VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

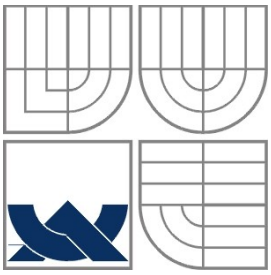
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

GENEROVÁNÍ KÓDU Z OBJEKTOVĚ ORIENTOVANÝCH PETRIHO SÍTÍ

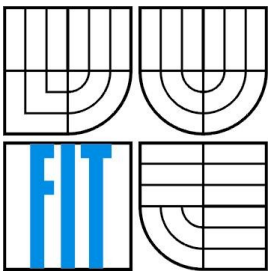
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. MARTIN HANÁK
AUTHOR

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

GENEROVÁNÍ KÓDU Z OBJEKTOVĚ ORIENTOVANÝCH PETRIHO SÍTÍ

CODE GENERATION FROM OBJECT ORIENTED PETRI NETS

DIPLOMOVÁ PRÁCE

MASTER THESIS

AUTOR PRÁCE
AUTHOR

Bc. MARTIN HANÁK

VEDOUCÍ PRÁCE
SUPERVISOR

ING. RADEK KOČÍ, PhD.

BRNO 2015

Abstrakt

Tato diplomová práce se zabývá generováním zdrojových kódů z jazyka PNtalk do standardních objektově orientovaných jazyků, např. C++ nebo Java. Rozebírá možnosti zjednodušeného modelování formalismu Objektově orientovaných Petriho sítí (OOPN) v prostředí těchto jazyků. Takový model by měl být jednodušší a efektivnější, než nabízí aktuální implementace v prostředí jazyka Smalltalk. Práce také uvádí návrh abstraktizace generátoru kódu tak, aby byl schopen generovat výsledný kód v různých jazycích.

Abstract

This thesis describes opinions about generating code from PNtalk language into more standard object orinted languages, such as C++ or Java. Goal is to construct model of the formalism of Object Oriented Petri Nets (OOPN), represented in PNtalk languege in mentioned languages. This model should be simlpier and more effective than actual implementation in Smalltalk language. Thesis also contains ideas about abstactization the code generator to be able to generate code in various languages.

Klíčová slova

objektově orientované Petriho sítě, PNtalk, generování kódu, překladač jazyka, optimalizace

Keywords

object oriented Petri nets, PNtalk, code generation, compiler, optimalization

Citace

Martin Hanák: Generování kódu z objektově orientovaných Petriho sítí, diplomová práce, Brno, FIT VUT v Brně, 2015

Generování kódu z objektově orientovaných Petriho sítí

Prohlášení

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

……
Martin Hanák
21.5.2015

Poděkování

Rád bych poděkoval vedoucímu práce Radku Kočímu za trpělivost a pomoc při formování zásad, z nichž tato práce vzešla.

Obsah

1 Úvod

Se stoupajícím rozvojem softwarového inženýrství stoupá také význam návrhu softwaru. Jak důraz na návrh roste, rostou i tendence vytvářet návrhy čím dál komplexnější. A platí, že čím je návrh dokonalejší, tím více je možné ovlivnit implementaci, popř. dokonce zavést postupnou automatizaci generování implementace. Za předpokladu, že nástroj pro návrh zároveň může sloužit jako simulátor naskýtá se možnost si nejdřív software vyzkoušet pomocí simulace. Pokud do návrhu dostatečně popíšeme i chování, existuje možnost mít kompletní aplikaci popsanou modelem. Pro to, aby tato myšlenka byla životaschopná, musí nejdřív vzniknout prostředek pro popis modelu, který bude v zásadě velmi jednoduchý, bude však skýtat možnosti popsané výše.

Jedním takovým projektem je i formalismus OOPN. OOPN je formován a vyvíjen na Fakultě informačních technologií již od roku 1993 prostřednictvím projektu PNtalk [1]. Je to prozatím experimentální projekt. Jeho základní myšlenka je, že vyšší modelovací nástroje mohou sloužit podobně jako klasické programovací jazyky. Má ambice stát se součástí implementace projektů a svým konceptem nabídnout odlišný přístup k tvorbě software.

1.1 Předmět diplomové práce

PNtalk je v současné době postupně rozvíjen především jako modelovací nástroj s širokými vyjadřovacími schopnostmi. V průběhu času se však ukázalo, že jeho aktuální podoba, tj. nástavba nad simulačním formalismem DEVS, navíc implementovaným v jazyce Smalltalk, je sice intuitivní, avšak samotný běh systému je poněkud náročný. Vzniká tedy idea, že PNtalk by mohl sloužit především jako vizuální specifikační nástroj, resp. jazyk. Výsledný program by však bylo efektivnější spouštět na jiné, standardnější a efektivnější platformě. Jako logickým východiskem se jeví dnes již standardní objektové jazyky jako C++ nebo Java.

Hlavní náplň práce tedy spočívá ve vytvoření generátoru kódu, který bude schopen systém specifikovaný pomocí jazyka PNtalk přetransformovat do zdrojového kódu běžnějšího objektově orientovaného jazyka, konkrétně C++.

Při návrhu takového generátoru je však vhodné, klást důraz na jeho rozšiřitelnost. Toto hledisko je důležité jak z pohledu možných budoucích rozšíření jazyka PNtalk, tak především z pohledu cílového jazyka. Tedy aby bylo možné relativně jednoduchým způsobem doimplementovat např. generátor pro jazyk Java, C# apod. Je tedy třeba klást důraz na univerzálnost použitých konstrukcí.

Petriho sítě jsou především simulační nástroj. Abychom mohli generovat kód, je nutné implementovat i simulační prostředí PNtalku pro každý cílový jazyk, který uvažujeme. Toto simulační prostředí pak bude součástí výsledného programu. Součástí diplomové práce je i návrh toho simulačního prostředí a vytvoření jeho UML modelu pro jednoduchou implementaci

Při generování kódu se nabízí možnost optimalizovat samotné generování pomocí identifikace standardních programovacích konstrukcí. Takové konstrukce jsou popsány v bakalářské práci *Knihovna vzorů Petriho sítí* [2], na kterou tato diplomová práce navazuje

1.2 Pojmy a jejich vztah

Během projektu budeme operovat s několika termíny, nástroji a jazyky, které spolu úzce souvisí a proto je vhodné je na začátku vyjmenovat, oddělit a určit mezi nimi vztahy. Jsou to:

- *Petriho sítě* – matematický formalismus popisující složité systémy na základě změn stavů.
- *Objektově orientované Petriho sítě (OOPN)* – rozšíření Petriho sítí o objektový přístup.
- *PNtalk* – jazyk pro popis OOPN, v rámci implementace PNtalku je implementován také simulátor modelů popsaných tímto jazykem.
- *Smalltalk* – objektově orientovaný jazyk, ve kterém je implementován PNtalk. PNtalk zachovává možnost použití Smalltalku při tvorbě systému.
- *Pharo* – nástroj pro programování ve Smalltalku. Není to pouze vývojové prostředí, ale zároveň virtuální stroj, ve kterém je Smalltalk implementován

1.3 Úrovně modelů OOPN

V rámci práce budeme popisovat a specifikovat různé modely systému vytvořeného pomocí OOPN. Jde o jeden a tentýž formalismus, budou se však lišit jazyky, pomocí nichž model specifikujeme, ale také účel existence modelu v té konkrétní podobě.
Modely budeme rozlišovat:

1. *Zdrojový model* v jazyce PNtalk – je pouze ve formě zdrojového textu a tvoří vstup pro generátor kódu. Hlavní rysy jazyka PNtalk budou popsány v kapitole 2.4.
2. *Interní model* překladače – bude reprezentován strukturou, která vzejde z analýzy zdrojového modelu. Struktura musí mít takovou podobu, aby v ní šly provádět logické úpravy pro účely výsledného systému. V zásadě lze ale říci, že struktura bude formou *Abstrakního syntaktického stromu.* Podobu tohoto stromu popíšeme v kapitolách 4.1 a 4.2. Interní model a generátor budeme implementovat v jazyce C++.
3. *Cílový model* v cílovém programovacím jazyce – jeho součástí budou třídy vygenerované ze zdrojového systému a také podpůrná knihovna, která bude implementovat simulátor systému v OOPN. Přesná specifikace se může lišit i v rámci různých cílových jazyků. Formální popis simulačního systému bude řešen v kapitole 3. Generování obsahu se pak zabývá kapitola 4.4.

2 Objektově orientované Petriho sítě

2.1 Petriho sítě

Petriho sítě sestávají z míst a přechodů, přičemž místo však nereprezentuje celý stav (tak jako například konečný automat), ale pouze jistou vlastnost systému. Hovoříme o tzv. parciálním stavu, přičemž množina všech aktuálních ohodnocení míst pak tvoří výsledný stav systému.

Označení místa se provádí graficky tečkou, ta se často nazývá *token*. Přechod je pak proveditelný, pokud systém splňuje výčet parciálních stavů, tzv. *Preset*. Přechod však také definuje *Postset*, což je množina parciálních stavů, jež systém nabude po provedení přechodu. Vazby postsetu a presetu k místům se nazývají *hrany*.

Fakt, že v jednu chvíli může být označeno více míst, nám tedy dává možnost modelovat paralelismus, což je asi největší přínos Petriho sítí. Avšak z hlediska čitelnosti modelu je dekompozice stavu na množinu vlastností, tedy míst, také nespornou výhodou.

Pro Petriho sítě existuje celá řada rozšíření, uvedu nejklasičtější:

- *prioritní přechody* – pokud jsou v jednu chvíli proveditelné dva přechody současně, vznikl by nedeterminismus, řeší se prioritním přechodem
- *přechod s pravděpodobností* – pokud je proveditelných více přechodů, lze jim stanovit pravděpodobnost, s jakou se provedou. Pokud povolíme nedeterministický stav, je to v podstatě ekvivalent pravděpodobných přechodů se stejnou pravděpodobností.
- *váha hrany* – dovoluje aby pro proveditelnost musel přechod odčerpat více než jeden token a také mohl po provedení distribuovat více než jeden token.
- *kapacita míst* – dovoluje stanovit maximální počet tokenů, které se v daném místě mohou vyskytovat

2.2 OOPN

Jedno z rozšíření také bývá vnesení objektově orientovaného přístupu. Existuje snaha v klasickém programování, aby struktury v programech co nejvíce odrážely skutečnou strukturu reálného světa, což vedlo k objektově orientovaném přístupu k programování. Využití této abstrakce má význam především pro lepší strukturování kódu, důsledkem čehož se zlepší čitelnost a přehlednost nebo znovupoužitelnost programů.

Modelování systémů Petriho sítěmi je sice elegantní, avšak při nárůstu složitosti systému se síť stává příliš složitá, nepřehledná a prakticky nečitelná. Při vhodném zapouzdření však lze dosáhnout značné abstrakce, a tím i zjednodušení. To je hlavní důvod snahy o zavedení objektového přístupu i do Petriho sítí. Jedním takto vytvořeným formalizmem je OOPN, definovaný v disertační práci *Modelování objektů Petriho sítěmi* [3].

OOPN je matematický formalismus a proto vyžaduje nějakou implementaci, a to buď příslušným jazykem (v našem případě PNtalk, definován také v [3]), nebo grafickou reprezentací[1]. Proto představíme obě reprezentace konstrukcí OOPN.

2.3 Vlastnosti OOPN

Nejdůležitějším atributem OOPN je, že samotná Petriho síť je objekt. Vytvořením objektu nějaké třídy (zavoláním konstruktoru new) se automaticky aktivuje jeho *objektová síť*, tj. Petriho síť popisující hlavní logiku objektu. Objektová síť je však skryta, nelze k ní přistupovat z vnějšku, dá se tedy říci, že je autonomní. Kromě ní může mít třída metody a každá metoda má svou vlastní Petriho síť, tzv. *síť metody*. Ta vzniká zavoláním metody a jejím skončením zase zaniká. Připouští se vícenásobné spuštění metody, což znamená, že síť metody může mít i více instancí.

Tokeny nejsou už pouze značky, ale mohou být i reference na objekty. Objekt může být buď datové primitivum (číslo, symbol, řetězec, apod.) nebo objekt Petriho sítě. Token může reprezentovat n-tici objektů. Zůstává zachována sémantika místa v Petriho sítích, tedy že přítomnost tokenu v místě značí parciální stav.

Přechody mezi místy jsou proveditelné, pokud platí podmínky preset, postset a stráž přechodu. V přechodu lze definovat akci, tj. posloupnost příkazů a volání, které se při přechodu zavolají. V přechodu lze manipulovat s proměnnými, které do přechodu vstupují pomocí hran. Lokální proměnné není třeba deklarovat, OOPN je volně typovaný a jmenný prostor se váže dynamicky.

Přechod nemusí být vykonán atomicky. Pokud během vykonávání akce přechodu dojde k volání metody Petriho sítě, a tedy aktivace příslušné sítě metody, zůstává přechod v tzv. *rozpracovaném stavu* a čeká na návrat ze sítě metody. Poznamenejme, že takto rozpracovaných může být libovolný počet navázaní proměnných.

Hrany jednak vyjadřují kardinalitu objektu, s nímž se manipuluje a také slouží k navázání objektu na proměnnou. Hrany vyjadřují podmínky preset a postset. V PNtalku jsou vyjádřeny pomocí konstrukcí *precond* (objekty vstupující do přechodu), *postcond* (objekty vystupující) a *cond* (objekt se naváže na proměnnou přechodu, ale v místě zůstává).

V sítích lze také namodelovat nedeterministické chování, tj. vnesení náhody do vykonávání programu. Existují dva způsoby jak toho lze dosáhnout. Prvním je situace, kdy existuje více možných navázání proměnných pro jeden přechod. Druhý pak nastane, pokud v jednu chvíli je z jednoho místa proveditelných více přechodů. Simulátor si pak musí jeden vybrat a tato činnost se děje náhodně.

Naše výsledná implementace se však nedeterminsticky nechová, po nalezení prvního proveditelného přechodu jej provede. Důvodem je, že bychom museli hledat všechna navázání proměnných a navíc je pro experimentování užitečné, že jeden model bude vykonán vždy stejně.

Z hlediska omezení jsou v OOPN nejdůležitější tyto vlastnosti:

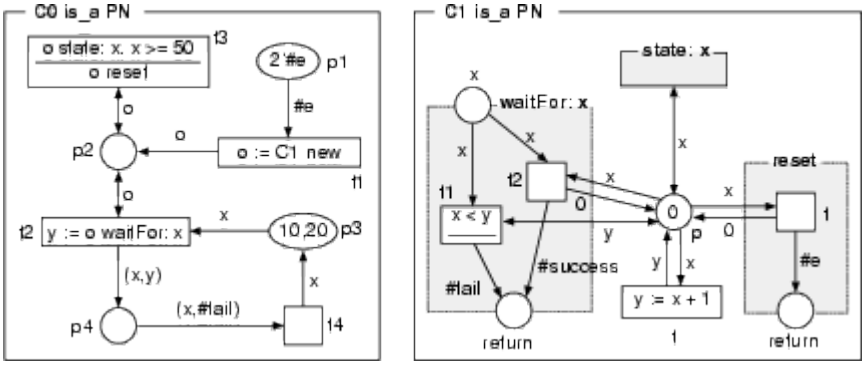
- Hranám nelze definovat prioritu ani pravděpodobnost
- Místům nelze definovat omezení kapacity

1 Striktně vzato je grafická reprezentace také formou jazyka, a jazyk PNtalk definuje jak graf. podobu, tak i textovou podobu

2.4 Jazyk PNtalk

V této kapitole zběžně popíšeme jazyk PNtalk, jež slouží pro popis systému modelovaném pomocí OOPN. Textová podoba jazyka PNtalk také slouží jako vstup pro generátor kódu.

Nejdůležitější prvky demonstrujeme na příkladu použití, k čemuž poslouží demostrativní systém na obrázku 1. Nebudeme se však zabývat významem modelu, jež reprezentuje. V následujících podkapitolách nás zajímají nejdůležitější konstrukce, které identifikujeme, popíšeme, popř. interpretujeme jejich sémantiku. Na příkladu si také můžeme všimnout grafické reprezentace.



Obrázek 1: Příklad konstrukcí OOPN

Jazyk PNtalk vychází z programovacího jazyka Smalltalk. Z tohoto jazyka přebírá datová primitiva (číslo, znak, textový řetězec a symbol) a je použit pro definování akcí přechodů.

2.4.1 Místo, přechod a inskripce přechodu

Místo definujeme pomocí klíčového slova *place*. Místo může mít počáteční značení i akci. Definice má tvar:

```
place jméno_místa (počáteční_značení) {počáteční_akce}
```

Přechod definujeme pomocí klíčového slova *trans* a identifikátoru přechodu. Může obsahovat následující konstrukce (v daném pořadí): *precond, cond, guard, action, postcond*. Každá konstrukce je uvozena pouze jednou, pokud obsahuje více položek, píší se za sebe a jsou odděleny čárkou. Všechny konstrukce jsou nepovinné, přechod však musí mít smysl.

Za klíčovými slovy pro typ hrany následují definice hran které mají tvar:

```
cílové_místo (hranový_výraz)
```

Hranový výraz a počáteční značení místa definujeme tzv. multimnožinou. Multimnožina je takový zápis množiny, kde prvek množiny má kvantifikátor, který určuje, kolikrát se daný prvek v množině vyskytuje. Má tvar:

$$n_1 \text{`} c_1, n_2 \text{`} c_2, \ldots, n_m \text{`} c_m,$$

kde n_i je kvantifikátor a c_i je prvek množiny. Kvantifikátor může však být definován proměnnou, musí však být vyhodnotitelný jako kladné celé číslo. V opačném případě je jeho hodnota automaticky nulová. Prvkem množiny mohou být datové primitiva, proměnné nebo seznamy.

Příklad multimnožiny:

```
2‘#e, x‘y, 5‘(x, y, $a), 6‘7.
```

Tento výraz reprezentuje multimnožinu, obsahující dva výskyty symbolu *#e*, x výskytů (kde *x* je proměnná) objektu referencovaného obsahem proměnné *y*, pět výskytů trojice (*x*, *y*, *$a*), a šest výskytů čísla *7*.

2.4.2 Stráž (guard)

Stráž je součást přechodu a odděluje se od akce přechodu vodorovnou čarou, přičemž stráž je nad čarou. Pomocí stráže lze realizovat testy na stav objektů před provedením přechodu. Toho lze dosáhnout voláním synchronních portů a inhibitorů a také vyhodnocovacích operací. Pokud lze provést nějaké navázání proměnných a zároveň všechny vyhodnocovací výrazy jsou úspěšné, pak lze přechod provést. Jinak je přechod pro navázání neproveditelný a objekty vrátí do stavu před zavoláním portu. Stráž má tvar:

```
guard {výrazy_stráže}
```

Pro definici výrazů stráže používáme stejný jazyk, jako pro akce přechodu. Ve stráži však platí, že výsledek každého dílčího výrazu musí být vyhodnotitelný jako pravdivostní hodnota.

Příklad: v síti *C0* v přechodu *t3* je stráž: *o state: x. x >= 50*. Její sémantika je: pokus se najít v místě *p2* objekt *o*, jehož synchronní port *state: x* vrátí objekt, jehož hodnota je větší než 50.

2.4.3 Synchronní port

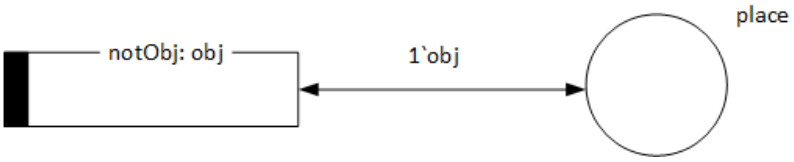
Zjednodušeně, synchronní port (nebo zkráceně jen *port*) slouží k propojení s jiným objektem. Může definovat *precond*, *cond*, *postcond* a *guard*. Od přechodu se tedy liší tím, že nedefinuje akci a především jeho provedení je podmíněno voláním z jiného objektu. Port je tak napůl metodou a napůl přechodem. Pokud je portu argumentem předána již navázaná proměnná, port je proveditelný pouze když je schopen nalézt navázání, tj. funguje jako test na přítomnost nějakého objektu v síti. Pokud je proměnná v argumentu nenavázaná, port provede některé možné navázání, pak slouží jako nástroj pro získání reference. V případě, že je možných více navázání, vybere se jedno náhodně, což je nedeterministické chování.

Příklad: síť *C1* definuje port *state: x*, který exponuje objekt z místa *p* a opět ho vrací. Tento port je volán ze stráže přechodu *t3* třídy *C0* pomocí nenavázané proměnné *x*. Port tedy naváže proměnnou *x* na některý objekt z místa *p*.

