VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

INTERPRET PETRIHO SÍTÍ

INTERPRETER OF PETRI NETS FORMALISM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **ONDREJ ŠAJDÍK**
AUTHOR

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

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Šajdík Ondrej**
Program: Informační technologie
Název: **Interpret Petriho sítí**
Interpreter of Petri Nets Formalism
Kategorie: Překladače

Zadání:

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

Literatura:

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

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2.

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

Vedoucí práce: **Kočí Radek, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 28. května 2020
Datum schválení: 31. října 2019

Abstrakt

Cieľom bakalárskej práce bolo vytvoriť nástroj schopný interpretovať Objektovo orientované Petriho siete. V teoretickej časti boli preskúmané Petriho siete, jeho jednoduché rozšírenia, Objektovo orientované Petriho siete a modelovanie v jazyku PNtalk. V praktickej časti práce bol navrhnutý a implementovaný nástroj pre načítanie modelu definovaného v jazyku PNtalk do vnútornej reprezentácie a vykonávanie jeho dynamiky. Na záver bola vytvorená sada automatizovaných testov pre zabezpečenie kvality a skupina modelov pre demonštráciu vlastností interpreta.

Abstract

The goal of bachelor thesis was to create tool which would be able to interpret Petri nets. In theoretical part was researched Petri nets, its simple extensions, Object-oriented Petri nets and modeling in language PNtalk. In practical part of thesis was designed and implemented tool for loading model defined in language PNtalk into inner representation and executing its dynamics. In the end was created set of automated tests for quality assurance and group of models for demonstration of interpret features.

Kľúčové slová

Petriho sieťe, Objektovo orientované Petriho sieťe, PNtalk, Interpret

Keywords

Petri nets, Object-oriented Petri nets, PNtalk, Interpret

Citácia

ŠAJDÍK, Ondrej. *Interpret Petriho sítí.* Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Kočí, Ph.D.

Interpret Petriho sítí

Prehlásenie

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

.....
Ondrej Šajdík
28. mája 2020

Poďakovanie

Rád by som týmto poďakoval doktorovi Radkovi Kočímu za vedenie pri vypracovávaní práce a cenné rady.

Obsah

Kapitola 1

Úvod

Petriho siete sú populárnym nástrojom pre návrh, modelovanie a analýzu distribuovaných a paralelných systémov. Existuje viacero variánt Petriho sietí vytvorených s cieľom uľahčiť modelovanie a statickú analýzu systému. Pre dynamickú analýzu pozorovaním behu modelovaného systému je potreba vytvoriť spustiteľný program reprezentujúci model. Takto vytvorený program spustiť a pozorovať výsledky. Použitím získaných znalosti o systéme upraviť model a následkom toho i spustiteľný program pre ďalšie pozorovanie. Tvorba spustiteľného programu umožňujúci túto činnosť, nie je vždy jednoduchá a zaberie čas. Preto vytvorením nástroja, ktorý by bol schopný spúšťať model priamo, by sa odstránila potreba vytvárať model i spustiteľný súbor zároveň a tak sa potencionálne zjednodušil tento proces.

V práci sú popísané Petriho siete. Ich štruktúra, dynamika a niekoľko jednoduchých rozšírení umožňujúcich lepší popis modelovaného systému. Na tomto základe je vysvetlený princíp Objektovo orientovaných Petriho sietí a jazyka PNtalk, pre ich zápis. Jedná sa o variantu Petriho sietí, ktorá pridáva možnosť využívať výhod objektovej orientácie, podporujúcej okrem iného aj rozdelenie systému na podsystémy a tým uľahčenie modelovania a analýzy rozsiahlejších, komplikovanejších systémov.

V praktickej časti je navrhnutý a implementovaný nástroj pre simuláciu Objektovo orientovaných Petriho sietí. Naviac výsledné riešenie rozširuje PNtalk o časti, ktoré sú nevyhnutné pre pozorovanie výsledkov simulácie a časti, ktoré uľahčujú modelovanie systému. Napríklad modelovanie času a náhody.

Pre zabezpečenie kvality a správneho správania implementovaného nástroja je vytvorená sada testov, zložených z jednoduchého modelu a očakávaného výsledku. Pre odstránenie nutnosti ich vykonávať manuálne, je vytvorený automatizovaný skript, ktorý modely spúšťa a kontroluje výsledky. Na záver je vytvorených niekoľko príkladov modelov pre demonštráciu dynamiky siete a popis ich interpretácie implementovaným nástrojom.

Kapitola 2

Objektovo orientované Petriho siete

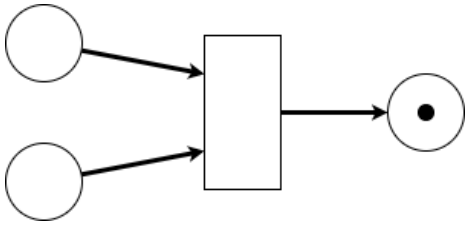
V tejto kapitole budú vysvetlené základné princípy Petriho sietí a niektoré z ich jednoduchých rozšírení. Na týchto základoch bude vysvetlená dynamika a štruktúra Objektovo orientovanej Petriho siete(skrátene OOPN). Následne bude popísaný jazyk PNtalk, ktorý je konkrétnou implementáciou OOPN so zameraním na jeho syntax a rozdiely oproti OOPN. Obsah tejto kapitoly bol prebraný z [6].

2.1 Petriho Siete

Petriho sieť je používaná na modelovanie diskrétnych distribuovaných systémov. Takýto systém môže obsahovať viacero procesov, kde každý proces má vlastný stav. Petriho sieť slúži na reprezentovanie stavu celého systému jednou sieťou. Okrem stavu Petriho sieť reprezentuje aj udalosti, ktoré v systéme môžu nastať a menia tak stav systému.

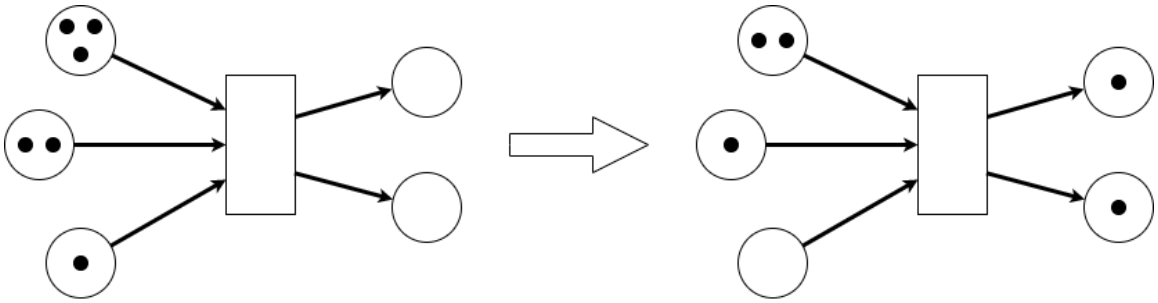
2.1.1 Základy

Petriho sieť má grafovú štruktúru. Skladá sa z dvoch typov uzlov a orientovaných hrán, ktoré uzly spájajú. Prvým typom uzlu je miesto. Miesto predstavuje stav a je reprezentované kruhom. Druhým typom je prechod reprezentovaný štvoruholníkom, ktorý spolu s naňho napojenými hranami reprezentuje udalosť systému. Hrana vždy spája práve jedno miesto s práve jedným prechodom. Takýchto spojení však môže mať aj prechod aj miesto viacero. To, akým smerom je hrana orientovaná určuje, či sa jedná o vstupnú alebo výstupnú podmienku. ak teda hrana je orientovaná z miesta do prechodu, tak sa jedná o vstupnú podmienku. ak z prechodu do miesta, tak sa jedná o podmienku výstupnú. Okamžitý stav systému je určený značkami v modeli reprezentovaných bodkami. Tieto značky sa vždy nachádzajú vo vnútri miest a v mieste sa môže nachádzať viacero značiek naraz.



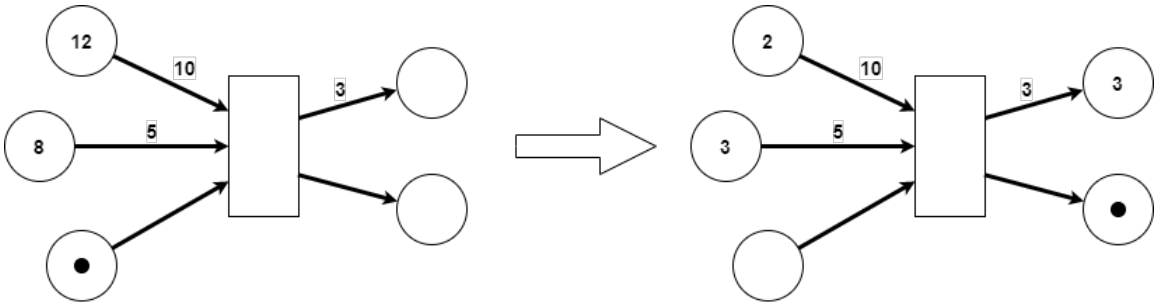
Obr. 2.1: Jednoduchá Petriho sieť

Udalosť, reprezentovaná prechodom, môže byť vykonaná práve vtedy, keď sú v jednom okamžiku splnené všetky vstupné podmienky. Vstupná podmienka prechodu je splnená vtedy, keď sa v mieste napojenom hranou orientovanou do prechodu nachádza značka. Vykonanie prechodu je operácia atomická a jej dôsledkom je, že sa z každého miesta vstupnej podmienky odstráni značka a do každého miesta výstupnej podmienky sa značka vloží.



Obr. 2.2: Vykonanie prechodu

Miesto môže byť s jedným prechodom spojené viacerými hranami. Prechod tak môže mať podmienku, ktorá odstráni viacero značiek z jedného miesta, alebo podmienku, ktorá vkladá viacero značiek na jedno miesto. Pre zjednodušenie grafu a väčšiu prehľadnosť sa taká situácia môže reprezentovať číslom u hrany značiac, koľko takých hrán sa tam nachádza. Rovnaký princíp môžeme aplikovať pre značky v miestach, kde namiesto množstva bodiek bude len číslo, reprezentujúce koľko značiek sa tam nachádza.

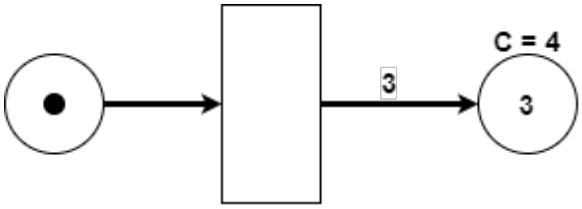


Obr. 2.3: Použitie čísel u hrán a v miestach

2.1.2 Rozšírenia Petriho sietí

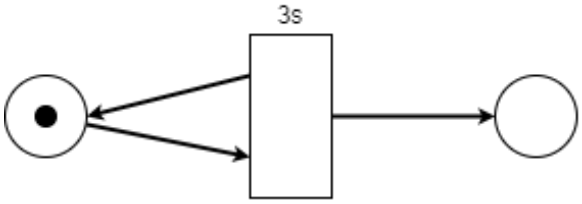
Takto zadefinovaná Petriho sieť sa dá rozšíriť o ďalšie možnosti. Ďalej budú uvedené také, ktoré majú vplyv na to, ktorý prechod môže byť vykonaný.

Prvým možným rozšírením je možnosť určiť kapacitu konkrétneho miesta. Urobíme to tak, že k miestu, ktorému má byť obmedzená kapacita pridáme informáciu o počte značiek, ktoré sa v ňom maximálne môže nachádzať. ak nie je kapacita miesta definovaná, tak sa predpokladá, že miesto je bez obmedzenia. Doteraz na vykonanie prechodu bolo potrebné mať požadovaný počet značiek v miestach, ktoré predstavujú vstupné podmienky a obsah miest výstupných podmienok nemal žiadny vplyv na to, či sa môže prechod vykonať. Toto rozšírenie má za následok, že aj výstupné podmienky môžu zamedziť vykonaniu prechodu, ak by vloženie značiek vykonaním prechodu prekročilo určenú kapacitu.



