VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

GENEROVÁNÍ KÓDU Z MODELŮ PETRIHO SÍTÍ

CODE GENERATION FROM OBJECT ORIENTED PETRI NETS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **EDUARD FRLIČKA**
AUTHOR

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

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

144765

Ústav:	Ústav inteligentních systémů (UITS)
Student:	**Frlička Eduard**
Program:	Informační technologie
Specializace:	Informační technologie
Název:	**Generování kódu z modelů Petriho sítí**
Kategorie:	Softwarové inženýrství
Akademický rok:	2022/23

Zadání:

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

Literatura:

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

Při obhajobě semestrální části projektu je požadováno:
První tři body zadání.

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

Vedoucí práce:	**Kočí Radek, Ing., Ph.D.**
Vedoucí ústavu:	Hanáček Petr, doc. Dr. Ing.
Datum zadání:	1.11.2022
Termín pro odevzdání:	31.7.2023
Datum schválení:	3.11.2022

Abstrakt

PNtalk je jazyk využívaný na modelovanie objektovo orientovaných petriho sietí. Existujúci simulátor implementovaný v jazyku Smalltak slúži na modelovanie a ladenie, pre nasadenie v praxi, je potrebné reprezentovať tieto modely efektívnejšie. Táto práca sa zaoberá práve tvorbou efektívnejšej reprezentácie, konkrétne transformáciou do jazyku C++. V práci boli riešené problémy paralelného vyhodnocovania petriho sietí, optimalizácie vyhodnocovania prechodov a dynamické typovanie.

Abstract

PNtalk is a language used for modelling of object-oriented Petri nets. The existing simulator implemented in Smalltalk language is used mainly for modelling and debugging. For practical use, however, a more efficient representation of Petri net models is needed. This work presents a C++ representation of object-oriented Petri nets.

Kľúčové slová

Petriho siete, PNtalk, C++, generátor kódu, prekladač

Keywords

Petri networks, PNtalk, C++, code generator, compiler

Citácia

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

Generování kódu z modelů Petriho sítí

Prehlásenie

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

.....
Eduard Frlička
31.7.2023

Poďakovanie

Rád by som poďakoval vedúcemu práce pánovi doktorovi Kočímu, za odborné vedenie a trpezlivosť.

Obsah

Zoznam obrázkov

Kapitola 1

Úvod

Modelovanie systémov nám dnes umožňuje konceptualizovať, simulovať, a lepšie pochopiť svet okolo nás. V spojení s informatikou sa využíva prevažne pri návrhu softwaru, kde nám umožnuje dizajnovať, analyzovať a simulovať komplexné systémy. Existuje mnoho jazykov a formalizmov využívaných pri modelovaní. Jedným z nich sú Petriho siete.

Petriho siete sú matematický formalizmus, umožňujíci modelovať paralelné systémy. Pri modelovaní zložitejších systémov sú však Petriho siete nedostatočné a preto vznikajú rôzne rozšírenia. Jazyk PNtalk rozširuje koncept Petriho sietí o triedy a objektovo orientovaný prístup. Pri vzniku bol inšpirovaný ako Petriho sieťami, tak aj programovacím jazykom Smalltalk.

Dnes pre jazyk PNtalk existuje simulátor naprogramovaný v Smalltalku, bežiaci pod prostredím Pharo. Tento nástoroj slúži prevažne pre návrh a prípadnú analýzu modelov. Umožňuje nám podrobne sledovať a prípadne zasahovať do behu modelu. To má však za následok príliš vyskoké nároky na zdroje. Jedným z riešení je využívať simulátor v prostredí Pharo len pre návrh a ladenie a implementovať nástroj pre transformáciu modelov PNtalku do efektívnejšej reprezentácie. Táto práca sa zaoberá práve návrhom a implementáciou tohto nástroja, prekladača.

Pre účely tejto práce bola ako výstupná reprezentácia zvolený jazyk C++. Ŭlohou prekladača bude teda vygenerovať zdrojové súbory jazyka C++, ktoré budú presne koírovať správanie modelu špecifikovaného v jazyku PNtalk. Táto práca zachytáva postup vývoja tohto nástroja od naštudovania definície PNtalku, cez návrh systému, implementáciu, až po jeho testovanie. Zaoberá sa nielen prekladom a generovaním kódu z modelu PNtalku, ale aj transformáciou niektorých základných tried jazyku Smalltalk.

Kapitola 2

Jazyk PNtalk

V tejto kapitole je stručne popísaný formalizmus Petriho sietí. Ďalej sú popísané konštrukcie jazyku PNtalk podľa definície z [3], konkrétne výrazy a ich vyhodnocovanie, multimnožiny, prechody, miesta, metódy, konštruktory a synchrónne porty.

2.1 Petriho siete

Petriho siete sú matematický modelovací jazyk používaný na modelovanie distribuovaných systémov. Používa dva druhy prvkov:, miesta a prechody. Miesto obsahuje diskrétny počet značiek resp. tokenov a reprezentuje tým určitú vlastnosť systému tzv. *parciálny stav*. Množina ohodnotení všetkých miest (parciálnych stavov) reprezentuje stav systému. Stav systému sa môže meniť pomocou prechodov, ktoré sú prepojené orientovanými hranami s miestami. Hrany smerujúce smerom do prechodu označujú vstupné miesta prechodu a opačné hrany označujú výstupné miesta. Prechod je uskutočniteľný pokiaľ všetky jeho vstupné miesta obsahujú dostatočný počet tokenov (určený kardinalitou hrany). Uskutočnenie prechodu spôsobí zníženie počtu tokenov vo vstupných miestach a zvýšenie vo výstupných miestach podľa kardinality.

Petriho sieťe majú taktiež mnoho rozšírení, ktoré zaručujú deterministickosť systému alebo zvyšujú ich vyjadrovaciu silu a to:

- **Priorita prechodu** – V prípade viacerých uskutočniteľných a vzájomne sa vylučujúcich prechodov určuje presné prechody, ktoré sa uskutočnia.
- **Pravdepodobnosť prechodu** – Pri výbere prechodu sa použije pravdepodobnosť namiesto priority.
- **Časované prechody** – V prípade, že je prechod uskutočniteľný, spustí sa časovač odpočítavajúci určený čas (môže byť aj náhodne vybratý pomocou funkcie náhodného rozloženia). Pokiaľ bol prechod počas celej doby behu časovača uskutočniteľný, uskutoční sa.

2.2 Termy

- Znak: Môže byť reprezentovaný literálom `$a, $4, $/ $$`
- Reťazec: Rôzne literály reťazcov. Pre vloženie znaku úvodzovky do reťazca, je potrebné ju zdvojiť. `'Ahoj', 'You''re'`

- Symbol: Literál symbolu je reprezentovaný buď mriežkou a postupnosťou znakov, a čísel alebo znakom mriežky a stringovým literálom`#santa, #'santa', #e, #ad5d`
- Celé číslo: Literály čísel sú postupnosťou číslic. Literály zapísané v inej ako desiatkovej sústave sú reprezentované najprv základom sústavy, znakom r a následne hodnotou literálu v danej sústave: Príklady literálov celých čísel: `4, 2r1001, 16r1A5`
- Číslo s desatinným rozvojom: Literály týchto čísel sa zapisujú ako postupnosť číslic nasledovaná bodkou a následne dalšou postupnosťou číslic. Podobne ako pri celých číslach je možné špecifikovať sústavu, v ktorej je literál zadaný. Rovnako je možné použiť vedecký zápis čísla, pomocou znaku e a nasledujúcej postupnosti číslic reprezentujúcej exponent. Príklady literálov: `1.3, 16r2A.45F, 1.57e-6`
- `true, false, nil`: Rezervované identifikátory reprezentujúce boolovské konštanty a nedefinovaný objekt.
- Názvy premenných: Začínaju vždy malým písmenom. Pokračujú rôznymi znakmi abecedy, čísel alebo odčiarkovník. `x, var, anObject`
- Názvy tried: Začínajú veľkým písmenom. Podobne ako pri názvooch premenných, nasledujú znaky abecedy, čisla alebo podčiarkovník `PN, Integer`
- Pseudopremenné: `self, super` Tieto premenné maju vyhradenú platnosť a rozličnú sémantiku od ostatných premenných. Premenná `self` sa používa pri zasielaní správy sebe samému, a pseudopremenná `super` zase pri zasielaní správy rodičovskej triede. To pri preťažení metódy umožňuje prístup k inak neprístupným metódam.

2.3 Výrazy

V jazyku PNtalk sú výrazy vyhodnocované pomocou zasielania správ alebo termov. Zaslanie správy má tvar:

<adresát> <správa>

kde adresát je výraz a správa je zložená zo selektoru a pripadných argumentov. V jazyku PNtalk, rovnako ako Smalltalk, vyhodnotenie správy závisí čisto na adresátovi. Rozlišujú sa 3 druhy správ:

1. **Unárna správa** – Selektor je reprezentovaný identifikátorom. Tento druh správy nemá žiaden argument. Príkadom zaslania unárnej správy je `5 factorial`. V tomto prípade je číslu 5 zaslaná správa `factorial`, kde `factorial` je selektor správy.
2. **Binárna správa** – Selektor je reprezentovaný binárnym operátorom. Za selektorom sa nachádza práve jeden výraz ako argument. Príkladom zaslania binárnej správy je `3 + 4`. V tomto prípade je číslu 3 zaslaná správa `+ 4`, kde `+` je selektor a `4` je argument. Podporované operátory sú:

   `+ - * / // \\= ~= == ~== < > <= >= & |`
3. **Správa s kľúčovými slovami** – Selektor správy je zložený z viacerých identifikátorov. Zaslanie takejto správy má identifikátory nasledované dvojbodkou a argumentom.

Príkladom zaslania tejto správy je `'ahoj' at: 4` alebo `'ahoj' at: 4 put:$a`. V prvom prípade je reťazcu „ahoj“ zaslaná správa so selektorom `at:` a jedným argumentom `4`. V druhom prípade je selektor `at:put:` a zaslané argumenty sú dva: `4` a `$a`.

Argument správy môže byť term alebo zaslanie inej správy. Je tak možné zasielania správ kombinovať. Správy sa v takom prípade vyhodnocujú nasledovne:

1. Zátvorky: Použitie zátvorky vynúti prioritné vyhodnotenie výrazu.
2. Unárne správy, zľava. Pri príklade `3 tan floor` by bola najprv zaslaná správa `tan` číslu 3 a následne by výsledku tohto zaslania bola zaslaná správa `floor`.
3. Binárne správy, zľava. Pri príklade `1+2*3` by bola číslu 1 zaslaná správa `+2` a výsledku by bola zaslaná správa `*3`. Jazyk PNtalk teda nemá žiadnu prioritu operátorov. Je však možné ju vynútiť pomocou zátvoriek: `1+(2*3)`.
4. Správy s kľúčovými slovami, zľava. Pre kombináciu týchto správ je potrebné použiť zátvorky. Ak by v príklade `'ahoj' at: 1 put: ('ahoj' at: 4)` neboli zátvorky, výraz by bol vyhodnotený len ako jedno zaslanie správy `at: 1 put: 'ahoj' at: 4` resp. zaslanie správy so selektorom `at:put:at:`.