2.4.4 Inhibitor

Tato struktura byla do OOPN dodefinována v [4]. Inhibitor je negativní port. Opačná sémantika takového portu spočívá v tom, že port je proveditelný, pouze pokud nelze nalézt příslušné navázání. Ostatní vlastnosti jsou shodné se synchronním portem. Význam inhibitoru je, že pokud máme dvojici inhibitor a synchronní port se stejně definovanými podmínkami, tvoří spolu vzájemně komplementární rozhodovací dvojici

Grafická reprezentace je na obrázku 2.



Obrázek 2: Příklad inhibitoru

2.4.5 Metody

Metoda je jedna z nejtypičtějších konstrukcí v objektově orientovaném přístupu. Také třídy v OOPN mají definované metody. Každá metoda má definovanou svoji síť metody. Instance sítě je vázána na jedno volání metody, tzn. při každém volání vznikne nová síť a po skončení metody zase zaniká. V jednu chvíli může existovat více instancí téže metody.

Každá metoda musí ve své síti definovat návratové místo *return*. Přítomnost tokenu v tomto místě indikuje konec vykonávání metody. Pokud je v místě *return* více tokenů, jako návratová hodnota slouží pouze jeden token. Ten by se vybral náhodně a když by objekty nebyly stejné, vzniklo by nám nedeterministické chování.

Metoda může definovat argumenty, ale také nemusí. Volání metody je možné z jakéhokoli přechodu modelu a z metody je možno přistoupit ke všem místům objektu. V *síti metody* však nelze definovat porty.

Příklad: třída *C1* definuje dvě metody. Metodu *waitFor: x* s argumentem *x* a metodu *reset* bez argumentu.

2.4.6 Třídy a dědičnost

Třídy mohou ze sebe dědit pomocí klíčového slova *is_a*. Aby byla třída Petriho sítě validní, musí dědit z bázové třídy *PN*, nebo z jiné validní třídy Petriho sítí. Dědičnost specifikujeme při deklaraci třídy, celá deklarace má pak tvar:[1]

```
Jméno_třídy is_a rodičovská_třída
```

Ve třídě můžeme definujeme objektovou síť, která musí být uvozena klíčovým slovem object. Dále může mít třída libovolný počet metod, synchronních portů a inhibitorů. Při dědění můžeme libovolně specifikovat nové prvky a navazovat na stávající. Máme možnost však i předefinovat stávající tím, že v nové třídě uvedeme stejné jméno prvku. Pro jednotlivé prvky platí následující chování:

- Místo – při předefinování můžeme specifikovat jiné počáteční značení a akci.
- Přechody – předefinováváme tělo přechodu ale i všechny hrany. Původní hrany na rodiči tedy nebudou platné.
- Synchronní porty a metody – dojde ke kompletnímu nahrazení.

1 Všimneme si, že první písmeno je velké, což je podmínka pro jméno třídy.

2.4.7 Inskripční jazyk akcí přechodů

Pro definování akcí přechodů používá PNtalk jazyk, který je silně inspirovaný Smalltalkem. Dokonce umožňuje využívat i některé třídy Smalltalku především proto, že PNtalk byl původně implementovaný ve Smalltalku.

Jazyk pro pro inskripci přechodu sestává z posloupnosti zaslání zpráv, přičemž výsledek může být uložen do proměnné. Zpráva je identifikována tzv. *selektorem* a může mít argumenty. Zaslání zprávy má tvar: <adresát > <zpráva>. Adresátem může být libovolný token a zpráva má jeden z následujících tvarů:

- unární zpráva – zpráva má pouze selektor bez argumentu.
  Příklad: `'text' print` (řetězcové konstantě `'text'` je zaslána zpráva `print`), `-6.3 abs` (číslu `-6.3` je zaslána zpráva `abs`)
- binární zpráva – zpráva s jedním argumentem, selektor zprávy má však omezení – může to být jeden z následujících:
  `+ - / * = < > ~= <= >= & | // \\ , == ~==`
  Příklad: `3+4` (číslu 3 je zaslána zpráva „+ 4“, přičemž číslo 4 je argument)
- klíčovaná zpráva – má alespoň jeden argument, každý argument má svůj klíč a selektor zprávy je konkatenací klíčů.
  Příklad: `obj put:'text' at:3` (objektu v proměnné `obj` je zaslána zpráva `put: 'text' at: 3` , přičemž `'text'` a `3` jsou argumenty a selektor zprávy má tvar `put:at:`)

Pomocí takto definovaných zpráv můžeme vytvářet *jednoduché zaslání zprávy*. Jazyk PNtalk však definuje i složené zaslání zprávy, kdy místo adresátu nebo argumentu může být jiné zaslání zprávy. U složeného zaslaní jsou pak definovány pravidla pořadí vykonávání dílčích jednoduchých zpráv:

1. Unární zprávy, zleva doprava.
2. Binární zprávy, zleva doprava.
3. Zprávy s klíčovými slovy, zleva doprava.

Příklad takového vyhodnocení (zprávy jsou očíslovány v závorkách) je:

```
3 factorial + 4 factorial between: 10 and: 100
  (1)     (2)      (3)       (4)
```

Zpráva *factorial* vypočítá faktoriál čísla a zpráva *between:and:* vrátí, zda číslo náleží intervalu. Vyhodnocení zpráv bude v pořadí (1), (3), (2), (4) a výsledek tedy bude true (3! + 4! = 30) Pořadí lze ovlivnit pomocí závorek, kdy pokud narazíme na vyhodnocení zaslání zprávy, která je uzávorkována, vyhodnotíme přednostně tuto závorku. Příklad:

```
(3 factorial + 4) factorial between: 10 and: 100
  (1)     (2)      (3)       (4)
```

Pořadí vyhodnocení bude pak (1), (2), (3), (4) s výsledkem false ((3! + 4) = 10 a 10! = 3628800). Vezměme zde na vědomí, že ani Smalltalk ani PNtalk neimplementují priority aritmetických operátorů, takže například ve výrazu `3 + 4 * 6` bude nejdříve vyhodnoceno sčítání.

2.5 Příklad

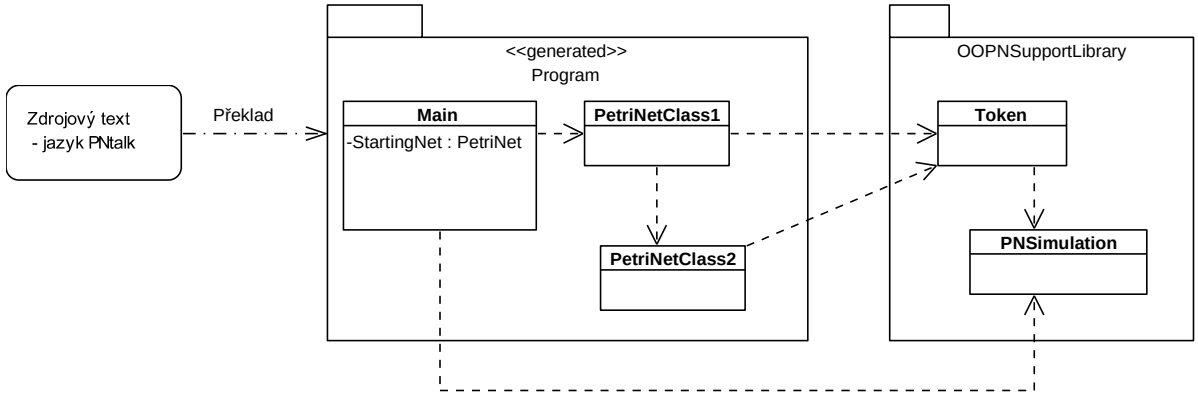
Následuje ukázka implementace systému z obrázku 1.

```
C0 is_a PN
object
  place p1(2`#e)
  place p2()
  place p3(10,20)
  place p4()
  trans t1
    precond p1(1`#e)
    action {o:= C1 new.}
    postcond p2(1`o)
  trans t2
    cond p2(o)
    precond p3(x)
    action {y := o waitFor: x.}
    postcond p4((x, y))
  trans t3
    cond p2(1`o)
    guard{o state:x. x >= 50.}
    action{o reset}
  trans t4
    precond p4((x,#fail))
    postcond p3(x)
```

```
C1 is_a PN
object
  place p(0)
  trans t
    precond p(x)
    action {y := x + 1.}
    postcond p(y)
  sync state: x
    cond p(1`x)
  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(1`x)
      postcond p(0),return(#e)
```

3 Návrh struktury simulátoru OOPN

Pro spuštění zdrojového kódu PNtalku je zapotřebí speciálního prostředí. Pro vygenerovaný kód budeme také potřebovat speciální knihovnu, která umožní běh OOPN. Tato knihovna bude mít charakter jednoduchého simulátoru, dále v textu ji tedy budeme nazývat zjednodušeně „simulátor“. Při implementaci simulátoru také definujeme, jakou strukturu budou mít třídy PNtalku v cílovém jazyce a nastíníme tak, co bude předmětem generování kódu samého.



Obrázek 3: Schéma generovaného kódu

Celkový proces je patrný z obrázku 3. Ze zdrojového textu v jazyce PNtalk vygenerujeme příslušné třídy a simulační kontext, tj. specifikace počáteční třídy, a volání spuštění simulace. Podpůrná simulační knihovna pak implementuje všechny konstrukce, na nichž je vygenerovaný kód závislý včetně simulační smyčky apod.

V následujících kapitolách popíšeme postup, který při implementaci simulátoru v cílovém jazyce budeme sledovat. Identifikujeme největší problémy, jež plynou z rozdílnosti zdrojové a cílové platformy, přičemž budeme brát v úvahu nejrozšířenější objektově orientované jazyky C++ a Java.

Jako nástroj pro popis simulátoru budeme využívat grafický jazyk UML.

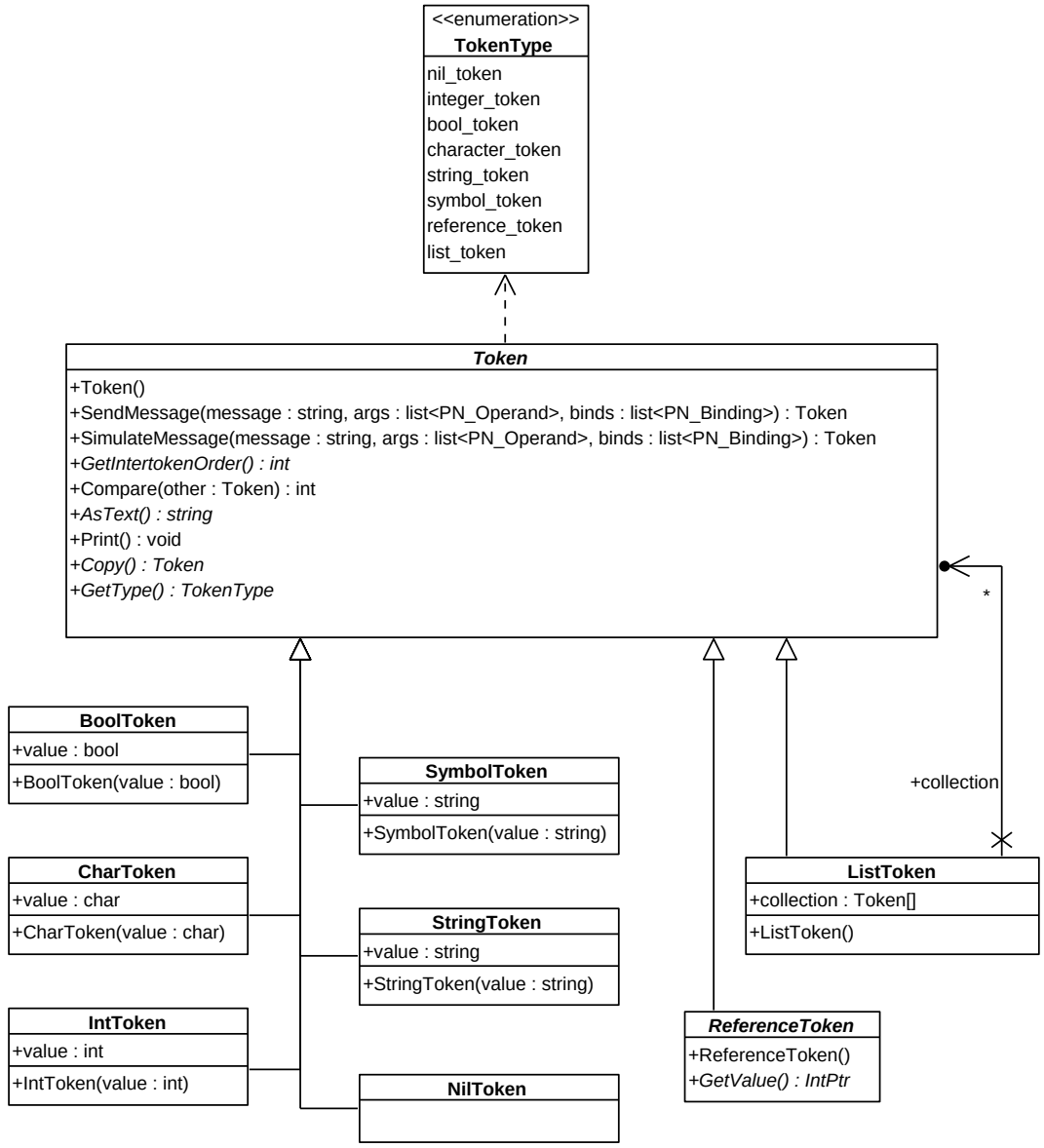
3.1 Konstrukce prvků OOPN

Tato kapitola se bude zabývat konstrukcí modelu, resp. jeho statické části. Popíšeme, jak modelujeme stav systému, jeho jednotlivé aspekty.

3.1.1 Token a primitivní datové typy

Jako základní nosič dat slouží abstraktní třída *Token* a její deriváty. Schéma popisuje diagram na Obrázku 4. V rámci formalismu OOPN jde o parciální ohodnocení stavu systému v konkrétním místě, v rámci našeho simulačního prostředí budeme navíc *Token* používat i v tělech přechodů jako součást navázání proměnných (více v kapitole 3.2.2)

Nejdříve si povšimneme potomků pro primitivní datové typy. Takový potomek nese hodnotu daného typu jako atribut. Třída *ListToken* slouží pro uložení strukturovaných n-tic libovolných Tokenů. *NilToken* je speciální token, který nenese žádná data. Abstraktní třída *ReferenceToken* slouží k uložení instancí objektů, které v systému tvoří především Petriho sítě, ale také další podporované objekty (viz. kapitola 3.1.10).



Obrázek 4: Třída Token a její deriváty

Na *Token* jsou z důvodů implementace kladeny následující požadavky:

a) Token se musí umět replikovat – implementace metody *Copy()*. Pro *ReferenceToken* je význam pouze kopie odkazu na instanci objektu, nikoli kopie objektu samého.
b) Nad třídou Token musí existovat částečné uspořádání. K tomu slouží metoda *Compare()*, která určí relativní pořadí tokenů nebo jejich rovnost. Způsob vyhodnocení:

1. Určí se, zda tokeny patří do stejné třídy pomocí metody *GetIntertokenOrder()* - každá implementace třídy Token musí vracet unikátní hodnotu, např. *IntToken* vrátí 1, *CharToken* vrátí 2, atd. Nejvhodnější je použít číselnou hodnotu enumerace *TokenType*, nebo alespoň zachovat uspořádání dané pořadím v této enumeraci. Pokud do stejné třídy nepatří, pak je relativní pořadí určeno rozdílem těchto hodnot.
2. Pokud patří do stejné třídy, vyhodnotí se pořadí v rámci datového typu. U primitivních typů využijeme prostředků cílového jazyka, u referenčních porovnáme hodnotu ukazatele, u seznamu provedeme lexikografické porovnání nad obsaženými tokeny a všechny instance *nil* jsou považovány za ekvivalentní.

c) Každému tokenu lze zaslat zpráva. K tomu slouží metody *SendMessage() a SimulateMessage()*. Jejich použití bude popsáno v Kapitole 3.2.4. Metoda *SimulateMessage()* je zde přítomná pro ty zprávy, které mají vedlejší efekt změny stavu objektu, kterému jsou zaslány. To je využito při kontrole stráží.

d) Každý token rozumí zprávě „print“[1] - vytištění tokenu na standardní výstup. Pro tuto funkčnost zavedeme metodu *AsText()*

3.1.2 Multimnožiny

Multimnožiny (angl. multisets) mají v návrhu podobnou strukturu jako jako tokeny, jejich význam a účel je však zcela odlišný. Zatímco tokeny reprezentují konkrétní data v konkrétním stavu, multimnožiny definují, jakým způsobem se stav bude měnit. Změna stavu je realizována odebráním nebo přidáním tokenů na místo v Petriho síti.

Multimnožinu tedy použijeme v případě:

- definice přechodu
- definice počátečního značení v místě.

V rámci multimnožin také zavádíme pojem proměnná. Proměnná hodnota slouží v OOPN k navázání hodnoty reprezentované tokenem na jméno, pomocí kterého budeme moci v přechodu k hodnotě přistoupit. Platnost jména proměnné je po celou dobu provádění přechodu, v průběhu provádění přechodu lze však definovat proměnné nové (více v kapitole 3.2.4). Referenční tokeny nelze v multimnožině specifikovat a jediný způsob, jak přemístit referenční token je pomocí proměnné.

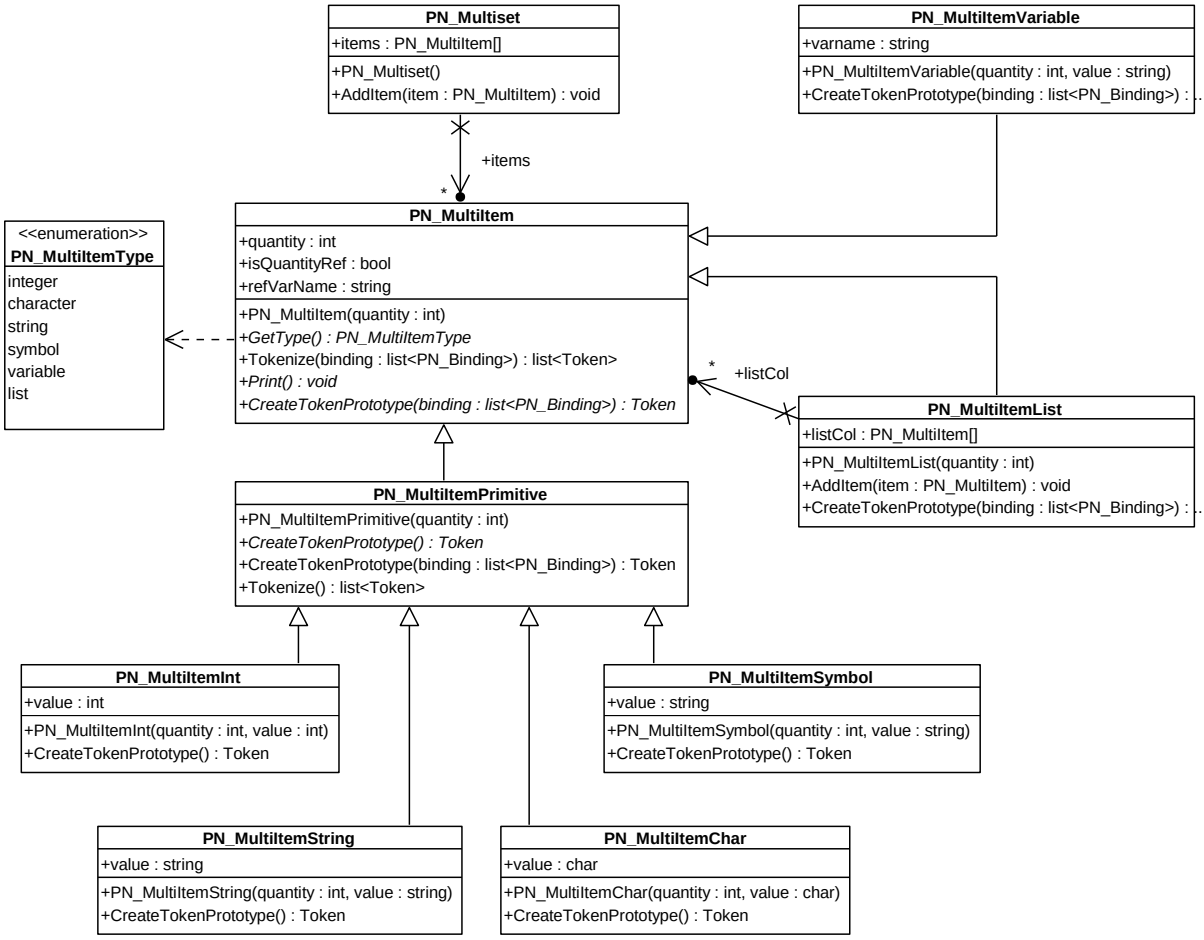
Proměnnou můžeme použít pro prvek množiny, ale také pro kvantifikátor samotný. Proměnná, pokud je nenavázaná, se snaží získat informaci o tom, kolik tokenů odpovídajících prvku multimnožiny se v místě nachází. Přesné chování je popsáno v kapitole 3.2.2.