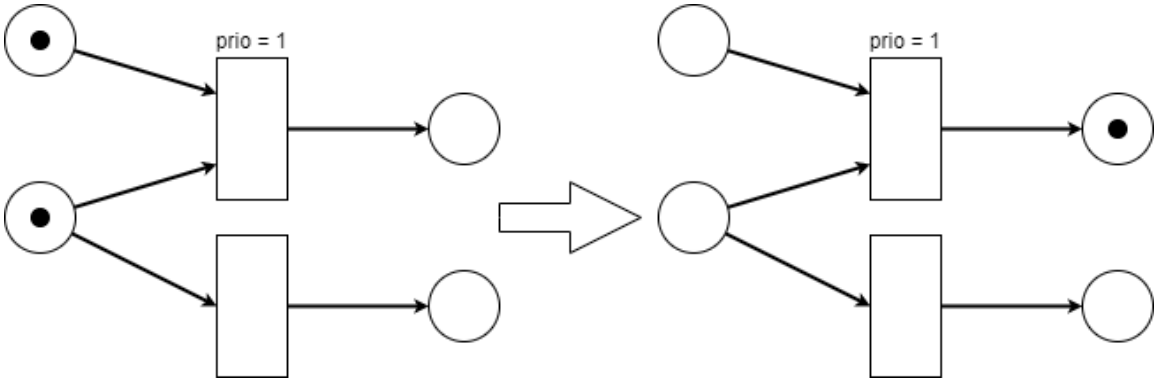
Obr. 2.4: Kapacita spôsobí, že sa prechod nevykoná

Možnosť modelovať dĺžku trvania činnosti je dôležitou požiadavkou pri vytváraní modelov množstva reálnych systémov. Časové obmedzenia je možné pridať miestu ale aj prechodu. Priradením časového obmedzenia miestu znamená, že značka, ktorá do tohto miesta prišla, musí v ňom zotrvať po zadanú dobu. až potom môže byť využitá pre vykonanie ďalšieho prechodu. Pri priradení časového obmedzenia prechodu sa určí po akú dobu musí byť možné prechod vykonať predtým, než sa naozaj vykoná. ak by niektorá podmienka počas tohto čakania prestala byť splniteľná, tak sa doterajšia dĺžka čakania vynuluje a znovu začne až budú všetky podmienky opäť splniteľné. Následkom tohto, ak napríklad jedna značka môže byť využitá v dvoch prechodoch a jeden je časovaný, tak sa vykoná ten, ktorý nie je časovo obmedzený.

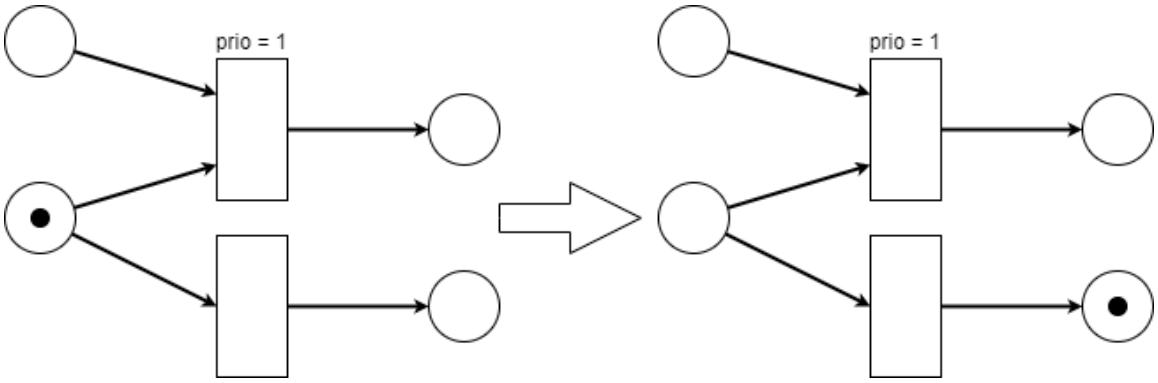


Obr. 2.5: Do systému sa pridá značka každé 3 sekundy

Priorita prechodu je ďalšou možnosťou ako rozšíriť modelovanie Petriho sieťami. Táto možnosť umožňuje rozhodnúť, ktorý prechod sa má vykonať na základe priority, ktorú pridelíme danému prechodu tak, že k nemu dáme informáciu v číselnej podobe. Platí, že čím vyššie číslo, tým je priorita väčšia a prechody bez priradenej priority majú prioritu najmenšiu možnú. Napríklad v prípade, že miesto so značkou, je vstupnou podmienkou dvoch prechodov s rozdielnou prioritou, tak sa prirodzene vykoná ten, čo ju má vyššiu. ale keďže prechod môže mať vstupné podmienky z viacerých miest, tak v prípade, že prechod s vyššou prioritou nemá aspoň jednu z takýchto podmienok splnenú a prechod s menšou prioritou má splnené všetky podmienky vykonania, tak sa vykoná ten.

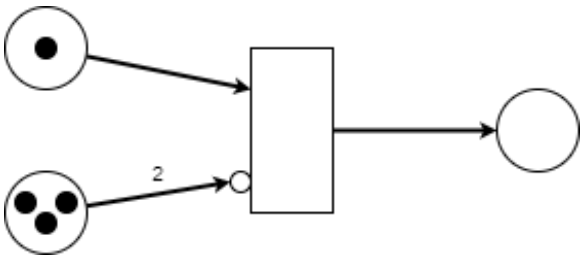


Obr. 2.6: Je vykonaný prechod s vyššou prioritou



Obr. 2.7: Je vykonaný prechod s nižšou prioritou

Inhibítory sú rozšírenie, ktoré pridáva nový spôsob ako podmieniť vykonanie prechodu. Je to typ orientovanej hrany smerujúcej z miesta do prechodu a označujeme ju tak, že pridáme na koniec koliesko a priradíme jej číslo. Toto číslo značí, že počet značiek v mieste tohto spojenia musí byť menší než dané číslo, aby sa prechod mohol vykonať. Okrem iného sa toto rozšírenie dá jednoducho použiť na testovanie počtu značiek.



Obr. 2.8: Inhibítor zabraňuje vykonaniu prechodu

2.2 OOPN

Pre Petriho Siete existuje viacero možných rozšírení, ktoré ich viac či menej rozširujú. Jedným z nich sú Objektovo orientované Petriho siete (skrátene OOPN). OOPN rozši-

rujú Petriho siete o objektovú orientáciu, čím umožňujú dedičnosť, zapuzdrenie, abstrakciu a ďalšie výhody objektovej orientácie, čím by sa mala zjednodušiť tvorba a analýza modelu.

2.2.1 Trieda

Trieda z pohľadu objektovo orientovaného programovania predstavuje šablónu pre vytváranie objektov. Táto šablóna obsahuje počiatočný stav objektu a jeho chovanie. v OOPN trieda obsahuje Petriho sieť spolu s jej počiatočným stavom, ktorým sú obsahy miest vo vnútri siete. Trieda taktiež obsahuje množinu správ, na ktoré je objekt schopný reagovať, a ako odpovede na tieto správy definuje metódy a synchrónne porty. Trieda samotná rozumie len jednému typu správy, ktorou je *new*, pre vytvorenie inštancie. Trieda môže byt odvodená od inej triedy. Hovoríme tak o dedičnosti. Trieda dedí sieť a metódy. Pripúšťa sa však len jednoduchá dedičnosť. Tzn. trieda dedí vždy len od jednej triedy. Môže tak vznikať hierarchia dedičnosti.

2.2.2 Objekt

Objekty v OOPN nahrádzajú značky. Máme dva typy objektov:

- Primitívne objekty
- Neprimitívne objekty

Primitívne objekty sú konštantné. Jedná sa o čísla, reťazce, symboly, konštanty a podobne. Takéto objekty nie je možné meniť. Neprimitívne objekty si držia stav a tento stav je aj možné meniť zasielaním správ danému objektu. Ten na ne reaguje definovaným chovaním vo forme metód. Definície týchto metód a počiatočný stav neprimitívneho objektu je určený triedou, ktorá ho vytvorila.

2.2.3 Metoda

Metoda z pohľadu objektovo orientovaného programovania predstavuje správu zaslanú objektu a chovanie metódy predstavuje odpoveď na tuto správu. v OOPN je chovanie metódy reprezentované Petriho sieťou a vzorom správy, na ktorú reaguje. Volaním metódy sa rozumie zaslanie správy objektu vo vzore volanej metódy. Pri zavolaní metódy sa vytvára inštancia Petriho siete. Sieť sa vykonáva dokiaľ sa nenaplní miesto return, ktoré predstavuje návratovú hodnotu. Po ukončení sa vytvorená inštancia odstráni zo systému a ako odpoveď na zaslanú správu vracia obsah miesta return.

2.2.4 Synchrónny port

Synchrónny port má vzor správy, na ktorú reaguje, takže je možné ho zavolať ako metódu. Narozdiel od metódy jeho chovanie nie je definované Petriho sieťou, ale len jedným prechodom v Petriho sieti objektu, ktorý na danú správu reaguje.

2.2.5 Petriho sieť

V OOPN sa samotná štruktúra Petriho siete viac menej nemení. Stále sa skladá z miest, prechodov a orientovaných hrán, ktoré predstavujú podmienky. ale od Petriho siete, tak ako bola zadefinovaná v predchádzajúcej kapitole, sa líši hlavne tým, že miesta neobsahujú len značky.

2.2.6 Miesto

Miesto v OOPN je tak ako v Petriho sieti úložiskom, do ktorého je možné vložiť značky. OOPN miesto jednoduchých značiek používa objekty. V mieste sa môže nachádzať viacero typov objektov, a tak ako to je u značiek, ten istý objekt sa tu môže nachádzať viackrát. Jedná sa teda o multimnožinu. Tzn. každému prvku je priradený počet výskytov daného prvku.

2.2.7 Prechod

Prechod je rozšírený o možnosť definovať akcie, ktoré sa majú vykonať spolu s prechodom. Týmito akciami, môžu byť aritmetické operácie, volanie metód a vytváranie nových objektov. Vzhľadom na to, prechody už nie sú vždy atomické. ak je volaná metóda, tak prechod musí čakať na odpoveď. v takom prípade, pri splnení všetkých podmienok, sa odoberú objekty zo vstupných miest a až po ukončení vykonávania všetkých akcií sa vložia objekty do výstupných miest. Odobraným objektom v rámci akcií je možné zasielať správy a špecifikovať objekty, ktoré sa vložia do výstupných miest.

2.2.8 Podmienky prechodu

Pre vykonanie prechodu musia byť splniteľné všetky vstupné podmienky. v OOPN vstupná podmienka môže požadovať nielen počet objektov vo vstupnom mieste, ale môže špecifikovať aj objekt, ktorý požaduje. Obdobne fungujú aj výstupné podmienky. v OOPN pre vykonanie prechodu vyžaduje, aby boli splnené stráže prechodu ak sú pre daný prechod špecifikované.

2.2.9 Stráže

Vykonanie prechodu je možné podmieniť novým typom podmienky. Týmto typom je stráž. Jedná sa o jeden alebo viacero výrazov, ktoré musia byť vyhodnotené ako pravdivé, aby bola táto podmienka splnená. Súčasťou týchto výrazov môžu byť volania synchrónnych portou. Tie sú vyhodnotené ako pravdivé práve vtedy, keď sú splnené všetky ich vstupné podmienky.

2.3 PNtalk

Jazyk PNtalk implementuje OOPN. Jeho meno je odvodené z *"Petri Nets & Small Talk"*. Jedná sa o programovací jazyk založený na objektovo orientovanom jazyku Small Talk, v ktorom je možné vytvárať modely OOPN. PNtalk konkretizuje časti OOPN, ako sú napríklad typy primitívnych objektov, ktoré neboli definované v OOPN. a taktiež prináša rozšírenia.

2.3.1 Termy

Reprezentujú objekty PNtalku. Jedná sa o najjednoduchšie výrazy.

1. **Literály**.

   Prvým typom termov sú literály. Sú reprezentáciou primitívnych objektov. Medzi literály patria čísla, znaky, reťazce, symboly, boolovské konštanty, nil.

- **Čísla**. Objekty reprezentujúce číselné hodnoty sa nazývajú čísla. Jedná sa o čísla celé, desatinné, kladné i záporné. Na oddelenie celej a desatinnej časti sa používa bodka. PNtalk umožňuje použiť taktiež notácie s exponentom. Čísla sú schopné reagovať na správy matematických operácii. Príkladom čísel v PNtalku môžu byť 8, 1.3, 152, -6, 1.85e11, 1.85e-11.
- **Znaky**. Objekty reprezentujúce jednotlivé písmená, cifry, symboly. Zápis znaku sa začína symbolom dolára a nasleduje akákoľvek graficky reprezentovatelná hodnota. Napríklad $3, $a, $b, $$, $#.
- **Reťazce**. Objekty reprezentujúce sekvenciu znakov. Sekvencia znakov je ohraničená apostrofmi. Pre vloženie apostrofu do sekvencie je potrebné ho napísať dvakrát za sebou. Reťazec reaguje na správu porovnávania s iným reťazcom a pre prístup k znakom reťazca. Príkladmi zápisu reťazcov môžu byť 'hello', 'I"m here!', '418 and half'.
- **Symboly**. Objekty používané pre jednoznačnú identifikáciu. Reagujú na správy pre porovnanie s ďalším symbolom. Ich zápis sa začína s # a nasleduje sekvencia znakov ohraničená apostrofmi (reťazec). ak sekvencia znakov neobsahuje prázdne miesta a interpunkciu, tak ohraničenie apostrofmi nie je potreba. Napríklad #symbol, #'other symbol', #53.
- **Boolovské konštanty**. Reprezentujú boolovské hodnoty a sú reprezentované pre nich vyhradenými identifikátormi *true* na *false*
- **Nedefinovaný objekt**. Jedná sa o hodnotu, nastavovanú pre ne-inicializované premenné. Môže reprezentovať aj stavy nič alebo nikde. Hodnota je reprezentovaná vyhradeným identifikátorom *nil*

2. **Premenné**.

