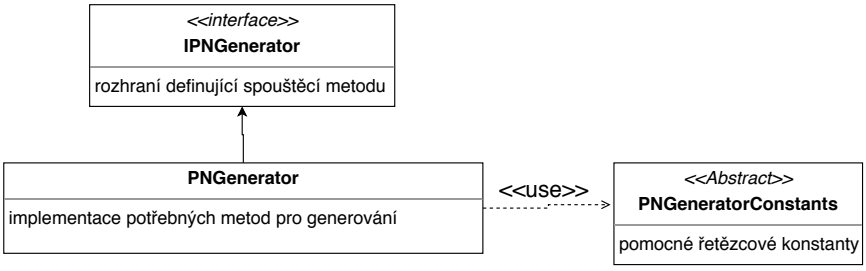
1. Source code - tzv. *zdrojový kód*. Jedná se o zdrojový kód s příponou `.java`. Takto vygenerovaný kód je čitelný a upravitelný, ale musí se před spuštěním přeložit.

2. Bytecode - již přeložený zdrojový kód s příponou `.class`. Výhodou bytecode je to, že už se nemusí překládat. Nevýhodou je, že je zcela nečitelný a pro modifikaci takto vygenerovaných tříd je zapotřebí využití reflexe, kterou jazyk Java umožňuje.

Vyzkoušeny byly obě varianty, kde v případě bytecode byla použita knihovna *Byte Buddy* [8]. U druhé varianty zdrojového kódu byla využita knihovna *JavaPoet* [6]. Vzhledem k cílu bakalářské práce, která by měla umožnit rozšíření pro modifikaci vygenerovaných modelů, byla vybrána varianta samotných zdrojových kódu, tedy varianta č.2.

4.3.1 Návrh generátoru

Generátor je implementovaný v třídě `PNGenerator`, který implementuje rozhraní `IPNGenerator` definující spouštěcí metodu generování. V atributu třídy `packageName` se při vytvoření nové instance uloží cesta balíčku, kam se budou ukládat vytvořené soubory. Vstupem metody `generateClasses` je vytvořená struktura tříd definovaná v kapitole 3.1.1 naplněná analyzátorem v předchozí kapitole 4.2. Pro generování je použita knihovna *JavaPoet* [6] a návrhový model *Builder* [2]. Musí se vytvořit třída a konstruktor, který obsahuje incializaci sítě. Vytvoření kostry třídy je implementované ve třídě `buildClassBuilderInit` a konstruktoru v `buildConstructorBuilderInit`. Postupným procházením struktury tříd se sestavuje `constructorBuilder`, který se na konci sestaví pomocí metody `build`. Poté se sestaví samotná třída a zapíše do souboru. Pro generování komponentů je využívaná abstraktní třída `PNGeneratorConstants`, která obsahuje často používané konstanty. Modely obou tříd jsou znázorněny na obrázcích 4.6. Detailní diagram tříd je znázorněn v příloze B.3.



Obrázek 4.6: Model tříd generátoru.

Generované soubory

1. třídy Petriho sítí - z návrhu třídy v kapitole 3.1.1 je vidět, že každá třída musí mít v hierarchii dědičnosti abstraktní třídu `PNClass`, v níž jsou definované struktury pro metody a synchronní porty.

2. metody Petriho sítí - obsahují místa a přechody, tudíž se generuje samostatný soubor. V hodnotě mapy je uložena zpráva na kterou musí síť reagovat. V případě invokace se z názvu metody a balíčku sestaví cesta k souboru. Metody dědí z třídy `PNMethod`, která přidává dva atributy. Prvním je třída nadtřídy a druhým je list argumentů.

4.3.2 Příklad generování

V následující ukázce bude představena práce s *Builder*, generování místa do konstruktoru a následné vytvoření souboru.

Inicializace třídy je implementovaná v metodě `buildClassBuilderInit`, která je znázorněná na obr. 4.7.

```
private TypeSpec.Builder buildClassBuilderInit(String className, Class<?>
    superClassType)
{
        LoggingUtil.trace("[T]: PNGenerator : buildClassBuilderInit(%s, %s)"
            , className, superClassType);

        return TypeSpec.classBuilder(className)
                .superclass(superClassType)
                .addJavadoc("Generated Petri Net Java file")
            .addModifiers(Modifier.PUBLIC, Modifier.FINAL);
}
```

Obrázek 4.7: Kód pro inicializaci Builder třídy.

V případě, že jméno třídy `className` je *TestovaciTrida* a třída, kterou dědí je *PNClass* bude výsledkem hlavička na obr. 4.8.

```
/**
* Generated Petri Net Java file */
public final class TestovaciTrida extends PNClass
```

Obrázek 4.8: Kód vygenerované hlavičky testovací třídy.

V případě konstruktoru se místo `TypeSpec.Builder` používá `MethodSpec.Builder`, ale princip je jinak stejný. Příklad práce se samotným skládáním *Builder* je znázorněn na generování místa, které má 2 značky - číslo `6` a symbol `#e`.

Pro jednoduchost budeme předpokládat, že místo nemá inicializační akci a v demonstraci kódu ji vynechám. Metoda pro generování místa je `buildPlace`. Na vstupu očekává `Builder` pro konstruktor a místo k vygenerování. Zapíše se komentář pomocí metody `addComment` pro lepší čitelnost zdrojového souboru. Název proměnné všech komponentů se skládá z konstanty a haše, tzv. *hash*. V případě místa to je `Place` z konstanty `PNGeneratorConstants.PLACE`
a *hashe* generovaného místa, tedy `place.hashCode`. Tímto je zaručená unikátnost názvu

proměnné. Dále se zapíše inicializace místa a jeho jména pomocí metody `addStatement`. Nyní zbývá vygenerovat vložení tokenů do místa. Princip generování vnořených komponentů spočívá v získání jména proměnné, která se bude generovat, zavolání metody pro generování samotného komponentu a vygenerování vložení. V případě značek u místa se pro každou značku sestaví jméno pomocí konstanty a *hashe*, zavolá metoda `buildToken` a zapíše vložení tokena do místa. Kód metody je znázorněn na obrázku 4.9. Princip generování je u všech metod stejný.

```
public void buildPlace(MethodSpec.Builder constructorBuilder,PNPlace place)
{
        LoggingUtil.trace("[T]: PNGenerator : buildPlace(%s)", place);

        constructorBuilder.addComment("Place: $N", place.getName());

        String placeName = PNGeneratorConstants.PLACE + place.hashCode();

        LoggingUtil.debug("[D]: Generated Place name: %s", placeName);

        constructorBuilder.addStatement("$T $N = $N $T()", PNPlace.class,
            placeName, PNGeneratorConstants.NEW, PNPlace.class);
        constructorBuilder.addStatement("$N.$N($S)", placeName,
            PNGeneratorConstants.SET_NAME, place.getName());

        for (IPNToken token : place.getTokens())
        {
                String tokenName = PNGeneratorConstants.TOKEN + token.
                    hashCode();
                buildToken(constructorBuilder, token);
                constructorBuilder.addStatement("$N.$N($N)", placeName,
                    PNGeneratorConstants.ADD_TOKEN, tokenName);
        }

        LoggingUtil.debug("[D]: Place '%s' generated.", placeName);
}
```

Obrázek 4.9: Kód pro sestavení místa pro generování.

Takto naplněný `Builder` se sestaví, přiloží do třídy, které se také sestaví a následně zapíše do souboru. Kód je znázorněn na obr. 4.10.

```
classBuilder.addMethod(constructorBuilder.build());
TypeSpec classToGenerate = classBuilder.build();
JavaFile javaFile = JavaFile.builder(packageName, classToGenerate).build();
File file = new File("generated");
javaFile.writeTo(file);
```

Obrázek 4.10: Kód pro sestavení a zapsání třídy do souboru.