Na obrázku 5 vidíme schéma pro multimnožiny. Abstraktní třída *PN_MultiItem* představuje jeden prvek multimnožiny. Všimneme si, že právě tato třída definuje atributy týkající se kvantity, a tedy všichni potomci mají kvantitu definovanou také.

1 Zpráva „print“ není definovaná ani v OOPN ani v jazyce PNtalk. Jde o konstrukci používanou čistě pro účely simulátoru zde popisovaného z důvodů demonstrace a testování. Metoda AsText() tedy není pro implementaci nezbytná, nicméně nějaký mechanismus výpisu je nanejvýš vhodný.

Na multimnožině zavedeme operaci *tokenizace*. Tokenizace je operace, kdy každý prvek multimnožiny vytvoří tolik instancí odpovídajících derivátů třídy *Tokenu*, kolik specifikuje jeho kvantifikátor. Proces tokenizace můžeme rozdělit do dvou kategorií:

- Instanciace konstanty
- Instanciace z proměnné



Obrázek 5: Multimnožiny

Výchozí metoda pro tokenizaci je *PN_MultiItem.Tokenize*. Parametrem je seznam všech aktuálních navázání proměnné. Protože instance třídy Token jsou klonovatelné, implementujeme metodu *CreateTokenPrototype()*, která vytvoří jeden prvek, který se pak ve výše zmíněné metodě Tokenize nakonuje na výslednou kvantitu.

Instanciace konstanty

Všechny prvky multimnožiny, které reprezentují fixní hodnotu (konstantu) odpovídají primitivním datovým typům používaných v simulátoru. Tyto prvky obaluje nadtřída *PN_MultiItemPrimitive*. Tokenizace takových prvků je triviální, ale především nevyžaduje seznam navázaných proměnných.

Toho lze využít při hledání všech možných navázání proměnných při hledání proveditelného přechodu (více kapitola 3.2.2).

Instanciace z proměnné

Při tokenizaci prvku multimnožiny, který specifikuje pouze jméno proměnné, budeme procházet všechna navázání proměnných a při shodě jmen proměnných pak klonujeme hodnotu spojenou s proměnnou. Na rozdíl od konstant této varianty nelze využít při hledání proveditelného přechodu, protože navázání proměnných ještě není kompletní.

Pro prvky multimnožiny typu seznam platí stejné pravidlo, protože seznamy mohou obsahovat proměnné.

3.1.3 Místa

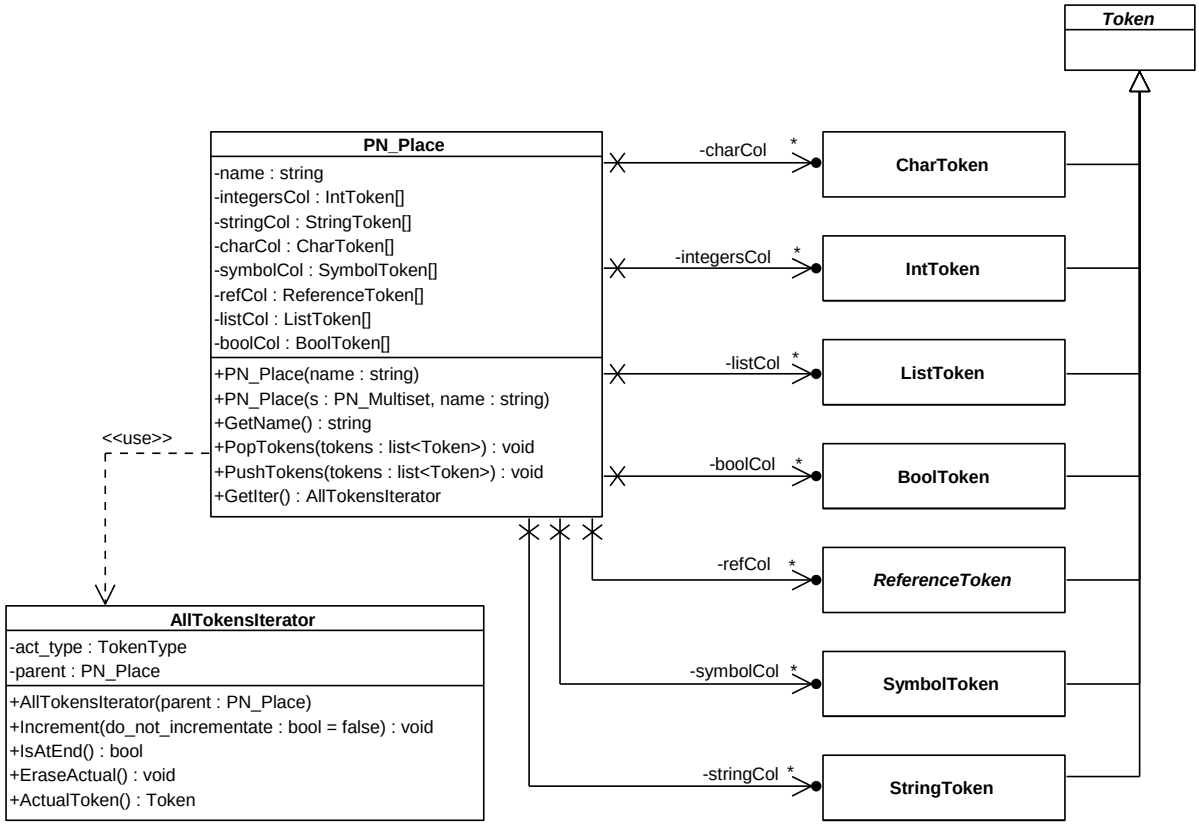
Místo v Petriho síti slouží k uložení tokenů a v simulátoru jej budeme modelovat pomocí třídy *PN_Place*. Každé místo v jazyce PNtalk musí mít jméno. Instance místa v rámci simulace ho vědět nemusí, protože místo bude definováno jako atribut v cílové třídě a právě jméno místa použijeme pro pojmenování tohoto atributu. Přesto je vhodné jméno implementovat i v rámci třídy *PN_Place*, např. pro účely ladění.

Povinné rozhraní místa musí obsahovat metody pro umístění tokenů do místa a odebrání tokenů z místa. Místo může být inicializováno pomocí multinožiny nebo inicializační akce, popř. Obojího. Inicializace multimnožinou je provedena jako proces tokenizace a následného uložení tokenu standardním způsobem. Inicializační akce pak spustí akci pro vytvoření případných navázání proměnných. V rámci diplomové práce není inicializační akce implementována, protože lze vytvořit ekvivalentní model, němž inicializační akci vytvoříme jako přechod z přidaného místa.

Způsob implementace metod pro vkládání a odebírání tokenů se může lišit. Nastíníme však jedno z možných řešení využívající faktu, že nad třídou *Token* existuje částečné uspořádání.

V místě bude pro každý typ tokenu existovat jeden seznam. Po vložení tokenů do místa musí být jednotlivé všechny seznamy tokenů uspořádány. Pro odebírání implementujeme speciální třídu *AllTokensIterator*. Tato třída slouží jako iterátor nad všemi seznamy a musí respektovat částečné uspořádání nad tokeny. Seznamy musí být tedy této třídě viditelné – lze použít buď modifikátor *friend* nebo techniku vnořených tříd (definici třídě ve třídě) popř. jinou techniku, kterou disponuje cílový jazyk. Odebrání pak probíhá tak, že nejdříve seřadíme seznam s tokeny k odebrání, a poté paralelně procházíme seznam tokenů k odebrání a všechny tokeny v místě pomocí iterátoru a v případě shody pomocí iterátoru vymažeme token z místa. Pokud pro některý token k odebrání iterátor doběhne na konec, nelze odebrání uskutečnit.

K tomuto řešení existuje varianta, kdy nepoužijeme oddělené seznamy, ale použijeme jeden seznam, na němž budeme řadit pomocí částečného uspořádání nad tokeny. Při tomto řešení však přicházíme o možnost využít řazení seznamů nad primitivními datovými typy, které může být na výsledné platformě optimalizované, ale také bychom museli řadit po každém vložení seřadit celý seznam. U řazení více seznamů jednak můžeme nedotčené seznamy neřadit, ale i samotná dekompozice přinese úsporu ve složitosti.



Obrázek 6: Modelace místa pomocí oddělených seznamů

3.1.4 Zprávy a operandy

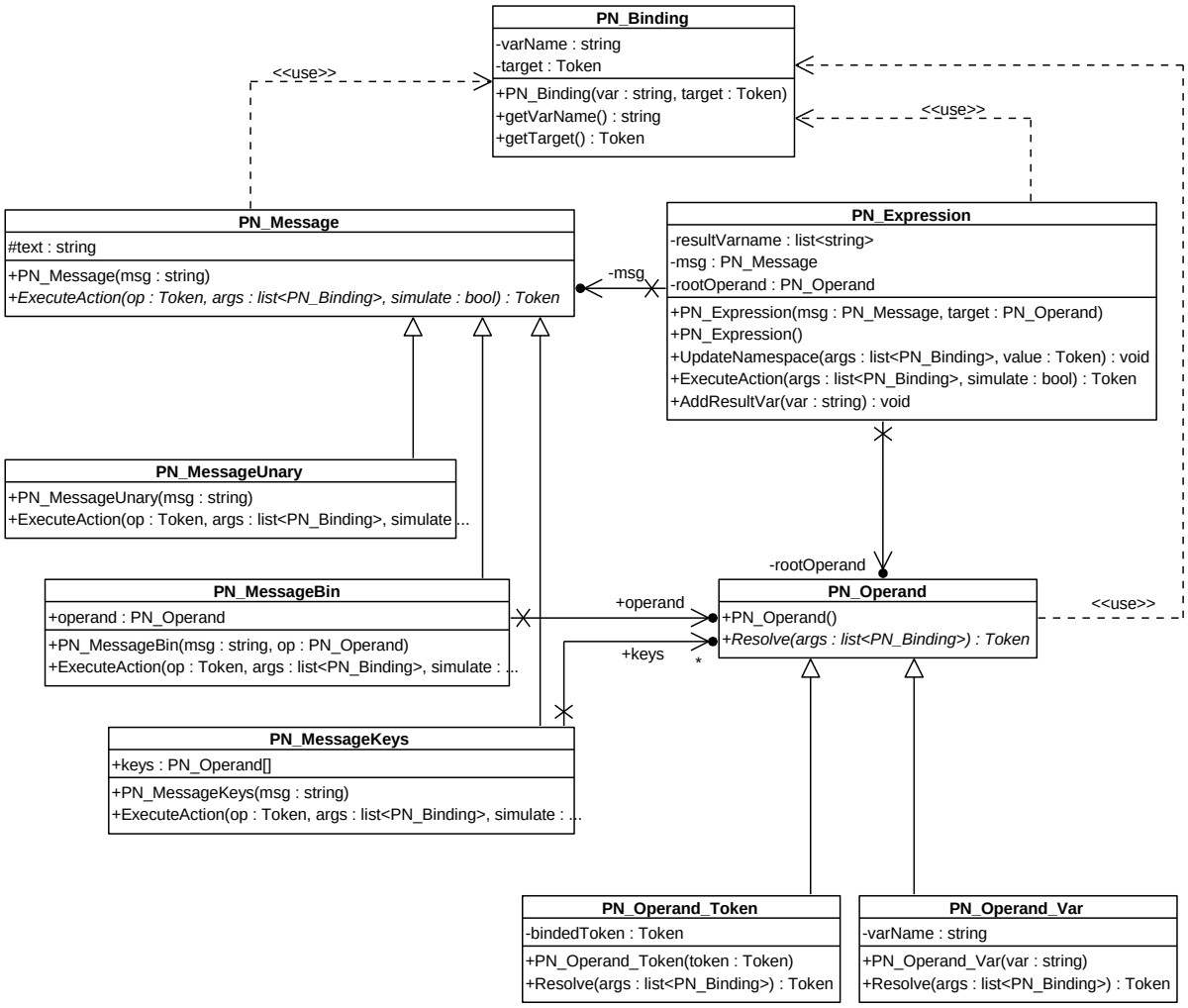
Než začneme modelovat přechody a akce přechodů, musíme definovat, co bude obsahem takových akcí. V jazyce PNtalk jsou akce inspirovány jazykem Smalltalk, kde operace definujeme pomocí zasílání zpráv. V našem simulátoru tedy budeme výpočetní model Smalltalku také simulovat.

Cílem zaslání zprávy je token, tedy instance třídy Token. V kapitole 3.1.1 jsme na třídě *Token* definovali metody *SendMessage* a *SimulateMessage*, potřebujeme ale také způsob, jak definovat zprávy samotné, jak definovat počet argumentů a způsob jejich předání.

Na obrázku 7 vidíme schéma pro definici zpráv. Vidíme zde také třídu *PN_Binding* nesoucí informaci o navázání tokenu na proměnnou. Typicky ji používáme v seznamu jako argument přenášející aktuální navázaní proměnných.

Zprávy jsou buď unární, binární nebo klíčované, viz kapitola 2.4.7. Operandem rozumíme jakýkoli token, který do komunikace vstupuje. Může to být buď cílový token nebo některý argument. Operandy specifikujeme jako:

- konstantní – operand je svázán s konkrétním tokenem – třída *PN_Operand_Token*
- proměnný – operand je svázán se jménem proměnné – třída *PN_Operand_Var*. Pro získání výsledného tokenu je potřeba projít všechna navázání proměnných.



Obrázek 7: Zprávy a operandy

K dosažení polymorfního chování operandu však operaci *Resolve()* definujeme už na předkovi *PN_Operand*. Konstantní operand pak navázání proměnných nebere v potaz a vrací vždy svůj token. Abychom mohli začít definovat zprávy jako součást přechodů, musíme zavést ještě třídu *PN_Expression*, která reprezentuje výraz v Petriho sítích. Výraz se skládá z:

- cílového operandu
- zprávy, jež bude cílovému oprandu zaslána
- množině jmen proměnných, jimž bude výsledná hodnota po vykonání výrazu přiřazena

Výrazy v našem simulátoru mohou specifikovat pouze jednoduché zprávy. Důvodem je složitost implementace zastavení vykonávání akcí v případě volání metod tříd PNtalku (viz kapitola 3.2.4). Gramatika PNtalku stejně jako Smalltalk však definují složené zprávy. Řešením je zavést operaci *redukce složených zpráv,* pomocí které získáme ze složené zprávy ekvivalentní posloupnost jednoduchých zpráv. Tuto operaci pak aplikujeme při generování kódu. Podrobně je tato operace popsána v kapitole 4.3.3

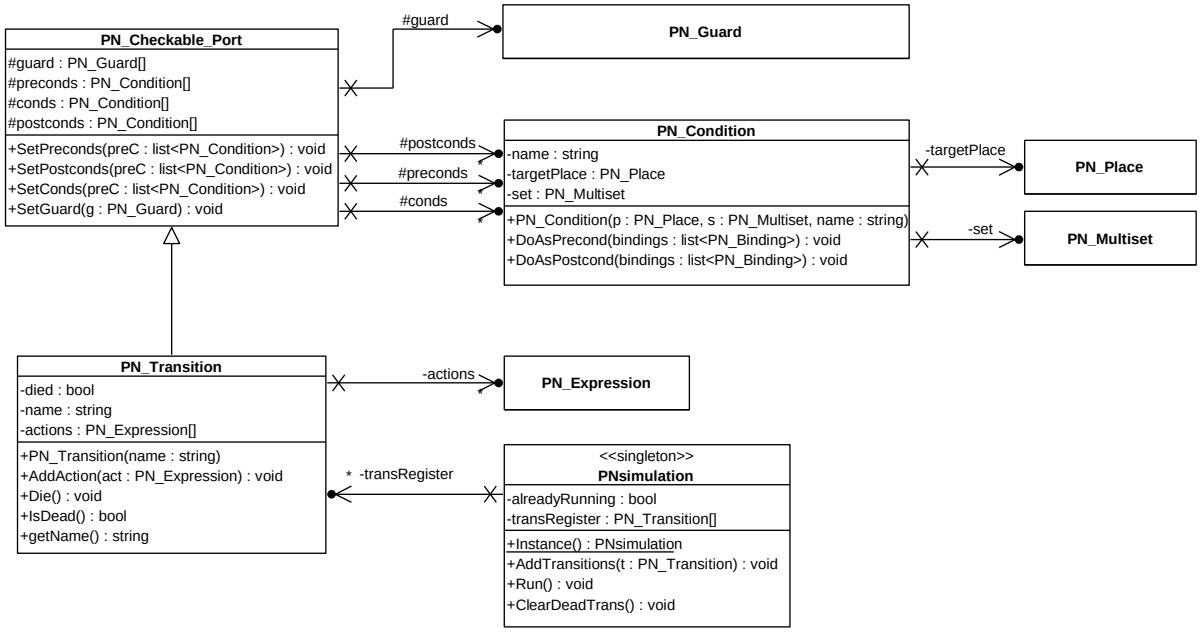
3.1.5 Přechody

V následující kapitole se budeme zabývat přechodem v Petriho sítích. Přechod (angl. Transition) je hlavním nástrojem pro provádění simulace. Zmíníme zatím statickou povahu přechodu, jeho definici, vazbu na ostatní prvky. Dynamickou část, tj. výběr proveditelného přechodu a provedení akce přechodu budeme řešit v kapitolách 3.2.2 a 3.2.4.

Přechod je definován jednak hranami k místům a inskripcí těchto hran a pak akcí/akcemi, jež budou spuštěny v případě vykonání přechodu. V souladu s obecnými Petriho sítěmi [Češ] definujeme tzv. preconds (objekty, které budou z cílového místa odebrány) a postconds (objekty, které do cílového místa uložíme). Navíc definujeme tzv. conds – kontrolujeme, zda objekty v místě jsou, ale není s nimi manipulováno.

Hrany modelujeme pomocí třídy *PN_Condition*, přičemž tuto třídu používáme pro všechny typy hran. *PN_Condtion* specifikuje:

- cílové místo, ke kterému se váže
- multimnožinu, která značí inskripci hrany
- implementuje metody, které realizují efekt hrany – *DoAsPrecond/DoAsPostcond* pro precond/postcond. Hrana typu cond nemá žádný efekt.



Obrázek 8: Přechod, inskripce hran a registr přechodů

Přechod realizujeme pomocí třídy *PN_Transition*, která dědí ze třídy *PN_Checkable_Port*. Třída *PN_Checkable_Port* specifikuje stráž (*PN_Guard* – viz. kapitola 3.1.7), a jednotlivé hrany

přechodu. Pro hrany přechodu platí, že mezi přechodem a místem vede maximálně jedna hrana od každého typu[1].

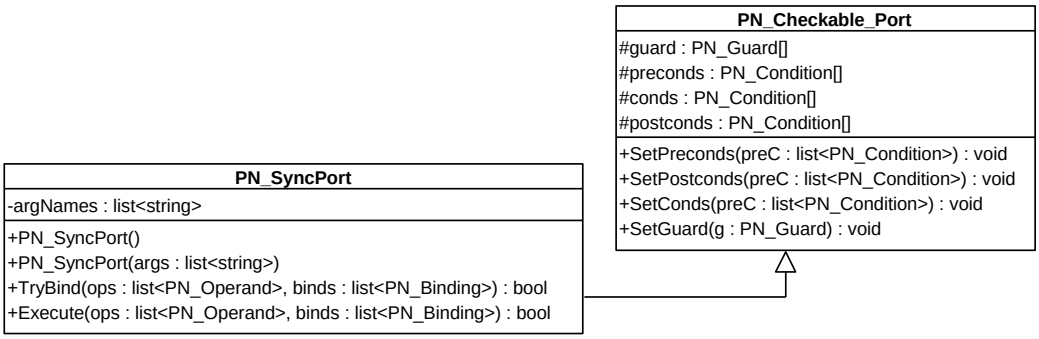
Třída *PN_Transition* pak nese informace o akci přechodu a také příznak, zda přechod aktivní, tedy zda odkazuje na platnou instanci sítě. Pokud přechod přestane být platný, např. při dokončení metody, nebo ztrátě reference na síť, volá vlastník přechodu metodu *Die()*. Simulátor pak před testováním proveditelnosti navíc kontroluje, zda je přechod aktivní[2].

Poslední součástí je tzv. registr přechodů, kterým je simulační kontext sám. V našem systému uvažujeme pouze jediný simulační kontext, který je realizován jako návrhový vzor *singleton,* a spravuje všechny registrované přechody.