2.4 Multimnožina

Multimnožiny sa využívajú na reprezentáciu hranových výrazov a počiatočných stavov miest. Skladajú sa z postupnosti čiarkou oddelených párov $n`t$, kde:

- n – Koeficient určujúci počet výskytov t. Je reprezentovaný literálom nezáporného celého čísla alebo názovom premennej. V prípade názvu premennej je očakávané, že sa jej hodnota vyhodnnotí na celé nezáporné číslo.
- t – Term alebo zoznam. Term je reprezentovaný ľubovoľným literálom alebo názvom premennej.

Zoznam je reprezentovaný zátvorkami, ohraničenou postupnosťou termov alebo zoznamov oddelených čiarkami. Prologovský zápis zoznamu je tiež akceptovaný, v tvare: $(e_1, e_2, ... \mid tail)$, kde e_i reprezentujú prvky zoznamu a *tail* reprezentuje zvyšok zoznamu. V prípade, že koeficient nie je špecifikovaný, je implicitne uvažovaný ako 1.

Príklad multimnožiny, ktorá obsahuje 5 výskytov znaku `a`, číslo `2`, x výskytov dvojice `('ahoj', 2)`, a trojicu `((#fail, y), 5, 4)`:

```
5`\$a, 2, x`('ahoj', 2), ((#fail, y), 5, 4)
```

2.5 Prechod

Prechod môže obsahovať stráž, akciu a naviazania na miesta pomocou hrán. Existujú 3 typy hrán:

- *Vstupná hrana* – Je orientovaná v smere z miesta do prechodu. Hranový výraz reprezentuje značky, ktoré majú byť pred vykonaním prechodu odobrané z miesta. V prípade, že tieto značky nie je možné odobrať, prechod nie je vykonateľný.

- *Výstupná hrana* – Je orientovaná v smere z prechodu do miesta. Hranový výraz reprezentuje značky, ktoré budú vložené do miesta po vykonaní prechodu.
- *Testovacia hrana* – Obojsmerná hrana. Hranový výraz reprezentuje značky, ktoré musia byť prítomné v mieste, avšak nie sú odobrané. V prípade, že sa tieto značky nenachádzajú v mieste, prechod nie je vykonateľný.

Stráž poskytuje možnosť špecifikácie komplikovanejších podmienok vyhodnotenie vykonateľnosti prechodu, ako aj možnosť naviazania na synchrónny port. Je reprezentovaná sekvenciou výrazov oddelených bodkami. Prechod je vykonateľný v prípade, že je konjukcia týchto výrazov vyhodnotená ako pravda.

Akcia je reprezentovaná výrazom alebo ich bodkou oddelenou postupnosťou. Umožnuje priradenie do premenných, a teda ovplyvnenie hranového výrazu pre výstupnú hranu. V prípade potreby je možné použiť dočasné premenné pre medzivýsledky.

2.6 Metóda

Objekt petriho siete, podobne ako všetky objekty v jazyku Smalltalk, môže reagovať na správy. Reakciu na správu určuje metóda. Metóda sa skladá zo vzoru správy a siete metódy. Vzor správy určuje na akú správu bude metóda odpovedať. Je tvorený selektorom správy a ľubovoľným počtom argumentov. Sieť metódy pozotáva z miest a prechodov, podobne ako sieť objektu. Pri prijatí správy objektom, je vytvorená inštancia metódy a argumenty sú vložené na tzv. parametrové miesta, ktorých mená korešpondujú s názvami argumentov. Pre odovzdanie výsledku správy je využívané tzv. výstupné miesto s názvom `return`. Prítomnosť tokenu v tomto stave indikuje ukončenie vyhodnocovania správy, a teda spôsobuje zánik inštancie objektu metódy.

2.7 Konštruktor

Každá sieť objektu môže byť inicializovaná pomocou zaslania správy `new` prislušnej triede. Trieda na túto správu reaguje vytvorením inštancie triedy s príslušnými počiatočnými stavmi v miestach. Pre parametrizovanú inicializáciu, je možné špecifikovať špeciálnu metódu konštruktoru. Pri zaslaní správy konštruktoru triede je najprv inicializovaná inštancia pomocou správy new a následne je jej zaslaná rovnaká správa. Metóda konštruktoru vždy vracia hodnotu self.

2.8 Snychrónny port

Synchrónny port umožňuje testovanie a prípadné zmeny stavu medzi objektami. Môže byť spojený s miestami pomocou hrán a môže obsahovať stráž. Podobne ako metóda má svoj vzor správy, na ktorý reaguje. Pri jej zaslaní môže byť argumentom aj voľná premenná. V takom prípade je jej naviazanie vyhodnocované rovnako ako pri prechode. V prípade, že port spĺňa s ohľadom na argumenty všetky hranové podmienky a stráž, vracia hodnotu `true`.

Kapitola 3

Návrh

V tejto kapitole je popísaný návrh výslednej implementácie modelu PNtalku. Bližšie sú rozobrané a oddôvodnené rozhodnutia, pri riešení problémov s konfliktnosťou a nekompabilitou typových kontrol jednotlivých jazykov, paralelizmu a optimalizáciám vyhodnocovania.

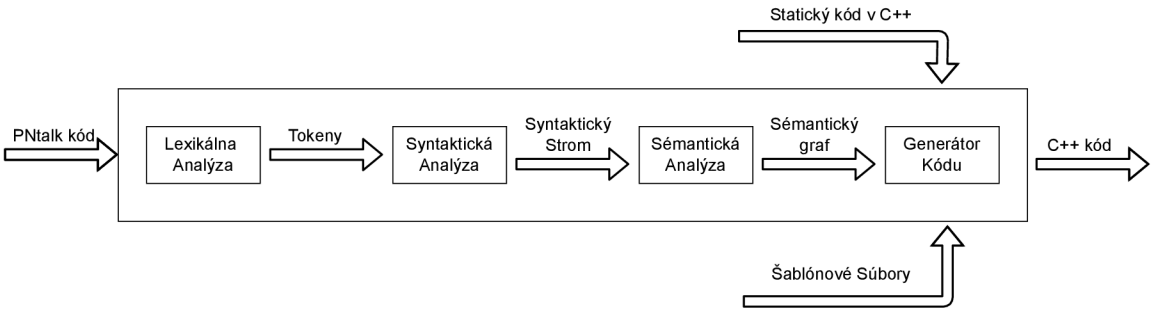
3.1 Prekladač

Vstupom prekladača sú súbory s kódom v jazyku PNtalk. Prekladač:

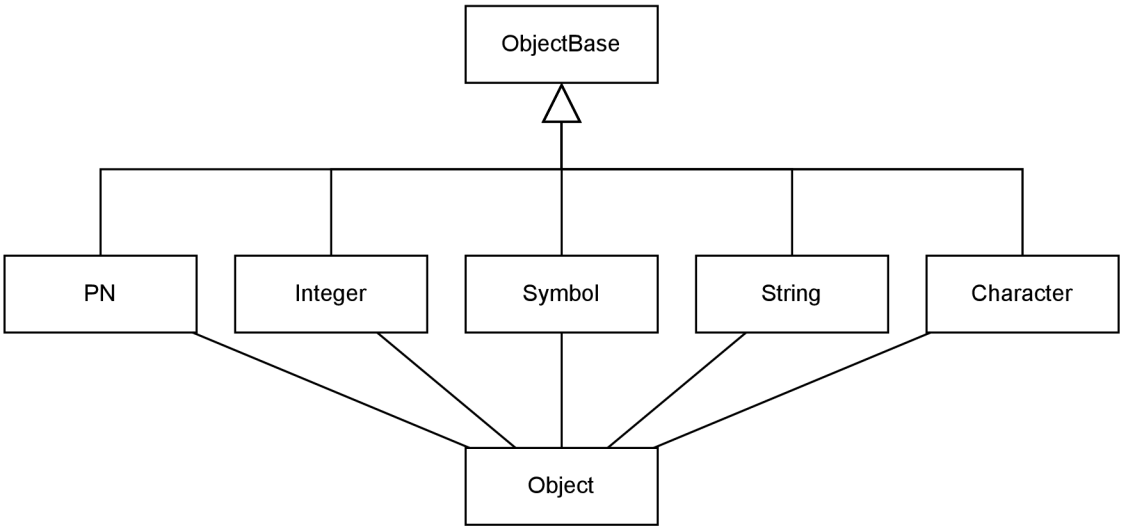
1. Vykoná lexikálnu analýzu a rozloží kód na tokeny.
2. Vykoná syntaktickú analýzu a zkonštruuje syntaktický strom.
3. Vykoná sémantickú analýzu a zkonštruuje sémantický strom.
4. Vygeneruje kód a vypíše ho do súborov v zadanom adresári.

Pokiaľ jedna z analýz nájde chybu, proces sa preruší a prekladač oznámi chybu na konzole.

Jazyk PNtalk vychádza a zároveň využíva jazyk Smalltalk. Keďže triedy jazyku PNtalk sa môžu odkazovať na rôzne triedy a dátové typy tohto jazyka, je potrebné implementovať niektoré z nich a súčasne poskytnúť jednoduchý spôsob, ako tieto triedy upravovať alebo pridať. Z toho dôvodu nieje možné, resp. je nepraktické, generovať kód zo statických reťazcov zo zdrojových súborov prekladača. Pre rieŠenie tohto problému budú zavedené šablónové súbory. Pomocou nich bude možné meniť alebo pridávať generovaný kód. Výhodou bude



Obr. 3.1: Diagram znázorňujúci vstupy, výstupy a komunikáciu medzi jednotlivými modulmi prekladaču.



Obr. 3.2: Diagram tried znázorňujúci vzťahy medzi triedami PNtalku.

aj jednoduchosť prípadnej zmeny výstupného jazyka, pre ktorú bude stačiť upraviť len tieto šablónové súbory. Šablónové súbory sa budú nachádzať na prístupnom mieste, kde ich bude môcť prekladač načítať. Vhodným riešením by bolo zahrnúť tieto šablóny aj do skompilovaného kódu pre prípad, že šablónové súbory nebudú k dispozícii.

3.2 Abstrakcia nad dátovými typmi

Keďže je PNtalk slabo typovaný jazyk a C++ naopak silno typovaný je potrebné zaviesť určitú abstrakciu nad jeho dátovými typmi. Táto abstrakcia musí poskytovať jednotné rozhranie pre zasielanie a delegovanie správ, možnosť ukladanie jednotlivých objektov do jedného dátového kontajneru jazyku C++ a súčasne možnosť prístupu k dátam jednotlivých typov resp. tried.