Název balíčku ve kterém je model uložený je implicitně odvozeno od názvu počáteční třídy. Aplikace podporuje nepovinné zadání názvu balíčku. Pro jednoduchost je tu znázorněna první varianta s počáteční třídou. Výsledkem generování je tedy třída *TestovaciTrida* uložená v balíčku *model.TestovaciTrida*, která dědí třídu *PNClass*. Příklad vygenerované třídy je zobrazen na obr. 4.11.

```
package model.TestovaciTrida

/* import potrebnych trid */

/**
* Generated Petri Net Java file */
public final class TestovaciTrida extends PNClass {
	public TestovaciTrida() {
		super();
		// Place: p
		PNPlace place380936215 = new PNPlace();
		place380936215.setName("p");
		IPNToken token142638629 = new PNTokenInteger(6);
		place380936215.addToken(token142638629);
		IPNToken token14263243 = new PNTokenSymbol(e );
		place380936215.addToken(token14263243);
		this.places.add(place380936215);
	}
}
```

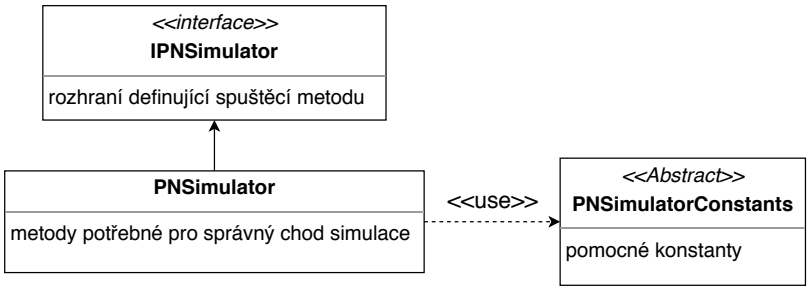
Obrázek 4.11: Kód vygenerované testovací třídy.

Princip generování ostatních komponentů zůstává stejný.

Kapitola 5

Simulátor

V této kapitole bude popsán návrh simulátoru, inicializační část, způsob iterování, navazávání proměnných včetně demonstračního příkladu, vyhodnocování proveditelnosti a provedení přechodu včetně volání metod a synchronních portů. Návrh simulátoru je zobrazen na obr. 5.1. Detailní diagram tříd simulátoru je v příloze B.4.



Obrázek 5.1: Model simulátoru.

5.1 Inicializace a iterace

Simulátor je inicializovaný pomocí cesty balíčku, ve kterém je obsažen model. Cesta balíčku může vypadat například: `model.Model1`.

Při spuštění simulace metodou `run(String classId)`, kde *classId* je jméno počáteční třídy. Simulátor si z názvu balíčku a jména počáteční třídy sestaví cestu a pomocí metody `getInstanceOfClassByPath(String classPath)` vytvoření instanci třídy, nad kterou začne iterovat. Taková třída je definovaná (kapitola 3.1.1) s metodami `getTransitions()`, `getPlaces()`, které představují objektovou síť a pomocnou metodu `getPlaceByName()`. Místa jsou implementované pomocí rozhraní typu množina, tzv. `Set`, která je implementována jako `HashSet`. Tato implementace zaručuje neduplikovatelnost záznamů a jejich náhodné seřazení, jelikož se záznamy vkládají podle hashované hodnoty, tzv. *hash code*. Přechody jsou implementované pomocí `ArrayList`.

Iterování nad takovou třídou probíhá v cyklu, dokud není list přechodů prázdný. Při špatně navrženém modelu by mohlo dojít k tzv. zacyklení. Z toho důvodu je implementována pojistka v souboru *config.xml*, kde se do záznamu `maxIterations` zadá hodnota maximálního počtu iterací.

Při každém iterování se volá metoda `runIteration`, která provádí daný krok. Vstupem jsou přechody a výstupem je modifikovaný list přechodů. V každé iteraci se prochází všechny přechody. Iterace je rozdělená na 2 fáze.

1. Test přechodu - metoda `testTransition`.
2. Provedení přechodu - metoda `doTransition`.

Pokud se během iterace nezmění žádný přechod, ukončí se model za již neprůchozí a simulace skončí.

5.2 Test přechodu

V rámci testování přechodu se zjišťuje, zda-li je přechod proveditelný pro nějaké návázání proměnných. V této fázi se testuje podmínka, vstupní podmínka a stráž přechodu. Na hranách podmínek mohou být objekty nebo proměnné, referencující na objekt (objektem se rozumí jak Petriho síť, tak primitivní datový typ).

Vyhodnocování probíhá v následujících krocích:

1. Sestavení všech možných hodnot proměnných a kontrola počtu značek v místech.
2. Kontrola kombinací proměnných a hledání funkčního navázaní. V případě, že se nenaleze žádné takové navázání, označí se přechod jako neproveditelný a pokračuje se v iteraci dalším přechodem. Pokud je navázání úspěšně nalezeno, použije se tato kombinace navázání pro provedení přechodu a změnu stavu.

5.2.1 Sestavení hodnot

Metoda `findAllPossibleBindings` očekává na vstupu testovaný přechod a mapu možných navázání. Tato mapa je navržena jako `Map<String, List<IToken»`, kde v klíčové hodnotě je název proměnné a v hodnotě této proměnné je list všech možných navázání. Taková mapa je implementována pomocí `HashMap` a je poslána jako argument funkce. Tato kolekce se postupně naplní ze vstupních a testovacích hran. Ke zpracování hran je metoda `findBindingsForCondition`, která očekává na vstupu danou podmínku a kolekci všech navázání. Z definice podmínek v kapitole 2.3.6, že podmínky multimnožiny, které definují počet a typ značek, které musí místo obsahovat, aby byl přechod proveditelný. Pro každou multimnožinu se zavolá metoda `findBindingsByPlace`, která tyto značky kontroluje. Každá multimnožina může obsahovat jednu z možností:

- Primitivní typ - v případě, že je v multimnožině primitivní typ popsaný v kapitole 2.3.4, kontroluje se pouze správný počet značek v místě. Pokud místo neobsahuje validní kvantitu, vyhodnotí se přechod jako neproveditelný. V opačném případě se kontrolují další elementy nebo multimnožiny hrany.
- Proměnnou - proměnná může obsahovat referenci na jakýkoliv objekt definovaný petriho sítěmi. Z toho důvodu se musí všechna možná navázání proměnné registrovat v kolekci `possibleBindings`. Proces probíhá ve funkci `updateBindingsForVariable`. Zjistí se, zda je daná proměnná v mapě všech možných navázání. Pokud ano, promažou se všechny stávající navázání dané proměnné, které nesplňují kvantitu hrany. Poté se do seznamu možných navázání přidá nová hodnota. V případě, že proměnná ještě není v mapě proměnných, přidá se i s novou hodnotou navázání.

Tímto způsobem se projdou hrany *precondition* a *condition*. Vytvoří se seznamy všech možných navázání proměnných, které budou sloužit při hledání proveditelné kombinace pro přechod. Pokud metoda *findAllPossibleBindings* vrátila hodnotu `false`, vyhodnotí se přechod jako neproveditelný. V opačném případě se zkontroluje, že mapa všech navázání obsahuje proměnné ke kontrole, hledá se proveditelná kombinace. Jinak se kontrolovaly pouze počty primitivních značek a proměnné nejsou potřebné, proto se vrací hodnota `true`.

5.2.2 Hledání možného navázání proměnných

Rekurzivní metoda `findBindingByGuard` očekává na vstupu všechny možné navázání proměnných, mapu sestavených proměnných `Map<String, IToken> binding`, která se bude naplňovat a stráž přechodu *Guard*, která se musí testovat. Metoda je navržená jako rekurzivní, takže sama sebe volá. Hledání kombinace je postavené na procházení proměnných z množiny všech možných navázání. Pro každé možné navázání proměnné se do hledané kobinace *binding* uloží hodnota a v případě, že existuje další proměnná, zavolá se znovu metoda `findBindingByGuard`. V situaci, kdy žádná další proměnná v množině není, se předpokládá, že je vytvořená kombinace hodnot všech proměnných.

Po vytvoření kombinace hodnot se musí otestovat její proveditelnost. V případě, že přechod neobsahuje žádnou stráž, se předpokládá, že je kombinace správná. V opačném případě se pro danou kombinaci musí otestovat stráž přechodu popsané v další podkapitole 5.4. Možných seskupení proměnných je více, což tvoří nedeterministické chování. Z toho důvodu je algoritmus implementován tak, že při nalezení první možné kombinace končí prohledávání.