3.1.6 Synchronní porty

Synchronní porty mají podobný charakter jako přechody, ale lze je volat explicitně ze stráže přechodu. Navíc disponují možností předat při volání metody argumenty.

Synchronní port má charakter přechodu i metody. V našem simulačním modelu upřednostňujeme charakter přechodu, především z toho důvodu, že synchronní port se účastní procesu hledání proveditelného přechodu a využívá stejné mechanismy při kontrole hran a také v něm lze specifikovat stráž. Proto mají jak přechod, tak i synchronní port stejného předka *PN_Checkable_Port*.



Obrázek 9: Synchronní port

Princip vázání proměnných a vykonání synchronního portu bude vysvětlen v kapitole 3.2.3.

3.1.7 Stráže

Stráž je konstrukce používaná k omezení proveditelnosti přechodů. Pokud máme korektní navázání proměnných, lze pomocí stráže specifikovat dodatečné podmínky, které musí navázání splňovat, aby byl přechod proveditelný.

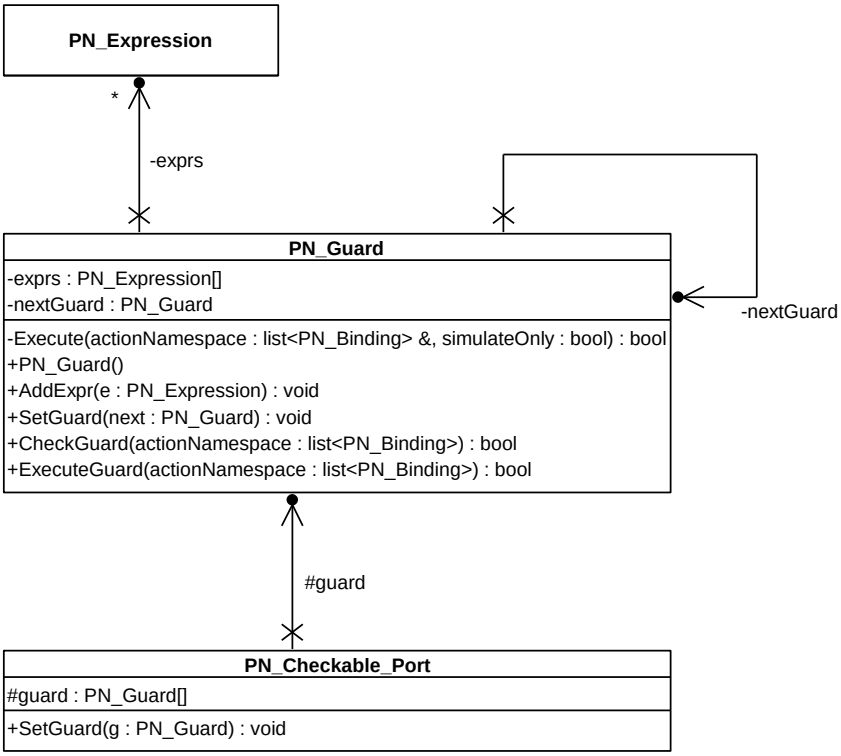
1 Pokud připustíme více hran, bude výsledkem součet multimnožin. Při implementaci hledání proveditelného přechodu musíme pak tuto skutečnost zohlednit. V našem případě však součet hran lze provést při generování kódu a v simulátoru tedy tuto možnost neuvažujeme.

2 Tuto funkčnost realizujeme především pro jazyky, které disponují vlastním garbage-collectorem, a u kterých tedy není zaručeno, kdy dojde k destrukci a uklizení objektu.

Stráž definujeme jako množinu složených výrazů, jejichž výsledkem musí být hodnota typu *bool,*[1] tedy vyhodnotitelná na pravdu či nepravdu. Stráž bude vyhodnocena kladně, pokud logický součin dílčích výsledků je vyhodnocen kladně. Tedy všechny složené výrazy musí být pravdivé.

Vzhledem ke způsobu vyhodnocení stráží si můžeme dovolit stráž modelovat jako posloupnost dílčích stráží. Pokud při vyhodnocení stráže je dílčí stráž vyhodnocena negativně, je i celá stráž vyhodnocena negativně, a vyhodnocení končí.

Pro výraz ve stráži platí stejné podmínky jako v případě akcí, tedy jako výraz můžeme použít pouze jednoduché zaslaní zprávy. Pro modelování složených výrazů, které jsou považovány za dílčí výsledek stráže, použijeme metodu redukce složených zpráv. Dílčí stráž je pak posloupnosti jednoduchých zaslání zprávy a výsledek posledního zaslání je kontrolován na pravdivost.



Obrázek 10: Stráž

Důležitou vlastností stráže je, že jako zaslání zprávy můžeme použít volání synchronního portu. V tomto případě je však nutné počítat s tím že pokud vykonání stráže selže, musí zůstat všechny objekty účastnící se vyhodnocování v tom stavu, v jakém byly před zahájením vyhodnocování. Jinými slovy, testování stráže nesmí mít žádný vedlejší efekt.

Tyto požadavky lze implementovat více způsoby. Prvním je varianta kdy vytvoříme *hluboké kopie* účastnících se objektů, tj. Musely by být zkopírovány všechny instance objektů[2]. Ty by se uchovaly pro případ neúspěchu. Druhou variantou je posílat zprávy tokenům v režimu simulace. V tomto režimu pak případné synchronní porty provedou pouze navázání proměnných, a efekt jejich

1 V naší implementaci se tedy jedná o instance třídy *BoolToken*.

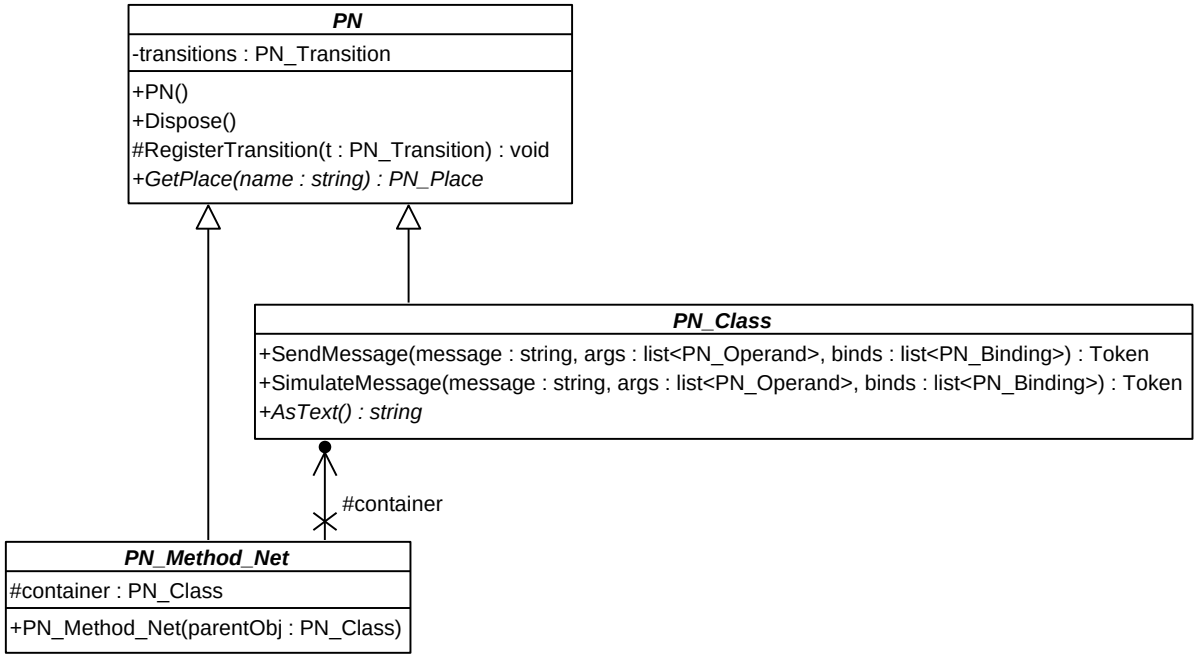
2 V kapitole 3.1.1 jsme definovali požadavek na replikovatelnost tokenu. Tato operace je však *mělká kopie*, tj. v případě referenčních typů vytváříme kopie referencí na instanci, nikoli instance samotné.

hran se provede až po vyhodnocení celé stráže na pravdu. Z důvodů náročnosti a neefektivnosti vytváření hlubokých kopií v naší implementaci volíme druhý přístup.

3.1.8 Petriho sítě a sítě metod

V předchozích kapitolách jsme definovali podobu jednotlivých prvků objektově orientovaných sítí. Nyní se zaměříme na definici sítí samotných. Podle OOPN síť může být definována jako:

- objektová síť – každá třída v OOPN má definovánu objektovou síť, která je vytvořena a aktivována při konstrukci instance této třídy.
- síť metody – třídy definují metody a voláním těchto metod se vytvoří instance sítě metody. Můžeme vytvořit libovolný počet instancí sítí metod, které mohou běžet paralelně.



Obrázek 11: Abstraktní třídy definující Petriho sítě

Na obrázku 11 vidíme systém abstraktních tříd používaných pro definici sítí. Výchozí třída je PN. Třídy PNtalku pak definujeme jako implementaci abstraktní třídy *PN_Class*, každou metodu pak definujeme jako implementaci abstraktní třídy *PN_Method_Net*. Definici sítě pak provádíme v konstruktoru konkrétní třídy, přičemž sledujeme postup definovaný výše. Způsob uložení přechodů a míst je:

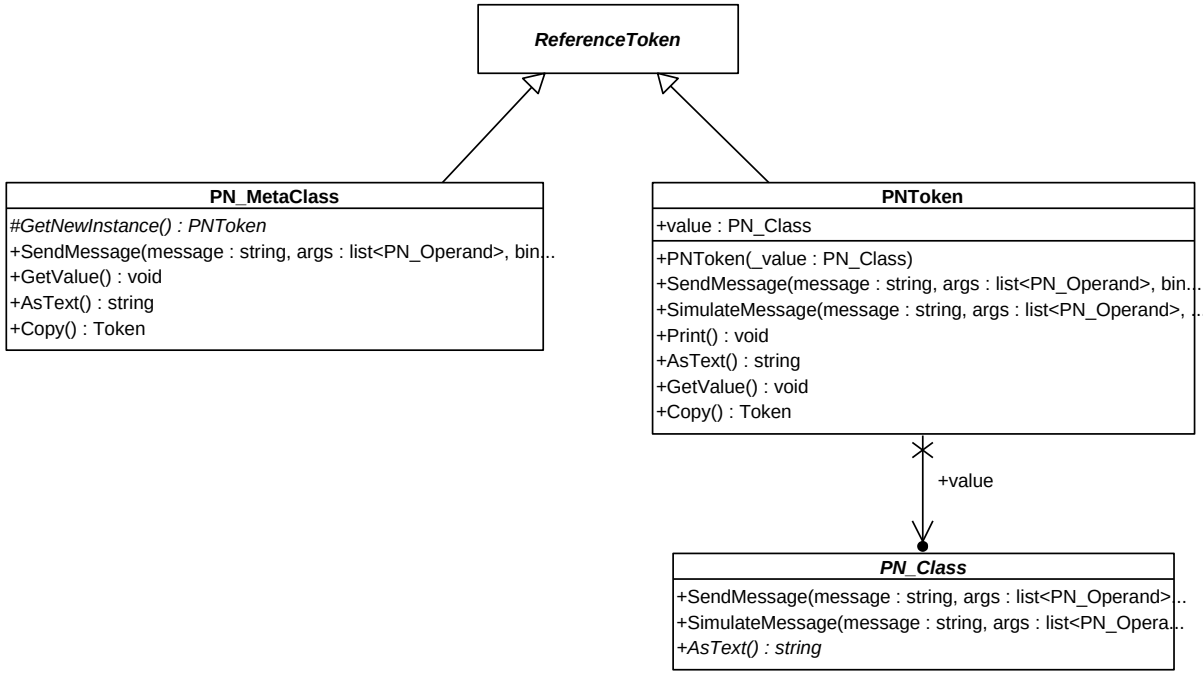
- přechody – instance přechodů ukládáme do připraveného seznamu přechodů. Při konstrukci sítě navíc musíme přechody registrovat v simulačním kontextu. Obojí implementujeme na úrovni PN v metodě *RegisterTransition*. Při uvolnění sítě pak označíme přechody sítě jako *neaktivní* (viz. Kapitola 3.1.5). Implementujeme na úrovni PN v metodě *Dispose*.

- místa – v konkrétních třídách definujeme jako atributy třídy. Místo je pak dostupné ve jmenném prostoru cílového jazyka. Jeho destrukce je provedena při destrukci objektu sítě samotné. Musíme počítat s jednou specialitou, a sice sítě metody mohou přistupovat k místům té instance třídy OOPN, z níž je volána. Z tohoto důvodu všechny třídy definující sítě musí definovat metodu *GetPlace*, která vrátí referenci na místo podle jména.

Metody jsou svázány s třídou (především kvůli dostupnosti míst), ke které náleží a ta je předána v konstruktoru při aktivaci sítě metody. Způsob invokace metody bude popsán v kapitole 3.2.4

3.1.9 Třídy v OOPN

V předchozích kapitolách jsme popsali, jak lze definovat a implementovat jednotlivé prvky jazyka PNtalk. Nyní naznačíme způsob, kterým budeme tvořit třídy jazyka PNtalk. Bude platit, že pro jednu třídu PNtalku[1] definujeme ve výsledném jazyce jednu třídu a navíc i příslušnou metatřídu. Výsledná třída dědí z bázové třídy *PN_Class* a metatřída z třídy *PN_MetaClass*.



Obrázek 12: Třída a metatřída

Metatřídy musíme modelovat, protože od systému vyžadujeme implementaci volání konstruktoru výsledných tříd, tedy způsob, jak vytvářet nové instance. V jazyce *PNtalk* tuto akci provedeme zasláním zprávy *new* metatřídě příslušící výsledné třídě. Příklad takové akce je:

```
o:= C1 new.
```

C1 se vyhodnotí jako metařída, protože v jazyk PNtalk vyžaduje, aby pojmenování tříd začínalo velkým písmenem, zatímco pojmenování proměnných začíná malým písmenem.

1 Myšleno třídu definující síť , tj. dědící z bázové třídy PN jazyka PNtalk

Metatřídu ve výsledném systému modelujeme jako referenční token. Nicméně metatřída je token pouze z důvodu implementace rozhraní pro příjem zpráv a kvůli tomu, že je tak interpretován gramatikou PNtalku. Token nesoucí metatřídu by však nikdy neměl opustit akci přechodu, v němž je volán. To v našem případě zajistíme implementací generátoru kódu.

Jak již bylo naznačeno, metatřída jako objekt rozumí jediné zprávě – new. Na tuto zprávu reaguje tak, že zkonstruuje objektu příslušné třídy a vrátí jej jako referenční token. Pro obalení reference na instanci třídy *PNtalku* slouží třída *PN_Token*.

3.1.10 Třídy pro podporu Smalltalku

V jazyce PNtalk je umožněno používat vybrané třídy Smalltalku pro podporu programování akcí přechodů. Vzhledem k tomu, že implementace původního simulátoru je v prostředí Pharo a je tedy na Smalltalku postavená, nebyl to větší problém. Při transformaci do jiného jazyka však musíme dodat podporu pro tyto třídy explicitně. Řešení převodu nastíníme na třídách Smalltallku *Transcript* a *OrderedCollection*. Základní postup, jak vytvořit podporu, je nalézt ekvivalentní konstrukce nebo třídy v cílovém jazyce a vytvořit k nim obalovací třídy, které umožní jejich použití v simulátoru. Na příkladech pak popíšeme převod do jazyka C++.

Třída *Transcript* slouží jako standardní výstup. Jejím ekvivalentem je v C++ standardní výstup *cout*. Podporované zprávy jsou:

Smalltalk	popis	C++
show: 'text'	zobrazí text na standardní výstup	cout << 'text'
lf	vloží znak '\n'	cout << endl

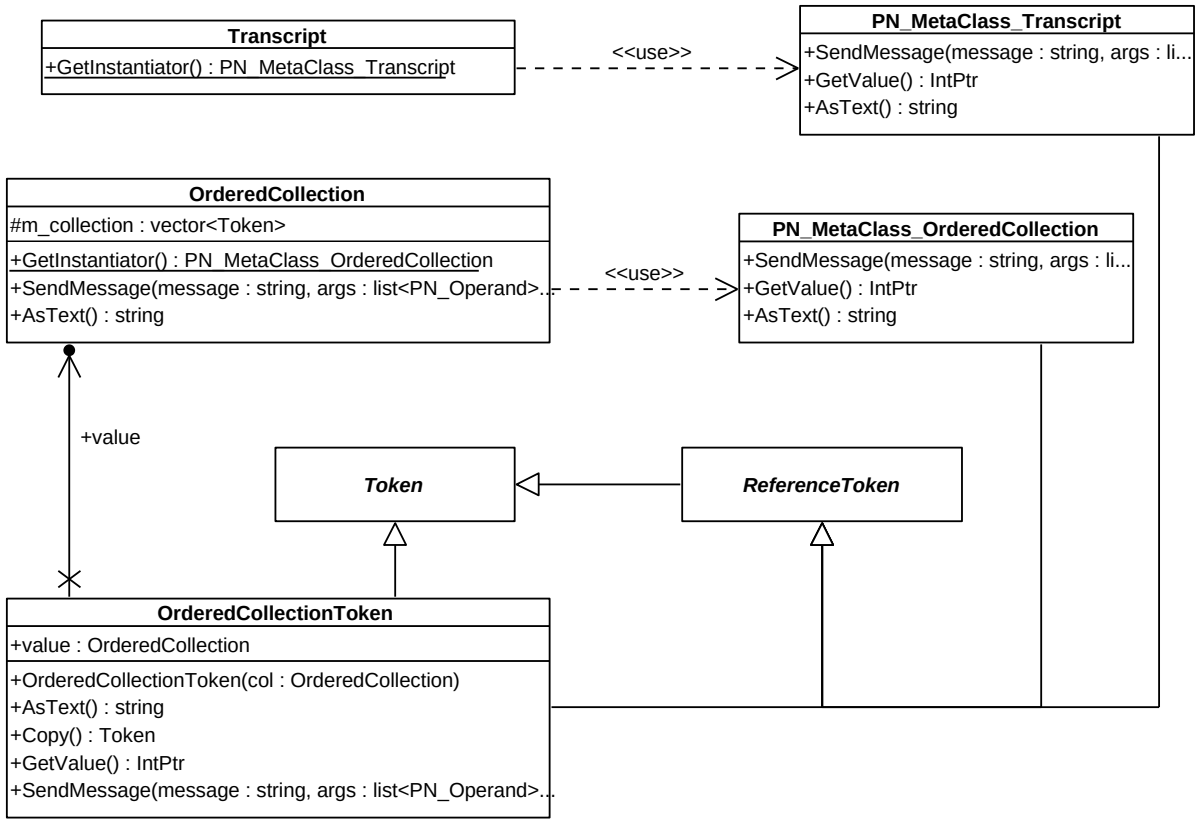
Třída *OrderedCollection* reprezentuje indexovaný seznam, přičemž do ní můžeme vkládat objekty libovolného typu. Kolekce je uspořádaná, nikoli seřazená; automaticky řazená kolekce by byla *SortedCollection*, tu však nepodporujeme. V C++ pro tyto účely použijeme třídu *std::vector<Token*>*. V cílovém systému omezíme obsah na jakýkoli token. Některé podporované zprávy jsou:

Smalltalk	popis	C++
add: item	vloží prvek	vector.push_back(item)
addFirst: item	vloží prvek na začátek	vector.insert(vector.begin(), item)
addLast: item	vloží prvek na konec	vector.push_back(item)
at: index	vrací prvek na daném indexu	vector[index]
at: index put:item	vloží prvek na zadaný index	vector[index] = item
RemoveAt: index	zruší prvek na daném indexu	vector.erase(vector.begin() + index)

Tyto řešení jsou pouze principiální a musí být implementována v příslušných metodách *SendMessage* na třídách které budou konstrukce reprezentovat. Na obrázku vidíme takové schéma pro naše třídy.

Definujeme třídy *Transcript* a *OrderedCollection*, které budou mít statické metody *GetInstatiator()* pro získání tokenu reprezentující odpovídající metatřídu. Všimneme si, že třída *Transcript* definuje pouze statické operace, proto ji budeme modelovat jako statickou, tedy neumožníme její instanciaci, a výše zmíněné zprávy implementujeme přímo na metatřídě.

U třídy *OrderedCollection* naopak chceme, aby byla instanciovatelná. Na metatřídě implementujeme odezvu na zprávu *new*. Ta bude vytvářet instance třídy *OrderedCollection* a vracet instanci třídy OrderedCollectionToken, která bude sloužit k přenášení kolekce v rámci Petriho sítě. Tato třída dědí ze třídy *ReferenceToken* a bude s ní zacházeno obdobně jako se třídou *PNToken*, která stejným způsobem přenáší instance tříd Petriho sítí.