   Premenná je identifikátor, ktorý reprezentuje objekt. To, aký objekt reprezentuje sa počas behu programu môže meniť. Identifikátor je sekvencia znakov bez medzier a interpunkcie, kde prvý znak je malé písmeno. Napríklad *var*, *x*, *x1*, *first_variable*.

3. **Pseudo-premenné**.

   v jazyku PNtalk sú to *self* a *super*. Objekty, ktoré reprezentujú, sú určené na základe kontextu a programovo ich nejde meniť.

4. **Mená tried**.

   Mená tried sú identifikátory, ktoré reprezentujú triedu. Zapisujú sa ako premenné, s tým rozdielom, že sa začínajú veľkým písmenom. Nie je ich možno meniť. Jedná sa o konštanty. Napríklad *Class*, *X*, *X1*, *First_class*.

2.3.2 Zasielanie správ

Stráže a akcie prechodu sú výrazy v PNtalku realizované zasielaním správ. Zaslanie správy má syntax: **<adresát> <správa>**. adresátovi je správa zaslaná a môžu ním byť:

- Primitívne objekty (len vstavané správy)
- Neprimitívne objekty (metódy, synchrónne prechody)
- Triedy (len správa *new*)

Správa sa skladá zo selektoru a argumentov.

- **Unárne správy**. Majú selektor reprezentovaný identifikátorom a neobsahujú žiadne argumenty. Príkladmi unárnych sprav sú *round*, *new*. Celé zaslanie správy tak môže vyzerať napríklad takto: *Class new*, *7.63 round*.

- **Binárne správy**. Majú práve jeden argument a selektor reprezentovaný jedným z týchto znakov:

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

  Príkladom binárnej správy môže byť „- 3“ a celé zaslanie správy môže vyzerať napríklad takto: „5 - 3“. adresátom je „5“, selektorom správy „-“ a „3“ je argument správy.

- **Správa s kľúčovými slovami**. Skladá sa z jedného alebo viac <kľúč>:<argument> dvojíc. Selektorom je spojenie kľúčov. Príkladom správy môže byť „insert:b times:3“. Selektorom takejto správy je „insert:times:“ a argumenty sú „b“, „3“. Celé zaslanie správy by vyzeralo napríklad takto „list insert:b times 3“.

V uvedených príkladoch zaslania správy boli adresátmi i argumentmi termy. Jedná sa o jednoduché zaslanie správy. ak je adresátom alebo argumentom ďalšie zaslanie správy, ide o zaslanie správy zložené. Zložené zasielanie správy sa implicitne vyhodnocuje v poradí zľava doprava a to u všetkých typov správ. Napríklad zložené zaslanie správy „3 + 2 * 4“ najprv zašle správu „+ 2“ číslu 3 a až potom zašle správu „* 4“ výsledku zaslania správy „3 + 2“. Výsledkom vyhodnotenia tohto zaslania správy je číslo 20. Pre uprednostnenie vyhodnotenia zaslania správy je nutné takéto zaslanie správy dať do zátvoriek. Napríklad „3 + (2 * 4)“ sa vyhodnotí najprv zaslanie správy „2 * 4“ a výsledok sa zašle ako argument v správe so selektorom „+“ číslu 3. Výsledkom vyhodnotenia tohto zaslania správy je číslo 11.

2.3.3 akcie a stráže

Stráž prechodu sa skladá z postupnosti výrazov. Výrazom je jednoduché či zložité zaslanie správy a jednotlivé výrazy sú oddelené bodkou. aby sa stráž vyhodnotila ako splnená a prechod sa mohol vykonať, musia sa všetky výrazy stráže vyhodnotiť ako *true*. Bodka oddeľujúca jednotlivé výrazy predstavuje logický operátor aND. teda o selektor správy &.

akcie prechodu sa tak ako u stráže skladá z postupnosti výrazov oddelených bodkou. ale naviac umožňuje použiť priradenie. Výsledok zaslania správy sa uloží do premennej. Operátor priradenia je reprezentovaný ako „:=“. Syntax výrazu priradenia má tvar:

<premenná> := <zaslanie správy>

Mená premenných majú rozsah platnosti obmedzený na stráž, akcie prechodu a výrazy na okolitých hranách.

2.3.4 Hranové výrazy

Hranové výrazy sa nachádzajú na hranách Petriho Sietí. Sú reprezentované multimnožinami a zapisujú sa v tvare

$$n_1`c_1, n_2`c_2, ..., n_i`c_i$$

n_i môže byť term a c_i môže byť term alebo zoznam. n_i reprezentuje počet a keď sa $n_i = 1$, tak sa môže vynechať. **Zoznam** sa zapisuje ako

$$(c_1, c_2, ..., c_i)$$

c_i je term alebo ďalší zoznam. Hranové výrazy sa nachádzajú aj vo vnútri miest, kde určujú počiatočný stav miesta.

2.3.5 Miesta a prechody

V PNtalku je sieť možné definovať v grafickej i textovej forme. Pre túto prácu je podstatná tá textová, kvôli možnosti strojového spracovania.

Miesto je v rámci siete jednoznačne identifikované identifikátorom a môže niesť počiatočné značenie. Spôsob zápisu miesta je následovný:

"placeïd "("[initmarking] ")"

Zápis sa začína kľúčovým slovom „place“, nasleduje identifikátora počiatočné značenie, ktoré má rovnakú syntax ako hranový výraz a je ohraničený zátvorkami. Hranaté zátvorky značia, že počiatočné značenie nemusí byť definované.

Prechod je v rámci siete jednoznačne identifikované identifikátorom a obsahuje definície hrán, stráže a akcií. Zapisuje sa následovne:

"transïd [cond] [precond] [guard] [action] [postcond]

Zápis sa začína kľúčovým slovom trans, nasleduje identifikátor a vlastnosti prechodu. Vlastnosti prechodu a to podmienky, stráže, akcie sú nepovinne.

Podmienky predstavujú hrany. Zápis podmienky sa skladá z identifikátora miesta, s ktorým je prechod spojený a hranového výrazu. Podla orientácie hrany sa podmienky rozdeľujú na tri typy. Syntax sa skladá z kľúčového slova určujúci orientáciu hrany a zoznamu podmienok.

- Vstupné podmienky. Predstavujú hrany smerujúce z miest do prechodu a určujú objekty, ktoré sa musia odobrať z miesta vykonaním prechodu. Zoznam vstupných podmienok sa začína kľúčovým slovom „precond“ a nasledujú podmienky oddelené čiarkou.

  "precondïd "("multiset ")"[",ïd"("multiset ")"]*

- Výstupné podmienky. Predstavujú hrany smerujúce z prechodu do miest a určujú objekty, ktoré sa musia vložiť do miesta vykonaním prechodu. Zoznam výstupných podmienok sa začína kľúčovým slovom „postcond“ a nasledujú podmienky oddelené čiarkou.

  "postcondïd "("multiset ")"[",ïd"("multiset ")"]*

- Testovacie podmienky. Predstavujú obojsmerne smerované hrany. Tzn. odstránia i vložia objekt do miest. Zoznam testovacích podmienok sa začína kľúčovým slovom „cond“ a nasledujú podmienky oddelené čiarkou.

  "condïd "("multiset ")"[",ïd"("multiset ")"]*

Stráž predstavuje špeciálnu podmienku vykonania prechodu a pre vykonanie výrazu sa musí vyhodnotiť ako splnená. Syntax sa začína kľúčovým slovom „guard“ a výrazy sú ohraničené zloženými zátvorkami.

"guard{ëxpr "}"

akcie sú operácie, ktoré sa majú vykonať vykonaním prechodu. Môžu použiť objekty získané zo vstupných podmienok a určiť, čo sa má vložiť výstupnou podmienkou. Ide tak urobiť použitím premenných definovaných vo vstupných a výstupných podmienkach. Syntax sa začína kľúčovým slovom „action“ a nasledujú akcie ohraničené zloženými zátvorkami.

äction{ëxpr "}"

2.3.6 Siete

Sieť je tvorená miestami, prechodmi a hranami, ktoré ich spájajú. v textovom zápise má sieť takúto syntax:

[place|transition]*

V PNtalku máme dva druhy sietí

- **Sieť objektu**. Špecifikuje sa ako súčasť triedy. Reprezentuje stav objektu, tzn. jeho atribúty v podobe miesta aa jeho vlastnú vnútornú aktivitu v podobe prechodov.
- **Sieť metódy**. Reprezentuje reakciu objektu na zaslanie správy. Je taktiež špecifikovaná ako súčasť triedy a skladá sa z miest, prechodov, a hrán. Trieda môže obsahovať viacero sietí metód. Musí obsahovať výstupné miesto *return* a miesta parametrov, ktorých mená odpovedajú formálnym parametrom zaslania správy. Sieť môže mať aj nulový počet miest parametrov. Zaslaním správy objektu sa vytvorí nová inštancia siete metódy a do miest parametrov sa vložia vstupné objekty. Vložením objektu do miesta *return*“ sa ukončuje vykonávanie metódy. ako odpoveď na zaslanie správy vracia obsah tohto miesta a sieť zaniká. Prechody siete metódy môžu byť napojené hranami na miesta objektu, a tak k objektu pristupovať s možnosťou čítať alebo meniť jeho stav. Zápis siete metódy v textovom tvare sa začína kľúčovým slovom „method“, nasleduje vzor správy a za ňou sieť metódy.

"method"message net

2.3.7 Konštruktor

Zaslaním správy *new* triede sa vytvorí nová inštancia danej triedy. Vytvorí sa sieť objektu, vložia sa počiatočné objekty do miest a ako odpoveď na túto správu sa vracia meno vytvoreného objektu. Správe *new* rozumejú všetky triedy a je možné ju zaslať akejkoľvek triede v systéme.

Konštruktor je špeciálny typ metódy, ktorý vždy vracia inštanciu objektu. Má teda vzor správy, ktorému rozumie, cez ktorý je možné jej predávať argumenty a tým parametrizovať tvorbu objektu. Vzoru správy konštruktoru rozumie i trieda aj objekt. Trieda, ako reakciu na takúto správu, vytvorí inštanciu zaslaním správy *new* a potom zašle prijatú správu novovzniknutému objektu. Objekt na takúto správu reaguje štandardne. v textovej podobe

sa konštruktor zapisuje rovnako ako metóda ale miesto kľúčového slova „method“ sa použije kľúčové slovo „constructor“.

"constructor"message net

2.3.8 Synchrónne porty

Súčasťou špecifikácie triedy sú aj synchrónne porty, ktoré tak ako metódy reagujú na zaslanie správy, ale neobsahujú miesta a prechody. Jedná sa o jeden prechod so vzorom správy, ktorý je možné volať zo stráže akéhokoľvek prechodu a testovať alebo meniť stav adresáta správy. Má podmienky prechodu, a tak sa dá testovať jeho splniteľnosť. Môže obsahovať taktiež stráž a volať ďalšie synchrónne porty. ako odpoveď na správu poslanú synchrónnemu prechodu sa vracia informácia boolovského typu o jeho splniteľnosti. Textový zápis je podobný zápisu prechodu, ale začína sa kľúčovým slovom „sync“, na mieste mena prechodu sa nachádza vzor správy a neobsahuje akcie.

"sync"message [cond] [precond] [guard] [postcond]

2.3.9 Triedy

Model v PNtalku sa skladá zo špecifikácii tried. Špecifikácia triedy obsahuje meno a sieť. Ďalej môže obsahovať siete metód, konštruktorov a synchrónne porty. Trieda je šablóna, z ktorej sa dajú vytvárať objekty a je schopná reagovať na správu *new* pre vytvorenie objektu. Každý takto vytvorený objekt má svoju vlastnú sieť a je schopný reagovať na vzory správ, ktoré sú špecifikované metódami, konštruktormi a synchrónnymi portami.

V každom modeli je práve jedna počiatočná trieda, ktorej inštancia je prítomná na začiatku simulácie. Určenie, ktorá trieda ňou je, sa zapisuje kľúčovým slovom „main“ a menom danej triedy.

"mainïd

Ďedičnosť. v PNtalku má každá trieda svojho predchodcu. Tvorí sa tak hierarchia tried a na vrchole je abstraktná trieda *PN*, ktorá priamo dedí od triedy *Object*. *Object* je taktiež abstraktná trieda a je vrcholom hierarchie dedičnosti. Okrem triedy *PN* od nej dedia aj všetky primitívne objekty. *Object* definuje reakcie na správy, na ktoré musia byť schopné reagovať všetky objekty. Ide o správy pre porovnanie identity. (== a ~==).