5.2.3 Příklad vyhledání správné kombinace

Test, který je zobrazený na obr. 5.2 slouží pro demonstraci navázání proměnných je implementovaný a integrovaný v sadě testů popsaných v kapitole testování 6. Navázání proměnných je podmíněno následujícími pravidly.

Dle hran:

- x - v místě `p1` se musí vyskytovat dvakrát vyskytovat stejná značka
- y - musí být taková hodnota, která je obsažená v místech `p2` a `p4`.
- z - jakkákoliv hodnota z místa `p3`.
- o - jakkákoliv hodnota z místa `p6`.
- poslední hranová podmínka pouze kontroluje, že v místě `p5` jsou dvě číselné značky s hodnotou 1.

Dle stráže:

- o - instance Petriho sítě obsažená v proměnné `o` musí mít v místě `p` stejnou hodnotu jako je v proměnné x.
- x - hodnota musí být větší než v proměnné y.
- y - hodnota musí být větší než v proměnné z.

```
main AdvancedInput6

class AdvancedInput6 is_a PN
    object
        place p1(1, 2, 2, 8, 8)
        place p2(#e, 4)
        place p3(10, 2)
        place p4(4)
        place p5(1, 1, 42)
        place p6(#e)

        trans t1
            precond p6(#e)
            action {o := C1 new}
            postcond p6(o)
        trans t2
            precond p1(2‘x), p2(y), p3(z), p5(2‘1), p6(o)
            cond p4(y)
            guard {o hasPlaceToken: x. x > y. y > z}
            action {x println. y println. z println}

    class C1 is_a PN
        object
           place p(8)

        sync hasPlaceToken: x
            cond p(x)
```

Obrázek 5.2: Zdrojový kód modelu pro demonstraci navázání proměnných.

Ze zdrojového souboru lze vyčíst, že model má následující funkcionalitu. Prvním přechod se provede pouze jednou, jelikož symbol `#e` je v místě `p6` po prvním průchodu nahrazen objektem Petriho sítě `TestBindingClass2`. Druhý přechod vytiskne hodnoty proměnných `x, y, z` pouze v případě, že tyto proměnné splní následující požadavky:

Aplikací jednotlivých metod vyjde kombinace navázání proměnných nebo to, že přechod je neproveditelný.

Prvním krokem je sestavení všech možných hodnot proměnných. V tomto kroce se vyfiltrují podmínky hran. Již zmíněna metoda `findAllPossibleBindings` nejdříve naváže proměnné z hran `Precondition` a zkontroluje počet značek v místě `p5`. Poté vyfiltruje proměnnou `y` dle hrany v `Condition`. Výsledkem je následující mapa proměnných `x = {2, 8}, y = {4}, z = {10, 2}, o = {TestBindingsClass2Instance}`. Instance druhé třídy bude dále pojmenována pomocí zkratky `TBC2Instance`.

Druhým krokem hledání kombinace proměnných je splnit podmínky stráže. Rekurzivní metoda `findBinding` postupně skládá kombinace a testuje podmínky stráže. První kombinací je `x = 2, y = 4, z = 10, o = TBC2Instance`. Tato kombinace selže už při prvním testu přes synchronní port, jelikož instance třídy `TestBindingsClass2` nemá v místě `p` hodnotu 2. Algoritmus tedy zkusí další kombinaci v pořadí. Vzhledem k pořadí zano-

řování rekurzivní funkce se zkouší kombinace od poslední proměnné. Pokud proměnná nemá žádné další možné hodnoty, vrátí se o volání zpět a a s novou hodnotou se volá znovu. Takto se zkouší všechny kombinace, dokud se metoda nevrátí do prvotního volání metody, která obsahuje proměnnou `x`. První kombinací, která splní alespoň první podmínku stráže je: `x = 8, y = 4, z = 10, o = TBC2Instance`. Zároveň se splní podmínka `x > y`, ale neplatí `y > z`. Zkusí se prakticky poslední možná kombinace před ukončením vyhledávání. Kombinace `x = 8, y = 4, z = 2, o = {TBC2Instance}` splňuje všechny podmínky stráže. Tato sestava proměnných je prohlášená za validní, tedy přechod je proveditelný a uloží se do množiny proměnných přechodu pomocí metody `setVariables(Map<String, IToken>`. V tomto stavu jsou proměnné inicializované a nastavené, přechod je označen za proveditelný a mohou se provést změny.

5.3 Provedení přechodu

Druhou fází iterace, po kontrole proveditelnosti přechodu, je jeho provedení. V tomto kroce se odeberou značky ze vstupní hrany. V případě, že stráž obsahovala synchronní port, se musí provést změny i v něm. Dále se provedou výrazy v akci a vložení značek do míst pomocí výstupní hrany. Metoda `doTransition` očekává na vstupu přechod k provedení a množinu míst, se kterými pracuje. Na obrázku 5.3 je znázorněn způsob volání metod jednotlivých částí přechodu.

```
PNCondition precondition = transition.getPrecondition();

if (precondition != null)
{
    doPrecondition(precondition, variables, places);
}
```

Obrázek 5.3: Kód zavolání metody pro vykonání vstupní podmínky.

`Variables` je kombinace proměnných z předchozího kroku uložené v přechodu: `Map<String, IPNToken> variables = transition.getVariables();`.

5.3.1 Hrany

Funkce vstupní hrany je odebrání značek z míst. Způsob zpracování takové hrany je znázorněno na obr. 5.4.

```
for (kazdou_mnozinu_z_multimnoziny_hrany)
    najdi_misto_k_modifikaci

    for (kazdou_polozku_z_mnoziny)
        nacti_znacku_z_polozky
        nacti_kvantitu_z_polozky

        if(znacka_je_promenna)
            nacti_hodnotu_promenne_do_znacky

        odeber_danou_kvantitu_znacek_z_mista
```

Obrázek 5.4: Pseudokód zpracování vstupní podmínky.

V případě výstupní hrany je algoritmus stejný, ale místo odebrání značek se značky vkládají. K tomu slouží metoda *putValuesIntoPlace*, která dle kvantity vytváří kopie metodou *clone*, jenž je implementovaná na každém tokenu zvlášť.

5.3.2 Akce

Akce přechodu je definovaná jako posloupnost výrazů, které se mohou přiřadit do proměnných. Z definice vyplývají události, které mohou nastat (2.2.7) a řeší se pomocí zaslání zprávy. Pseudokód je znázorněn na obr. 5.5.

```
for (kazdy_vyraz)
    nacteni_adresata_zpravy

    for (kazdy_vyraz_akce)
        zaslat_zpravu_adresatu_zrpavy

    for (kazdou_promennou_vyrazu)
        uloz_vysledek_zpravy
```

Obrázek 5.5: Pseudokód zpracování akce přechodu.

Zaslání zprávy je popsáno v kapitole 5.5. Pokud výraz žádnou zprávu neobsahuje, jedná se o klasické přiřazení `y := 4` a uloží se načtený *adresát zprávy*.

5.4 Stráž přechodu a synchronní port

V této části bude popsán stráž přechodu, synchronní port a způsob vyhodnocování a provádění změn. Stráž obsahuje list výrazů. Mezi těmito výrazy je operace konjunkce, což znamená, že všechny musí být pravdivé aby byl přechod proveditelný. Stráž je z hlediska implementace navržená na 2 fáze. V první - testovací - se kontroluje pouze, zda jsou výrazy pravdivé. V druhé, se musí provést změny synchronního portu, pokud ho stráž obsahuje. Zjednodušený pseudokód testování stráže je na obr. 5.6.

```
inicializace_promennych

for (kazdy_vyraz_straze)
    proved_vyraz_a_uloz_vysledek

    if (vysledek_je_booleovska_hodnota)
        logicky_soucin_hodnot
    else
        zpracuj_vysledek_jako_argument_synchronniho_portu

vrat_vysledek
```

Obrázek 5.6: Pseudokód otestování stráže přechodu.

Z pseudokódu lze vyčíst, že provedení výrazu může vrátit buď booleovské výrazy (*true*, *false*) a nebo cokoliv jiného. Samotné vyhodnocení výrazu je zaslání zprávy, které je popsáno v kapitole 5.5, jelikož je společné s vyhodnocováním akce přechodu popsané v kapitole 5.3.2. Druhá varianta je implementována z důvodu synchronního portu, kterému se může zaslat nevyhodncená proměnná a očekává se navázání.