Obrázek 13: Podpora tříd Smalltalku

3.2 Dynamika simulátoru OOPN

V následujících kapitolách se budeme zabývat implementací dynamiky simulátoru. Popíšeme simulační krok, specifikujeme způsob výběru proveditelného přechodu a popíšeme akci přechodu, a

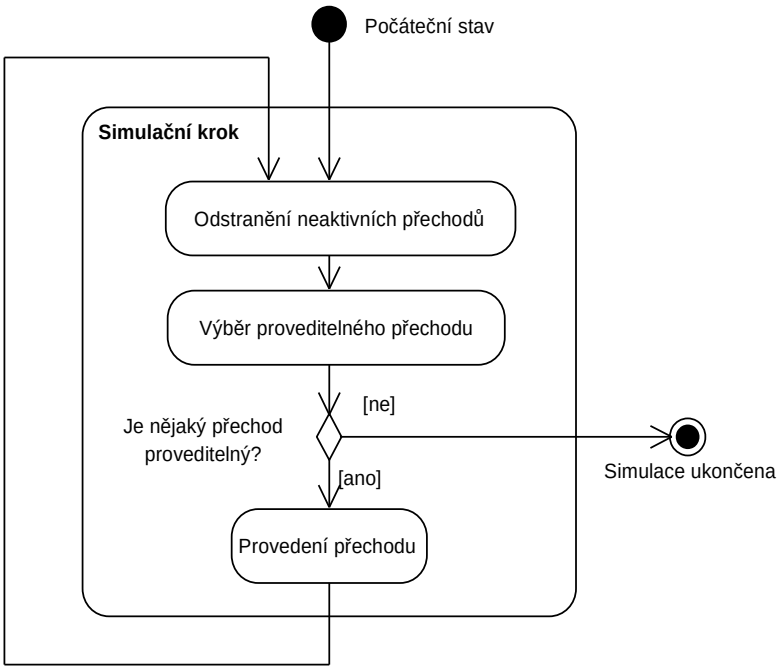
všechny varianty jejího vykonání. Kapitola předpokládá znalost předchozí kapitoly, tedy jakým způsobem modelujeme prvky jazyka

3.2.1 Inicializace a simulační krok

Simulátor běží ve smyčce, a v každé iteraci provede množinu operací, kterou budeme nazývat simulační krok. Jako jádro simulátoru nám slouží statická třída *PN_Simulator*. Tato třída funguje zároveň jako simulační kontext – spravuje všechny aktivní přechody.

Simulační kontext je třeba nejdříve inicializovat. Protože v implementaci uvažujeme pouze jeden možný simulační kontext, je inicializace přímočará – stačí registrovat přechody výchozí třídy. V kapitole 3.1.9 jsme popsali způsob definice prvků sítě a v rámci něho jsme požadovali, aby registrace přechodů byla provedena automaticky v konstruktoru třídy. Inicializace simulačního kontextu tedy spočívá ve vytvoření instance výchozí třídy.

Třída *PN_Simulator* má metodu *Run*, která slouží ke spuštění simulace. Implementace simulace je simulační smyčka, znázorněná na obrázku 14. V naší implementaci se omezujeme na deterministický simulátor, takže operace výběru proveditelného přechodu testuje přechody, dokud nenalezne první proveditelný přechod. Abychom dosáhli nedeterministického chování, museli bychom nalézt všechny proveditelné přechody a z nich pak náhodně jeden vybrat.



Obrázek 14: Simulační krok

Akci odstranění lze implementovat buď ve smyčce tak, jako na obrázku 14, variantou je však rušit neaktivní přechody při úklidu instancí Petriho sítí. Při této variantě však musíme mít jistotu, že k úklidu dojde včas po uvolnění instance, jinými slovy závisí, jakým způsobem je implementován garbage-collector.

3.2.2 Výběr proveditelného přechodu

Hledání proveditelného přechodu je úkol, při němž se snažíme pro přechod T nalézt takovou sadu S navázání proměnných na konkrétní tokeny, že:

- všechny proměnné, které se vyskytují ve vstupních a testovacích hranách přechodu T jsou obsaženy v sadě S
- pro všechny multimnožiny definující vstupní a testovací hrany přechodu T je v cílových místech dostatek tokenů (včetně navázaných proměnných)
- každá proměnná se v sadě S vyskytuje právě jednou; pokud se proměnná stejného jména vyskytuje ve více hranách přechodu T, musí být navázání totožné
- pokud přechod T specifikuje stráž, musí být vyhodnotitelná pro S kladně

Nyní nastíníme možný způsob řešení předchozích požadavků, tak jak jsou řešeny v rámci implementovaného simulátoru. Postup budeme popisovat od dílčích částí k celku. Jednotlivé úkoly pak budou:

- generování stavového prostoru všech možných navázání proměnných v rámci jedné hrany
- nalezení navázání pro jednu hranu
- sjednocení navázání proměnných různých hran
- kontrola stráží

Pro účely generování možných navázání pak zavedeme pojem „částečné navázání hrany“, což bude stav indikující, které proměnné byly navázány na které tokeny, ponese zatím nezpracovanou část multimnožiny definující hranu a obsah místa po odebrání tokenů z již zpracované části multimnožiny. Obsah místa budeme nazývat dle definice Petriho sítí značení. Při zpracování multimnožiny musíme zohlednit i fakt, že kvantifikátor prveku může být definovaný proměnnou. Mohou nastat dva případy:

1. Proměnná kvantifikátoru není navázána – proměnná se naváže na číselnou hodnotu značící, kolik tokenů příslušící prvku multimnožiny se v místě nachází.
2. Proměnná kvantifikátoru není navázána, navíc prvek obsahuje stejnou proměnnou – v takovém případě vyhodnotíme nejdříve prvek, až poté kvantifikátor.

Prohledávání začínáme s aktuálním značením místa v rámci stavu Petriho sítě, kompletní multimnožinou definující hranu a prázdnou množinou navázaných proměnných. Pro aktuálně zpracovávaný prvek multimnožiny a aktuální stav postupujeme takto:

1. Prvek odebereme z multimnožiny
2. Pokud je kvantifikátor proměnná, zjistíme zda je navázaná. Pokud ano, zkontrolujeme, zda je hodnota číselná. Pokud ne, je hodnota automaticky nulová a pokračujeme dalším prvkem mulimnožiny
3. Upravíme značení místa aktuálního stavu jedním z následujících způsobů:

a) Pokud je prvek primitivní token, pak z místa odebereme příslušný počet tokenů. Pokud je kvantifikátor nenavázaná proměnná, navážeme proměnnou na číselnou hodnotu odpovídající počtu tokenů v místě a odebereme je všechny.
b) Pokud je prvek proměnná, pak pro všechny možná navázání tokenů na proměnnou v místě vytvoříme nové stavy. Stav bude obsahovat značení místa bez navázání proměnné[1], seznam proměnných včetně nového navázání a aktuální stav nezpracované multimnožiny. Stavy pak uložíme na zásobník stavů. Pokud je nové navázání v konfliktu s některým z předchozích navázání, pak tento stav ignorujeme. V případě, že kvantifikátor je nenavázaná proměnná, postupujeme při jeho navazování obdobně jako ve variantě a) s tím, že přednost má navázání proměnné prvku, až poté navazujeme kvantifikátor.
c) Pokud je prvek seznam, pak kontrolujeme v místě všechny seznamy a případnou korektnost navázání proměnných uvnitř seznamu, čili zda není v konfliktu s již nalezeným navázáním. Pokud seznam odpovídá, odebereme jej z místa s respektem na kvantitu prvku. Všechna nová navázání proměnných přidáme do seznamu navázání. Pokud je kvantifikátor nenavázaná proměnná, řešíme její navázání stejně jako ve variantě b)

Pokud nelze splnit bod 2., čili v místě není dostatek tokenů, pak je stav označen jako nevalidní a v dalším prohledávání již nepokračujeme. V takovém případě se vrátíme poslednímu vygenerovanému stavu částečného navázání. Navázání proměnných pro jednu hranu je kompletní ve chvíli, kdy nezpracovaná část multimnožiny je prázdná. Pokud vyčerpáme všechny vygenerované stavy, pak na hraně nelze nalézt navázání.

Algoritmus bere v úvahu existující navázání proměnných, proto můžeme sjednocení navázání na více hranách implementovat, tak že algoritmus bere v úvahu nejen navázání v rámci své hrany, ale i navázání u již zpracovaných hran. Postupujeme následovně:

- Hrany uspořádáme[2]
- Pro každou hranu udržujeme zásobník jejích vygenerovaných stavů.
- Pokud nalezneme navázání pro jednu hranu, pokračujeme hledáním na následující hraně s tím, že jako výchozí stav použijeme nalezené navázání.
- Pokud nelze na hraně nalézt navázání, vrátíme se k předchozí hraně a pokračujeme jejím posledním vygenerovaným stavem. Pokud žádný nemá, vrátíme se předposlední hraně atd.

Tímto způsobem postupujeme, dokud nenalezneme navázání na poslední hraně. Pak přistoupíme k případné kontrole stráže. Pokud je stráž vyhodnocena negativně, vracíme se poslednímu vygenerovanému stavu.

1 Ve skutečnosti vytváříme v implementaci kopie míst. Pokud implementujeme garbage-collector, je vhodné u referenčních tokenů optimalizovat počet registrací tokenu. Viz. kapitola 3.3.1.

2 Uspořádaní může být libovolné, v implementaci bereme hrany v pořadí, v jakém jsou definovány.

3.2.3 Stráže a synchronní porty

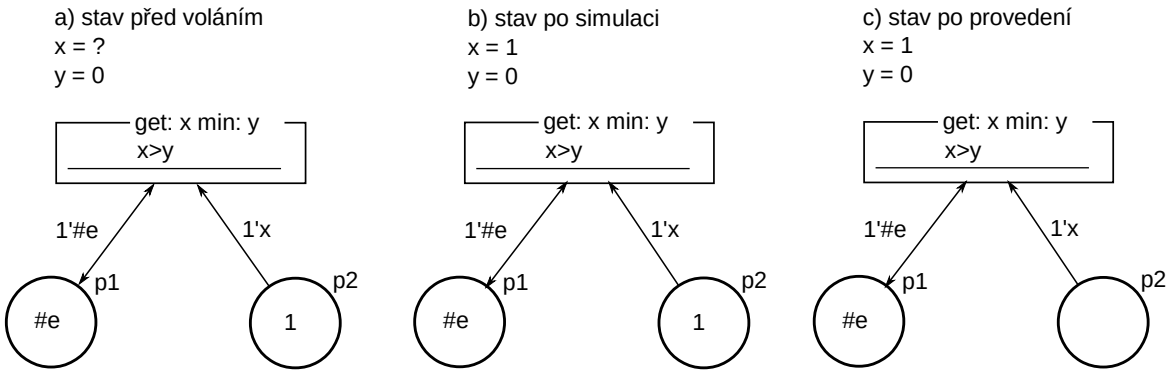
Stráž přechodu je konjunkce booleovských výrazů, která musí být platná, aby byl přechod proveditelný. Strukturu stráží jsme popsali v kapitole 3.1.7. Nyní se bude zabývat jejím provedením.

Protože dílčí výsledky stráže jsou v konjunkci, musí platit zároveň. Testujeme je však sekvenčně, tudíž musíme dbát na to, aby při testování stráže nedošlo k vedlejšímu efektu, který bychom nebyli schopni vrátit v případě, že by stáž selhala v některé své další části. Proto mají všechny *tokeny* možnost nejen zprávu přijmout, ale také ji pouze simulovat.

U tokenů nesoucí datová primitiva nemusíme simulaci implementovat, pokud efekt všech zpráv bude jiný token než adresát zprávy a ponese hodnotu výsledku. Takové chování je přirozenou vlastností běžných deklarativních jazyků, a platí to i pro Smalltalk a proto ho v naší implementaci také dodržíme.

Zajímavější je však vedlejší efekt na tokenech, jež jsou referencí na objekty, především pak Petriho sítě. V rámci naší implementace se soustředíme na vedlejší efekt synchronního portu. Do synchronního portu může vstupovat i nenavázaná proměnná, v takovém případě pak synchronní port hledá i navázání takové proměnné. Pokud takové navázání nelze najít, bude výsledek vykonání synchronního portu *BoolToken* s hodnotou *false*.

Na obrázku 15 vidíme postup, jakým synchronní port vyhodnotíme. Jako příklad poslouží port *get: x min: y*, který z místa *p2* odebere token splňující podmínku, že je větší než druhý parametr portu. Proměnná x bude před voláním portu nenavázaná a dojde k jejímu navázání na odebíraný token. To vše pouze pokud je v místě *p1* symbol *#e.*



Obrázek 15: Vázání proměnné v synchronním portu

Všimněme si, že synchronní port může sám obsahovat stráž. Nemá však tělo a nesmí v něm být volána metoda, protože synchronní port vykonáváme atomicky. V rámci simulace pouze provedeme navázání proměnných, využijeme algoritmus popsaný v kapitole 3.2.2 s tím, že jako výchozí navázání použijeme navázané argumenty portu (v příkladu proměnná *y = 0*). Algoritmus respektuje existující navázání a naváže všechny ostatní proměnné na hranách. Připomeňme, že součástí algoritmu je i kontrola případné stráže a platí to i v případě synchronního portu.

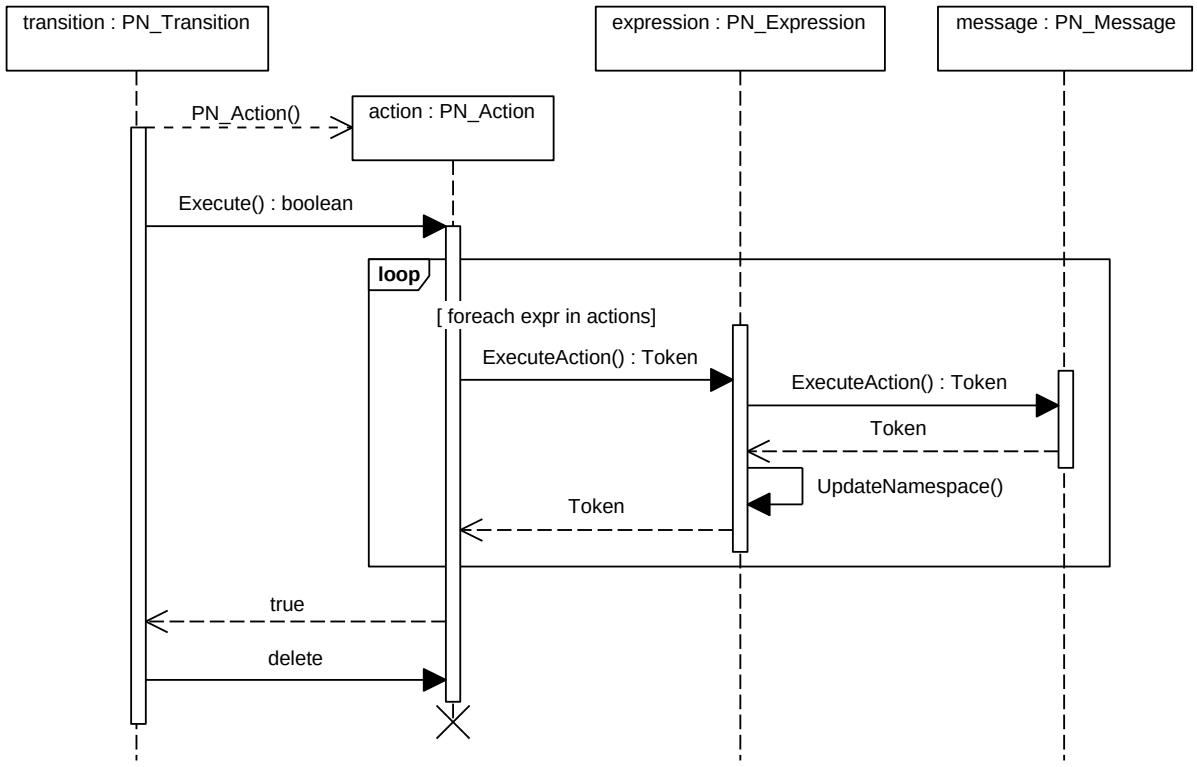
Ve chvíli, kdy jsou všechny výrazy konjunkce stráže volající synchronní port pravdivé, musíme ještě synchronní port vykonat. Při vykonání již pracujeme s navázanými proměnnými musíme provést efekty vstupních hran a provést stráž pro případ, kdy má stráž vedlejší efekt.

3.2.4 Zaslání zprávy a akce přechodu

Akci přechodu jsme definovali jako posloupnost jednoduchých výrazů, jež jsou v jazyce PNtalk jednoduché zaslání zprávy. V souladu s definicí OOPN a PNtalku [Jan] rozlišujeme :

- Událost typu A (atomic) - klasické provedení výrazů. A-událost může nastat tehdy, když adresátem zprávy v akci proveditelného přechodu je primitivní objekt nebo je akce prázdná.
- Událost typu N (new) - vytvoření nového objektu v rámci provedení přechodu. N-událost může nastat tehdy, když adresátem zprávy v akci proveditelného přechodu je třída a selektorem je zpráva *new*.
- Událost typu F (fork) – jeden z vykonávaných výrazů je předání zprávy, tedy vytvoření instance sítě metody. F-událost může nastat tehdy, když neprimitivní objekt, který je adresátem zprávy specifikované v akci proveditelného přechodu, má implementovanou metodu pro tuto zprávu.
- Událost typu J (join) – akceptování výsledku zprávy při současném zrušení instance sítě metody.

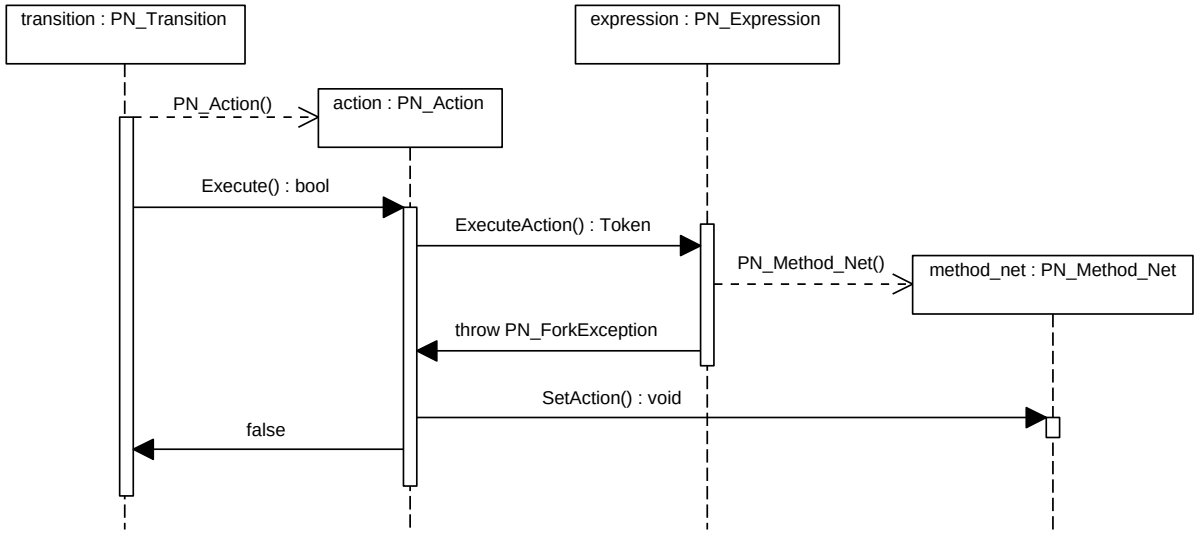
Původní definice těchto událostí předpokládá, že nastane právě jedna možnost a kvůli tomu omezuje akci přechodu na jediné zaslání zprávy. V naší implementaci připouštíme v akci sekvenci výrazů. Typy událostí tedy vztáhneme na jednoduché zaslání zprávy.