Trieda dedí vlastnosti a správanie svojho predchodcu a všetkých jeho predchodcov. v PNtalku ide o štruktúru siete a reakcie na správy. Trieda môže na zdedenú sieť napojiť ďalšie prvky alebo predefinovať vlastnosti svojho predchodcu, s výnimkou vlastností triedy *Object*. Dá sa to urobiť tak, že špecifikuje časti siete, ktoré sú pod rovnakým menom špecifikované aj u predchodcu. Rovnaký postup je možné použiť aj na pre-definovanie reakcie na zaslanie správy.

Textový zápis triedy sa začína kľúčovým slovom „class“, nasleduje meno siete, kľúčové slovo „is_a“, meno predchodcu, sieť a reakcie na správy. Jedná sa o syntax:

"classïd ïs_aïd [net] [method|constructor|sync]*

Kapitola 3

Návrh interpreta

V predchádzajúcej kapitole bola popísaná štruktúra a dynamika Objektovo orientovaných Petriho sietí a možnosť textového zápisu pomocou jazyka PNtalk. Dajú sa tak zapísať vlastnosti a počiatočný stav modelovaného systému. Pre simulovanie, tzn. experimentovanie s takto zadefinovaným modelom je potrebné byť schopný vykonávať dynamiku modelu a mať možnosť pozorovať výsledky. Je potreba vytvoriť nástroj, ktorý by bol toho schopný.

3.1 Požiadavky

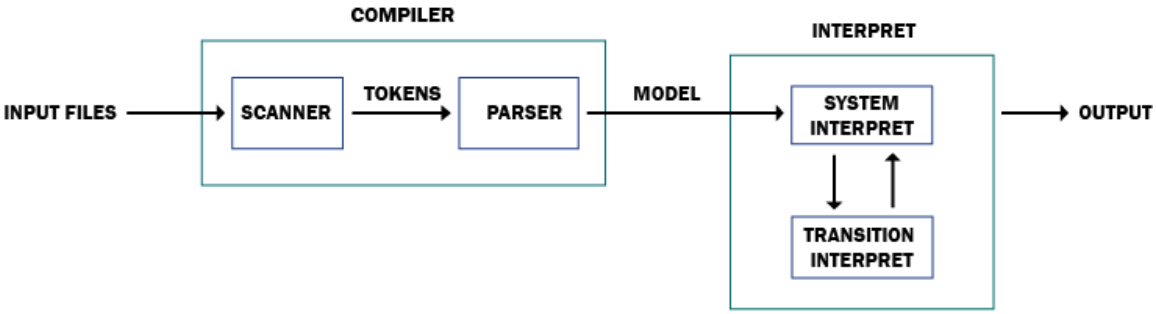
- **Interpretovať** model. Riešenie musí byť schopné interpretovať Objektovo orientovanú Petriho sieť. Vykonávať dynamiku siete, tak ako je zadefinovaná.
- **Načítať** model. Riešenie musí umožniť vykonávanie veľkého množstva rôznych modelov zapísaných v textovej podobe. Preto musí byť schopné spracovať súbor obsahujúci definíciu modelu v textovej podobe a načítať model pre interpretáciu. Definícia modelu môže byť rozdelená do viacerých súborov, preto musí byť táto možnosť podporovaná.
- Model definovaný v jazyku **PNtalk**. Riešenie musí podporovať načítanie a interpretáciu Objektovo orientovanej Petriho siete v textovej podobe jazyka PNtalk. Syntax jednotlivých častí PNtalku a ich požadované vlastnosti boli popísané v podkapitole 2.3. Syntax akejkoľvek konštrukcie nezahrnutej v tomto popise musí taktiež syntakticky zapadať do jazyka.
- **Pozorovať** interpretáciu modelu. Musí byť umožnené sledovať priebeh interpretácie, tak pozorovať vlastnosti modelu a prípadných výsledkov. Pre túto potrebu riešenie musí umožniť v akciách prechodu zaslanie správy pre výpis na štandardný výstup programu.
- **Čas** je nezanedbateľnou súčasťou veľkého množstva systémov. Riešenie musí umožniť modelovať beh času a interpretácia musí byť schopná bežať v reálnom i v modelovom čase. Modelovanie časovej udalosti preto bude možné zaslaním správy vo vnútri akcií prechodu. Reakciou na túto správu bude pozastavenie vykonávania prechodu na určený časový okamžik. Vykonávanie zvyšku siete musí pokračovať a musí byť zabezpečený návrat k vykonávaniu prechodu v chronologickom poradí.
- **Náhoda**. Pri tvorbe modelu systému je častokrát potrebné modelovať udalosti, ktoré sa dejú náhodne. Riešenie musí poskytovať nástroje pre jej simuláciu. Pre modelovanie

náhody riešenie musí umožniť zaslanie správy v akciách prechodu pre vygenerovanie náhodného čísla.

3.2 Návrh riešenia

Navrhovaným riešením takého nástroju by bol program, ktorého vstupom sú súbory obsahujúce model systému zapísaný jazykom PNtalk. Program simuluje dynamiku modelovaného systému a výstupom sú pozorované vlastnosti. z hľadiska architektúry tohto riešenia ho môžme rozdeliť na tri časti.

- **Prekladač** pre načítanie súborov do vnútornej reprezentácie.
- **Model** pre uloženie načítaného modelu vo vnútornej reprezentácii.
- **Interpret** pre interpretáciu modelu využitím vnútornej reprezentácie.



Obr. 3.1: architektúra riešenia

3.3 Model

Časť programu, ktorá vnútorne reprezentuje vstupný model a poskytuje prístup k jeho prvkom a ich vlastnostiam. Jednotlivé prvky sú na seba naviazané tak, aby bolo možné ich efektívne využívať. Model počas interpretácie svoj stav nemení. Jedná sa len o šablónu pre vytváranie inštancií jeho prvkov a programového vybavenia, ktoré tak vykonávajú.

3.4 Prekladač

Pred tým, než bude možné interpretovať model, je potrebné ho načítať do vnútornej reprezentácie. Túto úlohu plní prekladač. Načíta model v textovej forme jazyka PNtalk do vnútornej reprezentácie. Robí tak **lexikálny analyzátor**, ktorý číta zdrojové texty a prekladá ich na elementy jazyka PNtalk. Napr. čísla, identifikátory a podobne. Tieto elementy sú posielané priamo do druhej časti prekladača, ktorým je **syntaktický analyzátor**. Ten kontroluje, či elementy reprezentujú štruktúry PNtalku a ak áno, tak vytvorí model systému, ktorý je výstupom prekladača. [7]

3.5 Interpret

Hlavná časť programu. Volá model pre vytvorenie inštancií prvkov OOPN a vykonáva ich činnosť. Skladá sa z dvoch častí.

1. **Interpret systému** spravuje inštancie prvkov siete, čím drží stav systému. Hľadá prechody, ktoré by sa mohli vykonať a volá interpret prechodu pre ich vykonanie. Jeho vstupom je model systému, ktorý využíva pre vytváranie nových inštancií a ich vlastností, ktorými sú napríklad požiadavky pre vykonanie prechodu.

2. **Interpret prechodu** interpretuje správanie vykonania prechodu a tak mení stav systému. Taktiež môže iniciovať vytvorenie novej siete z predpisu triedy alebo metódy. Prechod nie je vždy atomický, preto umožňuje uložiť stav interpretácie prechodu a až to bude možné, v tom pokračovať.

3.6 Nástroje

Implementácia sa realizuje v jazyku C++ pre jeho objektovú orientovanosť a vysoký výkon vďaka prekladu do strojového jazyka.

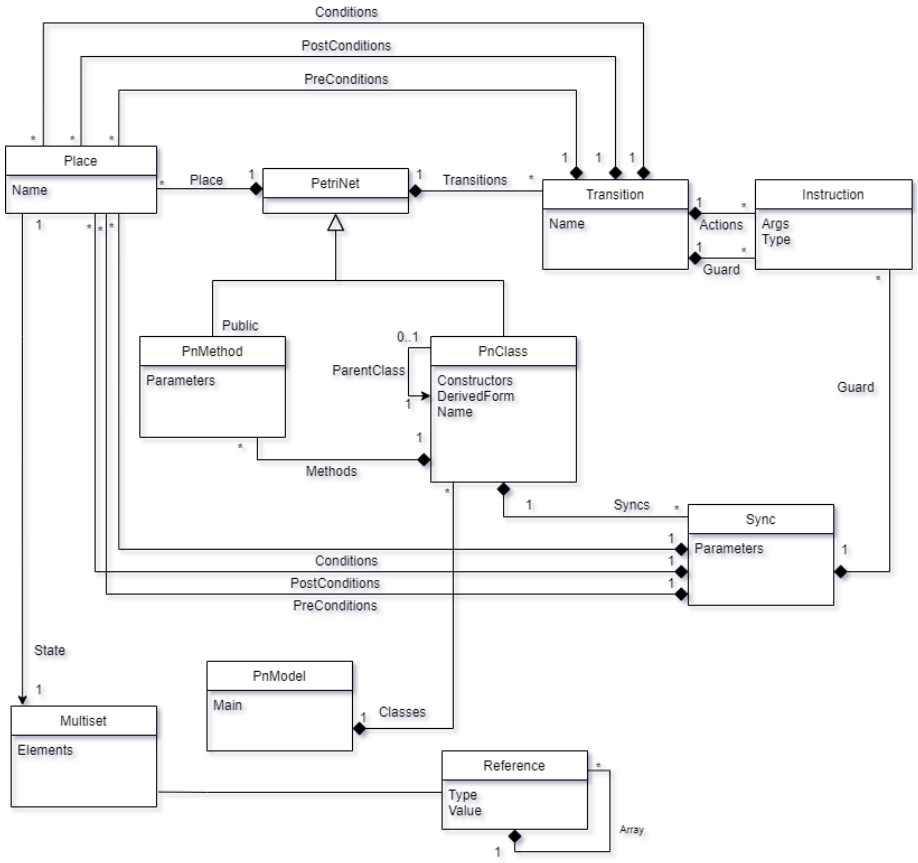
Kapitola 4

Vnútorná reprezentácia modelu

V tejto kapitole je popísaná vnútorná reprezentácia modelu. Spôsob uloženia jednotlivých častí, ako vnútorne fungujú. Spôsob načítania zdrojových súborov použitím prekladača.

4.1 Model

Vnútorná reprezentácia modelu získaného zo zdrojových textov. Je implementovaný triedou *PnModel*.



Obr. 4.1: Triedy vnútornej reprezentácie a ich vzťahy

Časti modelu sú implementované ďalšími triedami, ktoré je možné vidieť na obrázku 4.1. Trieda model ukladá zoznam definovaných tried a informáciu o tom, ktorá trieda je počiatočná. Uložené triedy ukladajú ich sieť, synchrónne porty a metódy, ktoré ukladajú zase ich obsah, atď. Inštancia triedy *PnModel* tým zahrňuje všetky časti načítaného modelu.

Vzťahy tried a informácie, ktoré uchovávajú je možné vidieť na obrázku 4.1. Trieda prvku siete na obrázku ukazuje aké ďalšie prvky uchováva. Prípadne je vyznačené, ak obsahuje viacero prvkov toho prvku.

4.1.1 Referencia

Pre ukladanie termov a polí je implementovaná trieda *Reference*. Trieda si uchováva tri hodnoty.

1. **Typ** hodnoty. (nil, identifikátor, bool, celé číslo, reálne číslo, reťazec, znak, symbol, pole)
2. **Hodnota** uložená v textovej podobe. Napr. „15“, „príklad“, „true“
3. **Pole** referencií. ak typ referencie je pole. Inak je prázdne.

Trieda implementuje reakcie na unárne a binárne správy.

4.1.2 Miesto a multimnožina

Multimnožina reprezentuje obsah miesta. Implementovaná triedou *Multiset*, ukladá prvky v slovníku, tzn. zoznam dvojíc kľúč, hodnota, kde kľúč je Referencia, identifikujúca element a hodnota je Referencia určujúca počet výskytov kľúča. Trieda poskytuje niekoľko metód pre prácu s jej obsahom.

- *remove* metóda dostáva ako argument *Multiset*, ktorý je jeho podmnožinou a odstráni ich prienik.
- *insert* metóda pre vloženie obsahu iného *Multisetu*. ak sa prvok v mieste nenachádza, je vložený s daným počtom výskytov, inak je iba zvýšený počet výskytov.

Miesto implementované triedou *Place* zapuzdruje multimnožinu a má meno pre jej identifikáciu. Poskytuje rovnaké metódy ako multimnožina.

4.1.3 Inštrukcia

akcie a stráže prechodu sú v prekladači prevedené do elementárnych inštrukcií. Prekladač vytvára inštancie triedy *Instruction*. Inštancia obsahuje informáciu o akú operáciu sa jedná a argumenty uložené ako zoznam referencií. Prvý argument je vždy identifikátor premennej, do ktorého sa uloží výsledok operácie. Príkladom môže byť inštrukcia typu *add*. Do premennej s menom uloženým v prvom argumente vloží výsledok sčítania druhého a tretieho argumentu.