Synchronní port

Generátor popsaný v kapitole 4.3 naplňuje synchronní port jako mapu, která má v klíči název portu, podle kterého je vyhledáván. V hodnotě ukládá vzor zprávy, na který reaguje. Synchronní port může být volán s i bez parametru. Implementace testování proveditelnosti portu je v metodě `testSynchronousPort`, zatímco metoda `doSynchronousPort` obsahuje logiku pro provedení změn s platným navázáním proměnných (vstupní podmínka, stráž - pokud obsahuje další synchronní port a výstupní podmínka). Pseudokód testování portu je na obr. 5.7.

```
inicializace_promennych
inicializace_promennych_z_argumentu_portu

if (promenna_je_navazana)
    kontrola_znacek_mista
else
    vyhledani_navazani_promennych

if(ma_byt_vracena_navazana_hodnota_promenne)
    vraceni_hodnoty
else
    vraceni_booleovske_hodnoty
```

Obrázek 5.7: Pseudokód otestování synchronního portu.

Příklad vyhodnocení synchronního portu z disertační práce [3] je:

```
o state: x. x >= 50
```

Příklad znamená, že objekt `o`, který je referencí na Petriho síť, má definovaný synchronní port se vzorem zprávy `state: x`. Tedy reaguje na zprávu `state` s parametrem `x`. Odpovědí musí být číselná hodnota, která se v dalším výrazu porovnává s hodnotou 50.

Příklad je tedy pravdivý pouze v případě, že Petriho síť, jenž má definovaný tento synchronní port obsahuje hodnotu x, která je větší nebo rovna 50.

5.5 Zaslání zprávy

Zasílání zpráv definované v 2.3.4 se dělí na 3 typy. Diagram tříd je popsán v 3.1.4. Implementace v metodě `doMessage`, která dle typu zprávy volá metody pro zpracování jednotlivých typů zpráv. Na vstupu očekává adresáta `IPNToken target`, kterému je zasílána zpráva `IPNMessage message`, množina navázaných proměnných `Map<String, IPNToken> variables` a informace, zda je zaslání volané ze stráže přechodu. Tato informace je dále posílána pro rozlišení synchronního portu a volání metody Petriho sítě.

5.5.1 Unární zpráva

Provedení unární zprávy závisí pouze na *selektoru* zprávy. Implementace přepínače *switch* v metodě `doMessageUnary` pomocí *selektoru* rozhoduje, které zpracování zprávy se má provést. Implementované možnosti jsou:

- Tisk bez nového řádku, které musí rozumět každá značka (`callMethodPrint`).
- Tisk s novým řádkem (`callMethodPrint`).
- Vytvoření nové instance Petriho sítě pomocí klíčového slova *new*(`callMethodNew`). Zároveň se aktivuje objektová síť třídy.
- Zaslání zprávy objektu Petriho sítě - metoda a synchronní port (`doMethodSynchronousPort`).

Pro vytvoření nové instance Petriho sítě je potřeba znát cestu k souboru. Cesta k souboru je navržena pomocí názvu balíčku, který se skládá z `model`, za kterým následuje název modelu - jméno počáteční třídy, tedy: `mainClassId` zakončeným názvém invokované třídy. Výsledná cesta má tvar `model.mainClassId.classToBeCalled`. K vytvoření nové instance je využíván mechanismus reflexe. Kód pro vytvoření nové instance je zobrazen na obr. 5.8.

```
Class<?> clazz = URLClassLoader.getSystemClassLoader().loadClass(
    fullClassName);
Constructor<?> constructor = clazz.getConstructor();
Object object = constructor.newInstance();

if (object instanceof PNClass)
{
        result = PNClass.class.cast(object);
}
else
{
        throw new InvalidPnClassException();
}
```

Obrázek 5.8: Kód vytvoření nové instance třídy.

Kontrolou, zda je *object* instací třídy *PNClass* je ověřené, že tuto třídu dědí. Ošetření vyjímek je v implementaci ošetřeno. Provedení metody a synchronního portu je popsáno v samostatných kapitolách 5.6 a 5.4.

5.5.2 Binární zpráva

Binární zpráva se může liší dle typu adresáta. Pomocné rozhraní `SimulatorConstats` má definované seznamy povolených operací *NUMERIC_OPERATIONS* a *BOOLEAN_OPERATIONS*. Práce s těmito konstantami je znázorněna na obr. 5.9.

```
if (SimulatorConstants.NUMERIC_OPERATIONS.contains(selector))
{
        msgResult = doNumericOperation(targetToken, selector, operandToken);
}
else if (SimulatorConstants.BOOLEAN_OPERATIONS.contains(selector))
{
        msgResult = doBooleanOperation(targetToken, selector, operandToken);
}
else
{
        throw new NotSupportedYetException();
}
```

Obrázek 5.9: Kód pro zpracování operace zprávy.

V implementaci je znázorněná logika pro provádění operací. Každý list možností musí mít ekvivalentní metodu s implementovanám přepínačem pro zpracování. Momentálně jsou povoleny matematické a logické operace pro znázornění funkčnosti. Pro rozšíření např. ře-

tězcových operací by se musel přidat list *STRING_OPERATIONS* do rozhraní *SimulatorConstants* a metoda pro zpracování *doStringOperation(...)*.

5.5.3 Klíčová zpráva

Implementace klíčových zpráv je omezena na metody a synchronní porty Petriho sítí. Provedení je totožné jako u *unární zprávy* s rozšířením o argumenty. Argumenty jsou řazeny dle vzoru zprávy, tudíž se do seznamu argumentů vkládají pouze hodnoty. Provedení metody a synchronního portu je popsáno v samostatných kapitolách 5.6 a 5.4.

5.6 Volání metody

Metoda Petriho sítí, dle definice 2.3.2, může obsahovat místa a přechody. Z toho důvodu je navržena jako samostatna třída, podobně jako *třída* Petriho sítí. Rozdíl je v nadtřídě `PNMethod` a cesty k souboru. Metoda má *selektor*, tedy vzor zprávy, na který Petriho síť reaguje. Pro generování souboru jsem navrhl názvosloví, kdy cesta k souboru se skládá stejně jako u normální třídy Petriho sítě - balíčku *generated.model.mainClassId* zakončené názvem souboru. V případě metod je název složen z třídy, která ho definuje a samotného názvu. Výsledná cesta je *generated.model.mainClassId.ownerClassId_MethodNameId*.

V metodě `callMethod` je implementované ošetření existence metody ve třídě a poskládání správné cesty. Poté volá `runMethod`, která má za úkol vytvořit instanci metody, zpracovat argumenty a provést přechody.

5.6.1 Vytvoření instance

Implementace odvozená od vytvoření instance Petriho sítě (5.5.1) s tím rozdílem, že se využívá parametrizovaný konstruktor, který je na obr. 5.10.

```
Class<?>[] constructorArguments = new Class[2];
constructorArguments[0] = IPN.class;
constructorArguments[1] = List.class;
```

Obrázek 5.10: Kód parametrizovaného konstruktoru.

5.6.2 Inicializace míst z argumentů

V případě klíčové zprávy načte metoda `initPlacesWithArguments` argumenty metody v pořadí ze vzoru zprávy. Nejdříve se načtou formální hodnoty zprávy. Pokud se počet argumentů, které jsou zaslány, nerovnají počtu formálních argumentů zprávy, tak je to vyhodnocené jako chyba. V opačném případě se vloží patřičné argumenty do místa.

5.6.3 Provedení metody

Průchod metodou je implementovaný v *runMethodIteration* stejně, jako průchod Petriho sítí (5.3). Rozšířením je kontrola výstupního místa *return*, což indikuje událost ukončení metody.

Kapitola 6

Testování

V této kapitole bude popsán způsob testování aplikace, možnosti konfigurace a příklad implementace testů. Používán je framework JUnit 4.12, který umožňuje psát jednotkové testy. Aplikace je vyvíjena v jazyce Java 8.0, tudíž je potřeba mít odpovídající JDK. Struktura aplikací, které budou v kapitole popisovány včetně příkazů jsou popsány v kapitole C.