Možným riešením, ktoré využíva aj jazyk Smalltalk, je zaviesť základný prázdny dátový typ pre prázdny objekt s virtuálnymi prototypmi metód a od neho derivovať všetky ostatné dátové typy. K objektom by sa pristupovalo pomocou ukazateľa, aby bolo možné využiť virtuálnu tabuľku metód, a teda pri zaslaní správy zavolať správnu metódu. Pri tomto prístupe je však komplikované mať jednotné rozhranie pre prístup k dátam daného objektu. Bolo by potrebné mať uloženú informáciu o type objektu a podľa nej pretypovať daný ukazateľ na objekt. Ďalej by bolo potrebné mať virtuálny prototyp správ všetkých objektov v spomínanom t.j. základnom dátovom type. Viacero tried v jazyku Smalltalk implementuje správy s rovnakým názvom avšak s rozličným návratovým typom, čo sa tiež javí ako problém.

Z toho dôvodu bol zvolený iný prístup. Je stále zavedený základný dátový typ, ktorý deklaruje základné rozhranie pre zasielanie a príjmanie správ. Na obrázku 3.2 je možné sledovať, že od tohto typu sú derivované všetky ostatné triedy. Obsahuje asociatívne pole, kde sa nachádzajú páry *<reťazec:ukazateľ na metódu>*. Toto pole slúži pre vyhľadávanie metód, ktoré sú schopné spracovať a odpovedať na správu. Zaslanie správy bude vykonané volaním metódy, ktorá sa pokúsi nájsť daný selektor správy v spomínanom asociatívnom poli. Pokiaľ je objekt schopný na správu odpovedať, bude zavolaná príslušná metóda a pokiaľ nie, bude preposlaná správa triede, od ktorej je daný objekt derivovaný. Navyše

bude zavedená trieda, ktorá bude zjednocovať všetky dátové typy a poskytne možnosť preposielať im správy, zistiť ich dátový typ či získať jeho inštanciu. Je tým eliminovaná potreba deklarácie prototypov v základnom dátovom type a je poskytnuté rozhranie pre získavanie dát objeu.

3.3 Správy

Zasielanie správ bude vykonávané pomocou volania metódy `message`. Táto metóda poskytuje jednotné rozhranie pri zasielaní jak unárnych, binárnych ale aj správ s kľúčovými slovmi. Z toho dôvodu bude príjmať pole argumentov, ktoré predáva metódam implementujúcim jednotlivé správy. Jednotlivé metódy spracuvávajúce tieto správy budú svoje argumenty extrahovať z tohto poľa vo vopred určenom poradí.

3.4 Plánovanie

Keďže vyhodnocovanie Petriho sietí je paralelizovaný proces, je potreba paralelizmus zahrnúť aj do generovaného kódu. Je vhodné aby sa kontrolovanie vykonateľnosti prechodov vykonávalo len pri zmene stavu Petriho siete, resp. pri zmene parciálneho stavu, ktorý má vplyv na vykonateľnosť prechodu.

Pre riešenie tohto problému bude zavedený systém plánovania. V hlavnom vlákne programu bude kontrolovaná vykonateľnosť naplánovaných prechodov. V prípade vykonateľného prechodu bude vytvorené nové vlákno, v ktorom sa prechod uskutoční. Pri vytvorení novej inštancie siete, budú všetky jeho prechody naplánované pre kontrolu. Následne pri každej zmene parciálneho stavu, teda pri zmene stavu miesta, toto miesto naplánuje kontrolu všetkých prechodov, ktorých vykonateľnosť ovplyvňuje.

Kapitola 4

Prekladač

Beh prekladača je rozdelený na viacero procesov.

- Lexikálna analýza
- Syntaktická analýza
- Sémantická analýza
- Generácia kódu

4.1 Lexikálna analýza

Úlohou lexikálnej analýzy je skontrolovať a rozložiť čítaný text na menšie časti (tokeny), s ktorými sa nášmu prekladaču bude dalej lepšie pracovať. Každý takýto token nesie so sebou informácie o tom, akého je typu, kde sa nachádza (názov súboru, číslo riadku a pozíciu v riadku) a z akého textu pozostáva. Analyzátor je implementovaný pomocou regulárneho automatu. V prípade, že automat zostane v koncovom stave, je nastavený typ tokenu podla daného stavu. Pokiaľ automat neskončí v koncovom stave znamená to, že prečítaný vstup bol chybný. Užívateľ je upozornený na chybu formou chybovej hlášky v konzole a beh prekladaču je ukončený. V prípade, že bolo pre preklad zvolených viacero súborov, tak sú postupnosti tokenov z každého súboru spojené do jednej.

4.2 Syntaktická analýza

Úlohou syntaktickej analýzy je skontrolovať syntaktickú správnosť postupnosti tokenov a skonštruovať syntaktický strom. Zvolený postup je analýza zvrchu nadol pomocou rekurzívneho spádu. Pre každé pravidlo je implementovaná funkcia, ktorá skontroluje terminály daného pravidla a pre neterminály volá prisluchajúce funkcie.

4.2.1 Analýza výrazov

Syntaktická analýza výrazov je implementovaná pomocou precedenčnej analýzy za pomoci precedenčnej tabuľky. Precedenčná tabuľka určuje, čo sa má stať na základe vrchného terminálu na zásobníku a tokenu na vstupe.

V určitých prípadoch je však dvojrozmerná precedenčná tabuľka nedostatočná. Pri analýze výrazu: `array at: i + 1 put: 8` by sa pomocou dvojrozmernej tabuľky výraz vyhodnotil ako: `array at: i + (1 put: 8)` čo je samozrejme chybné. Pre vysporianie s týmto problémom je zavedená trojrozmerná tabuľka. V tejto tabuľke je vyhľadávané podľa vrchného terminálu na zásobníku a dvoch tokenov zo vstupu. Samozrejme vo väčšine prípadov pre rozhodnutie stačí len jeden token.

Keďže Smalltalk a aj PNtalk pri vyhodnocovaní výrazov nezohľadňujú prioritu matematických operácií, odráža to aj tabuľka precedencie. V prípade potreby je možné tabuľku vymeniť a obohatiť tak jazyk PNtalk o túto prioritu.

Precedenčná tabuľka vyberá medzi následujúcimi akciami:

- r – Aplikuje sa redukcia podľa jedného z redukčných pravidiel. Toto pravidlo sa vyberie podľa prvkov od najvrchnejšej zarážky až po vrchol zásobniku. Počas redukcie sa zo zásobníku vyberie spomínanná zarážka. V prípade, že žiadne pravidlo nevyhovuje, výraz nie je syntakticky správny.
- p – Do zásobníku sa vloží zarážka za vrchný terminál a na vrchol sa vloží prečítaný token.
- e – Na vrchol zásobníku sa vloží prečítaný token bez vkladanaia záložky.
- f – Na základe tokenu sa vyberie funkcia, ktorej sa dočasne prenechá analýza výrazu.
- x – Značí syntakticky nesprávny výraz.

Redukčné pravidlá

- E . E -> E – Konkatenácia výrazov.
- E . -> E – Ukončenie výrazu.
- E ; id -> E – Unárna kaskáda.
- E ; op E -> E – Binárna kaskáda.
- E ; (id: E)+ -> E – Kaskáda s kľúčovým slovom.
- id -> E – Čítanie hodnoty z premennej, prípadne. trieda pri volaní konštruktoru.
- lit -> E – Literál.
- (E) -> E – Zátvorky.
- E id -> E – Unárny výraz.
- E op E -> E – Binárny výraz.
- E (id: E)+ -> E – Zaslanie správy s kľúčovým slovom.
- id := E -> E – Priradenie hodnoty do premennej.

Vysvetlenie syntaxu redukčných pravidiel:

- E – výraz

- id – Identifikátor. V tejto fáze sa identifikátory premenných a identifikátory tried zatiaľ nerozlišujú.
- lit – Ľubovoľný literál primitívneho objektu. Akceptované sú taktiež rezervované identifikátory true, false a nil.
- op – Ľubovoľný binárny operátor.
- (...)+ – Iterácia. Znamená viacnásobný výskyt ... oddelený zarážkami.

4.3 Sémantická analýza

Sémantická analýza sa vykonáva prechodom abstraktného syntaktického stromu (ďalej len AST) zhora-nadol. Jej výsledkom je abstraktný sémantický graf (ďalej len ASG), ktorý ma obdobnú štruktúru ako AST. Rozdiel medzi ASG a AST spočíva prevažne v dátach, ktoré uchováva. Nachádzajú sa v ňom meta informácie o identifikátoroch ako napríklad rozah ich platnosti, miesto ich deklarácie...

4.4 Generácia kódu

Generátor kódu má za úlohu transformovať ASG na kód. Generátor prechádza strom v smere zhora-nadol. Pre každý uzol je vygenerovaný príslušný kód a následne je skombinovaný s kódom z uzlov pod ním. Kód pre jednotlivé uzly je generovaný pomocou šablónových súborov. Následne sa vygenerovaný kód skombinuje so statickým kódom. V tomto statickom kóde sa nachádza napr. implementácia primitívnych dátových typov PNtalku, implementácia základnych štruktúr a tried ako sú miesta, prechody alebo celé siete. Vygenerovaný kód je tak schopný používať a dediť metódy z týchto tried.

4.4.1 Šablónové súbory

Šablónové súbory majú príponu `.t` a nachádzajú sa v zložke `src/templates/compiler`. Tieto súbory sú pred generovaním všetky prečítané, spracované a uložené v manažéri šablón, pristupné pre generátor podľa názvu.

Počas generovania kódu zo šablóny je šablóne predané asociatívne pole textových náhrad. Šablóna pre každú dvojicu kľúč–hodnota z asociatívneho poľa vykoná náhradu všetkých výskytov. Pre nahradenie sa najprv vytvorí regulárny výraz pre slot pridaním predpony a prípony (`__`). Všetky výskyty tohoto slotu sú nahradené za reťazec z hodnoty. Pre príklad, šablóna v tvare:

```
class __class_name__ : public __base_class_name__ {
    __class_name__();
};
```

by po aplikácii asociatívneho poľa `{"class_name": "MyClass"}` vygenerovala kód:

```
class MyClass : public __base_class_name__ {
    MyClass();
};
```

alebo po aplikácii `{"class_name": "MyClass", "base_class_name":"Base"}` vygenerovala:

```
class MyClass : public Base {
    MyClass();
};
```

Pokiaľ je potrebné určité časti kódu len v určitých prípadoch, je možné použiť nepovinný slot. Nepovinný slot obsahuje klasický slot a v jeho okolí sa môže nachádzať podmienený kód (`/*? ... __slot_name__ ... */`); V prípade použitia tohto slotu, je slot odkomentovaný a nahradený za príslušnú hodnotu. Všetky nevyužité nepovinné sloty sú pred zápisom do súborov odstránené. Šablónu pre triedu je teda možné upraviť tak, aby nevyžadovala super triedu.

```
class __class_name__ /*? : public __base_class_name__*/{
__class_name__();
};
```

Takáto šablóna by po aplikácii `{"class_name": "MyClass"}` vygenerovala:

```
class MyClass {
    MyClass();
};
```

Pre zjednodušenie práce so šablónami a zvýšenie ich možností, sú zavedené rôzne direktívy. Všetky direktívy sa nachádzajú v špeciálnom blokovom komentári vo formáte: `/*! <direktiva> */`

- *Súbor.* Táto direktíva umožňuje upraviť cieľový súbor pre kód. Je možné skombinovať viacero direktív pre nastavenie súboru a pomocou toho generovať kód do viacerých súborov z jednej šablóny. Direktívu je možné upravovať pomocou slotov, čo umožňuje generovanie názvov súborov. Pri kombinovaní kódu z viacerých uzlov sa skombinuje kód patriaci do rovnakého súboru. Pokiaľ pre určitý súbor vygeneroval kód len jeden uzol, kód sa len skopíruje bez akéhokoľvek kombinovania. V prípade, že nie je zadaný názov súboru, kód bude spracovaný ako tzv. voľný kód. Voľný kód nie je viazaný na žiaden súbor a je vložený do slotu v ktoromkoľvek súbore. Direktíva je vo formáte `/*!file:<názov súboru>*/`.

  ```
  /*!file:__class_name__.h*/
  class __class_name__ : public __base_class_name__ {
      __class_name__();
  };

  /*!file:__class_name__.cpp*/
  __class_name__::__class_name__(){
  }

  /*!file:*/
  __class_name__();
  ```

Takáto šablona vygeneruje:

  - deklaráciu triedy v náležitom hlavičkovom súbore
  - definíciu konštruktoru v náležitom zdrojovom súbore
  - volanie konštruktoru ako voľný kód, ktorý môže byť vložený do ktoréhokoľvek súboru.