4.1.4 Prechod

Prechod je v modeli ukladaný ako inštancia triedy *Transition*. Obsahuje meno pre identifikáciu. Vstupné, výstupné a testovacie podmienky sú ukladané ako rady miest, pretože

miesto má rovnaký spôsob uloženia ako podmienka. Tzn. pomenovaná multimnožina. akcie a stráž prechodu sú uložené ako rady inštrukcií.

Synchrónny port je v uložený ako prechod, ale neobsahuje meno. Na identifikáciu používa predpis správy, na ktorú reaguje. Okrem toho synchrónny port nemôže obsahovať akcie.

4.1.5 Trieda a metóda

Užívateľom definované triedy a metódy sú ukladané ako inštancie tried *PnClass* a *PnMethod*. Obe dedia od triedy *PetriNet*. Preberajú tak jej vlastnosť ukladať sieť ako radu prechodov a radu miest.

PnClass k tomu ukladá meno triedy, odkaz na triedu od ktorej dedí, množiny metód, konštruktorov a synchrónnych prechodov.

PnMethod okrem siete ukladá predpis správy, na ktorú reaguje. Predpis správy je uložený ako rada párov dvoch reťazcov. Kde prvý reťazec je selektor a druhý argument.

4.2 Prekladač

ako už bolo spomenuté v návrhu, jedná sa o časť riešenia, ktorá načíta zdrojový text do vnútornej reprezentácie. Skladá sa z dvoch častí.

4.2.1 Lexikálny analyzátor

Lexikálny analyzátor je modul, ktorý priamo číta zdrojové texty, prevádza nad ním lexikálnu analýzu a vystupujú z neho elementy jazyka tzv. tokeny. Napr. mená tried, mená premenných, čísla, reťazce, operátory, zátvorky a ďalšie.

V riešení je implementovaný triedou *Scanner*. Inštancia tejto triedy si drží aktuálny token a umožňuje prístup k jeho vlastnostiam pomocou *„get“* metód. analýzu vykonáva konečným automatom, ktorý prečíta časť zdrojového textu a ak nebola nájdená chyba, nastaví hodnotu aktuálneho tokenu na ďalší následujúci. v opačnom prípade vyvolá výnimku. Toto chovanie sa vyvolá metódou *next_token()*. Pre možnosť určenia súboru a prípadnú zmenu, z ktorého sa číta slúži metóda *open(path)*. Token obsahuje dve informácie:

1. **Typ**. Určuje o aký element jazyka sa jedná. Napr. meno premennej, reťazec, číslo.
2. **Hodnota** (textová reprezentácia). Obsahuje načítaný text, ktorý v závislosti od typu je potrebný na následovné spracovanie. Príklady hodnôt možu byť: „var“, „Hello World", „153".

4.2.2 Syntaktický analyzátor

Syntaktický analyzátor je modul, ktorý využíva Lexikálny analyzátor pre získanie tokenov, vykonáva syntaktickú analýzu, identifikuje štruktúry modelu a ukladá ich do vnútornej reprezentácie.

Implementovaný triedou *Parser* poskytujúci metódu *„load_model()“*, ktorá na vstup zoberie zoznam súborov so zdrojovým textom a vracia model.

Kapitola 5

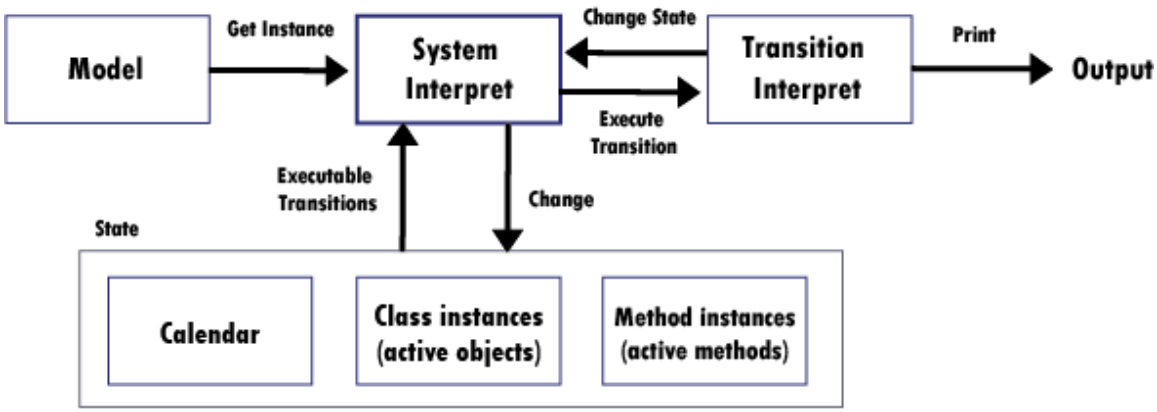
Interpretácia

Popis implementácie jazyka PNtalk, prípadných rozšírení a rozdielov. ako prebieha interpretácia jednotlivých prvkov a vnútorné fungovanie interpreta s využitím modelu popísaného v predchádzajúcej kapitole.

5.1 Interpret

Interpret je časť riešenia, ktorá interpretuje model, tak ako je načítaný vo vnútornej reprezentácii. Model systému je privedený na vstup a je používaný ako šablóna pre vytváranie inštancií.

- **Interpret systému** je jadrom interpreta a riadi beh interpretácie. Spravuje stav interpretácie, vyberá prechody pre vykonanie a komunikuje s ostatnými časťami.
- **Stav** je časť, ktorá ukladá stav inštancií tried a metód. Súčasťou je aj kalendár, skladajúci časovo závislé udalosti a stav prechodu, ktorý má pokračovať vo svojom vykonávaní v dobe udalosti.
- **Interpret prechodov** na podnety interpreta systému vytvára inštancie prechodov a vykonáva ich dynamiku. Mení tým stav interpretácie. Zasiela späť správy pre vytvorenie inštancií alebo časovej udalosti. Jeho funkciou je aj vypisovať výstup, keďže táto akcia je vyvolaná z akcií prechodu.



Obr. 5.1: Komunikácia častí interpreta

5.2 Vytváranie inštancií

Interpret berie na vstup model vo vnútornej reprezentácii. Obsahuje triedy, ich metódy a označenie, ktorá trieda je počiatočná. Umožňuje vytváranie inštancií tried a metód zaslaním správy z akcií prechodu.

1. **Trieda**. Pre vytvorenie inštancie tried, tzn. vytvorenie objektu, je potrebné iba meno triedy. Takto vytvorený objekt obsahuje kópiu miest triedy, spolu s ich obsahom, ktorým je počiatočný stav a odkaz na triedu, z ktorej bol vytvorený, pre prístup k prechodom siete. Inštancia je uložená do slovníku objektov, kde prístupovým kľúčom je vygenerované unikátne identifikačné číslo. Vytvorenie inštancie triedy je reakcia na správu *new*. Prechodu je ako odpoveď na túto správu zaslané identifikačné číslo objektu.

2. **Metóda**. Vytvorenie inštancie metódy je reakcia na zaslanie správy. Zaslanie správy sa skladá z adresáta, ktorým je identifikačné číslo objektu a správy, ktorou je zoznam argumentov. Pre nájdenie príslušnej metódy, ktorá je odpoveďou, sa najprv podľa identifikačného čísla vyberie objekt zo slovníka objektov a následovne sa použije jeho odkaz na triedu pre vybranie metódy, ktorá zodpovedá správe. Táto metóda rovnako ako pri vytváraní inštancie triedy vytvorí kópiu miest metódy a obsahuje odkaz na metódu s prechodmi. Inštancia metódy sa uloží do slovníku metód s vygenerovaným unikátnym identifikačným číslom. k tomu ale do vstupných miest vloží ešte vstupné argumenty obsiahnuté v správe. Keďže vykonanie metódy z pohľadu interpretácie nie je operácia atomická. Tak aj prechod, z ktorého je správa zaslaná nemôže byť atomický. Preto metóda obsahuje aj odkaz na tento prechod aby sa ihneď po jej ukončení mohlo pokračovať v jej vykonávaní a metóda vedela kam zaslať návratovú hodnotu.

5.3 Interpretácia systému

Dynamika modelovaného systému je reprezentovaná vykonávaním prechodov. Prechod sa vykoná vtedy, keď sú splniteľné všetky jeho podmienky a stráž.

Inicializácia. Interpret vytvorí inštanciu triedy, ktorá je v modely označená ako počiatočná(main).

Beh. Cyklicky prechádza všetky inštancie objektov a inštancie ich metód. Testuje ich prechody, či majú splnené podmienky. ak sú podmienky splnené tak sa vytvorí inštancia prechodu, ktorá vykoná definované chovanie. Odstráni objekty zo vstupných miest (precond) ale aj z testovacích miest (cond). Začne sa vykonanie prechodu a keď skončí, tak sa vložia objekty do výstupných miest (postcond) a späť do testovacích miest.

Ukončenie. Interpretácia sa ukončuje vtedy, keď sa v posledom cykly testovania prechodov všetkých inštancií nevykonal ani jeden a kalendár udalostí je prázdny.

5.3.1 Interpretácia prechodu

Vykonanie prechodu nemusí byť vždy atomické, v prípade volania metódy z časti akcií musí čakať na jej vykonanie. Preto je potrebné byť schopný uložiť stav prechodu počas tohto čakania. Prechod má taktiež priestor s premennými obsiahnutých v podmienkach a v akciách, ktorých stav je potrebné držať. Preto je potrebné vytvoriť inštanciu prechodu.

- **Inštancia prechodu** je realizovaná triedou *IntTransition* a obsahuje slovník premenných, číslo aktuálnej inštrukcie a odkaz na prechod, ktorého je inštanciou. Trieda umožňuje testovať či je možné vykonať prechod, vykonávanie podmienok a akcií prechodu.

- **Vstupné podmienky** sa skladajú z miesta a multimnožiny, ktorú musia obsahovať. Multi-množina obsahuje zoznam dvojíc počet a hodnota. Oba prvky dvojice môžu byť premennými. Pri vyhodnocovaní splniteľnosti podmienky sa postupne odstraňujú požadované objekty zo vstupného miesta. ak sa to podarí, tak je podmienka splnená. Premenná tu značí, že akákoľvek hodnota je prijateľná. Napr. podmienka o dvojici (3‘vstup) požaduje odstránenie akéhokoľvek objektu 3-krát. Sú tak štyri typy definovania požiadavky na miesto:

  1. **Počet - Hodnota**. Požaduje presne stanovený objekt, presne stanovený počet krát.
  2. **Premenná - Hodnota**. Požaduje presne stanovený objekt a odoberie všetky. ak sa v mieste objekt nenachádza, podmienka je vyhodnotená ako splnená a do premennej je vložená nula. z miesta nie je odobraný žiaden objekt.
  3. **Počet - Premenná**. Požaduje akýkoľvek objekt a odoberie ho stanovený počet krát. Objekt sa v mieste môže nachádzať aj viackrát a viacero objektov môže vyhovieť tejto podmienke. ale použije sa len prvý nájdený. Do premennej je uložený odobraný objekt.
  4. **Premenná - Premenná**. Požaduje akýkoľvek objekt a odoberie všetky jeho výskyty v mieste. akýkoľvek obsah vyhovie tejto požiadavke. ak je miesto prázdne, do premenných sú uložené hodnoty nula a *nil*.

  Pre rozličné vlastnosti jednotlivých požiadaviek je potrebné, aby im bolo vyhovené v správnom poradí. Hodnoty odobrané premennými sú v inštancii prechodu uložené v slovníku premenných a tým je umožnené ich použitie v ďalších častiach prechodu.

- **Výstupné podmienky** sa rovnako ako vstupné podmienky skladajú z miest a multimnožín. Pri testovaní splniteľnosti prechodu na ne nie je braný ohľad. Vkladajú obsah

do miest tak , ako je definovaný v multimnožinách. Taktiež sa tu môžu nachádzať premenné a na začiatku vykonávania prechodu sú inicializované na hodnotu *nil*. Pri vkladaní je na mieste premennej použitá príslušná hodnota zo slovníka premenných.

- **Testovacie podmienky** zdieľajú vlastnosti vstupných a výstupných podmienok. Na začiatku vykonávania prechodu je z miest odstránený obsah rovnako ako u vstupných podmienok a na konci zasa vložený. Použitím premenných sa tak môže zmeniť obsah testovaného miesta.

- **akcie** sú reprezentované radou inštrukcií, ktoré sú postupne vykonávané. Inštancia prechodu má uložený index aktuálnej inštrukcie a po každej vykonanej inštrukcii sa inkrementuje. Pre každý typ inštrukcie je implementovaná metóda pre vykonanie jej chovania. Inštrukcie nastavujú hodnoty premenných uložených v inštancii prechodu a tým menia jeho stav. v prípade, keď je hodnota ukladaná do premennej, ktorá nie je definovaná, je premenná vytvorená. Prístup k nedefinovanej premennej spôsobí ukončeniu interpretácie s chybou.