6.1 Konfigurace

Pro konfigurace výpisu slouží soubor *config.xml* ve kterém lze nastavit následující parametry:

1. Výpisy pro ladění - výpisy, které zobrazují informace o aktuálním dění v aplikaci.
2. Výpisy pro trasování - výpisy, které zobrazují informace o aktuální metodě, ve které se aplikace nachází.

Oba parametry lze zapnout/vypnout nastavením *ENABLED* či *DISABLED*. Pro vypisování je implementovaná pomocná finální třída `LoggingUtil`, která při vytvoření první instance zkontroluje soubor *config.xml* a zapne či vypne vypisování. Pro výpis se používají statické metody `debug` či `trace`.

6.2 Vývojové testy

Jsou testy překladače, které byly využívány při vývoji. Na těchto testech lze vidět příklad použití struktur, která je klíčová pro překladač i simulátor. Pro otestování chodu aplikace slouží integrační testy.

Překladač obsahuje tyto testy:

1. Jednotkové testy, které slouží pro otestování metod.
2. Celkové testy - otestování funkcionality analyzátoru.
3. Doplňující testy - například testy pro kontrolu *PNSyntaxValidator*.

Testy v příkladech slouží pouze pro demonstraci principu testování.

6.2.1 Jednotkový test výrazu

Test má zpracovat výraz `y := 42` a zkontrolovat, že se struktura naplnila dle očekávání. Implementace testu na obr. 6.1 je jeden z testů implementovaných v třídě `ParserUnitTests`.

```
@Test
public void testExpression()
{
        String expressionsInput = "y := 42";
        List<PNExpression> expressions = compiler.parseExpressions(
            expressionsInput);

        assertEquals(0, expressions.size());
        PNExpression expression = expressions.get(0);

        assertEquals(1, expression.getVariables().size());
        assertEquals("y", expression.getVariables().get(0).getValue());
        assertEquals(PNTokenInteger.class, expression.getTarget().getClass()
            );
        assertEquals(42, expression.getTarget().getValue());
}
```

Obrázek 6.1: Kód pro testování zpracování výrazu analyzátorem.

6.2.2 Validace syntaxe

Kontrola validace syntaxe je implementovaná v `ParserSyntaxValidatorTests`. Ukázka na obr. 6.2 testuje korektní zápis *ID*, což představuje název proměnné. Proměnná z definice syntaxe musí začínat malým písmenem, za kterou následuje libovolný počet písmen a číslic.

```
@Test
public void testId()
{
        assertEquals(true, PNSyntaxValidator.isId("a"));
        assertEquals(true, PNSyntaxValidator.isId("a1"));
        assertEquals(true, PNSyntaxValidator.isId("abc"));

        assertEquals(false, PNSyntaxValidator.isId("1"));
        assertEquals(false, PNSyntaxValidator.isId("#"));
        assertEquals(false, PNSyntaxValidator.isId("_42"));
}
```

Obrázek 6.2: Kód pro testování syntaxe identifikátoru.

6.2.3 Celkový test analyzátoru

Celkový test analyzátoru je takový, který na vstupu dostane cestu k souboru obsahující model a musí validně naplnit strukturu. Testy jsou implementovány v třídě `ParserTests`. V testu na obr. 6.3 se testuje vstup `Input1`, který má model s 1 třídou, která má v objektové síti 2 místa `p1` a `p2`. V prvním místě je značka číslo s hodnotou `3`. Dále má přechod se vstupní a výstupní podmínkou. Test kontrolu validní zpracování těchto komponentů.

```
private static final String INPUT_1_PATH = "examples/development/input1";

@Test
public void testInput1()
{
        PNClasses classes = compile(INPUT_1_PATH);
        assertNotNull(classes);

        assertEquals("Input1", classes.getMainClassId());
        assertEquals(1, classes.getClasses().size());
        PNClazz clazz = classes.getClasses().get(0);

        // object net
        IPNNet objectNet = clazz.getObjectNet();
        assertNotNull(objectNet);
        assertEquals(2, objectNet.getPlaces().size());

        // place
        PNPlace p1 = objectNet.getPlaces().stream().filter(p -> p.getName().
            equals("p1")).findFirst().get();

        assertEquals(1, p1.getTokens().size());
        assertEquals(3, p1.getFirstToken().getValue());

        // transition
        assertEquals(1, objectNet.getTransitions().size());
        PNTransition transition = objectNet.getTransitions().get(0);

        assertNotNull(transition.getPrecondition());
        assertNotNull(transition.getPostcondition());
}
```

Obrázek 6.3: Kód pro otestování zpracování třídy analyzátorem.

6.2.4 Spuštění testů

Testy je možné spustit pomocí příkazu `ant runDevTests` v aplikaci *PNCompiler*, které spustí vývojové testy pro syntakticky a lexikální analyzátor. Výsledkem je výpis, který zobrazuje hromadné informace o testech jednotlivých tříd. Příkladem může být:

```
TestSuite: bp.pntalk.codegenerator.tests.development.parser.PNParserUnitTests
Tests run: 14, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.032 sec
```

6.3 Integrační testy a spuštění aplikace

Integrační testy slouží jako testovací modely pro aplikaci. Jedná se o 18 navržených modelů, jejichž složitost se postupně zvyšuje. Testy jsou uloženy ve složce *examples/integrated* v aplikaci `PNCompiler`. Generování souborů z modelů je implementováno jak manuálně, tak automaticky. Simulování je však čistě manuální záležitost. Dále budou popsány úrovně testů, příklad obou přístupů generování souborů a jednotlivé kroky pro spuštění aplikace včetně demonstrační ukázky.

6.3.1 Rozdělení testů

1. Základní (*basic*) - otestování modelu s jednou třídou, která obsahuje místa a přechody, jednoduché navázání proměnných, vyhodnocování stráží, unární a binární zasílání zpráv, práce s proměnnými.

2. Středně pokročilý (*intermediate*) - přidání více výrazů, složených zasílání zpráv, inicializace značek v místě pomocí proměnné a počáteční akce, složité navázání proměnných viz. příklad v kapitole 5.2.3, vyhodnocení složených výrazů ve stráži.

3. Pokročilý (*advanced*) - přídání podpory více tříd, synchronních portů bez parametru i s parametry, metody bez parametrů i s parametry, aktivace objektové sítě nové instance třídy.

Mechanismus je rozdělen do 3 částí (kapitola 3.1). Aplikace *PNCompiler* se stará o překlad souborů a vytvoření knihovny pro simulátor, jelikož obě části využívají stejných struktur. Druhá aplikace - *PNSimulator* slouží pro vytvoření knihovny, která se stará o chod simulace. Poslední aplikace, *PNModelApp* slouží pro spouštění modelů. Jednotlivé sekce včetně možností příkazů, překladů a spuštění budou popsány v následujících podkapitolách.

6.3.2 Příklad spuštění aplikací

Následuje příklad na kterém se demonstrují jednotlivé kroky potřebné pro vygenerování a simulování modelu. Předpokladem je, že aplikace neobsahují knihovny potřebné pro chod. V případě, že knihovny již mají, nejsou kroky s vytvářením a ukládáním knihoven potřebné. Model pro demonstraci spuštění z obr. 6.4 je obsažen v přiložené sadě příkladů ve složce *examples/integration/advanced*.

```
main AdvancedInput2

class AdvancedInput2 is_a PN
    object
        place p1(#e)
        place p2()

        trans t1
            precond p1(#e)
            action {o := C1 new}
            postcond p2(o)
        trans t2
            precond p2(o)
            guard {o containsToken42}
            action {'Correct' print}

class C1 is_a PN
    object
        place p(42)

    sync hasPlace42
        cond p(42)
```

Obrázek 6.4: Zdrojový kód modelu pro demonstraci spuštění aplikace.

Model, jehož počáteční třída je `AdvancedInput2` v prvním přechodu vytvoří novou instanci třídy `C1`. V druhém přechodu zkontroluje, zda tato vytvořená instance má v místě `p` hodnotu 42. Pokud ano, vytiskne se na výstup `Correct`, jinak se neděje nic.