Obrázek 16: Smyčka s atomickými událostmi

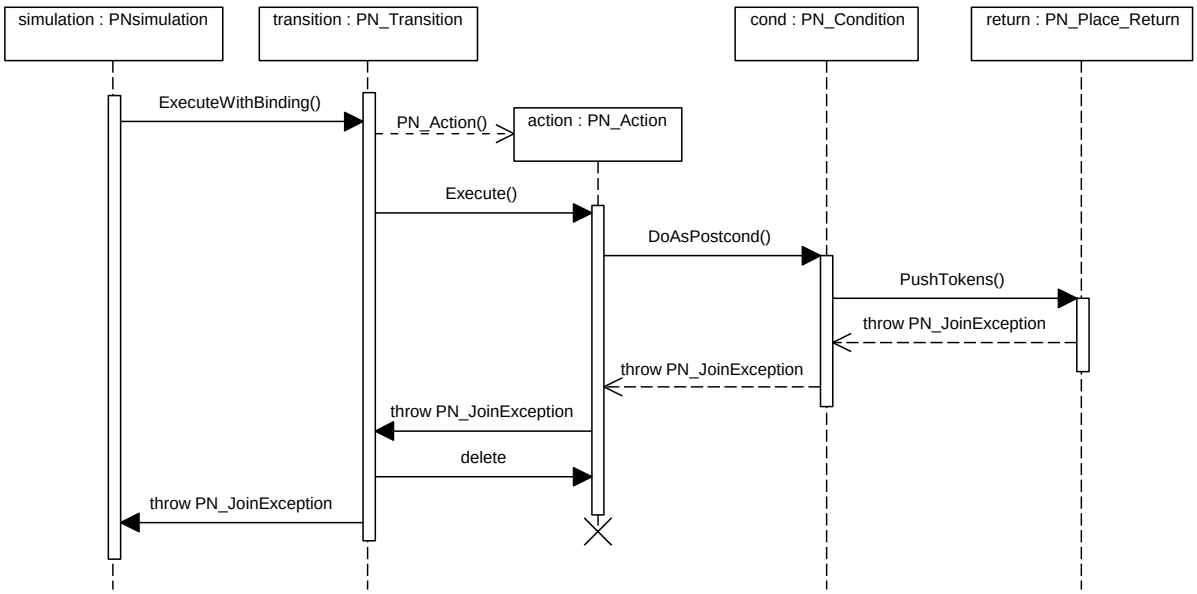
Přikládáme implementaci všech typů událostí. Události typu A a N jsou implementovány identicky. Třída *PN_Action* je instance akce, v rámci událostí typu A a N je instance omezena na dobu vykonávání události.

Na obrázku 16 vidíme ideální běh přechodu, pokud jsou všechny události atomické. Událost typu N budeme považovat za atomickou, protože parametrizovaný konstruktor nepřipouštíme, viz kapitola 3.3.4. Na obrázku je také vidět smyčka vykonávající všechny atomické výrazy. Následující diagramy v této kapitole již smyčku neobsahují, protože je považujeme za speciální případ vyhodnocení výrazu.



Obrázek 17: Událost typu F

Událost typu F vytváří síť metody. Kontext akce získá referenci na novou síť metody pomocí vyjímky typu *PN_ForkException*. Třída *PN_Expression* nevolá konstruktor sítě metody přímo, síť metody se konstruuje ve zpracování zprávy na příslušné třídě – viz. metoda *SendMessage*, kapitola [doplnit]. Pro demonstraci běhu je však volání na obrázku 17 tímto způsobem zjednodušena.



Obrázek 18: Odchycení události typu J

Důležité je, že instance akce *PN_Action* zůstává aktivní a je svázána s instancí sítě metody. Instance akce v tuto chvíli nese informaci o tom, na kterém výrazu akce přechodu byla metoda invokována a po návratu z metody můžeme tedy ve vykonávání akce pokračovat.

Pro návrat z metody slouží událost typu J. Každá síť metody definuje speciální místo *return*, které, když do něj umístíme *token*, tak invokuje událost typu J. V implementaci je jeho ekvivalentem třída *PN_Place_Return*, která speciálním případem třídy *PN_Place*. Při umístění tokenu do tohoto místa je vyhozena výjimka *PN_JoinException*.

Tato výjimka nese informaci o akci, která původně metodu invokovala a také o návratovou hodnotu. Vyjímka je postupně vyhazována až na úroveň simulačního kontextu, který zpracovává simulační krok. Díky tomu simulátor ví, že po provedení přechodu došlo k návratu z metody a pokračuje ve vykonávání akce, která metodu invokovala. Je důležité si uvědomit, že tato akce také může být vykonávána v rámci metody a tuto metodu ukončit. Taková situace je znázorněna na obrázku 18.

3.3 Problémy a omezení simulátoru

V následujících kapitolách prodiskutujeme identifikované problémy, které plynou z popisované implementace. Jde především o zdůraznění rysů a možností jazyka PNtalk, které umožňuje. Tyto okrajové možnosti nejsou implementovány z důvodů jednoduchosti. Vycházíme z toho, že tyto rysy lze doimplementovat, nebo se problematickému chování lze vyhnout vytvořením ekvivalentního modelu.

3.3.1 Garbage-collector

V současné implementaci není garbage-collector. Navrhneme však postup pro jeho implementaci. Nejjednodušší implementace garbage-collectoru je tzv. reference-counting [5]. V případě našeho simulátoru je implementace následující:

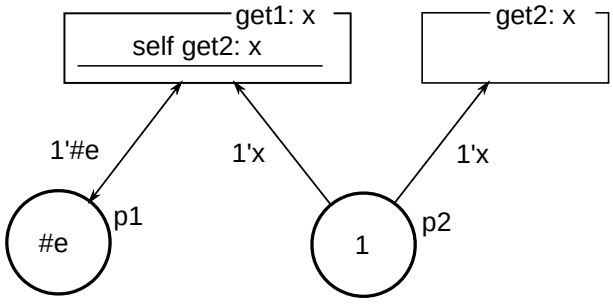
1. Zavedeme třídu PN_GC, která bude splňovat návrhový vzor singleton.
2. Obsahem této třídy bude seznam dvojic, kde první člen je číselná hodnota ukazatele a zároveň klíč záznamu, druhý pak počet aktivních referencí.
3. Třída bude mít metody *IncreaseCounter(IntPtr)* a *DecreaseCounter(IntPtr)* které, budou zvyšovat a snižovat počet referencí. Pokud při zvýšení není klíč v seznamu, přidá ho, pokud při snížení reference klesne na nulu, zavolá na odkaz destruktor.
4. Volání těchto metod zakonponujeme do konstruktoru a destruktoru třídy *ReferenceToken*
5. Zkontolujeme, že jsou na kritických místech zániku instance *ReferenceToken* korektně rušeny, především jde o úklid tokenů po provedení akce (rušíme ty tokeny, které nejsou součástí žádné výstupní hrany)

Tato implementace neřeší cyklus, tedy situaci, kdy dva objekty mají na sebe vzájemně referenci. Objekty nelze uvolnit, přestože nejsou z kořene dostupné, protože díky vzájemné referenci neklesne čítač nikdy na nulu. Pro řešení takové situace musíme zvolit některý ze složitějších algoritmů (např. *mark and sweep* [5])

Při implementaci je vhodné se zamyslet nad složitostí prováděných operací. Seznam dvojic by měl být implementován jako vyhledávací strom a při hledání proveditelného přechodu je vhodné tyto operace neprovádět, jelikož počet generovaných stavů může být vysoký. Řešením je zvýšit počitadlo až po nalezení proveditelného přechodu.

3.3.2 Vnořený efekt synchronního portu

Synchronní port umožňuje definovat vlastní stráž. Simulátor popsaný v této kapitole však neřeší situaci, kdy volání stráže má vedlejší efekt (typicky synchronní port) a ten je v konfliktu s efektem volajícího synchronního portu. Situace je znázorněna na obrázku 19.



Obrázek 19: Vnořené synchronní porty

Vidíme, že synchronní port *get1: x* bude odebírat navázaný *token* z místa *p2* a ve stráži pak volá další synchronní port *get2: x*, který má stejný efekt. Ve fázi simulace proběhne vše v pořádku,

protože navázání *x = 1* je jediné možné. Při vykonávání však dojde ke konfliktu, protože bychom token *1* z místa *p1* chtěli odebrat dvakrát - jednou portem *get1* a v rámci volání stráže pak portem *get2*.

Podobný problém nastane, pokud z volání synchronních portů vytvoříme cyklus. Stejně tak můžeme dojít ke konfliktu i v případě konjunkce výrazů stráže.

3.3.3 Dědičnost

Z hlediska dědičnosti je podpora simulátoru uspokojující. Pokud podědíme, tím, že místa jsou navržena jako *protected,* budou dostupná i v potomcích. Pro definici nových míst, přechodů, metod a portů je podpora nativní, jelikož využíváme polymorfismu cílového jazyka; při zaslání zprávy třída jí buď rozumí a zpracuje nebo ji deleguje na rodiče. Vznikají dvě problematické situace, které simulátor neřeší. Obě jsou způsobeny tím, že zpravidla je zavolán nejdříve konstruktor rodiče až poté potomka. Jsou to:

- inicializace předefinovaného místa – pokud u místa předefinováváme inicializaci, budou v implementaci spuštěny obě. Musíme tedy nejdříve místo korektně uvolnit, včetně jeho obsahu.
- Předefinování přechodu – budou existovat dvě instance přechodu, rodiče i potomka. Při registraci přechodu potomka je tedy nutné nejdříve odregistrovat přechod rodiče.

3.3.4 Omezení

V simulátoru nejsou z důvodů zjednodušení implementovány následující možnosti:

- parametrizovaný konstruktor – jelikož parametrizovaný konstruktor je v jazyce Smalltalk implementovaná jako zaslání zprávy *new* a následném zaslaní zprávy invokující metodu, můžeme chování takového konstruktoru simulovat stejným postupem.
- datový typ float – typ number definovaný jazykem PNtalk (popř. Smalltalk) je tedy omezen na celá čísla.

4 Generátor kódu

V následující kapitole nastíníme podobu výsledného programu. Jako vstup pro generátor kódu bude sloužit zdrojový kód modelu zapsaný v textové verzi jazyka PNtalk. Budeme brát ohled na fakt, že chceme docílit možnosti generovat výsledné systémy v různých jazycích. Proto zavedeme systém primitivních generátorů, které se budou starat o výsledné generování kódu. Pro jednoduché komunikování se šablonami využijeme návrhového vzoru *AbstractFactory* [6].

4.1 Základní struktura

Průběh překladu a generování nového kódu lze schematicky popsat následujícím způsobem:

1. *Lexikální analýza* zdrojového kódu
2. *Syntaktická analýza* vytvoří abstraktní syntaktický strom (AST), jehož uzly budou tvořit prvky interního modelu systému
3. *Sémantická kontrola* korektnosti stromu – probíhá kontrola sémantiky, např. zda třída dědí z PN nebo jiných existujících tříd a některé další sémantické odvození, které nejsou zachyceny gramatickými pravidly syntaktické analýzy.
4. *Generovaní výsledného kódu* z uzlů interního modelu

Při generování textu samotného budeme postupovat iterativně. To znamená že budeme výsledný text tvořit postupně podle potřeb a struktury cílového jazyka. Jedním z možných východisek pro tuto práci bylo použít tzv. *generativní programování* [7]. Při tomto postupu každému pravidlu gramatiky přísluší kostra složená z výsledného textu, může však obsahovat reference na další nonterminály gramatiky. Výsledný text se pak injektuje do takových uzlů. Tento přístup má pro naše účely několik problémů. Jednak v případě C++ není výsledný text celistvý, ale rozdělený do více souborů. Důležitějším důvodem, proč nepoužijeme tento postup, však je fakt, že budeme na syntaktickém stromě provádět složitější úpravy, např. redukci složených zpráv na posloupnost jednoduchých. Z důvodu jednoduchosti bylo však od tohoto návrhu upuštěno.

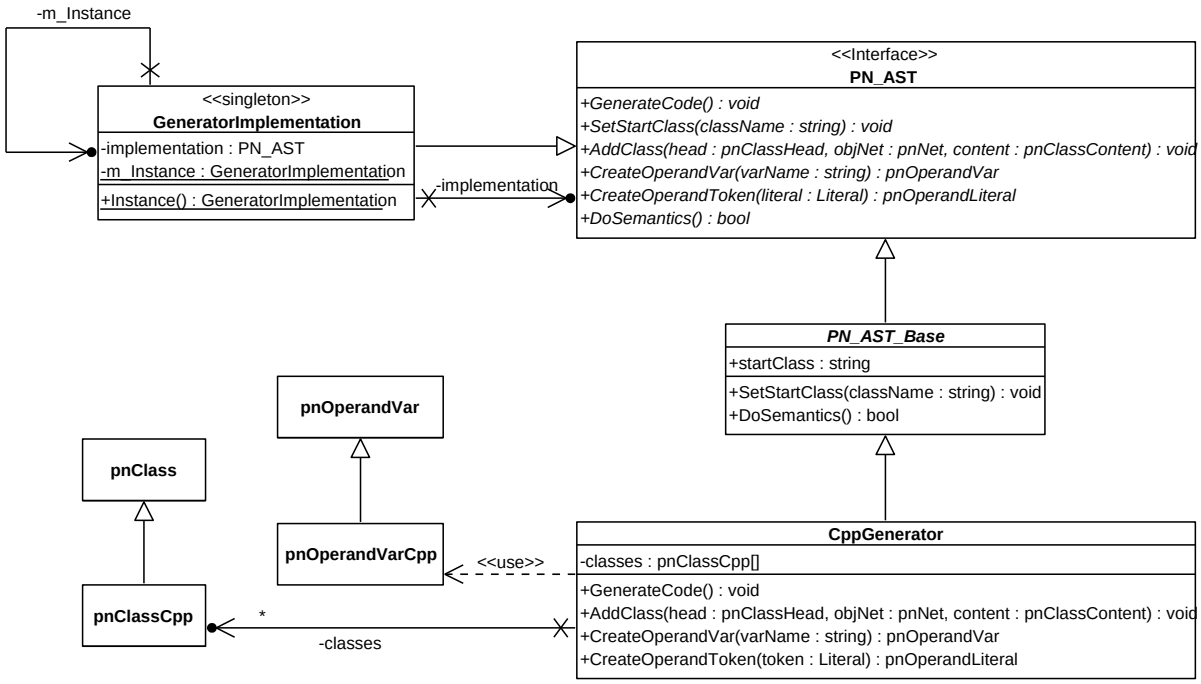
4.1.1 Podpora pro více cílových jazyků

Při návrhu systému, který bude generovat kód do různých cílových jazyků čelíme problému, kdy chceme mít co nejuniverzálnější systém na jedné straně a zároveň musíme respektovat strukturu tohoto jazyka. Při řešení se opíráme o třídy simulátoru navrženého v kapitole 3. Způsob implemetace totiž implikuje, jak bude vypadat generovaný kód. Nezanedbatelnou motivací při implementaci generátoru také bude snaha využít syntaktických možností cílového jazyka.

Nejdříve popíšeme způsob, jakým budeme spravovat implementace generátorů pro více cílových jazyků. Využijeme zde dvou návrhových vzorů, a to *Singleton* a *AbstractFactory*.[6] Způsob implementace je na obrázku 21.

Výchozí je rozhraní *PN_AST*, které definuje všechny operace, které budeme využívat při sestavování syntaktického stromu. Dále zde existuje třída *GeneratorImplementation*, která je *sigleton* (čili je zajištěno, že je dostupná právě jedna její instance). Tato třída v sobě uchovává aktuání implementaci generátoru a zprostředkovává všechny operace definované rozhraním *PN_AST*, protože toto rozhraní implementuje.

Pro každý cílový jazyk pak implementujeme vlastní verzi syntaktického stromu a generátorů primitiv. Z důvodu snahy o co nejuniverzálnější návrh se primitivní generátory omezují především na ty uzly stromu, které vykazují podstatné rysy polymorfního chování. V příkladu na obrázku 21 si můžeme takto všimnout, že generátor vyrábí dva typy operandů – jeden je definován literálem a druhý proměnnou. Z kapitoly 3.1.4, kde jsme definovali operandy v rámci simulátoru, vyplývá, že jinak se operandy chovají podobně a v simulátoru už vystupují jen jako obecný předek. Při definici konkrétních generátorů v kapitole (doplnit) tedy obdobně zavedeme třídu *pnOperand* jako předka pro oba typy operandů. Abychom dodrželi možnost generovat do více jazyků, identifikujeme všechny situace, za jakých se konkrétní typy operandů mohou vyskytovat v cílovém kódu a vytvoříme takové rozhraní, které umožní v daných situacích generovat kód.



Obrázek 20: Abstract factory pro generování syntaktického stromu

Můžeme si všimnout, Konkrétní generátor má ještě předka PN_AST_Base. Tento předek implementuje ty vlastnosti syntaktického stromu, u kterých nepředpokládáme závislost na cílovém jazyku. Jsou to především ty uzly stromu které nejsou popsány polymorfnímy generátory.

4.2 Lexikální a syntaktická analýza

Při lexikální a syntaktické analýze sestavujeme syntaktický strom reprezentující program zapsaný pomocí jazyka PNtalk. Tento syntaktický strom je sestaven pomocí dvojice nástrojů Lex a Yacc [8] Výsledkem je pak odpovídající interní struktura tvořená třídami reprezentující jednotlivé uzly.

4.2.1 Lex a Yacc

Lex a Yacc jsou programy, které z konfiguračních souborů umí vygenerovat C/C++ zdrojový text implementující parser zadané gramatiky. Pomocí programu Lex specifikujeme lexémy jazyka a pomocí programu Yacc specifikujeme gramatická pravidla jazyka.

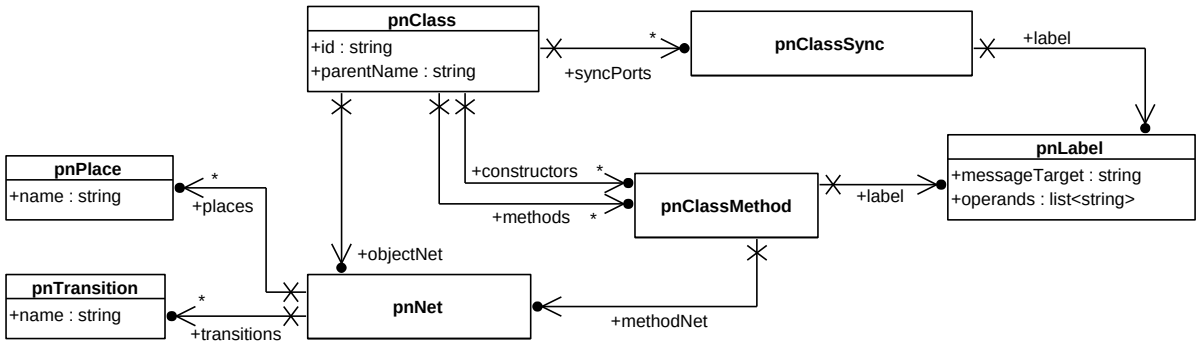
V konfiguračním souboru programu Yacc pak navíc u každého pravidla můžeme připojit zdrojový kód v C/C++, který bude vykonán při redukci pravidla. Této vlastnosti využíváme při konstrukci našeho interního syntaktického stromu.

Použité lexémy a gramatiku jazyka PNtalk nalezneme v příloze.

4.2.2 Syntaktický strom

Nyní provedeme čtenáře syntaktickým stromem. Budeme popisovat jednotlivé elementy tak, jak jsou v syntaktickém stromu reprezentovány třídami. Čtenář může sledovat analogii s gramatikou definující jazyk PNtalk tak, jak je popsána v příloze, popř. i s modelem simulátoru definovaným v kapitole 3. Jen podotkněme, že na rozdíl od simulátoru, který jsme sestavovali zdola nahoru, gramatiku dle zvyklostí popisujeme shora dolů.

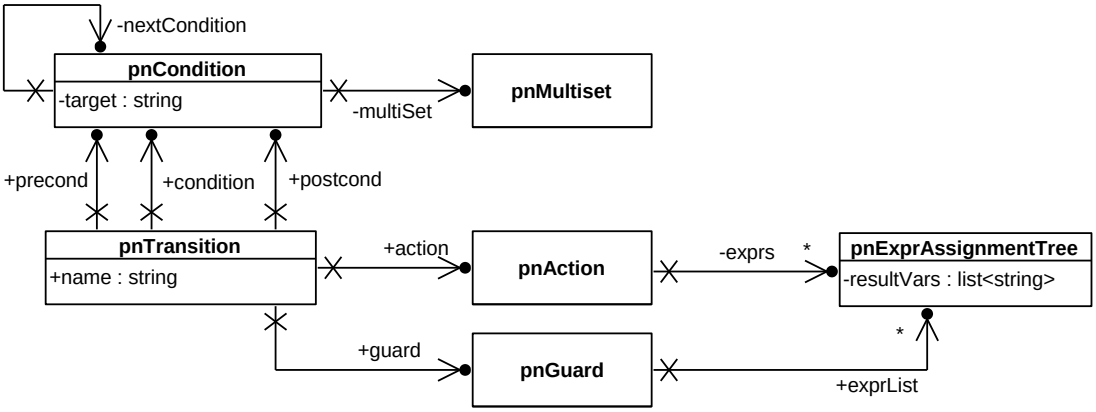
Výchozím elementem je třída Petriho sítí. Celý syntaktický strom je zjednodušeně pouze seznam tříd, které jsou v programu nadefinovány a navíc nese ještě informaci, která třída bude pro chod aplikace výchozí (viz obrázek 21).