- *Substitúcia.* Pomocou substitúcie je možné nahradiť slot za akýkoľvek text. Táto operácia je vykonaná v dobe načítania šablóny. Umožňuje zjednodušiť a skrátiť zápis napr. dlhých ciest súborov. Direktíva je vo formáte `/*!slot:<názov slotu>:<text>*/`

  ```
  /*!slot:dirname:__class_name__/*?/__net_name__*//__trans_name__*/

  /*!file:__dirname__/init.cpp*/
  std::string dirname = "__dirname__"
  ```

  Takáto šablóna je po uplatnení substitúcie spracovaná ako:

  ```
  /*!file:__class_name__/*?/__net_name__*//__trans_name__/init.cpp*/
  std::string dirname = "__class_name__/*?/__net_name__*//__trans_name__"
  ```

  a po aplikovaní asociatívneho poľa `{"class_name": "C0", "transition_name": "t2"}` a následnom odstránení nepovinných slotov bude vyzerať:

  ```
  /*!file:C0/t2/init.cpp*/
  std::string dirname = "C0/t2"
  ```

- *Ignoruj.* Táto direktíva ovplyvňuje čítanie šablónového súboru. Spôsobuje preskočenie nasledujúceho neprázdneho riadku kódu. Vďaka tejto direktíve je možné špecifikovať deklarácie a definície pre integrované vývojové prostredie. Direktíva je vo formáte `/*!ignore*/`

- *Biele znaky.* Dve direktívy pre ovládanie bielych znakov. Umožňujú vložiť alebo vynechať znak nového riadku. Tieto direktívy nemajú žiaden funkčný vplyv na výsledný kód. Direktívy sú vo formáte `/*!inline*/` a `/*!newline*/`.

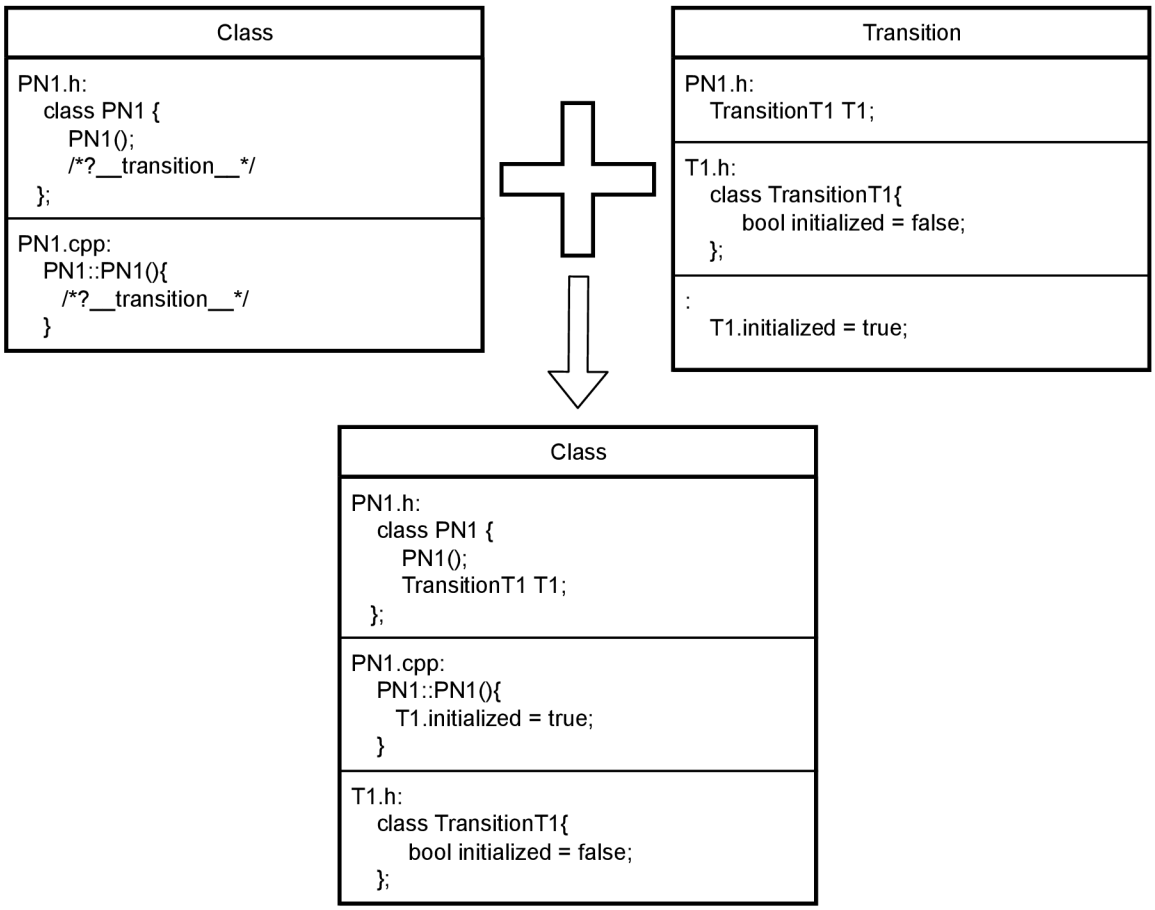
4.4.2 Kombinovanie kódu

Kód pre jeden uzol je vnútorne reprezentovaný ako asociatívne pole názov_súboru:kód. Pri kombinácii kódu sa nad príjemcom volá metóda `apply`. Táto metóda dostáva ako argument názov slotu a aplikovaný kód, s ktorým sa má skombinovať.

1. Pre každý súbor, ktorý obsahuje príjemcu, je na miesto špecifikovaného slotu vložený aplikovnaný kód. V prípade, že sa v aplikovanom kóde daný súbor nenachádza, použije sa voľný kód.

2. Každý súbor z aplikovaného kódu, ktorý sa v príjemcovi nenachádza, je skopírovaný do príjemcu.

3. V prípade, že sa voľný kód z aplikovaného kódu nepoužil, je vložený na miesto slotu alebo prekopírovaný[1] do voľného kódu príjemcu.



Obr. 4.1: kombinovanie dvoch kodov

4.4.3 Statický kód

Pod pojmom statický kód je myslený kód, ktorý sa nemení pre rôzne prekladané súbory. Ide o súbor zdrojových kódov pre triedy jazyku PNtalk resp. jazyku Smalltalk. Všetky súbory sú po vygenerovaní kódu prečítané. Následne sú skombinované s vygenerovaným kódom pomocou slotu `__generated__`. Týmto krokom je umožnené zasahovať a pridávať kód aj do statických súborov. Keďže je potrebné počas kompilácie zasahovať do týchto súborov, nie je možné ich skompilovať do zdielanej knižnice a používať z viacerých skompilovaných programov.

Súbory pre statický kód sa nachádzajú v zložke `src/templates/static`. Táto cesta môže byť ovplyvnená prepínačom z konzoly.

[1] Len v prípade, že voľný kód v príjemcovi je prázdny.

Kapitola 5

Generácia kódu

V tejto kapitole je podrobne popísaná implementácia tried, pre jednotlivé konštrukcie PNtalku. Vačšinou sa jedná o abstraktné triedy, ktoré sú derivované do konkrétnych vygenerovaných tried, reprezentujúcich model PNtalku. Mimo to, táto kapitola popisuje spôsob riadenia toku výsledného programu. Na obrázku 5.1 sú znázornené vzťahy, medzi jednotlivými triedami, spoločne reprezentujúcimi triedu jazyka PNtalk.

Stručný prehľad používaných pojmov:

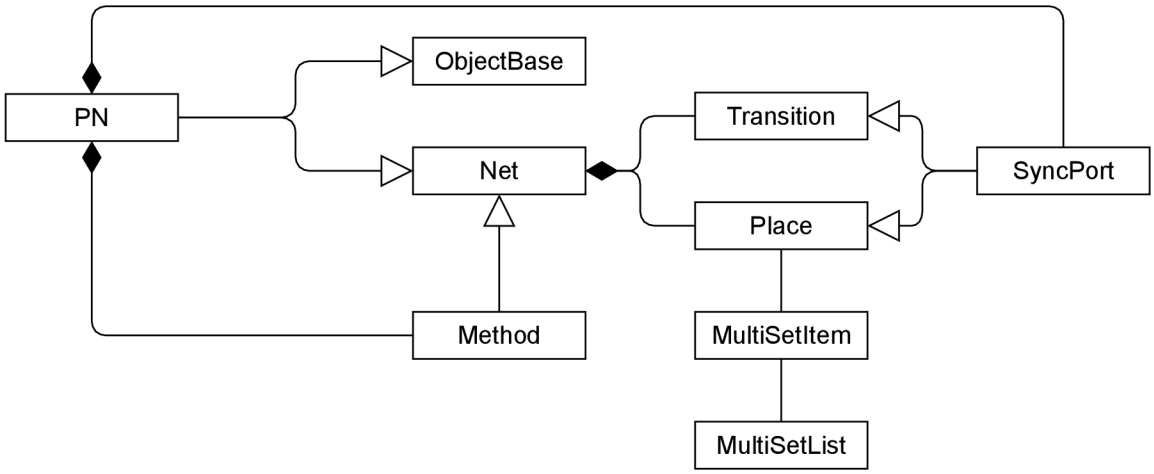
- Zdieľaný ukazateľ. Trieda `shared_ptr` štadnardnej knižnice c++, ktorá implementuje ukazateľ so správou pamäte. Tento objekt počíta počet ukazateľov na objekt, ktorý reprezentuje a v prípade, že dosiahne 0, automaticky dealokuje tento objekt zavolaním jeho deštruktoru.
- Slabý ukazateľ. Trieda `weak_ptr` štandardnej knižnice c++, ktorá je kompatibilná s triedou `shared_ptr` a implementuje ukazateľ na objek, ktorý však nevlastní tento objekt. V prípade, že na objekt neukazuje žiaden zdieľaný ukazateľ, je objekt dealokovaný a slabý ukazateľ sa tým stáva neplatným.
- Asociatívne pole. Trieda `map` štandardnej knižnice c++, ktorá implementuje binárny vyhľadávaci strom.
- Pole. Pod pojmom pole je zvačša rozumená inštancia triedy `deque` zo štandardnej knižnice c++. Táto trieda poskytuje možnosť pridávať a odstraňovať prvky zo začiatku ale aj z konca v konštantnom čase, pričom zachováva ukazatele na všetky ostatné prvky platné.