Kroky pro spuštění simulace

1. Překlad modelů - *PNCompiler*
   (a) Překlad aplikace - zdrojové soubory se přeloží do složky *build* příkazem `ant compile`.
   (b) Vytvoření knihovny - knihovna `xfryct00-bp-compiler.jar` se uloží do složky *dest* příkazem `ant jar`.
   (c) Spuštění překladu modelu - provést generování modelu lze několika způsoby.

   ```
   ant run -Darg0=examples/integration/advanced/advancedInput2.
   ```

   Pro změnu názvu modelu lze použít druhý argument jako:

   ```
   ant run -Darg0=examples/integration/advanced/advancedInput2
   -Darg1=NewModelName.
   ```

   Pro usnadnění testování je implementován integrační test, který vygeneruje všechny modely. Spustit lze:

   ```
   ant runIntTests
   ```

2. Knihovna simulátoru - *PNSimulator*

   (a) Vložení knihovny překladače - soubor `xfryct00-bp-compiler.jar` ze složky *dest* aplikace překladače se vloží do složky knihoven *lib* v simulátoru.

   (b) Vytvoření knihovny simulátoru - knihovna `xfryct00-bp-simulator.jar` se uloží do složky *dest* příkazem `ant jar`. Příkaz automaticky provede překlad pomocí `compile`, tudíž jej není nutné manuálně provádět.

3. Spuštění simulace modelu - *PNModelApp*

   (a) Vložení knihovny simulátoru - soubor `xfryct00-bp-simulator.jar` ze složky *dest* aplikace simulátoru se vloží do složky knihoven *lib* v aplikaci modelu.

   (b) Vložení zdrojových souborů modelu - model uložený v adresáři *generated/model* aplikace `PNCompiler` se vloží do adresáře *src/model* spouštěcí aplikace.

   (c) Překlad aplikace - zdrojové soubory se přeloží do složky *build* příkazem `ant compile`.

   (d) Spuštění simulace - Aplikaci lze spustit pomocí jednoho či dvou argumentů. V případě, že název modelu odpovída názvu hlavní třídy, stačí použít pouze první argument. V opačném případě je potřeba specifikovat i název modelu.

```
ant run -Darg0=AdvancedInput2

ant run -Darg0=AdvancedInput2 -Darg1=NewModelName
```

Výsledkem těchto kroků je zobrazení řetězce `Correct` v konzoli.

Pro překlad a spuštění aplikace `PNModelApp` bez *Ant* lze využít příkazy:

```
mkdir build

javac -cp "lib/xfryct00-bp-simulator.jar:" -d build src/model/App.java
javac -cp "lib/xfryct00-bp-simulator.jar:" -d build src/model/**/*.java
java -cp "lib/xfryct00-bp-simulator.jar:build:" model.App "AdvancedInput2"
```

Nebo v případě dvou argumentů:

```
java -cp "lib/xfryct00-bp-simulator.jar:build:" model.App "AdvancedInput2"
   "NewModelName"
```

Kapitola 7

Závěr

Cílem této práce bylo navrhnout mechanismus transformace modelů z formalismů OOPN do programovacího jazyka Java. Cíle se podařilo dosáhnout navržením takového mechanismu, který umožňuje překládat modely zapsané v jazyce PNtalk a následně je generovat do programovacího jazyka Java. Implementace obsahuje syntaktickou validaci modelů. Takto vygenerované soubory jsou čitelné a lze je ručně modifikovat. V rámci implementace je dodána i knihovna simulátoru. Aplikace podporuje objektové sítě, jednoduché i složené zaslání zpráv, více výrazů v akcích i přechodech, matematické a logické operace, více tříd, metody bez parametru i s parametry, synchronní porty bez parametru i s parametrem, hledání možné kombinace navázání proměnných se zapojením podmínek i vstupních podmínek, použití proměnných i v kvantitě hran.

Výstupem je tedy překladač, simulátor a spouštěcí aplikace, která tyto aplikace spojuje. Všechny části jsou popsány včetně diagramů tříd a příkladů funkcionality. Přiloženo je 18 modelů k otestování chodu aplikace s popisem spuštění v kapitole 6.3.2. Mimo to jsou navrženy JUnit testy pro překladač. Výhoda jazyka Java spočívá v nezávislosti na platformě, tudíž aplikace přeložená na systému Windows OS je spustitelná i na systému UNIX OS.

Aplikace má pár omezení, které jsou popsána v další podkapitole. Stejně tak jsou navržena rozšíření pro jednotlivé bloky mechanismu.

7.1 Omezení

V rámci implementace byly vynechány některé možnosti, které jazyk PNtalk nebo Smalltalk nabízí. Zde je souhrn věcí, které nejsou podporovány, uvedení příkladu možné implementace a návrh na rozšíření funkcionality bakalářské práce.

V příloze A je popsaná gramatika jazyka PNtalk. Ze syntaxe není podporováno:

1. Závorky ve výrazech.
2. Identifikátory *temps* v počáteční akci místa a výrazu akce.
3. List v multimnožinách.
4. Pole *array* včetně konstanty pole *arrayconst*.
5. Posloupnost zpráv *cascade* oddělené středníkem ve výrazech.

V případě doplnění implementace pole *array* je zapotřebí rozšířit:

- Implementace metod přichystané značky. `PNTokenArray` pro práci s hodnotami.
- Přidání metody `isArrayconst` v třídě `PNSyntaxValidator`, která zkontroluje syntax symbolu `#` a následného pole.
- Přidání kontroly pole v `return isNumber(token) || isString(token) || isCharconst(token) || isSymconst(token)` v metodě `isLiteral`.
- Přidání podmínky v metodě `parseToken` analyzátoru `PNParser`. Je-li token pole, pak zpracuj pole a vytvoř novou instanci tokenu.
- Rozšíření implementace metody `doKeyMessage` pro podporu zasílání zpráv v `PNSimulator`.

7.1.1 Generování dědičnosti

Definice PNtalk v kapitole 2.3.1 umožňuje dědit ostatní třídy Petriho sítí. Aktuální implementace umožňuje zpracování pouze v překladači. Generování umožňuje pouze dědění třídy *PN*. V případě rozšíření by bylo potřeba rozšířit generátor o následující části:

1. Přeložení vygenerované třídy z které se má dědit.
2. Načtení přeložené třídy.
3. Úprava dat ve struktuře tříd.

Vzhledem k návrhu generovaných tříd se potřebné data sdílí skrz klasickou dědičnost. Každý konstruktor nejdříve volá konstruktor svého předka, tudíž by se inicializovala děděná třída a poté se přepsaly hodnoty. Místa jsou implementované jako množiny pomocí `HashSet`, které neumožňují duplicitní záznamy. To znamená, že kdyby se přepisovaly tokeny v místě, tak by se nahradilo celé místo a tím se nahradilo. Přechody jsou implementované pomocí `ArrayList`, takže by se musely vyhledat a upravit. Pseudokód s příkladem metod pro přeložení generovaných tříd a načtení takových tříd je na obr. 7.1.

```
poskladani_cesty_k_prekladanemu_souboru
inicializace_systemoveho_prekladace
prelozeni_tridy
kontrola_uspesnosti_prekladu
```

Obrázek 7.1: Pseudokód přeložení zdrojových souborů modelu.

Pro přeložení souboru lze využít nástroje `JavaCompiler`. Překlad by vypadal dle kódu na obr. 7.2.

```
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
int check = compiler.run(null, null, null, classPath);
if (check != 0)
{
    throw new CompileNotPossibleException();
}
```

Obrázek 7.2: Kód přeložení třídy.

Načtení souboru by se dalo implementovat pomocí funkce *forName* v rozhraní tříd *Class*.

Dále nebyly nebyly implementovány konstruktory, jelikož jejich funkcionalitu lze nahradit metodami.