Obrázek 21: Modelovaní tříd Petriho sítí v syntaktickém stromu

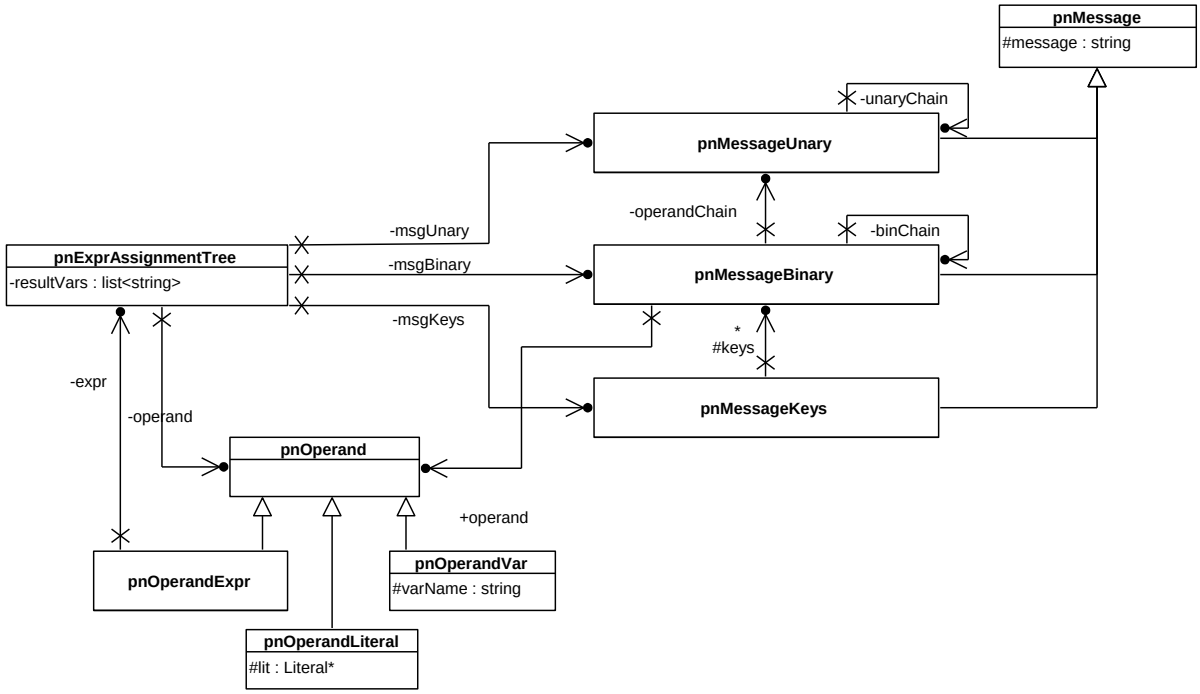
Element pnClass definuje třídu Petriho sítí. Nese své jméno (id) a jméno rodiče v hierarchii tříd. Třída definuje objektovou síť *pnNet*, synchronní porty pnClassSync, metody pnClassMethod a konstruktory, které jsou jen speciální metody. Poznamenejme, že gramatika zde definuje

objektovou síť a libovolný mix synchronních portů, konstruktorů a metod. Pro sestavení nepoužíváme polymorfismus ale pomocné médium *pnClassContentHolder*, z něhož podle příznaku přetypujeme na výsledný uzel. Důvodem je vysoká složitost generovaného kódu z těchto uzlů.



Obrázek 22: Syntaktický strom přechodu

Synchronní porty a metody navíc definují selektor zprávy, která je invokuje. Ten je definovaný pomocí třídy *pnLabel*.[1] Metoda definuje svou síť metody a síť je obecně tvořena místy *pnPlace* a přechody *pnTransition.*



Obrázek 23: Syntaktický strom složeného zaslání zprávy

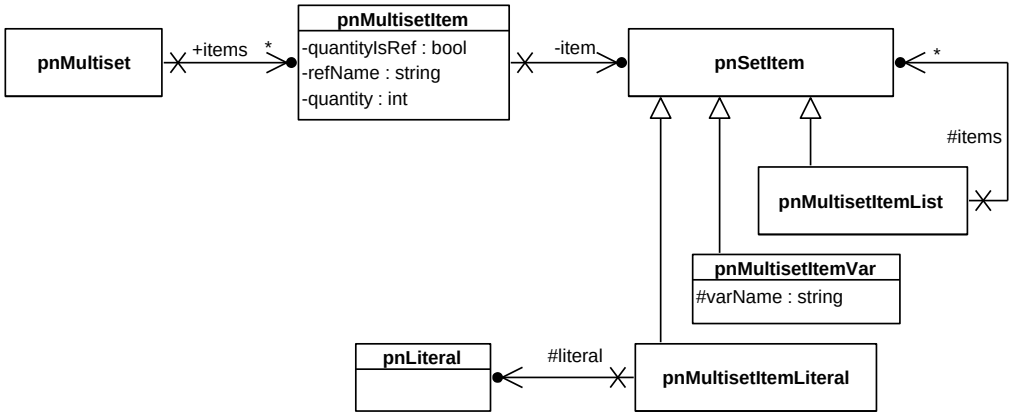
Přechod definuje hranové výrazy cond, precond, postcond, které jsou všechny reprezentovány třídou *pnCondition*. Pokud je hranových výrazů daného typu více, skládají se do řetězce. Hranový

1 V gramatice odpovídá nonterminálu *message* a zastřešuje všechny tři typy zpráv užívané v jazyce PNtalk.

výraz je definován multimnožinou *pnMultiSet*. Přechod pak definuje ještě stráž a přechod, které jsou definovány posloupností *složených zaslání zpráv* s možností uložit výsledek do několika proměnných.[1] Náš simulátor však vyžaduje, aby zaslání bylo jednoduché. Redukce složených zpráv popisuje kapitola 4.3.3. Dodejme, že gramatika přechodu vyžaduje přesné pořadí svých členů a to: cond, precond, guard, action, postcond.

Obrázek 23 popisuje schéma syntaktického stromu pro složené zaslání zprávy. Modelujeme tři typy zpráv, přičemž si všimneme, že klíčovanou zprávu modelujeme jako kolekci binárních zpráv, přičemž binární zpráva zde plní roli selektoru klíče pro argument. Dále si všimneme, že unární a binární zprávy se mohou řetězit a že binární zpráva může obsahovat odkaz na unární řetěz, který pak aplikuje na svůj operand. Samotné zaslání pak může definovat všechny tři typy, abychom mohli modelovat pořadí vykonávání zpráv, tak jak jsme jej popsali v kapitole 2.4.7. Jako operand pak může vystupovat literál, proměnná nebo vnořené zaslání zprávy, které implementuje možnost uzávorkování.

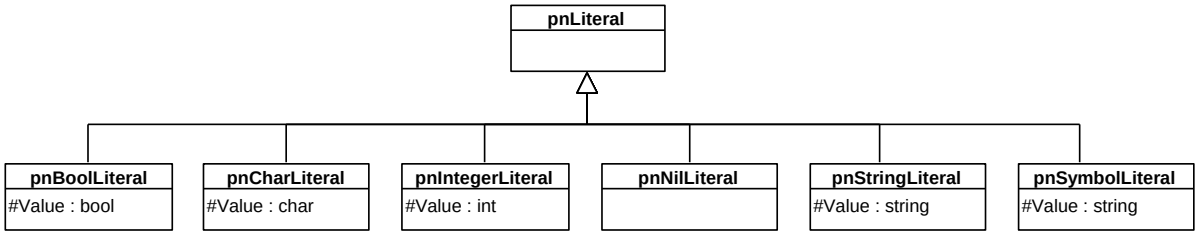
Mulimnožinu modelujeme podobně jako v simulátoru, ovšem je zde několik odlišností, které reflektují povahu jazyka PNtalk. Především prvky seznamu pnMultisetItemList neumožňují specifikovat kvantitu. Seznam je tedy n-tice, kde záleží na pořadí prvků. Implementace simulátoru toto omezení neřeší, protože je odchyceno už na úrovni syntaxe jazyka. Dále si můžeme všimnout, že nevytváříme typy prvků multimnožiny pro literály, ale literály referencujeme. Tato skutečnost vychází jednak z povahy gramatiky a jednak protože generování kódu pro multimnožiny přenecháme na literály.



Obrázek 24: Syntaktický strom pro multimnožinu

Poslední modelovanou skupinou jsou literály a jak vidíme na obrázku 25, jde o velmi přímočarou skutečnost. Jen podotkněme, že literál typu bool a typu nil nejsou z jazyka PNtalk generovány přímo, ale jsou interpretovány jako rezervovaná proměnná. Při implementaci překladače pak tuto interpretaci provádí tovární metoda třídy *PN_AST*, konkrétně *CreateMultisetItemVar*.

1 Oproti původní implementaci z [3] stráž připouští navíc konkatenaci složených výrazů a možnost uložit výsledek. Stále však platí, že výraz musí být vyhodnotitelný jako pravdivostní hodnota. Toto je však kontrolováno až za běhu programu, gramatika totiž nespecifikuje návratový typ.



Obrázek 25: Syntaktický strom pro literály

4.3 Sémantická analýza

V následující kapitole shrneme sémantické operace prováděné nad syntaktickým stromem. Jde především o kontrolu korektnosti modelu a také některé transformace, nezbytné pro správný chod výsledného programu.

4.3.1 Speciální interpretace uzlů

Jazyk PNtalk rozlišuje jméno proměnné a název třídy pouze podle toho, zda je první písmeno identifikátoru velké (pak je to název třídy). Gramatika PNtalku toto však po syntaktické stránce nerozlišuje a považuje všechny identifikátory za stejné. V případě definice jména třídy v její hlavičce můžeme provést kontrolu identifikátoru na první velké písmeno, už při konstrukci stromu. Dále už budeme jméno třídy používat jen jako metatřídu.

K interpretaci dochází až v případě, že je identifikátor použit v operandu typu proměnná. Pak platí následující pravidla:

- operand je vyhodnocen jako metatřída, pokud má počáteční velké písmeno a je použit jako adresát zprávy
- ve všech ostatních případech je operand proměnná a musí mít malé počáteční písmeno

Dále se budeme zabývat dostupností míst v síti metody. Platí, že přechody v síti metody mohou hranovými výrazy přistupovat jak do míst definovaných v metodě, tak do míst volajícího objektu, popř. i do míst jeho rodiče.

Při definici sítě v rámci simulátoru jsme nastínili, že místa jsou ve výsledné třídě definována jako atributy objektu. Při definování hran přechodu jsou tedy přímo dostupné pod svým jménem. Ovšem síť metody nemá přímo přístup do sítě volající třídy a navíc i tak musíme vědět, kde místo hledat. Jednou možností je využít pro každé místo metodu *GetPlace()* která místo podle jeho názvu vyhledá a to i ve volající síti. Druhou možností je použít tuto metodu pouze pro místa, u nichž víme, že nejsou definovány v síti metody.

V případě druhého řešení však musíme v rámci každého přechodu zkontrolovat všechny hrany, jestli odkazují na místo dané sítě metody.

4.3.2 Kontrola korektnosti

Kontrola korektnosti je fáze sémantické analýzy, kdy provádíme dílčí interpretace, které mají uživateli poskytnout přesnější informace o chybách v jeho modelu. Pokud bychom kontrolu neprovedli a byla v něm chyba, model by nešlo přeložit, ale uživatel by měl k dispozici pouze chybové hlášení překladače cílového jazyka. Bez detailní znalosti simulátoru by mohl jen obtížně zjišťovat podstatu chyby.

Jedna z důležitých kontrol je kontrola jmenného prostoru tříd. V rámci této skupiny operací definujeme tyto požadavky:

- Třída definující Petriho síť musí dědit z bázové třídy *PN* nebo jiné třídy definující Petriho síť.
- Pokud voláme v akci přechodu metatřídu (tj. identifikátor začínající velkým písmenem), musí odpovídat některé z nadefinovaných tříd nebo některé z podpůrných tříd Smalltalku. Třída PN však nelze instanciovat
- Třída definovaná zdrojovým kódem nesmí předefinovat vestavěnou třídu

Řešení: třída *pnClass* definuje svého rodiče pomocí textového řetězce a zároveň definuje své vlastní jméno. Abstraktní syntaktický strom pak nese seznam všech nadefinovaných tříd. Z tohoto seznamu můžeme tedy zkontrolovat, zda rodič třídy existuje, nebo je *PN* a zároveň zkontrolovat, zda třída nepředefinuje vestavěnou třídu. Nemusíme rekurzivně kontrolovat prarodiče, protože selhání u kterékoli třídy implikuje nekonzistenci systému a překlad dále nepokračuje.

Jiným problémem je fakt, že metoda třídy musí implementovat pro každý svůj argument místo se stejným názvem, kam při své inicializaci vloží hodnotu argumentu.

4.3.3 Redukce složených zpráv

V kapitole 4.2.2 jsme popsali syntaktický strom pro složené zaslání zprávy. V definici simulátoru však předpokládáme, že budeme modelovat pouze jednoduché zaslání zprávy. V následující kapitole popíšeme způsob, kterým redukujeme složenou zprávu na ekvivalentní posloupnost jednoduchých zpráv.

Budeme postupovat jako interpret jazyka Smalltalk, ze kterého je systém zpráv převzat a aplikujeme pravidla pro vykonání složené zprávy popsané v kapitole 2.4.7. Připomeneme:

1. Unární zprávy, zleva doprava.
2. Binární zprávy, zleva doprava.
3. Zprávy s klíčovými slovy, zleva doprava.

Místo vykonání zprávy však vytvoříme nový uzel syntaktického stromu reprezentující jednoduché zaslání zprávy. Pro udržení výsledku operace použijeme speciální proměnné. Je vhodné použít takové proměnné, které jazyk PNtalk neumožňuje definovat avšak v simulátoru je použít můžeme. Zvolíme tedy proměnné jejichž součást je podtržítko_.

Budeme používat následující proměnné[1]:

- `__primary__` - proměnná udržující hodnotu adresáta původní zprávy, po vykonání výsledné posloupnosti
- `__binArg__` - proměnná udržující hodnotu která bude použita jako argument binární zprávy
- `__key0__`, `__key1__` … - proměnné pro hodnoty jednotlivých argumentů klíčované zprávy

Navíc musíme počítat i s uzávorkováním, protože vnořené zprávy by mohly hodnoty v pomocných proměnných přepsat. V rámci uzávorkovaného výrazu budeme používat jiné proměnné. Modifikujeme definované pomocné proměnné přidáním podtržítka před jméno proměnné navíc. Pro vnoření první úrovně tedy použijeme např. ___primary__ kde v prefix tvoří 3x znak podtržítka.

Pro ilustraci redukce použijeme modifikovaný příklad z kapitoly 2.4.7:

```
o := 3 factorial + (4 factorial - 2) factorial between: 10 and: 100
```

Výsledek redukce pak bude:

```
__primary__ := 3 factorial
___primary__ := 4 factorial
___primary__ := ___primary__ - 2
__binArg__ := ___primary__
__binArg__ := __binArg__ factorial
__primary__ := __primary__ + __binArg__
__primary__ := __primary__ between: 10 and: 100
o := __primary__
```

4.4 Generátory

Nyní již máme sestavený a sémanticky upravený syntaktický strom, ze kterého můžeme začít generovat kód. Při generování kódu samotného se budeme opírat o simulátor navržený v kapitole 3. Předmětem generovaného kódu budou třídy definované zdrojovým kódem zapsaným v jazyce PNtalk.

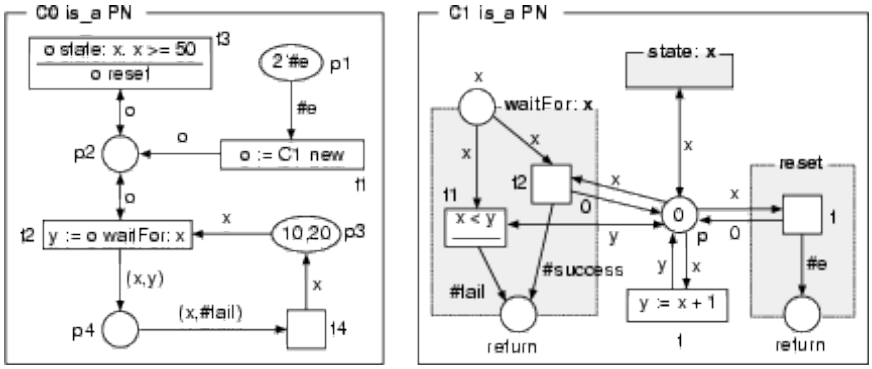
V ukázkové implementaci, která je programovou částí této práce, je implementovaný generátor pro jazyk C++ a následující kapitoly ilustrují jeho vygenerovaný výsledek. Vygenerovaný obsah je však závislý na cílovém jazyku a na způsobu, jakým je v něm implementován simulátor.

4.4.1 Generovaný obsah

Hlavním generátorem je samotná implementace syntaktického stromu. Samotný proces je pak překladačem spuštěn pomocí metody *Generate()*. Tato metoda by měl v ideálním případě vygenerovat soubory pro každou třídu zvlášť. V případě C++ bude tedy výstupem pro každou třídu

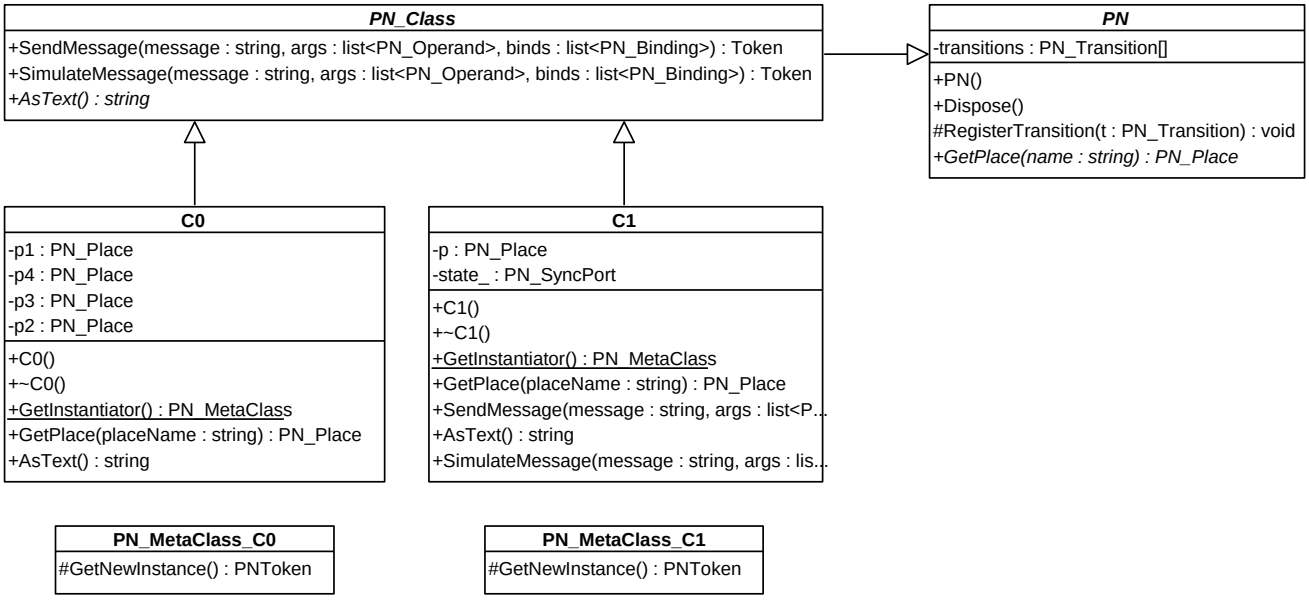
1 Prefixy i postfixy jsou tvořeny 2x znakem podtržítka _

hlavičkový soubor *.h definující třídu, její atributy, třídy pro sítě metod a také její metatřídu a pak soubor *.cpp obsahující implementaci všech metod nadefinovaných hlavičkovým souborem. Pro jiné jazyky však může být výstup odlišný.



Obrázek 26: Příklad tříd PNtalku

Generovaný výsledek budeme demonstrovat na příkladě z obrázku 26. Výsledné schéma pak vidíme na obrázku 27. Při porovnání vidíme, že dědičnost je jiná; v PNtalku je *PN* bázová třída pro třídy Petriho sítí, zatímco v simulátoru je to bázová třída pro sítě obecně a roli předka pro třídy Petriho síti plní třída *PN_Class*.