5.1 Plánovač

Plánovač (`Scheduler`) je trieda zaobstarajúca tvorenie vlákien pre vykonávanie prechodov. Jeho inštancia je globálna a prístupná odšadiaľ. Obsahuje frontu zdieľaných ukazateľov na prechody a pole inštancií vnorenej triedy `TransitionThread`. Pre plánovanie prechodu je možné využiť metódy `schedule`. Táto metóda bezpečne vloží ukazateľ na prechod na koniec fronty. Pre bezpečnosť prístupu k fronte a plánovaniu je použitý semafor.

Plánovač implementuje aj metódu `run` ktorá riadi beh programu:

1. Bezpečne sa vyberie prvý ukazateľ na prechod z fronty.



Obr. 5.1: Diagram tried popisujúci vzťahy medzi jednotlivými triedammi reprezentujúcimi model petriho siete.

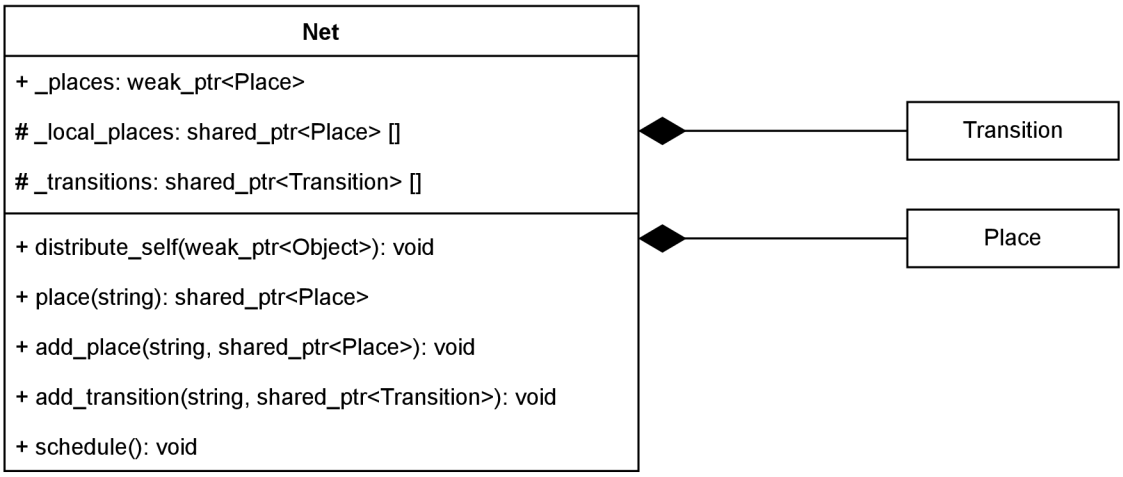
2. Zavolá sa virtuálna metóda `scheduler_check`, ktorá skontroluje uskutočniteľnosť prechodu.
3. Pokiaľ je prechod uskutočniteľný, je vytvorená inštancia triedy `TransitionThread` a tá je vložená do poľa `threads`;
4. Iteruje sa naprieč polom `threads`, pre každý prvok je zavolaná metóda `try_join` a v prípade že bolo vykonávanie vlákna už ukončené, zmaže tento prvok z poľa.
5. V prípade, že je fronta a aj pole vlákien prázdna, je ukončené vyhodnocovanie – model sa stabilizoval. Inak pokračuj bodom 1.

Trieda `TransitionThread` pri svojej inicializácii, vytvorí vlákno, v ktorom je spustená metóda `scheduler_execute` pre daný prechod. Vlákno po skončení vyhodnocovania tejto metódy nastaví atribút `finished` na pravdu. Pri volaní metódy `try_join` je prečítaná hodnota z tohto atribútu a na základe jej hodnoty je možné určiť či vykonávanie prechodu skončilo.

5.2 Sieť

Sieť (`Net`) je trieda reprezentujúca prázdnu Petriho sieť. Ako je možné pozorovať na obrázku 5.2, táto trieda obsahuje vnorené triedy pre miesto a prechod. Sieť obsahuje:

- Pole zdieľaných ukazateľov na miesta `_local_places`, kam sú ukladané ukazatele na inštancie miest danej siete.
- Asociatívne pole slabých ukazateľov na miesta `_places`, kam sú uložené ukazatele na všetky miesta ku ktorým má sieť prístup. V prípade vnorených sietí, ako je sieť metódy alebo konštruktoru, je toto asociatívne pole predané vnorenej sieti pri inicializácii. Vďaka tomu je možné z vnorených sietí pristupovať k miestam vrchnej siete.
- pole zdieľaných ukazateľov na prechody `_transitions`, kam sú uložené ukazatele na inštancie prechodov danej siete.



Obr. 5.2: Diagram tried reprezentujúci čisú petriho sieť.

Trieda siete implementuje aj metódy pre pridávanie prechodov a miest: `add_place`, `add_transition`. Pri zániku inštancie siete sú všetky prechody z poľa `_transactions` a všetky miesta z poľa `_local_places` korektne uvoľnené.

Trieda pre motódy a konštruktory (`Method`) je derivovaná od tejto triedy a neimplementuje nič navyše.

5.3 Multimnožina

Na obrázku 5.3 je znázornená trieda reprezentujúca multimnožinu, spoločne s pomocnými triedami.

5.3.1 MultiSet

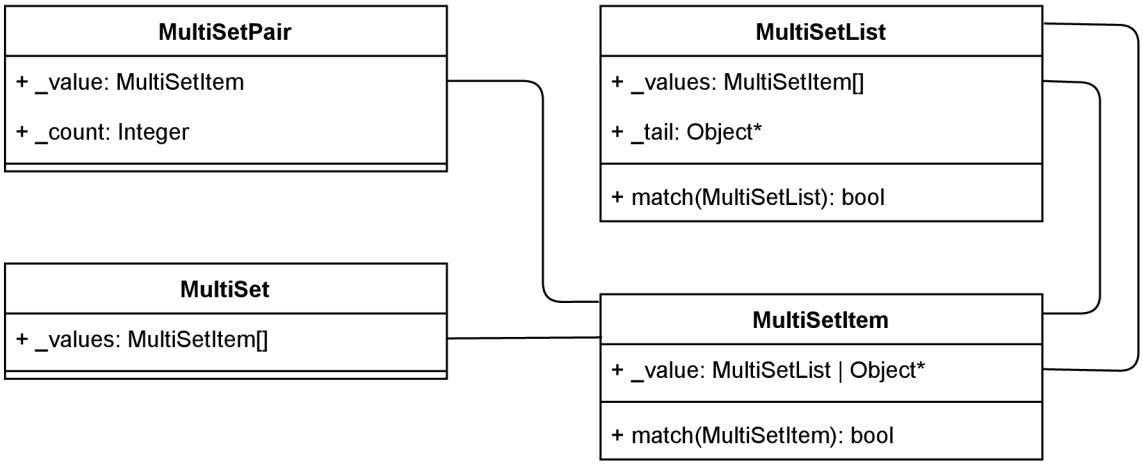
Trieda MultiSet je vrchnou triedou používanou pre reprezentáciu multimnožín. Je inicializovaná buď prázdna alebo pomocou poľa párov n‘t reprezentovných triedou `MultiSetPair`. Počas inicializácie sú tieto páry rozbalené, a teda v prípade páru `5‘2`, je do poľa s hodnotami vložený ukazateľ na číslo 2 práve päť krát. Niesú tvorené žiadne kópie objektov. Trieda obahuje už spomenuté pole hodnôt, v ktorom sa nachádzajú inštancie triedy `MultiSetItem`.

5.3.2 MultiSetPair

Trieda `MultiSetPair` je pomocná trieda pre inicializáciu multimnožín. Inicializuje sa buď dvojicou n‘t, kde n je zdieľaný ukazateľ na objekt celého čísla a t je buď ukazateľ na akýkoľvekt objekt PNtalku alebo inštancia triedy MultiSetList.

5.3.3 MultiSetItem

Trieda `MultiSetItem` je úniou obsahujúcou buď inštanciu triedy `MultiSetList` alebo zdieľaný ukazateľ na objekt PNtalku. Implementuje metódu `match` ktorá je využívaná pri naviazaní premenných z miesta do prechodu. Táto metóda má ako argument ďalšiu inštanciu triedy `MultiSetItem`, s ktorou má byť porovnaná, a podľa ich hodnôt má rôzne chovanie:



Obr. 5.3: Diagram tried reprezentujúcich multimnožinu.

- Pokiaľ obidve inštancie obsahujú list, teda inštanciu triedy `MultiSetItem`, je predané riadenie metóde match triedy `MultiSetList`.
- Pokiaľ obidve inštancie obsahujú voľnú premennú, ide o chybu, vypíše sa príslušná chybová hláška a vracia sa false.
- Pokiaľ obidve inštancie obsahujú platný ukazateľ na objekt, je pomocou zaslania správy vyhodnotená ich rovnosť a vrátena odpovedajúca hodnota.
- Pokiaľ jedna inštancia obsahuje voľnú premennú a druhá platný ukazateľ na objekt, je tento objekt naviazaný na voľnú premennú a vrátená hodnota true.
- Pokiaľ jedna inštancia obsahuje list a druhá nie, je tento stav vyhodnotený ako false.

5.3.4 MultiSetList

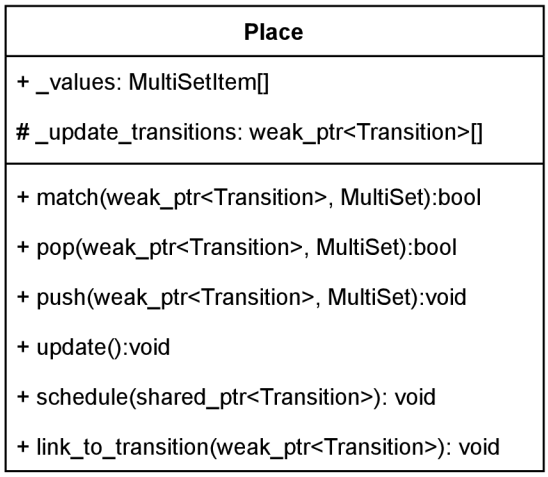
Trieda `MultiSetList` obsahuje pole inštancií triedy `MultiSetItem`. Implementuje metódu `match`, ktorá má argument dalšiu inštanciu triedy `MultiSetList`. Táto metóda súčasne prechádza cez polia hodnôt obidvoch inštancií a naväzuje ich hodnoty pomocou metódy `match` z triedy `MultiSetItem`. V prípade, že sa nájde dvojica prvkov, ktorá nevyhovuje alebo je počet prvkov v poliach rodzielny, je vyhodnocovanie ukončené a vrátená hodnota false. Pokiaľ však všetky prvky vyhovovali, je vrátená hondota true.