7.2 Navrhovaná rozšíření

Několika rozšířeními jsou:

1. Optimalizace generovaných souborů - generování již vyhodnocených jednoduchých zpráv. Například `y := 1 + 2` se vyhodnotí již za překladu a vygeneruje se pouze `y := 3`.
2. Modifikace vygenerovaných souborů - implementace nástrojů pro načtení, úpravu a vytvoření nových modelů z již vygenerovaných
3. Více módů spuštění - mód simulace krokování iterací
4. Rozšíření kontroly sémantické analýzy - kontrola typů argumentů při zasílání zpráv
5. Implementace vývojových testů - aplikace je natolik obsáhlá, že by bylo vhodné doimplementovat komplexnější vývojové testy pro všechny bloky, které z důvod velkého rozsahu práce nebyly prozatím implementovány.

Literatura

[1] Alexander Meduna, R. L.: *Modelování a simulace* - studijní opora. [Online; citováno 23. 4. 2019].
URL https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FIFJ-IT%2Ftexts%2FOporaIFJ.pdf&cid=11438

[2] Eric Freeman, K. S. B. B., Elisabeth Robson: *Head First Design Patterns*. O'Reilly Media, 2004, ISBN 978-0-596-00712-6.

[3] Janoušek, V.: *Modelování objektů Petriho sítěmi* - disertační práce. [Online; citováno 10. 4. 2019].
URL http://www.fit.vutbr.cz/~janousek/publications/phdthesis.pdf

[4] Meduna, A.: *Automata and languages : theory and applications*. London : Springer, 2000, ISBN 1-85233-074-0.

[5] Peringer, P.: *Modelování a simulace* - studijní opora. [Online; citováno 23. 4. 2019].
URL
https://wis.fit.vutbr.cz/FIT/st/cfs.php/course/IMS-IT/texts/opora-ims.pdf

[6] square: *JavaPoet* - knihovna. https://github.com/square/javapoet, 2015, [Online; citováno 25. 4. 2019].

[7] Vladimír Janoušek, R. K.: *Simulace a vývoj systémů*. [Online; citováno 1. 5. 2019].
URL http://perchta.fit.vutbr.cz/research-modeling-and-simulation/3

[8] Winterhalter, R.: *ByteBuddy* - knihovna. http://bytebuddy.net/#/, 2019, [Online; citováno 25. 4. 2019].

[9] Zbyněk Křivka, D. K.: *Principy programovacích jazyků a objektově orientovaného programování* - studijní opora. [Online; citováno 23. 4. 2019].
URL https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FIPP-IT%2Ftexts%2FIPP-II-ESF-1_3b_hyperlinks.pdf&cid=11469

Příloha A

Syntax jazyka PNTalk

V této příloze je popsána syntaxe jazyka PNtalk, definovaná v disertační práci [3].

```
classes: [class]* "main" id [class]*
class: "class" classhead [objectnet] [methodnet|constructor|sync]*
classhead: id "is a" id
objectnet: "object" net
methodnet: "method" message net
constructor: "constructor" message net
sync: "sync" message [cond] [precond] [guard] [postcond]
message: id | binsel id | [keysel id]+
net: [place|transition]*
place: "place" id"(" [initmarking] ")" [init]
init: "init" "{" initaction "}"
initmarking: multiset
initaction: [temps] exprs
transition: "trans" id [cond] [precond] [guard] [action] [postcond]
cond: "cond" id"(" arcexpr ")" ["," id"(" arcexpr ")"]*
precond: "precond" id"(" arcexpr ")" ["," id"(" arcexpr ")"]*
postcond: "postcond" id"(" arcexpr ")" ["," id"(" arcexpr ")"]*
guard: "guard" "{" expr3 "}"
action: "action" "{" [temps] exprs "}"
multiset: [n "‘"] c ["," [n "‘"] c]*
n: [dig]+ | id
c: literal | id | list
list: "(" [c ["," c]* ["|" [id | list] ]] ")"
temps: "|" [id]* "|"
```

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

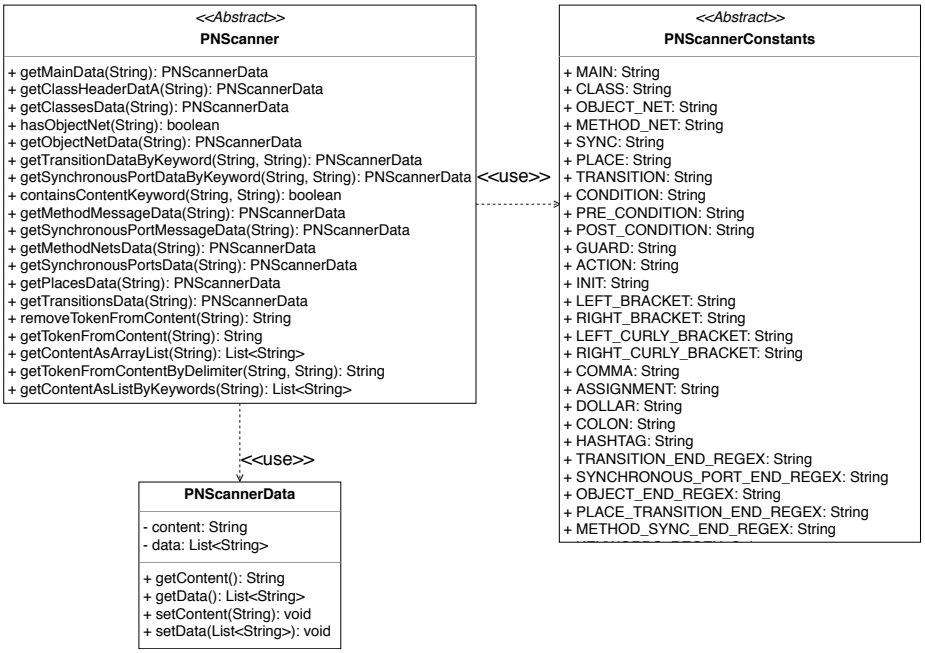
Příloha B

Detailní modely mechanismu transformace

V této příloze jsou přiloženy detailní návrhy překladače a simulátoru.

B.1 Lexikální analyzátor

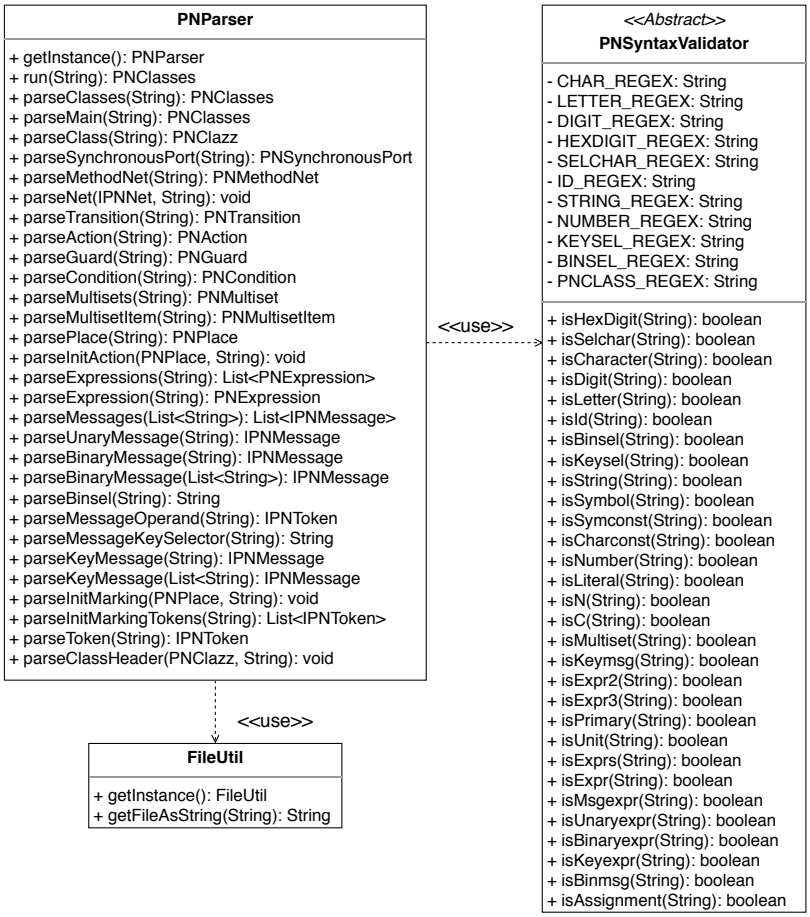
Přehled metod, konstant a pomocné třídy pro získávání dat o modelu.