- **Stráž** (Guard) je uložená v modeli ako rada inštrukcií, ktoré sa musia vykonať. Ich interpretácia sa vykonáva rovnako ako pri akciách. ak má výsledok v tomto okamžiku hodnotu *true*, tak sa stráž vyhodnotí ako splnená. Na rozdiel od akcií, sa v stráži nenachádzajú priradenia hodnôt do premenných a volania metód. v stráži sa môžu nachádzať volanie synchrónneho portu.

5.3.2 Volanie metód

Metódy je možné volať z akcií prechodov. Jedná sa o inštrukciu *call*. Interpret rozoznáva dva typy metód. Vstavané metódy a užívateľské metódy. Metóda, ktorá interpretuje túto inštrukciu najprv zistí o ktorý typ sa jedná a podľa toho reaguje.

Užívateľské metódy sú definované vo vstupných súboroch, uložené v modeli a ich súčasťou je sieť. Volaním takejto metódy sa vytvára inštancia siete. Prechod čaká, dokiaľ sa neukončí vykonávanie metódy. Koniec vykonávania metódy nastáva, keď miesto *return* nie je prázdne. ako návratovú hodnotu vracia prvú nájdenú hodnotu v mieste *return*.

5.3.3 Konštruktor

Konštruktor je špeciálny typ metódy. adresátom môže byť trieda a chová sa rovnako ako je definované v jazyku PNtalk. adresovanej triede sa zašle správa *new* pre vytvorenie inštancie a vytvorenej inštancii sa zašle správa konštruktoru. Návratovou hodnotou volania nie je obsah miesta *return*, ale vždy identifikačné číslo objektu.

Na rozdiel od jazyka PNtalk riešenie umožňuje adresovať priamo inštanciu. v tomto prípade sa konštruktor chová ako metóda.

5.4 Vstavané metódy

Vstavané metódy sú povinnou súčasťou interpreta. Neobsahujú sieť. Syntax a chovanie týchto metód je inšpirované jazykom Smalltalk, ale v niektorých prípadoch sa líši.

- **new** je správa, ktorej rozumejú všetky triedy definované užívateľom. Metóda nájde v modeli triedu, vytvorí jej inštanciu a ako návratovú hodnotu vracia vygenerované identifikačné číslo.

- **hold:arg** je správa, ktorej rozumejú všetky inštancie tried. Po zadanú dobu zastaví vykonávanie akcií prechodu.
- **halt** je správa, ktorej rozumejú všetky inštancie tried. Ukončí interpretáciu s návratovou hodnotou 0. Vrátenie inej hodnoty je možné použitím správy *halt:arg*, kde *arg* je číslo.

Volanie vstavanej metódy je v modeli uložené rovnako ako volanie užívateľsky definovanej metódy inštrukciou *call*. Pri interpretácii sa najprv zistí porovnaním so vzormi správ, či sa jedná o vstavanú metódu. Chovanie vstavaných metód je definované metódami, ktoré sú implementované ako súčasť interpreta prechodu.

5.4.1 Transcript

je vstavaná trieda, ktorá slúži pre textový výstup. Trieda je inšpirovaná GNU Smalltolkom, kde *Transcript* je objektom výstupu. [3] Rozumie následujúcim správam. [4]

- **show:arg/error:arg** vypíše textovú reprezentáciu hodnoty *arg* na štandardný výstup(*stdout*)/štandardný chybový výstup(*stderr*) a vracia hodnotu *nil*. U správy *error* sa výpis ukončí novým riadkom.
- **cr** vypíše nový riadok na *stdout* a vracia hodnotu *nil*.

Pre umožnenie riadenia priebehu simulácie užívateľom bola naviac implementovaná možnosť čítať zo štandardného vstupu. Čítanie sa iniciuje zaslaním následujúcich správ triede *Transcript*. Ukončuje sa načítaním konca riadku.

- **read** načíta jeden riadok zo štandardného vstupu a vracia ho ako reťazec.
- **readNumber** načíta jeden riadok zo štandardného vstupu. Načítaný obsah je prevedený na číslo ak je to možné. Pri neschopnosti prevodu, je vrátená hodnota *nil*. Inak je vrátené číslo.

Čítanie zo štandardného vstupu je z pohľadu interpretácie siete operácia atomická. Tzn. od začiatku, do konca čítania sa nič iné v sieti nedeje. Výnimkou je len posun reálneho času.

5.4.2 Generovanie pseudo náhodných čísel

Pre podporu simulovania náhody, riešenie obsahuje vstavaný generátor pseudonáhodných čísel. Jeho použitie sa realizuje zaslaním správy **between: min and: max** triede **Random**. ako odpoveď vracia číslo z intervalu <min, max>.

Riešenie implementuje i triedy pre generovanie čísel náhodných rozložení:

- **Normálové rozloženie**. Zaslaním správy **between: min and: max** triede **Normal** vracia **reálne** číslo z intervalu <min, max> tohto rozloženia.
- **Uniformné rozloženie**. Zaslaním správy **between: min and: max** triede **Uniform** vracia **celé** číslo z intervalu <min, max> tohto rozloženia.
- **Poissnovo rozloženie**. Zaslaním správy **mean: number** triede **Poisson** vracia **celé** číslo tohto rozloženia so zadanou strednou hodnotou generovaných čísel *(number)*.

Generátory sú implementované metódami *random(), normal(), uniform(), poisson()* a sú súčasťou *IntTransition* triedy. Využívajú C++11 knižnicu *random* pre generovanie čísel. [1]

5.4.3 Pole

array(pole) je vstavaná trieda. Preberá správanie z jazyka Smalltalk.[2] Rozumie správe **new: size** pre vytvorenie inštancie pola o zadanej veľkosti. Počiatočná hodnota všetkých prvkov je *nil* a vracia vytvorenú inštanciu. Inštancia pola rozumie následujúcim správam.

- **at: index**. Prístup k prvku na zadanom indexe. Vracia hodnotu prvku.
- **at: index put: value**. Nastavenie prvku na mieste s indexom na zadanú hodnotou. Vracia pole s nastaveným prvkom.

Toto riešenie naviac rozširuje inštanciu pola o možnosť reagovať na následujúce správy:

- **append: value**. Vloží prvok so zadanou hodnotou na koniec poľa a tým zväčší jeho veľkosť o jedna. Vracia pole s vloženým prvkom.
- **size**. Vracia veľkosť pola.

Vytváranie inštancií je možné i vymenovaním prvkov a to dvomi spôsobmi.

1. Zaslaním [**with: value**]+ správy triede *array*. Počet dvojíc *with: value* v správe určuje veľkosť pola a zadané prvky určujú počiatočné hodnoty prvkov pola. ako odpoveď na správu trieda vracia vytvorenú inštanciu.
2. v tvare zoznamu. Tzn. (v_1, v_2, ... , v_i), kde v_i je prvok. Prvkom môže byť i ďalší zoznam alebo výraz, ktorý sa vyhodnotí pred vytvorením poľa. Pri použití v akciách prechodu sa funkcionálne nijako nelíši od zápisu vo forme správy. Napr. výrazy *array with: 15 with: 3* a *(15, 3)* vytvoria rovnaký objekt. Tento zápis je ale naviac možné použiť i pri definícii počiatočného stavu miest a v podmienkach prechodu, kde nie je možné zasielanie správ.

V riešení je pole ukladané ako referencia typu *pole*. Prvky pola sú uložené ako zoznam priamo v referencii a sú reprezentované ďalšími referenciami.

5.4.4 Pole ako podmienka prechodu

Pole je možné použiť ako súčasť podmienky prechodu. Pole definované ako zoznam prvkov. Pre splnenie podmienky to znamená, že v mieste musí byť ekvivalentné pole. Uloženie poľa ako referencie toto jednoducho umožňuje a platí, že dve polia sú ekvivalentné práve vtedy, keď sú ekvivalentné všetky ich prvky a to v rovnakom poradí.

V podmienkach je taktiež možné vložiť ako požiadavku identifikátor. Tým povedať, že akákoľvek hodnota je prijateľná a uložiť ju do premennej. Identifikátor je možné takýmto spôsobom použiť i vo vnútri zoznamu. Definuje sa tak, že prvok na danom umiestnení v poli, môže mať akúkoľvek hodnotu a je uložená do premennej. Veľkosť pola musí stále zodpovedať.

5.5 Kalendár udalostí

Riešenie umožňuje modelovať chovanie systému v čase. Pre simuláciu tohto chovania je implementovaný kalendár udalostí.

V akciách prechodu je možné zaslať správu **hold: arg** a tým pozastaviť vykonávanie akcií a tak i celého prechodu po zadanú dobu. Vytvorí sa udalosť v kalendári, ktorá obnoví vykonávanie prechodu až sa dosiahne daného časového okamžiku.

V závislosti od času, ktorý je použitý, kalendár umožňuje pracovať vo dvoch režimoch:

1. **Modelový čas**. Čas kalendára sa pohybuje skokovo.
   - Práca v modelovom čase sa nastaví zadaním argumentu '–simulation'.
   - v momente, keď interpret nie je schopný vykonať žiadny prechod, vyberie sa najbližšia udalosť, vykoná sa a aktuálny čas sa posunie na čas tejto udalosti.
   - Čas novo vytvorenej udalosti je získaný sčítaním aktuálneho modelového času a zadanej doby.
2. **Reálny čas**. Čas sa pohybuje plynule spolu s reálnym časom.
   - Prednastavený režim.
   - Udalosť v kalendári je vykonaná vtedy, keď reálny čas prekročí časový okamžik udalosti.
   - Čas novo vytvorenej udalosti sa vypočíta sčítaním aktuálneho času so zadaným časom. Jednotka zadaného času je milisekunda.

Kalendár udalostí je implementovaný triedou *Calendar*. Udalosti sú ukladané do zoradenej štruktúry. Obsahujú čas udalostí, podľa ktorého sú zoradené a stav prechodu, z ktorého bola udalosť vytvorená pre návrat k jeho vykonávaniu.

Pre prácu s ním poskytuje štyri metódy.

1. *set_event(duration, transition)* metóda vytvorí udalosť v kalendári, ktorá sa vykoná za zadanú dobu *duration*. Udalosť bude pokračovať vo vykonávaní zadaného prechodu *transition*.
2. *get_events()* metóda slúži pre prácu v reálnom čase. Vracia zoznam prechodov, ktorých vykonávanie má pokračovať.
3. *next_event()* metóda pre prácu v modelovom čase. Vracia prechod následujúcej udalosti a posúva modelový čas na čas udalosti.
4. *empty()* metóda slúži pre overenie, či sa v kalendári nachádzajú udalosti.

Kalendár udalostí rozširuje podmienku pre ukončenie interpretácie modelu, nie je možné vykonať žiadny prechod, podmienkou, že musí byť aj kalendár udalostí prázdny pre to, aby sa mohla interpretácia ukončiť.

5.6 Dedičnosť

Trieda definovaná užívateľom musí mať definovanú triedu ,od ktorej dedí. Dediť je možné od základnej triedy *PN* alebo od inej užívateľom definovanej triedy.

Dedenie siete, tzn. miest a prechodov prebieha pridaním miest a prechodov rodičovskej triedy do siete. ak trieda, ktorá dedí už miesto alebo prechod s rovnakým názvom obsahuje, nepreberá ho. Dedenie siete prebieha na konci načítania vstupných súborov. Trieda má tak v modeli uloženú celú sieť pred začiatkom interpretácie a pri vytváraní inštancie tak len vytvorí kópiu miest.

Dedenie metód, konštruktorov a synchrónnych portov tento prístup neumožňuje, pretože je možné pristupovať aj k tým, ktoré majú rovnakú identifikáciu u rodiča i potomka. Zaslaním správy objektu, objekt reaguje tak, ako je implementované v jeho triede. ak trieda nemá implementovanú reakciu na danú správu, rekurzívne volá rodiča pre získanie reakcie.

Objekt môže zasielať správy sebe samému z prechodov triedy a metód. Pri tomto zaslaní správy sa ako adresát určuje kľúčovým slovom **self**. Ukazuje tak na seba. Pre cielenie reakcie na správu rodičovskej triedy sa adresát určuje kľúčovým slovom **super**. Vtedy objekt rovnako ukazuje na seba, ale rekurzívne hľadanie reakcie na zaslanie správy začína rodičom svojej triedy.

Kapitola 6

Testovanie

V tejto kapitole je popísaný spôsob testovania vlastností riešenia pre zaistenie kvality a je uvedených niekoľko príkladov, pre demonštrovanie jeho vlastností.

6.1 Typy testov

Správne chovanie riešenia je zaisťované sériou testov. Testy overujú či sa riešenie správa tak, ako je zadefinované a popisujú správanie v hraničných a nezadefinovaných prípadoch. z hľadiska výsledku interpretácie rozdeľujeme testy na dva typy:

1. **Pozitívne testy** overujú správanie riešenia v situáciach, kde sa očakáva úspešné ukončenie interpretácie. Tzn. interpret vráti nulový návratový kód. Sú zamerané na kladné scenáre. Overujú, či interpret reaguje na jednotlivé štruktúry tak ako sú zadefinované. Vkladajú interpretu na vstup súbor s platným modelom a verifikujú jeho správanie.