5.4 Miesto

Miesto (`Net::Place`) je vnorená trieda implementujúca miesto v Petriho sieti. Obsahuje pole tokenov `_values` reprezentujúce jeho stav a pole slabých ukazateľov na prechody, ktoré k nemu pristúpili `_update_transitions` reprezentujúce prechody, s ktorými je spojený pomocou vstupnej alebo testovacej hrany.

Na obrázku 5.4 môžeme pozorovať, že táto trieda implementuje viaceré metódy:2

- `match` a `pop`. Tieto metódy príjmaju ako parametre slabý ukazateľ na prechod a objekt multimnožiny. Ukazateľ je uložený do poľa `_update_transitions` a multimnožina je



Obr. 5.4: Diagram znázorňujúci triedu miesta.

naviazaná na pole tokenov. Metóda `pop` navyše tieto tokeny z miesta odstráni. V prípade, že je naviazanie uskutočniteľné, metódy vracajú hodnotu true.

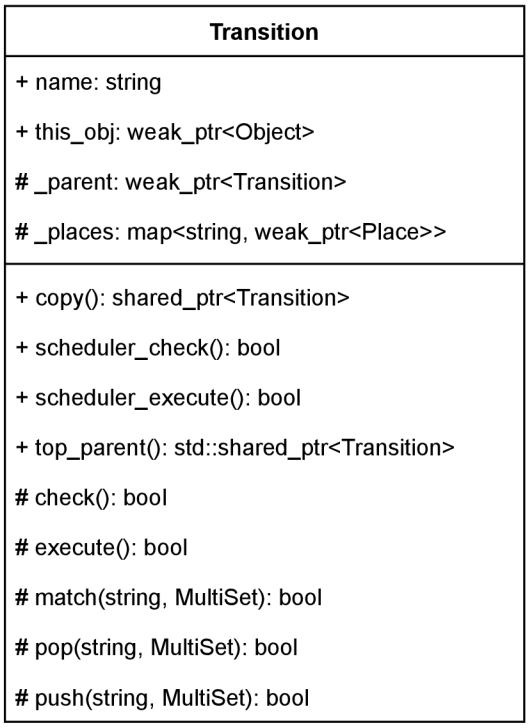
- `push`. Táto metóda príjma ako parameter multiset, z ktorého hodnoty vloží do poľa tokenov.
- `update`. Táto metóda je volaná pri zmene stavu miesta, teda pri zmene poľa `_values`. Prechádza pole `_update_transitions` a pre každý ukazateľ z neho:
  1. skontroluje či ukazateľ ukazuje na existujúcu inštanciu prechodu [1]. V prípade, že nie, zmaže tento prvok z poľa.
  2. podľa premennej `_name` z inštancie prechodu skontroluje či prechod už bol touto metódou naplánovaný. V prípade že už bol naplánovaný, preskakujú sa ďalšie kroky.
  3. vytvorí sa nová inštancia prechodu, pomocou metódy prechodu `copy`. Meno tohto prechodu je uložené do dočasného poľa naplánovaných prechodov. Táto inštancia je predaná plánovaču pre vyhodnotenie.

5.5 Prechod

Trieda prechodu (`Net::Trasition`) je vnorenou triedou implementujúcou prechod siete. Už z obrázku 5.5 sa dá vypozorovať, že táto trieda obsahuje viacero členov:

- reťazec s názvom typu daného prechodu `name`. Platí, že každá zderivovaná trieda má unikátny názov. Tento názov pozostáva z názvu triedy Petriho siete, prípadného názvu siete metódy alebo konštruktoru v prípade, že sa prechod nachádza v ich sieti a z názvu prechodu. Tento atribút je potrebný pre zamedzenie plánovania rovnakých prechodov jedným miestom, teda ide skôr o optimalizáciu.
- slabý ukazateľ na objekt tiredy, v ktorej sa nachádza `this_obj`. Tento atribút je potrebný v prípade zaslaní správ pseudopremennej self.

[1] V prípade, že prechod ukončil vykonávanie a bol len kópiou, prechod je dealokovaný.



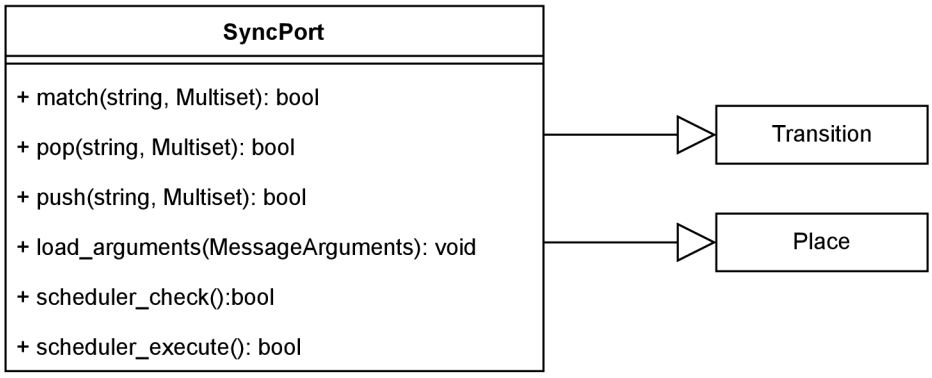
Obr. 5.5: Diagram znázorňujúci triedu reprezentujúcu prechod.

- asociatívne pole slabých ukazateľov na miesta `_places`. Pomocou týchto ukazateľov sa bezpečne pristupuje k miestam. Neplatný ukazateľ na miesto značí, že daná sieť bola dealokovaná. Tento stav môže nastať v prípade, že je prechod naplánovaný, teda jeho inštancia je vlastnená plánovačom, a skôr než sa vyhodnotí, je jeho sieť dealokovaná iným prechodom.
- deklaráciu virtuálnych metód `check`, `execute`, ktoré slúžia pre kontrolu uskutočniteľnosti a uskutočnenie prechodu. Tieto metódy sú exkluzívne virtuálne čo znamená, že je vyžadované, aby všetky zderivované triedy obsahovali ich implementáciu.
- deklaráciu a definíciu virtuálnych metód `scheduler_check` a `scheduler_execute`, ktoré sú len zapuzdreným volaním metód `check` a `execute`[2].
- deklaráciu a definíciu metód `match`, `pop` a `push`, ktoré berú ako argument reťazec s názvom miesta a multimnožinu reprezentujúcu hranový výraz. Zadané miesto nájdu v asociatívnom poli `_places` a volajú nad ním príslušnú metódu s rovnakým názvom.
- deklaráciu exkluzívne virtuálnej metódy `copy`, ktorá inicializuje novú inštanciu prechodu a vracia zdieľaný ukazateľ na túto inštanciu.

5.6 Synchrónny port

Trieda reprezentujúca synchrónny port `PN::SyncPort` je derivovaná od tried `Net::Place` a `Net::Transition`. To má za následok, že pri interakcii s prechodom, sa táto trida chová ako miesto a pri interakcii s miestom, sa chová ako prechod.

[2]Význam týchto metód je vysvetlený v 5.6



Obr. 5.6: Diagram reprezentujúci triedu synchrónneho portu.

Pre potreby plánovania prechodov, ovplyvnených synchrónnymi portmi je možné uložiť slabý ukazateľ na prechod do triedy Object. To umožňuje každému prechodu pri zasielaní priložiť redundandný posledný argument do správy, a to ukazateľ na seba samého, resp. na prechod, ktorý danú správu zasiela. Vďaka tomu môže príjemca správy zistiť, či a prípadne ktorý prechod danú správu zaslal. V prípade, že je príjemca inštancie vygenerovanej triedy Petriho siete a zaslaná správa odpovedá volaniu synchrónneho portu, je tento argument extrahovaný a uložený do inštancie synchrónneho prechodu do poľa `_update_transitions`. To má za následok, že všetky prechody, ktoré volajú tento synchrónny port, sú prihlásené k naplánovaniu, pri volaní metódy `update`.

Kedže pri zaslaní správy odpovedajúcej synchrónnemu portu bol synchrónny port otestovaný na uskutočniteľnosť a uskutočnený rovnako ako prechod, pokusom o naviazanie premenných na miesta, miesta s ktorými je spojený hranou ho už majú uložený medzi prechodmi, ktoré je potrebné naplánovať pri zmene stavu. Pokiaľ nebol synchrónny port volaný ani raz znamená to, že nieje naviazaný na žiaden prechod, nebude vyhodnotený ani raz a tým pádom ho nebudú plánovať ani miesta, s ktorými je spojený.

Vďaka derivácii triedy pre prechod znázornenej na obrázku 5.6, je možné vložiť ukazateľ na objekt synchrónneho portu do plánovaču a ten môže volať preťažené metódy `scheduler_check` a `scheduler_execute`.

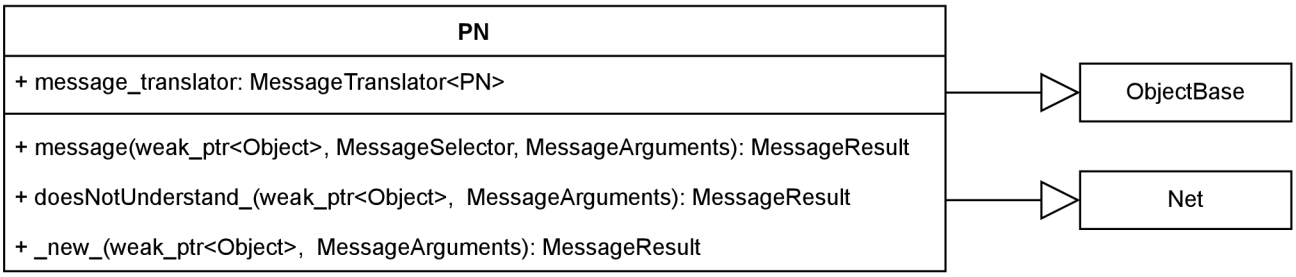
- `scheduler_check` – Vracia vždy true. To znamená, že scheduler vždy vytvorí vlákno a v ňom zavolá metódu `scheduler_execute`.
- `scheduler_execute` – Táto preťažená metóda vykoná metódu `update` z triedy `Net::Place`, a tým naplánuje všetky prechody, ktoré sú na daný synchrónny port naviazané.

5.7 Trieda PNtalku

Trida `PN` je vrchná trieda reprezentujúca sieť PNtalku. Je derivovaná od triedy `Net`, teda spĺňa všetky vlastnosti siete a od triedy `BaseObject`. Navyše obsahuje vnorené triedy `Method` a `SyncPort`. Na obrázku je možné pozorovať, že táto trieda už implementuje metódu `message`, ktorá má na starosti spracovávanie správ.