Obrázek B.1: Model tříd lexikálního analyzátoru.

B.2 Syntaktický analyzátor

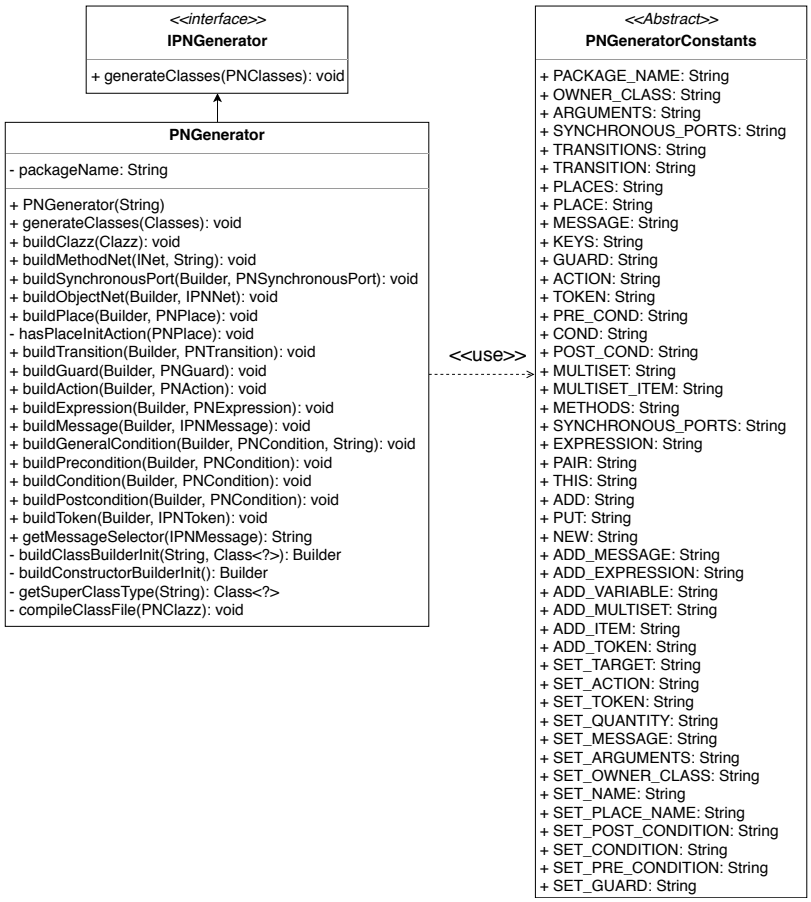
Přehled metod pro zpracování dat o modelu.



Obrázek B.2: Model tříd syntaktického analyzátoru.

B.3 Generátor

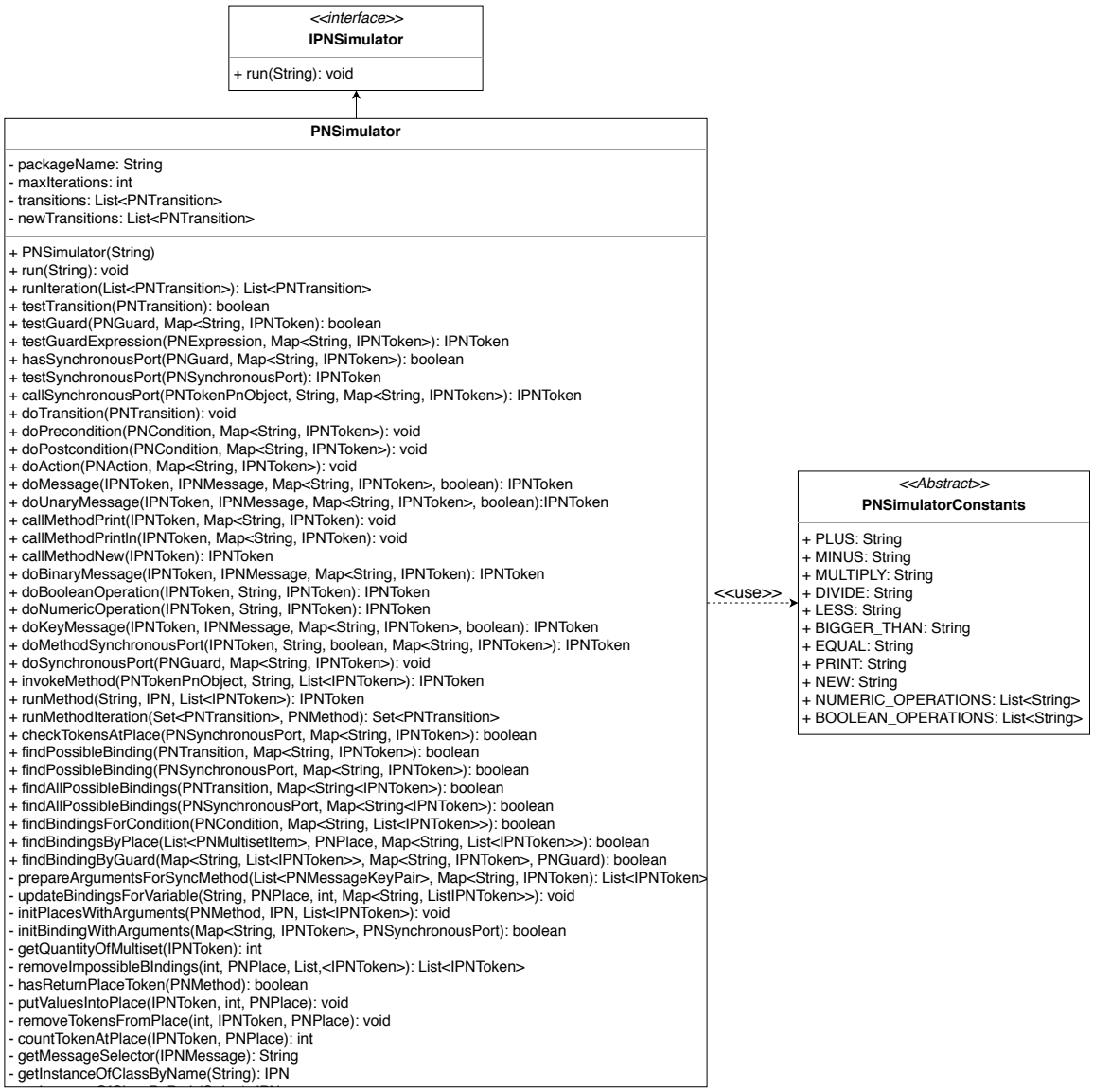
Přehled metod a konstant implementované v generátoru pro generování zdrojových souborů.



Obrázek B.3: Model tříd generátoru.

B.4 Simulátor

Přehled metod a konstant, které se starají o chod simulace modelu.



Obrázek B.4: Model tříd simulátoru.

Příloha C

Obsah CD

- Aplikace přeladače.
- Aplikace simulátoru.
- Aplikace pro provedení simulace modelů.
- Písemná práce ve formátu PDF.
- Zdrojové kódy a soubory písemné práce.

Struktura překladače

1. build - obsahuje přeložené soubory.
2. examples - obsahuje příklady modelů a modely pro testování.
3. generated - zde se ukládají vygenerované modely.
4. lib - knihovny *JavaPoet* [6], *junit* a *harmcrest* pro testování.
5. src - zdrojové soubory.
6. dest - obsahuje knihovnu překladače.
7. build.xml - soubor definující překlad a spuštění aplikace.
8. config.xml - konfigurace výpisů.
9. README.md - popis aplikace.

Struktura simulátoru

1. build - obsahuje přeložené soubory.
2. dest - obsahuje knihovnu simulátoru.
3. src - zdrojové soubory.
4. lib - knihovna překladače.

5. build.xml - definice překladu a spuštění aplikace.
6. README.md - popis aplikace.

Struktura spouštěcí aplikace

1. build - přeložené zdrojové soubory
2. lib - knihovna simulátoru pro simulaci
3. src - zdrojové soubory modelu, vygenerované z generátoru
4. build.xml - definice překladu a spuštění
5. config.xml - konfigurace výpisů
6. README.md - popis aplikace