2. **Negatívne testy** overujú správanie riešenia v situáciach, kde sa očakáva chybové ukončenie interpretácie. Do súboru s platným modelom je zavedená chyba. Súbor s chybným modelom je vložený na vstup interpreta. Sleduje sa schopnosť odhaliť chybu a správne na ňu reagovať.

6.2 automatizované testovanie

Pre odstránenie nutnosti testovať ručne bol vypracovaný skript, ktorý tento proces automatizuje. Pre jeho implementáciu bol použitý jazyk Python3, ktorý umožňuje multiplatformové použitie skriptu.

Skript pre spustenie vyžaduje Python s verziou najmenej 3.6 a spustiteľný súbor interpretu, ktorý je zadaný skriptu ako argument pri spúšťaní.

Test vykonávaný skriptom sa skladá zo vstupného súboru s modelom a súborov s očakávaným výstupom. Všetky súbory testu sú pomenované názvom testu a príponou, o aký typ testu sa jedná. Vstupný súbor je určený príponou „.in“ Výstupné súbory majú príponu „.out“, pre očakávaný štandardný výstup a príponu „.err“ pre očakávaný chybový výstup. Výstupné súbory nie sú povinnou súčasťou testu, ak sa neočakáva žiadny výstup daného typu. Test je uložený v zložke s pozitívnymi testami, alebo v zložke s negatívnymi testami. To, v ktorej sa nachádza, určuje aká je očakávaná návratová hodnota. Všetky časti testu sú uložené v tej rovnakej zložke.

Hlavná časť skriptu sa skladá z troch častí:

1. **Vytvorenie zoznamu testov**. Vytvoria sa dva zoznamy. Zvlášť pre pozitívne a negatívne testy. Každý zoznam sa naplní názvami testov získaných z príslušnej zložky. Test je pridaný do zoznamu vtedy, ak je nájdený vstupný súbor testu s príponou „.in“.

2. **Spustenie testov**. Testy sú spúšťané po jednom zo zoznamu spustením interpreta so vstupným súborom testu. Skript čaká na koniec interpretácie a zachytáva výstup. Pre prípad chyby má skript určenú maximálnu dobu čakania. Po uplynutí času ukončuje test ako neúspešný.

3. **Vyhodnotenie výsledkov**. Porovnáva sa návratový kód interpreta s očakávaným kódom, ktorý je nula pre pozitívne testy a jedna pre negatívne. Zachytený výstup je uložený do súborov. Ich obsah je porovnaný so súbormi očakávaného výstupu testu ak sú definované. Skript vypíše na štandardný chybový výstup zoznam testov, ktoré boli neúspešné a počet úspešných testov.

Pre niektoré testy takýto prístup nie je vhodný. Napr. pri testovaní viacerých vstupných súborov naraz. Testy tohto typu sú taktiež súčasťou skriptu, ale ich priebeh je odlišný.

6.3 Príklady použitia

Súčasťou testovania bolo vytvorenie sady modelov pre demonštrovanie chovania interpreta.

6.3.1 Výstup

Prvým príkladom je jednoduchý model zložený len z jednej triedy *Printer* obsahujúcej len jeden prechod. Cieľom je demonštrovať použitie výstupných operácii. Implementáciu modelu je možné vidieť na obrázku 6.1.

```
class Printer is_a PN
object
  trans print
    action{
      Transcript show: 'standard output'.
      Transcript cr.
      Transcript error: 'error output'.
      self halt: 0.
    }
main Printer
```

Obr. 6.1: Implementácia triedy Printer

Prechod *print* triedy *Printer* nemá žiadnu podmienku. Prechod je možné vykonať vždy. Po inicializácii modelu, sa ihneď začne vykonávať.

akcie prechodu sú vykonávane postupne jedna po druhej. Prvá akcia je zaslanie správy *show: 'standard output'* triede *'Transcript'*. Tá na správu reaguje tiskom textu „standard output“ na štandardný výstup. Všetky metódy triedy Transcript sú atomické operácie. Tzn. nevyžadujú z hľadiska interpretácie žiadny čas pre ich vykonanie a nič sa neudeje dokiaľ nie

sú ukončené. Preto sa vykonávanie prechodu nezastaví a pokračuje ďalej na ďalšiu akciu, ktorou je zaslanie správy *cr* triede *Transcript*. Výsledkom je tisk nového riadku na štadardný výstup.

Nasleduje zaslanie správy *error: 'error output'* rovnako triede *Transcript*. Výsledkom je tisk textu podobne ako u *show*, ale je vypísaná na štandardný chybový výstup a je zakončená novým riadkom.

Druhou časťou správ *show:* a *error:* môže byť akákoľvek hodnota. v prípade premennej je vypísaný jej obsah. ak je hodnotou objekt, je vypísané jeho identifikačné číslo.

akcie sú zakončené zaslaním správy *halt: 0* pre ukončenie interpretácie s návratovým kódom nula. Okrem nuly je možné použiť i iné kladné celé číslo. ak by interpretácia nebola takýmto spôsobom ukončená, prechod by sa vykonával stále dokola, keďže všetky podmienky pre jeho vykonanie sú vždy splnené a interpretácia by sa nikdy neukončila.

6.3.2 Podmienky prechodu

Pre demonštráciu ako sú vyhodnocované podmienky je vytvorený model siete tvorený jedinou triedou *Condition*. Sieť sa skladá z troch miest a troch prechodov. Model je navrhnutý tak, aby sa každý prechod vykonal práve raz a vykonanie prechodu bolo umožnené vykonaním predchádzajúceho prechodu. Implementáciu je možné vidieť na obrázku 6.2 a výstup interpretácie tohto modelu na obrázku 6.3.

```
class Condition is_a PN
object
  place start('start')
  place stage1(2`2, 8`3, 4`5)
  place stage2(#token)

  trans first
    precond start(var)
    action{
      Transcript show: var. Transcript cr.
    }
    postcond stage1(1`1)
  trans second
    precond stage1(last1`last2, 1`1, count`2, 8`value)
    cond stage2(#token)
    action{
      Transcript show: count. Transcript cr.
      Transcript show: value. Transcript cr.
      Transcript show: (last1 + ' and ' + last2).
      Transcript cr.
    }
    postcond stage1('finish'), stage2(2`#token)
  trans third
    precond stage1(var), stage2(3`#token)
    action{
      Transcript show: var.
      Transcript cr.
    }

main Condition
```

Obr. 6.2: Implementácia triedy Condition

```
start
2
3
4 and 5
finish
```

Obr. 6.3: Obsah štandartného výstupu po interpretácii Condition modelu

- Prechod *first*. Vykonanie je podmienené jednou vstupnou podmienkou, ktorá vyžaduje objekt v mieste *start*. Podmienka obsahuje identifikátor *var*. Môže sa tak jednať o akýkoľvek objekt. Odoberie sa z miesta a uloží do premennej *var*. Miesto *start* a ani podmienka nemajú určený počet inštancií, pretože počet jedna je implicitný.

  Ukončením prechodu sa vykoná výstupná podmienka. Vloží číslo jeden do miesta *stage1* a jeho obsah je tak následujúci: (1‘1, 2‘2, 8‘3, 4‘5).

- Prechod *second* má vstupnú podmienku, kde požaduje od miesta *stage1* aby obsahovalo, číslo jeden aspoň raz a akýkoľvek objekt osem-krát. Ďalej sa z miesta odoberú všetky inštancie čísla 2 a všetky inštancie jedného zo zvyšných objektov v mieste.

  Jednotlivé požiadavky sa vyhodnocujú postupne tak, aby boli všetky podmienky splnené, ak je to možné.

  1. Vyhovie sa požiadavke „1‘1“ a odoberie sa z miesta *stage1* číslo jeden. v mieste ostáva obsah: (2‘2, 8‘3, 4‘5)
  2. Vyhovie sa požiadavke „count‘2“, pre odobranie všetkých inštancií čísla dva. i v prípade, keď sa číslo dva v mieste nenachádza, je táto podmienka splnená. Do premennej *count* je uložené číslo, reprezentujúce počet odobraných inštancií. v mieste ostane (8‘3, 4‘5)
  3. Požiadavka *8‘value* odoberie osem inštancií objektu, ktoré sa v mieste nachádza aspoň osemkrát. Objekt je uložený do premennej *value*. v mieste ostáva len (4‘5)
  4. Posledná požiadavka odoberie akýkoľvek objekt a odoberie všetky jeho inštancie. Do premenných *last1*, *last2* je vložený odobraný počet a odobraný objekt. v tomto prípade sa tak odoberie zvyšný obsah miesta. v prípade, ak by bolo miesto prázdne, požiadavka je stále splnená a do oboch premenných je uložená hodnota *nil*.

  Prechod je podmienený i testovacou podmienkou, ktorá vyžaduje aby miesto *stage2* obsahovalo aspoň jeden *#token*. Ten je na začiatku interpretácie prechodu odobraný a po jeho ukončený vrátený do miesta.

  Výstupné podmienky vložia do miesta *stage1* reťazec „finish“ a do miesta *stage2* dva-krát *#token* a tým zvýši jeho počet na tri.

  Stav miest po vykonaní tohto prechodu je následujúci:
  start(), stage1(’finish’), stage2(3‘#token)

- Posledný prechod *third* má vstupné podmienky pre odstránenie akejkoľvek inštancie z miesta *stage1*, vloží ju do premennej *var* a odstránenie troch inštancií symbolu *#token*. Prechod nemá žiadne výstupné podmienky.

Vykonaním posledného prechodu *third* sú všetky miesta prázdne. Žiaden prechod nie je možné vykonať pre nesplnenie podmienok, preto sa interpretácia ukončuje.

6.3.3 Zasielanie správ

Trieda môže mať definované správy, ktorým ona alebo jej inštancia rozumie a reakcie na tieto správy. Jedná sa o správy typu konštruktor, metóda a synchrónny port. Pre demonštráciu ich použitia a správania je vytvorený model, ktorý je možné vidieť na obrázku 6.4 a obsah štandardného výstupu na obrázku 6.5.

```
main Main
class Main is_a PN
object
  place start(#e)
  place created_objects()
  trans create
    precond start(#e)
    action{
      obj_by_new := Calculator new.
      obj_by_constructor := Calculator create: #initiated.
    }
    postcond created_objects(obj_by_new, obj_by_constructor)

  trans calculate
    precond created_objects(obj)
    guard {obj is: #initiated}
    action{
      result := obj add: 5 with: 3.
      Transcript show: ('5 + 3 = ' + result). Transcript cr.
    }

  trans init
    precond created_objects(obj)
    guard { obj is: #new }
    action{
      returned := obj create: #updated.
      Transcript show: 'Constructor returns: '.
      Transcript show: returned. Transcript cr.
      result := obj add: 1 with: 2.
      Transcript show: ('1 + 2 = ' + result).
      Transcript cr.
      }
    postcond created_objects(obj)
```

```
class Calculator is_a PN
object
  place state(#new)

constructor create: arg
  trans init
    precond state(#new), arg(new_state)
    postcond state(new_state), return(0)

sync is: expected_state
  precond state(s)
  guard{ s == expected_state }

method add: num1 with: num2
  trans run
    precond num1(first), num2(second)
    action {
      result := first + second.
    }
    postcond return(result)
```

Obr. 6.4: Implementácia demonštračného modelu na zasielanie správ

```
Constructor returns: 0
1 + 2 = 3
5 + 3 = 8
```

Obr. 6.5: Štandardný výstup vykonaním demonštračného modelu na zasielanie správ

Model definuje dve triedy.

1. Trieda **Calculator** má sieť o jednom mieste *state* určujúci stav a implementuje reakcie na tri správy:

   - Metóda *add: num1 with: num2* sčíta čísla *num1*, *num2* a vracia výsledok. Realizuje to sieťou o troch miestach a jednom prechode. Miesta *num1* a *num2* sú definované správou a výstupné miesto *return* je povinná súčasť každej metódy. Preto nemusia byť explicitne uvedené.

Metóda na začiatku naplní miesta hodnotami zo správy, čím sa umožní vykonanie prechodu. Výstupnou podmienkou sa vloží výsledok do miesta *return*. Tým sa ukončí vykonávanie metódy a vráti sa výsledok.

- Konštruktor *create: arg* nastavuje stav objektu. Správu je možné zasielať triede i objektu.

  ak je zaslaný triede, vytvorí sa nová inštancia triedy a jej sa zašle konštruktor, tak ako by bol metóda, s tým rozdielom, že nevracia hodnotu z miesta return, ale vytvorený objekt.

  Pri zaslaní správy objektu je reakcia úplne rovnaká, ako by sa jednalo o metódu.
- Synchrónny port *is: expected_state* testuje pomocou stráže, či má objekt očakávaný stav *expected_state*. Vstupná podmienka synchrónneho portu neodstraňuje obsah z miesta *state*. Synchrónny port je možné volať len zo stráže.