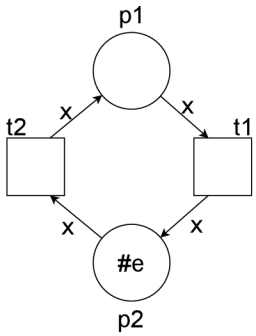
5.8 Dynamika plánovania prechodov

Majme jednoduchú sieť znázornenú ako na obrázku 5.9:



Obr. 5.7: Diagram reprezentujúci triedu PN

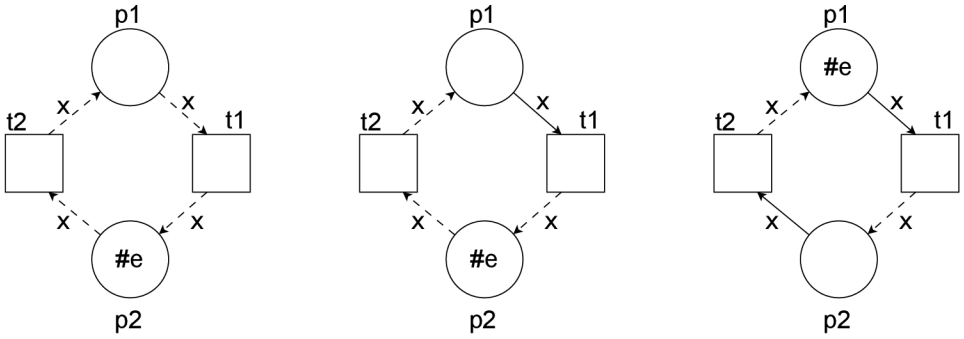
```
class SchedulerDynamic is_a PN
    object
        place p1()
        place p2(#e)
        trans t1
            precond p1(x)
            postcond p2(x)
            trans t2
            precond p2(x)
            postcond p1(x)
```



Obr. 5.8: Jednoduchá petriho sieť pre ukážku dynamiky plánovania

Pri inicializácii tejto siete sú vytvorené všetky miesta a prechody. Pre účely ukážky, na obrázku5.9 reprezentujú plné hrany existujúce prepojenia pre budúce plánovanie, resp. symbolizujú prítomnosť ukazateľa na prechod v poli `_update_transitions` a prerušované čiary značia len prepojenie, umožňujúce naväzovanie premenných. Po inicializácii 5.9a je zavolaná metóda `PN::schedule`, ktorá naplánuje vŠetky prechody v sieti.

Plánovač vyberie prechod *t1*. Pomocou volania metódy `Net::Transition::schedule_check` skontroluje jeho uskutočniteľnosť 5.9b. Počas tejto kontroly je volaná metóda `Net::Place::match`, nad miestom *p1*. Výsledok tejto metódy síce značí nemožnosť uskutočniť prechod, avšak ako vedľajší účinok má miesto *p1* uložený pokus o prístup od prechodu *t1*.



(a) Hrany po inicializácii siete (b) Hrany otestovaní t1 (c) Hrany vykonaní t2

Obr. 5.9: Jednoduchá petriho sieť pre ukážku dynamiky plánovania

Plánovač vyberie *t2*. Je skontrolovaná jeho uskutočniteľnosť a podobne ako pri predchádzajúcom prechode je ako veľajší účinok prístupu k miesto vytvorenie prepojenia 5.9c. Tento prechod však je uskutočniteľný, a teda je vytvorené nové vlákno a volaná metóda vykonávajúca prechod. Po vložení prechodu, pri volaní metódy `push` nad miestom *p2*, je volaná aj metóda `update`. Keďže miesto už má vytvorené prepojenie s prehodom *t1*, je tento prechod naplánovaný.

5.9 Súborová štruktúra

V tejto podkapitole je stručne popísaná štruktúra vygenerovaného kódu. Štruktúra bola navrhnutá tak, aby umožnila jednoduchú orientáciu vo veľkom množstve vygenerovaného kódu a teda zjednodušila a umožnila proces, manuálneho odstránenia chýb. Každá dôloežitá trieda sa náchadza vo vlastnej zložke v ktorej je zvačša osobitný súbor pre každú dôležitejšiu funkciu. Súčasťou vygenerovaných súborov je aj Makefile zabezpečujúci preklad vysledných zdrojových súborov.

5.9.1 Trieda siete

Každá vygenerovaná trieda jazyku PNtalk sa celá nachádza v zložke s rovnakým názvom. Táto zložka obsahuje hlavičkový subor `header.h` v ktorom sa nachádzajú všetky potrebné deklarácie pre miesta, prechody, metódy a konštruktory, snchrónne porty, metódy spracúvavajúce správy atď. Nachádzajú sa tu aj všetky zložky vnorených tried. Ďalej táto zložka obsahuje súbory:

- `init.cpp` – kde sa nachádzajú inicializácie miest, prechodov, synchronnych portov a asociatívneho poľa pre preklad správ.
- `message.cpp` – kde sa nachádzajú definíce metód pre spracovanie správ. Nájdeme tu implementáciu metódy message, ktorá má za úlohy vybrať a zavolať správnu metódu a metód pre spracovanie správ: rovnosti (porovnanie typu), doesNotUnderstand správy vypisujúcej chybovú hlášku pri neporozumení správy, new implicitného konštruktoru ...
- `message_*.cpp` – kde sa nachádzajú vygenerované metódy volané pri volaní metódy, konštruktoru alebo synchrónneho portu.

Každá trieda navyše vygeneruje kód do súborov s názvami `object/object_variant.h`, `object/object.h` a `object/object_message.cpp`, ktorý umožňuje prístup k tejto triede z triedy `Object`. Vďaka tomu je možné inštanciám vygenerovaných tried zasielať správy alebo ich aj použiť ako argumenty pri zasielaniach správ.

5.9.2 Metóda

Každá metóda špecifikovaná v generovanej triede PNtalku sa nachádza vo svojej podzložke. Táto zložka obsahuje hlavičkový súbor s potrebnými deklaráciami, súbor `init.cpp` obsahujúci inicializáciu miest a prechodov siete metódy a zložky pre jednotlivé miesta a prechody. V zložke triedy je vygenerovaný súbor `message_<názov_prechodu>.cpp`, v ktorom je implementovaná funkcia, ktorá:

1. alokuje voľnú premennú pre výsledok

2. vytvorí a inicializuje novú inštanciu triedy, reprezentujúcej danú metódu
3. naplánuje prechody siete metódy k vyhodnoteniu.
4. pasívne čaká dokiaľ nebude miesto return prázdne
5. naviaže voľnú premennú pre výsledok na hodnotu z miesta return.
6. vracia premennú reprezentujúcu výsledok

5.9.3 Prechod

Každý prechod sa nachádza vo vlastnej podzložke či už v zložke metódy alebo v zložke triedy. Obsahuje hlavičkový súbor s deklaráciami, súbor `init.cpp` obsahujúci alokácie premenných, a súbory:

- `check.cpp` obsahujúci definíciu funkcie vykonávanjúcu kontrolu vykonateľnosti prechodu. Pre každú vstupnú a testovaciu hranu, sa tu nachádza volanie metódy `Net::Transition::match`, ktorá sa pokúsi naviazať multimnožinu na dané naviazané miesto. V prípade špecifikovanej stráže prechodu je po naviazaní premenných vyhodnotená konjukcia všetkých výrazov zo stráže, vrátane volaní na synchrónne porty.
- `copy.cpp` obsahujúca definíciu funkcie `copy`, ktorá tvorí novú inštanciu pre daný prechod. Táto funkcia naalokuje novú inštanciu a následne jej predá asociatívne pole pre vyhľadávanie miesta. Všetky ostatné atribúty sú naninicializované prázdne resp. na predvolené hodnoty.
- `execute.cpp` obsahujúca definíciu funkcie `execute`, ktorá vykoná prechod. V tejto funkcii sa nachádzajú:
  1. volania metódy `Net::Transition::pop` pre každú vstupnú hranu prechodu
  2. volania metódy `Net::Transition::match` pre každú testovaciu hranu
  3. kód vyhodnocujúci stráž prechodu
  4. kódy vyhodnocujúci akciu prechodu
  5. volania metódy `Net::Transition::push`, pre každú výstupnú hranu

5.9.4 Miesto

V podzložke pre miesto sa nachádzajú vždy práve 2 súbory: hlavičkový súbor s potebnými deklaráciami a súbor init.cpp s inicializáciou miesta.

Kapitola 6

Transformované triedy Smalltalku

Keďže aktuálna implementácia simulátora PNtalku beží v rámci prostredia Pharo, sú objektom PNtalku prístupné triedy jazyku Smalltalk. Pre zachovanie kompability je teda potrebné tieto triedy transformovať do jazyku C++ tak, aby ich pretransformované modely mohli využívať. V tejto kapitole je popísaná implementácia niektorých týchto transformovaných tried Smalltalku. Ďalej je podrobne popísaný postup prípadnej implementácie ďalších tried.

6.1 Objekt a premenná

Všetky objekty PNtalku môžu byť priradené do inštancie triedy `Object`. Pod triedami PNtalku rozumieme implementované triedy Smalltalku (`Character`, `Integer`, `Bool`, `String`, `Array` a pod.), a všetky triedy vygenerované prekladačom, resp. všetky triedy derivované od triedy `PN`. Vnorené pomocné triedy triedy `PN` ako sú `Net`, `Place`, alebo `Method`, sú triedou `PN` iba využívané a nieje možné k nim pristupovať pomocou triedy `Object`.

Trieda `ObjectBase` je abstraktnou triedou obsahujúcou základné deklarácie a definície atribútov a metód pre všetky triedy PNtalku.

Ako je vidno aj na obrázku 3.2 každá trieda PNtalku je derivovaná od tejto triedy. Deklaruje pomocné substitúcie typov, napr: `MessageResult` špecifikujúci návratovú hodnotu metód, implementujúcich spracovávanie správ alebo `MessageTranslator<T>` reprezentujúcu asociatívne pole, využívané k výberu správnej metódy na spracovanie správy. Táto trieda implementuje operátory `==` a `~==`. Tie vyhodnotia identitu objektu porovnaním ukazateľov na seba samého a argument prijatý v správe.

Trieda `Object` implementuje úniu nad týmito triedami s využitím kontajneru `variant` zo štandardnej knižnice C++. Pri inicializácii alokuje novú inštanciu daného typu a ukladá len zdieľaný ukazateľ na túto novo vzniknutú inštanciu. Pokiaľ by priamo kopírovala objekt, prikladanie argumentov do volaní správ by spôsobovalo kopírovanie týchto objektov, a teda možné nedefinované chovanie alebo chyby pri preklade.[1]

6.2 Implementované triedy Smalltalku

Implementované triedy Smalltalku odpovedajú dokumentácii z [1].