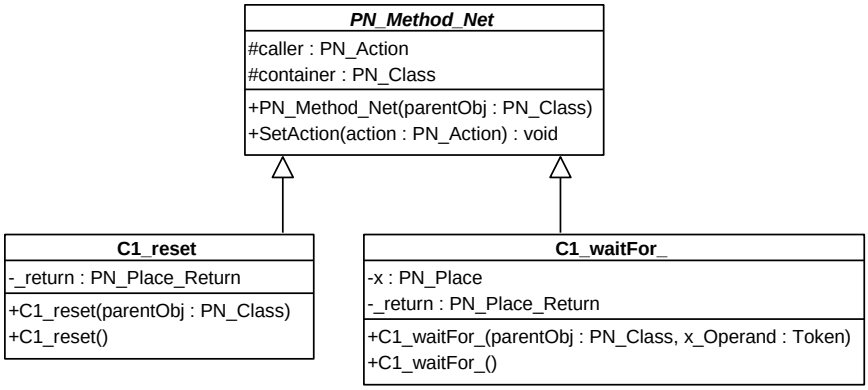
Obrázek 27: Schéma vygenerovaných tříd

Místa jsou v cílových třídách jako privátní atributy, jména míst jsou zachována. Třída však kvůli dostupnosti míst i ze sítí metod musí implementovat metodu *GetPlace*, která vrátí referenci na místo podle jeho jména. Pro každou třídu vygenerujeme její metatřídu a instanci metatřídy pak vracíme pomocí statické metody *GetInstantiator*. Každá třída tuto metodu implementujeme, při referenci na metatřídu z operandu se tedy na to můžeme spolehnout. Předek pro metatřídy implementuje zpracování zprávy *new*, koncové metatřídy implementují pouze metodu *GetNewInstance*, která vrátí token s referencí na novou instanci příslušné třídy.

Metoda *SimulateMessage* bude simulovat provedení synchronního portu a metoda *SendMessage* provede efekt synchronního portu a vykonání metod. Pokud třída tyto konstrukce nedefinuje, může příslušné metody vypustit, protože jsou nadefinovány také v bázové třídě.

Musíme také vygenerovat konstruktor a destruktor. Konstruktor slouží pro inicializaci sítě (viz dále) a destruktor uvolní všechny zapouzdřené objekty, což jsou místa a také přechody. Pro uvolnění přechodů použijeme metodu *Dispose* předka *PN*.

Pro každou metodu třídy pak musíme vygenerovat třídu, která ji reprezentuje. Sítím metod však nelze zaslat žádné zprávy ani nemohou definovat synchronní porty. Při své konstrukci však musí nastavit třídu, která metodu invokuje a navíc inicializovat místa pomocí parametrů, se kterými byla spuštěna.



Obrázek 28: Vygenerované třídy pro OOPN metody

4.4.2 Generování inicializace sítě a primitivní generátory

Každá síť je definována místy a přechody. V naší implementaci budeme při inicializaci sítě postupovat následovně:

1. Vytvoříme instance všech míst.
   a) Vytvoříme inicializační multimnožinu pro místo.
   b) Vytvoříme instanci místa pomocí jeho jména a multimnožiny.
2. Definujeme přechody.
   a) Vytvoříme instanci přechodu.
   b) Přechod registrujeme v simulačním kontextu.

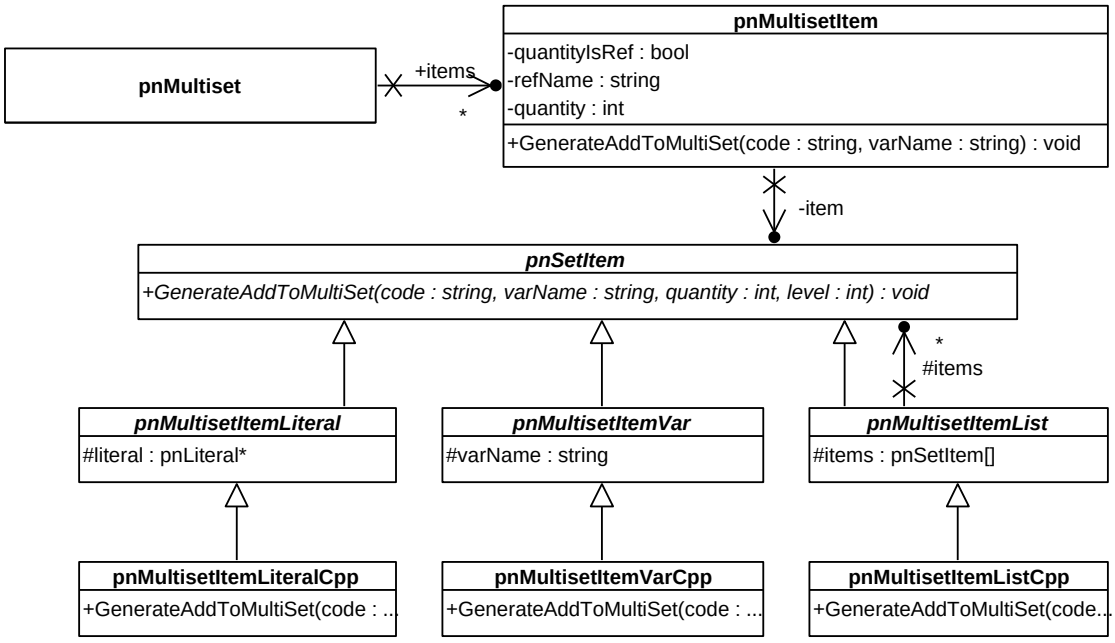
c) Přechodu vytvoříme všechny hrany – hrany definujeme po skupinách: precond, postcond, cond
   i. Vytvoříme multimnožinu definující hranu.
   ii. jako cíl hrany použijeme místa definovaná v bodě 1.
d) Definujeme stráž
e) Definujeme akci přechodu

Při definici některých prvků zmíněných výše můžeme využít připravených primitivních generátorů. Tyto generátory jsou navrženy s maximálním ohledem na nezávislost na cílovém jazyku. Není však vyloučeno, že pro nějaký programovací jazyk přesto nebudou vhodné a pak musíme kód generovat přímo.

Multimnožiny

Multimnožiny používáme při definici hrany nebo inicializačního značení místa. Vygenerujeme ji tak, že v cílovém kódu vytvoříme unikátní proměnnou a vytvoříme prázdnou multimnožinu. Pro každý prvek multimnožiny v syntaktickém stromu pak voláme metodu *GenerateAddToMultiSet* a jako parametry pošleme již vygenerovaný text a jméno proměnné v níž je uložena prázdná multimnožina.

V jednotlivých implementacích pak implementuje metodu *GenerateAddToMultiSet*, ve které generujeme kód přidání prvku do multimnožiny uložené v proměnné, přičemž z návrhu simulátoru víme, že třída *PN_MultiSet* má metodu *AddItem*. Zohledníme parametr kvantity a navíc parametr *level,* který značí požadované odsazení v cílovém kódu. Odsazení zavádíme kvůli prvku typu seznam. Pro definici seznamu budeme potřebovat pomocnou proměnnou, ale seznam může obsahovat vnořený seznam nebo jich může být více. Abychom nemuseli řešit konflikt pomocných proměnných, budeme ho generovat do uzavřeného bloku (v C++ uvozeného pomocí '{' a '}'), proto odsazení.



Obrázek 29: Primitivní generátory multimnožiny

Dodejme, že prvek typu literál neimplementuje generování kódu přímo, ale deleguje na stejnou metodu ve třídě *pnLiteral*.

Zprávy

V případě zpráv a další primitiv už nebudeme ilustrovat kompletní koncept, protože je podobný jako u multimnožin. Zmíníme však metody určené ke generování kódu a situace, v nichž se používají:

- *GenerateSetAction* – má za úkol vytvořit novou instanci třídy *PN_Expression*, která bude inicializovaná odpovídající zprávou s případnými argumenty. Lze využít při definici zaslání zprávy v rámci akce nebo stráže přechodu.

Operandy

Operandy definují jedinou metodu pro generování kódu:

- *GenerateNewInstance* – metoda, která má za úkol vygenerovat novou instanci operandu. Je volána v rámci inskripce přechodu vždy, když definujeme argument, adresát nebo cílovou proměnnou. V případě literálu opět delegujeme na třídu *pnLiteral*

Literály

Literály definují následující metody pro generování kódu:

- *GenerateAddToMultiSet* – jde o stejnou funkci jako v případě multimnožin tedy generování hran a inicializace místa.
- *GenerateNewInstance* – metoda, která má za úkol vygenerovat novou instanci literálu. Je volána v rámci inskripce přechodu vždy, když vystupuje jako operand.

4.4.3 Generování reakce na zprávu

Předmětem je kód, který bude vygenerován v metodách *SendMessage* a *SimulateMessage* . V rámci Petriho sítí může být zasláním zprávy vyvolán pouze synchronní port nebo metoda. Vstupy pro obě metody jsou:

- *string* message – obsahuje selektor zprávy
- *list<PN_Operand>* args – argumenty předané zprávou (proměnné nebo tokeny)
- *list<PN_Binding>* binds – aktuální navázání proměnných

V těchto metodách pro každou zprávu, které má třída rozumět, vytvoří následující kód (zapsáno psedukódem). Pro synchronní port využijeme připravených metod:

```
if (message == selektor)
{
  return new BoolToken(TryBind(args, binds))     // pro SimulateMessage
  - nebo -
  return new BoolToken(Execute(args, binds))    // pro SendMessage
}
```

V případě metod pak navíc musíme vyřešit operandy resp. jejich nalezení v navázání proměnných. Pro metodu s N operandy bude psedokód:

```
if (message == selektor)
{
  for (int i, i<N, i++)
  {
      Token argi = args[i].Resolve(binds);
  }
  throw new PN_ForkException(new MethodNetClass(this, arg1, arg2,…, argn));
}
```

5 Srovnání rychlosti implementací

Ve chvíli, kdy máme implementovaný jak simulátor, tak i generátor, nabízí se možnost srovnat rychlost vykonání programu v původním prostředí PNtalku, tj. vývojové prostředí *Pharo* a vygenerovaného programu v C++. Při kompilaci knihovny v C++ navíc můžeme využít vestavěného optimalizátoru překladače *g++*.

Pro účely testování jsou připraveny dva testy.

Prvním testem je implementace Ackermannovy funkce, která je typickým příkladem rekurzivní funkce, která není primitivně rekurzivní. Ackermannovu funkci definujeme jako:

$$A(m,n)=\begin{cases} n+1, & \text{pokud } m=0 \\ A(m-1,1), & \text{pokud } m>0 \text{ a } n=0 \\ A(m-1,A(m,n-1)), & \text{jinak} \end{cases}$$

Tato funkce roste extrémně rychle a pro naše účely použijeme hodnoty argumentů $A(3,2)$. Test je zaměřen na rekurzívní volání metod, otestujeme tedy i návrat z vykonání metody.

Druhý testovací příklad je iterativní test o 10 000 iteracích. V rámci jedné iterace je pak testován synchronní port v rámci stráže.

Zdrojové texty obou testů nalezne čtenář v příloze B.

Metodika

Testy byly provedeny v třech variantách – v systému Pharo, na programu přeloženém bez optimalizace a na optimalizované variantě. Měření času je řešeno voláním interních stopek na obou systémech v rámci vykonání programu. Přesnost stopek je v milisekundách. Test tedy není zatížen inicializační fází spuštění programu. Testy byly provedeny desetkrát pro každou variantu a následně byl vytvořen aritmetický průměr hodnot.

Výsledek

	Pharo	C++ (bez optimalizace)	C++ (s optimalizací)
Ackermannova funkce	4,02s	0,493s	0,153
Iterativní test	14,045s	1,52s	0,667

Již na první pohled je vidět, že implementace v C++ je rychlejší i neoptimalizované variantě, a to 8x až 9x. Se zapnutou optimalizací se potom dostáváme na více než 20x rychlejší vykonání. I když tyto testy zdaleka neobsáhnou všechny aspekty jazyka PNtalk, zrychlení výpočtu se zdá být přesvědčivé.

6 Závěr

Předmětem práce bylo navrhnout a implementovat generátor kódu, který bude z modelu specifikovaného jazykem PNtalk generovat programy v jiném objektově orientovaném jazyce. Jako výchozí jazyk byl zvolen C++.

Pro dosažení výsledku, byl zvolen přístup používající podpůrnou knihovnu plnící funkci simulátoru, vygenerovaný kód pak tuto knihovnu používá. V rámci práce je implementována simulační knihovna v programovacím jazyce C++. Knihovna je však popsána pomocí modelačního jazyka UML, její implementace do jiných programovacích jazyků by tedy měla být maximálně usnadněna.

Hlavním výstupem je samotný generátor kódu. Je implementovaný v jazyce C++ a v současné době implementuje generování opět do jazyka C++. Při implementaci však byl brán zřetel na to, abychom mohli implementaci doplnit i o generování do jiných jazyků. Samotným generovaným obsahem pak jsou třídy, pomocí nichž modelujeme objektové Petriho sítě.

Při implementaci generátoru byla navržena taková struktura syntaktického stromu, kde významné uzly tvoří primitivní generátory do výsledného jazyka. Tím je docíleno vyšší strukturovanosti a i psaní generátorů do jiných cílových jazyků by mělo být jednodušší.

Systém jako celek se podařilo s drobnými omezeními zprovoznit. Jde však většinou o okrajové možnosti jazyka PNtalk, všechny hlavní rysy jsou funkční. Při porovnání rychlosti simulátoru v C++ s původní implementací ve Smalltalku bylo zaznamenáno podstatné zrychlení. To je dáno jednak tím, že C++ je jazyk na nižší úrovni abstrakce a pak také tím, že můžeme využít existujících optimalizací překladače *g++*.

Možné rozšíření

I generátor kódu do jiného jazyka nabízí další rozšíření. Jednou z největších výhod jazyka PNtalk a jeho implementace v prostředí Pharo je možnost využít grafické reprezentace jazyka PNtalk. Po překladu do cílového jazyka a spuštění běhu simulátoru již nemáme možnost sledovat graficky stav sítí. Hlavní možné rozšíření je implementace opačného překladu, tedy ze stavu simulace vygenerovat opět kód v jazyce PNtalk. Výhodou by bylo, že struktura sítě zůstane zachována, změní se pouze iniciační značení míst.

Jiný možný vývoj se týká paralelizace. Aktuální simulátor počítá s jediným simulačním kontextem a nezabývá se paralelním zpracováním. Při paralelním zpracování by bylo zapotřebí řešit především otázku hledání proveditelného přechodu, tedy zajistit, aby vybraný přechod byl proveden včas a nedošlo ke kritické změně stavu objektů v důsledku provedení jiného přechodu.

Seznam použité literatury

[1]: PNtalk project [online] , cit. [2015-05-20]. Dostupné na: <http://http://perchta.fit.vutbr.cz/pntalk2k/1>

[2]: Hanák, M.: Knihovna vzorů Petriho sítí, Brno, FIT VUT, 2011

[3]: Janoušek, V.: Modelování objektů Petriho sítěmi, Brno, UIVT FEI VUT, 1998

[4] : Mazal, Z., Janoušek, V., Kočí, R., Enhancing the PNtalk language with Negative Predicates, MOSIS '08, Ostrava, MARQ, 2008, ISBN 978-80-86840-40-6

[5] : Wilson, P.R., Uniprocessor Garbage Collection Techniques, Proceedings of the International Workshop on Memory Management, London, UK, Springer-Verlag, 1992, ISBN 3-540-55940-X

[6] : Gamma, E., Helm, V., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software, Boston, USA, Pearson Education, 1994, ISBN 978-0-321-70069-8

[7] : Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools and Applications, Boston, USA, Addison-Wesley Professional, 2000, ISBN 978-0201309775

[8] : Levine, J., Mason, T.,Brown, D.: Lex & Yacc, Sebastopol, CA, USA, O'Reilly Media, 1992, ISBN 1-56592-000-7

Seznam ilustrací

Přílohy

A. Gramatika jazyka PNtalk

ROOT : classes MAIN id classes

classes : classes class |
class : CLASS classhead objectnet classcontent | CLASS classhead classcontent
classhead : id INHERITANCE id
classcontent : classcontentitem classcontent |
classcontentitem : methodnet | constructor | sync

objectnet : OBJECT net
methodnet : METHOD message net
constructor : CONSTRUCTOR message net
sync : SYNC message cond precond guard postcond

message : id | binsel id | keys
keys : keys keysel id | keysel id

net : net place | net transition |
place : PLACE id '(' multiset ')' init | PLACE id '(' ')' init

init : INIT '{' actioncontent '}' |
actioncontent : expr_assignment_concat | temps expr_assignment_concat
transition : TRANSITION id cond precond guard action postcond

cond : COND condS |
precond : PRECOND condS |
postcond : POSTCOND condS |
condS : id '(' multiset ')' | condS ',' id '(' multiset ')'

guard : GUARD '{' expr_concat '}' |
action : ACTION '{' actioncontent '}' |
multiset : multisetitem | multiset ',' multisetitem

multisetitem : DIGIT '`' c | INTEGER '`' c | id '`' c | c
c : literal | id | list

```
list : '(' listcontent ')'
listcontent : c | listcontent ',' c

temps : '|' tempscontent '|'
tempscontent : tempscontent id |

operand : id | literal | '(' expr_assignment ')'

unaryMsg : id
unaryChain : unaryMsg unaryChain |
binMsg : binsel operand unaryChain
binChain : binMsg binChain  |
keysel : KEYSEL
keyMsgSegment : keysel operand unaryChain binChain
keyMsg : keyMsgSegment | keyMsg keyMsgSegment

expr_concat : expr | expr_concat '.' expr
expr_assignment_concat : expr_assignment | expr_assignment_concat '.' expr_assignment
expr_assignment : expr | id ':' '=' expr_assignment
expr : operand | operand msgchain |

msgchain : unaryMsg unaryChain binChain
         | unaryMsg unaryChain binChain keyMsg
         | binMsg binChain
         | binMsg binChain keyMsg
         | keyMsg

binsel : BINSELCHAR  | '=' | binsel BINSELCHAR  | binsel '='
literal : DIGIT | INTEGER | string | charconst | symconst | arrayconst
arrayconst : '#' array
array : '(' arraycontent ')'
arraycontent : arraycontent string | arraycontent id | arraycontent array | arraycontent charconst |

string : STRING_LITERAL
charconst : '$' char  | '$' SMALLLETTER  | '$' CAPLETTER
symconst : '#' symbol
symbol : id | string
id : ID | SMALLLETTER | CAPLETTER
letter : SMALLLETTER | CAPLETTER
char : DIGIT | BINSELCHAR | '='  | ',' | '['  | ']'  | '{' | '}' | '(' | ')' | '_' | '^' | ';' | '$' | '#' | ':' | '.' | '`'
```

B. Obsah CD

Na přiloženém CD nalezneme zdrojové kódy implementovaných programů a sadu příkladů, na kterých si může čtenář vyzkoušet překlad do cílového systému. Programy byly implementovány ve vývojovém prostředí Code::Blocks a ve složkách jsou i projektové soubory pro toto prostředí – mají příponu *.cbp. Jsou přiloženy soubory Makefile pro spuštění v prostředí UNIX.

Struktura

- PNTranslator – složka, v níž je umístěn samotný generátor
- PNTranslator/PN_Sim – složka obsahující knihovnu simulátoru
- Examples – složka s příklady

PNTranslator

Přeložíme jej pomocí přiloženého souboru Makefile. Makefile podporuje následující parametry:

- all – přeloží optimalizovanou verzi simulační knihovny a poté přeloží překladač.
- debug – přeloží neoptimalizovanou verzi simulační knihovny s debugovacími symboly
- clean – uklidí přeložené soubory

Pokud bude čtenář zkoušet různé verze knihoven, měl by vždy nejdříve provést *make clean.* Překlad využívá programů Lex a Yacc, konkrétně jejich verzí *flex* a *byacc*. Musí být tedy v systému naistalovány.

Program PNTranslator pak spouštíme:

```
./PNTranslator -in infile [-dir outDir="output"] [-out progName= outDir]
```

- -in infile – povinný parametr určující zdrojový text v jazyce PNtalk
- -dir outDir – cílová složka, do které bude program včetně příslušného Makefile vygenerován, pokud není zadána, použije se „output"
- -out progName – jméno vygenerovaného programu, resp. programu, který vznikne spuštěním příslušného Makefile. Pokud není specifikován, bude stejný jako název výstupní složky

PN_Sim

Simulační knihovna má vlastní .cbp projekt, který slouží především pro experimentování se systémem. Také Makefile je pro knihovnu oddělen.

Příklady

Příklady jsou ve formě zdrojového kódu v jazyce PNtalk a mají příponu *.pn. Příklady obsahují i testy na výkonnost z kapitoly 5. Bližší popis příkladů včetně očekávaného výstupu nalezne čtenář v souboru */Examples/readme.txt*