2. Trieda **Main** je počiatočnou triedou. Vytvára inštancie triedy *Calculator* a zasiela jej správy. Sieť obsahuje tri prechody:

   - *create* pre vytvorenie dvoch inštancií. Jednu správou *new* a druhú správou konštruktoru *create: #initiated*.
   - *calculate* je vykonaný, ak je objekt v stave *#initialized*. Pre overenie stavu používa definovaný synchrónny port. Prechod volá metódu *add:5 with:3* objektu a vypisuje výsledok.
   - *init* je vykonaný, ak objekt je v stave *#new*. Rovnako ako prechod *calculate* pre overenie používa synchrónny port objektu. Zasiela správu konštruktoru na inštanciu modelu a vypisuje návratovú hodnotu. Na koniec volá metódu *add:1 with:2* a vypisuje výsledok.

6.3.4 Dedičnosť

Trieda môže dediť od inej triedy a tým prebrať jej vlastnosti. Je vytvorený model pre demonštráciu dedenia a operácií, ktoré s tým súvisia. Implementáciu modelu je možné vidieť na obrázku 6.6 a štandardný výstup po vykonaní modelu na obrázku 6.7.

```
class A is_a PN
object
  place name()
  place class_name('Class A')
  trans print_name
    precond name(var), class_name(cls)
      action{
        Transcript show:'A transition'.
        Transcript cr.
        Transcript show:(cls + ' : ' + var).
        Transcript cr.
        }
  constructor create: new_name
    trans init
      precond new_name(n)
      postcond name(n),return(#e)
  method get
    place return('A')

class B is_a A
object
  place class_name('Class B')
  trans print_name
  precond name(var), class_name(cls)
    action{
      Transcript show: 'B transition'.
      Transcript cr.
      Transcript show: (cls + ' : ' + var).
      Transcript cr.
    }
  method get
    trans run
      action{
        retVal := super get.
        retVal := retVal + ' and B methods'.
      }
    postcond return(retVal)
```

```
class Main is_a PN
object
  place start(#e)
  place state()

  trans create_object
    precond start(#e)
    action{
      o := B create: 'The Name'.
    }
    postcond state(o)

  trans run
    precond state(o)
    action{
      Transcript show: (o get).
      Transcript cr.
    }

main Main
```

Obr. 6.6: Implementácia demonštračného modelu dedenia

```
B transition
Class B : The Name
A and B methods
```

Obr. 6.7: Štandardný výstup po vykonaní demonštračného modelu dedenia

Model sa skladá z troch tried, kde trieda *B* dedí od triedy *a* a počiatočná trieda *Main*, ktorá triedu používa tak, aby boli vidieť jej vlastnosti získané dedením.

- Zaslaním správy **B create: 'The Name'** sa vytvorí inštancia triedy *B*. Tá by tejto správe bez dedenia nerozumela, ale dedením túto schopnosť získava a vykoná sa konštruktor definovaný triedou *a*. Konštruktor vloží do miesta *name* hodnotu argumentu správy. Miesto *name* je taktiež získané dedením z triedy *a*.

- Obe triedy *a*, *B* majú definovaný prechod s názvom **print_name**. v tomto prípade je platný iba ten definovaný triedou *B*. Vďaka vloženiu mena do miesta *name* pri vytváraní inštancie, je možné teraz prechod vykonať.

- Zaslaním správy **o get** je volaná metóda *get*. Tá je definovaná zasa obomi triedami. Je vybraná definícia triedy *B*. Tá v akciách svojho prechodu obsahuje zaslanie správy **super get**. Kľúčové slovo *super* odkazuje na seba. Je tak znova volaná metóda *get*, ale teraz definícia triedy od ktorej trieda *B* dedí. Tzn. od triedy *a*.

6.3.5 autobusová zastávka

Pre demonštráciu modelovania reálneho systému je vytvorený model autobusovej zastávky inšpirovaný problémom *The Senate Bus problem* z knihy *The Little Book of Semaphores*. [5] Na zastávku prichádzajú pasažieri a čakajú na príchod autobusu. autobus prichádza prázdny na zastávku. Lúdia čakajúci na zastávke nastúpia a autobus odíde. autobus má maximálnu kapacitu 50 ľudí. ak je na zastávke ľudí viac, zvyšný čakajú na ďalší príchod autobusu.

Model systému je možné vidieť na obrázku 6.8 a skladá sa z dvoch tried:

1. **Bus_stop** je trieda, ktorá simuluje chovanie na autobusovej zastávke. Je to zároveň počiatočná trieda. Inštanciovaná na začiatku interpretácie. Sieť sa skladá zo štyroch miest a troch prechodov.

   Miesto *assigned_bus* ukladá autobus, ktorý k zastávke jazdí a miesto *waiting_riders* obsahuje ľudí, ktorý čakajú na zastávke. Počiatočným stavom je 60 ľudí.

   Prechod *rider_arrive* generuje ľudí. Simuluje tak ich príchod na zastávku. Použitie miesta generátor v testovacej podmienke zabezpečuje, že prechod sa v jednom okamžiku vykonáva len raz, ale tým že je to jediná podmienka prechodu sa zabezpečuje jeho nepretržitý beh. Vďaka čomu je možné regulovať prichádzanie ľudí na zastávku.

   Prechod *bus_board* slúži pre simulovanie nástupu ľudí do autobusu. Obsahuje stráž, v ktorej prostredníctvom synchrónneho portu kontroluje či je autobus na zastávke. Zasiela správu autobusu pre nástup pasažierov.

2. **Bus** je trieda, ktorá simuluje autobus jazdiaci na zastávku. Má obmedzený počet jázd. Množstvo ostávajúcich jázd je uložené v mieste *rides*. Zároveň si ukladá pasažierov, ktorých previezol behom týchto jázd do miesta *passengers*. Sieť triedy má dva prechody a synchrónny port *in_station*, ktorý slúži pre testovanie prítomnosti autobusu na zastávke.

   Prechod *ride* simuluje čas, keď je autobus mimo zastávku a odčítava počet jázd, ktoré ostávajú.

   Prechod *finish* vypíše počet pasažierov, ktorý boli prevezený a ukončuje simuláciu. Jeho vykonanie je podmienené strážou, ktorá požaduje aby počet ostávajúcich jázd bol nulový.

   autobus taktiež implementuje metódu pre nástup pasažierov ako reakciu na správu *board: ammount_in_station*, ktorá simuluje nástup pasažierov. Jej sieť obsahuje dva prechody *take_maximum* a *take_all*. Oba ukončujú metódu. Preto sa môže vždy vykonať iba jeden a to, ktorý sa vykoná je určené podľa počtu osôb, ktoré majú nastúpiť. Metóda vracia počet ľudí, ktorý nenastúpili. Po jej ukončení je autobus na ceste a môže začať jazda.

```
class Bus is_a PN
object
    place rides(5`#e)
    place passengers()
    place on_road()
    place in_station(#e)
    trans ride
        precond on_road(#e),rides(#e)
        action {
            self hold:(Random between:80 and:120).
            Transcript show:('Bus arrived to station').Transcript cr.
        }
        postcond in_station(#e)
    trans finish
        precond passengers(amount`#rider),
                rides(left_rides`#e)
        guard { left_rides == 0 }
        action {
            Transcript show:('Bus transported '+amount+' riders').
            Transcript cr.
            self halt.
        }
sync in_station
    precond in_station(#e)
method board: amount_in_station
    place return()
    trans take_maximum
        precond amount_in_station(amount),in_station(#e)
        guard { amount > 50 }
        action {
            left_amount := amount - 50.
            Transcript show:('50 riders boarded to bus ').
            Transcript show:('and '+left_amount+' were left in station').
            Transcript cr.
        }
        postcond return(left_amount), passengers(50`#rider),
                 on_road(#e)
    trans take_all
        precond amount_in_station(amount), in_station(#e)
        guard { amount <= 50 }
        action {
            Transcript show:( amount+' riders boarded to bus').
            Transcript cr.
        }
        postcond return(0),passengers(amount`#rider),on_road(#e)
```

```
main Bus_stop

class Bus_stop is_a PN
object
    place start(#e)
    place waiting_riders(60`#rider)
    place assigned_bus()
    place generator(#e)
    trans create_bus
        precond start(#e)
        action {
            bus := Bus new.
        }
        postcond assigned_bus(bus)
    trans rider_arrive
        cond generator(#e)
        action {
            wait := (Random between:1 and:10).
            self hold:wait.
                }
        postcond waiting_riders(#rider)
    trans bus_board
        cond assigned_bus(bus),
             waiting_riders(riders`#rider)
        guard { bus in_station }
        action {
            riders := bus board:riders.
        }
```

Obr. 6.8: Implementácia autobusovej zastávky

Cieľom tohto príkladu bolo demonštrovať použitie základnej syntaxe, volania metód, výstupu, podmienok, vstavaných metód a priebeh interpretácie. Príklad výstupu interpretácie tohto modelu je možné vidieť na obrázku 6.9.

```
50 riders boarded to bus and 10 were left in station
Bus arrived to station
29 riders boarded to bus
Bus arrived to station
18 riders boarded to bus
Bus arrived to station
21 riders boarded to bus
Bus arrived to station
18 riders boarded to bus
Bus transported 136 riders
```

Obr. 6.9: Príklad výstupu modelu autobusovej zastávky

Kapitola 7

Záver

V bakalárskej práci boli preskúmané Petriho siete ako nástroj pre modelovanie systémov diskrétnych distribuovaných systémov. So zameraním na ich štruktúru a dynamiku. Súčasťou prieskumu boli aj viaceré jednoduché rozšírenia, ktoré pridávajú viacej možností pri modelovaní. s týmito základmi bolo preskúmané závažnejšie rozšírenie, ktoré okrem iného rozširuje Petriho siete o objektovú orientáciu. Ide o objektovo orientované Petriho siete a jazyku pre definovanie modelov objektovo orientovaných Petriho sietí PNtalk založený na jazyku Smalltalk. Umožňujúce pri modelovaní využiť vlastností objektovo orientovaného programovania ako napríklad zapuzdrenie, dedičnosť, abstrakcia a ďalšie.

V praktickej časti práce bol navrhnutý a implementovaný nástroj, ktorí interpretuje objektovo orientovanú Petriho sieť zapísanú v jazyku PNtalk. Implementácia bola vykonaná v jazyku C++17. Riešenie bolo rozdelené do troch častí. Prekladač ako prvá časť slúži pre načítanie modelu zo zdrojových súborov zapísaných v jazyku PNtalk do vnútornej reprezentácie použitím lexikálnej a syntaktickej analýzy. Druhá časť riešenia, model, slúži pre vnútornú reprezentáciu načítaného modelu implementovanú skupinou tried v stromovej štruktúre s triedou *model* ako koreň tohto stromu. Slúži ako šablóna, z ktorej sú vytvárané inštancie prvkov modelu. Počas priebehu interpretácie sa nemení. Posledná, tretia časť riešenia ,interpret, vedie interpretáciu načítaného modelu. Vytvára inštancie prvkov siete a simuluje ich dynamiku.

Nasledovne bolo vytvorené množstvo testov, pre kontrolu správneho fungovania a reakcií na neočakávané vstupy. Pre zjednodušenie testovania a ďalšieho vývoja bol vytvorený skript implementovaný v jazyku Python, pre automatizované spúšťanie testov. Testovanie bolo zakončené vytvorením série modelov a popisom priebehu ich interpretácie. s cieľom demonštrovať spôsob použitia prvkov PNtalku. Riešene umožňuje interpretáciu modelov v jazyku PNtalk. Pre nevyhnutnú potrebu pozorovania výsledkov interpretácie a pre uľahčenie modelovania bola implementovaná sada nástrojov, inšpirovaná jazykom Smalltalk pre výpis výsledkov, modelovanie času a náhody.

Literatúra

[1] Random. *Cplusplus* [online]. Dostupné z: http://www.cplusplus.com/reference/random/.

[2] An array in Smalltalk. *GNU smalltalk* [online]. Dostupné z: https://www.gnu.org/software/smalltalk/manual/html_node/Arrays.html.

[3] The Output Stream. *GNU smalltalk* [online]. 1998 [cit. !2019/11/11]. Dostupné z: https://www.gnu.org/software/smalltalk/manual/html_node/The-output-stream.html.

[4] TextCollector: accessing. *GNU smalltalk* [online]. 1998 [cit. !2019/11/11]. Dostupné z: https://www.gnu.org/software/smalltalk/manual-base/html_node/TextCollector_002daccessing.html.

[5] Downey, A. *The Little Book of Semaphores*. Green Tea Press, 2016. Dostupné z: https://open.umn.edu/opentextbooks/textbooks/83.

[6] Janoušek, V. *Modelování objektů Petriho sítěmi*. Brno, 1998. 121 s. Dizertačná práca. FEI VUT v Brně.

[7] Meduna, A. a Lukáš, R. *Formální jazyky a překladače: Studijní opora*. Božetěchova 2, 612 00 Brno-Královo Pole: FIT VUT v Brně, 2015.