- `Character` – Implementované základné porovnávacie operácie

[1] Napr trieda `Net::Place`, obsahuje semafór typu `std::mutex`, ktorý nepodporuje kopírovanie

- `Integer` – Implementované základné bitové a numerické operácie
- `String` – Implementované základné metódy. Porovnávanie.
- `Array` – Implementované základné metódy. Pridávanie, odstránenie prvkov, vyhľadávanie a iterácia.
- `Symbol` – Implementované základné porovnávacie operácie
- `Transcript` – Implementované metódy `shwo:` a `cr`

6.3 Transformácia novej triedy

V tejto podkapitole je rozobratý postup implementácie tried Smalltalku ako krokovaný návod:

1. Začneme vytvorením súborov pre novú triedu. Súbory sa musia nachádzať v zložke `src/templates/static` alebo musí byť cesta k týmto súborom zadaná ako argument `static_templates` v konzole. Na konkrétnej podzložke nezáleží, avšak aj táto zložka má svojú štruktúru. V podzložke `net` sa nachádzaju zdrojové súbory tried popísané v kapitole5, využívané modelmi Petriho sietí. V podzložke `object` sa nachádzajú súbory transformovaných tried Smalltalku a voľne sú niektoré výnimočné súbory, nezapadajúce do tohto rozdelenia, ako napriklad súbor `main.cpp`.

2. Do hlavičkového súboru deklarujeme triedu, v mennom priestore PNtalk, so správnym názvom. Je nevyhnutné, aby sa transformovaná trieda volala rovnako[2].

3. Hlavičkový súbor obsahujúci deklaráciu novej triedy zahrnieme do hlavičkového súboru `object/object_variant.h` pomocou direktívy include.

   Do definície typu `ObjectVariant` zahrnieme aj nadeklarovanú triedu. To umožní do objektu `Object` uložiť inštanciu novej triedy.

4. V súbore `object/object.h` deklarujeme nový konštuktor triedy `Object` ktorý bude mať našu novú triedu ako argument a do súboru `object/object.cpp` tento konštruktor definujeme.

5. V súbore `object/object_message.cpp` pridáme do metódy message, podmienku pre presmerovanie správy novej treide.

```
if (is_type<PNtalk::NovaTrieda>())
    return get<PNtalk::NovaTrieda>().
        message(shared_from_this(), message_selector, arguments);
```

6. Definujeme triedu. Trieda musí byť derivovaná z triedy `ObjectBase`, no nemusí byť derivovaná priamo. Musí obsahovať deklarácie týchto členov:

[2] Záleží aj na rozličovaní veľkých a malých písmen.

```
NovaTrieda();
MessageTranslator<NovaTrieda> message_translator;
MessageResult message(ThisObj this_obj,
    const MessageSelector &message_selector,
    MessageArguments arguments);
MessageResult doesNotUnderstand_(ThisObj this_obj,
    MessageArguments arguments);
    \begin{verbatim}
```

7. Definujeme predvolený konštruktor `NovaTrieda()` a v ňom budeme inicializovať asociatívne pole `message_translator`. Pre každú správu, ktorej rozumie objekt Smalltalkovej triedy, ktorú implementujeme, je potrebné vložiť dvojicu selektor – metóda nasledujúcim spôsobom:

```
message_translator["doesNotUnderstand_"] =
    &NovaTrieda::doesNotUnderstand_;
```

8. Definujeme metódu `message` nasledujúco:

```
MessageResult NovaTrieda::message(ThisObj this_obj,
        const MessageSelector &message_selector,
        MessageArguments arguments) {
    if (message_translator.find(message_selector) ==
            message_translator.end()) {
        return super::message(this_obj,
            message_selector, arguments);
    }
    return (this->*(message_translator[
        message_selector]))(this_obj, arguments);
}
```

Táto definícia využíva substitúciu typu super. Pre pridanie tejto substitúcie je potrebné pridať do definície: `typedef SuperTrieda super`, kde je názov `SuperTrieda` nahradený za názov triedy, z ktorej je naša nová trieda derivovaná.

Kapitola 7

Testovanie

Pre testovanie prekladača neboli implementované žiadne automatizované testy alebo veľké testovacie sady. Prekladač zvládol preložiť aj komplexnejšie modely, ktoré boli overované manuálne. Samozrejme pri paralelizovanom vyhodnotenocovaní nieje možné všetko overiť. Výsledkom pokusu o porovnanie rýchlosti implementácií požitom aj v [2] bolo približne 5-násobné zrýchlenie. Tento výsledok však je len informatívny. Porovnanie totiž neprebehlo za rovnakých podmienok. Trieda `Time` nieje implementovaná, a preto nemohli byť použité interné stopky. Pri teste Ackremanovej funkcie je vždy potrebné čakať na výsledok metódy a teda všetky bežiace vlákna len pasívne čakajú na výsledok. Naopak je pre alokáciu vlákien v tomto prípade spotrebovaný čas navyše. Samozrejme pre plnohodnotné otestovanie by bolo potrebné implementovať radu rôzne komplexných testov.

Pri testovaní bolo odhalených niekoľko nedostatkov či už v návrhu alebo implementácii.

- Sémantická analýza pri dedičnosti. Počas sémantickej analýzy sú dáta o identifikátoroch kontrolované len pre aktuálnu triedu. Pri derivovaní z triedy inej ako je `PN` teda sémantická analýza nepočíta s dedením prvkov zo super triedy. Paradoxne táto chyba neovplyvňuje funkcionalitu vygenerovaného kódu, keďze prekladač jazyku C++ takýto problém nemá.

  Ďalším nedostatkom sémantickej analýzy je inicializovanie premennej pomocou synchrónneho portu. Pokiaľ premenná nemá priradenú hodnotu alebo nieje na zozname dočasných premenných pred volaním portu, sémantická analýza vypíše varovanie. V dobe sémantickej analýzy však nieje možné rozlíšiť volanie metódy konštruktoru alebo portu, keďže ku všetkým sa pristupuje pomocou správy, ktorú spracúvava príjemca.

- Plánovanie prechodov pri testovacej hrane. Počas vykonania prechodu, ktorý je spojený s miestom testovacou hranou, je premenná len naviazaná. V prípade, že by v rámci prechodu bola tejto premennej zaslaná správa, ktorá akokoľvek upravuje hodnotu objektu, na ktorý ukazuje, neboli by naplánované prechody naviazané na toto miesto.

- V určitých modeloch môže dojsť k uviaznutiu. V určitých situáciách pri volaniach metód, keďže metódy čakajú na výsledok a tým pozastavia vlákno, je možné, že dôjde k uviaznutiu. Tento stav by sa dal chápať aj ako chyba modelu nie prekladaču. Každopádne by však bolo korektnejšie buď sa uviaznutiu vyvarovať alebo byť schopný ho detekovať a následne ukončiť vyhodnocovanie.

Kapitola 8

Záver

Cieľom práce bolo navrhnúť a implementovať prekladač z jazyka PNtalk do C++. V práci bol navrhnutý a implementovaný prekladač, ktorý analyzuje kód z jeho vstupu a vygeneruje z neho fungujúci kód. Jednotlivé moduly prekladaču sú implementačne oddelene a ich medzivýsledky sú jasne definvané, vďaka čomu je možne v prípade potreby upraviť len chybový modul. Vďaka systému šablónových súborov, je možné výslednú implementáciu jednoducho modifikovať, bez zložitého vyhľadávania v zdrojových súboroch jazyka C++. Ďalej je možné jednoducho implementovať chýbajúce triedy jazyku Smalltalk.

V budúcnosti by bolo vhodné opraviť spomínané nedostatky sémantickej analýzy. Pre túto zmenu by bolo potrebné prepísať celý modul sémantickej analýzy a zmeniť spôsob a štruktúry ukladaných dát.

Možným rozšírením by mohla byť aj optimalizácia generovaného kódu. Aktuálna implementácia využíva veľa reťazcov a dynamicky alokovaných pamäťových blokov. Práca s týmito dátovými typmi, môže mať za následok pokles rýchlosti implementácie.

Vítaným rozšírením by bola taktiež implementácia ďalších, netriviálnych tried jazyku smalltalk, ktoré by dodržovali hierarchiu tried zo Smalltalku. To by malo za následok odstránenie duplicitného kódu vďaka využitiu dedičnosti.

Literatúra

[1] Free Software Foundation, I. *GNU Smalltalk* [online]. 2019 [cit. 2019-10-02]. Dostupné z: https://www.gnu.org/software/smalltalk/.

[2] Hanák, M. *Generování kódu z Objektově orientovaných Petriho sítí*. Brno, CZ, 2015. Master's thesis. Brno University of Technology, Faculty of Information Technology. Dostupné z: https://www.fit.vut.cz/study/thesis/17932/.

[3] Janoušek, V. *Modelování objektů Petriho sítěmi*. Brno, CZ, 1999. Ph.D. thesis. Brno University of Technology, Faculty of Information Technology. Dostupné z: https://www.fit.vut.cz/study/phd-thesis/124/.

Príloha A

Precedenčná tabuľka

	$	id	lit	(	)	.	;	:=	:	[	#	+	-	*	/	//	%	&	|	===	!==	==	!=	<	>	<=	>=
$		p	p	p	x	p	p	x	x	x	{2}	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p
id	r	r	r	x	r	r	r	e	e	x	x	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
lit	r	r	r	x	r	r	r	x	x	x	x	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
(	r	p	p	p	e	x	x	x	x	x	{2}	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p
)	r	x	x	x	r	r	r	x	x	x	x	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
.	r	p	p	p	x	r	p	x	x	x	{2}	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p
;	x	e	x	x	x	x	x	x	x	x	x	e	e	e	e	e	e	e	e	e	e	e	e	e	e	e	e
:=	r	p	p	p	r	r	p	x	x	x	{2}	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p
:	r	p	p	p	r	r	r	x	x	f	{2}	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p
[	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
#	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
+	r	{1}	p	p	r	r	r	x	x	x	{2}	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
\-	r	{1}	p	p	r	r	r	x	x	x	{2}	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
*	r	{1}	p	p	r	r	r	x	x	x	{2}	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
/	r	{1}	p	p	r	r	r	x	x	x	{2}	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
//	r	{1}	p	p	r	r	r	x	x	x	{2}	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
%	r	{1}	p	p	r	r	r	x	x	x	{2}	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
&	r	{1}	p	p	r	r	r	x	x	x	{2}	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
|	r	{1}	p	p	r	r	r	x	x	x	{2}	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
===	r	{1}	p	p	r	r	r	x	x	x	{2}	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
!==	r	{1}	p	p	r	r	r	x	x	x	{2}	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
==	r	{1}	p	p	r	r	r	x	x	x	{2}	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
!=	r	{1}	p	p	r	r	r	x	x	x	{2}	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
<	r	{1}	p	p	r	r	r	x	x	x	{2}	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
>	r	{1}	p	p	r	r	r	x	x	x	{2}	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
<=	r	{1}	p	p	r	r	r	x	x	x	{2}	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
>=	r	{1}	p	p	r	r	r	x	x	x	{2}	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Tabuľka A.1: Precedenčná tabuľka

	$	id	lit	(	)	.	;	:=	:	[	#	+	-	*	/	//	%	&	|	===	!==	==	!=	<	>	<=	>=
{1}	p	p	p	p	p	p	p	p	r	x	x	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p
{2}	x	x	x	f	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Tabuľka A.2: Substitúcie v precedenčnej tabuľke