VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

GRAFICKÝ EDITOR NÁVRHOVÝCH MODELŮ

GRAPHICAL EDITOR FOR DESIGN MODELS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. DANIEL ŠVUB**
AUTHOR

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

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2021/2022

Zadání diplomové práce

Student: **Švub Daniel, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Inteligentní systémy
Název: **Grafický editor návrhových modelů**
Graphical Editor for Design Models
Kategorie: Softwarové inženýrství

Zadání:

1. Prostudujte formalismus Objektově orientovaných Petriho sítí (OOPN).
2. Navrhněte nástroj pro editaci a ladění modelů OOPN. Nástroj musí umožnit ukládání a načítání modelů ve vhodném formátu. Musí být připraven na vzdálenou práci se simulačním serverem, tj. získávat modely ze serveru, posílat modely na server a spustit, zastavit a krokovat simulaci a zobrazit aktuální stav simulace.
3. Implementujte nástroj v prostředí jazyka Java. Rozšiřte nástroj o možnost exportu modelů do formátu SVG (Scalable Vector Graphics).
4. Vytvořte manuál a sadu příkladů demonstrující možnosti nástroje, zejména komunikaci se simulačním serverem.
5. Diskutujte dosažené výsledky a navrhněte možná rozšíření nástroje. Výsledky také prezentujte formou posteru.

Literatura:

- Janoušek, V.: Modelování objektů Petriho sítěmi, Brno, CZ, UIVT FEI VUT, 1998

Při obhajobě semestrální části projektu je požadováno:

- První dva body.

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 2021
Datum odevzdání: 29. července 2022
Datum schválení: 3. listopadu 2021

Abstrakt

Cílem této práce byl návrh a vývoj uživatelsky přívětivého grafického editoru Objektově orientovaných Petriho sítí (OOPN). Pro popis tohoto formalismu je použit jazyk PNtalk, který spojuje Petriho sítě s objektově orientovaným jazykem Smalltalk. Výsledná aplikace má uživatelům umožnit modelování objektů a jejich chování za pomocí OOPN a simulování přímo v editoru.

Abstract

The target of this thesis was a design and a development of an user friendly graphical editor of Object oriented Petri nets (OOPN). For description of this formalism was used PNtalk language, which combines Petri nets with the object oriented language Smalltalk. The resulting application should allow users to design object and their behaviour via OOPN and simulate directly in the editor.

Klíčová slova

Petriho sítě, objektová orientace, Objektově orientované Petriho sítě, OOPN, PNtalk, Java, JavaFX, simulace

Keywords

Petri nets, object orientation, Object oriented Petri nets, OOPN, PNtalk, Java, JavaFX, simulation

Citace

ŠVUB, Daniel. *Grafický editor návrhových modelů*. Brno, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Kočí, Ph.D.

Grafický editor návrhových modelů

Prohlášení

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

.....
Daniel Švub
27. července 2022

Poděkování

Děkuji Ing. Radkovi Kočímu, Ph.D. za vedení a odbornou pomoc při tvorbě této práce.

Obsah

Kapitola 1

Úvod

Petriho sítě jsou široce používaným formalismem, jejich možnosti popisu reálného světa jsou však omezené, neboť při popisu rozsáhlých konstrukcí se rychle stanou nepřehlednými. Tyto nedostatky odstraňují Objektově orientované Petriho sítě, které spojují výhody Petriho sítí s výhodami objektově orientovaných programovacích jazyků. Díky tomu mohou sloužit k popisu objektově orientovaných programů nezávislém na platformě.

Na Fakultě informačních technologií Vysokého učení technického v Brně dlouhodobě probíhá projekt PNtalk. Jedná se o implementaci Objektově orientovaných Petriho sítí využívající prvky jazyka Smalltalk. V rámci tohoto projektu je mimo jiné vyvíjen i simulační server, jenž model v jazyce PNtalk umožňuje spustit a sledovat jeho stav krok po kroku.

Tato práce si klade za cíl vytvořit grafický editor, který umožní modelování a simulování modelů popsaných v jazyce PNtalk. Navazuje na diplomovou práci Ing. Antonína Neužila z roku 2020 a usiluje o rozšíření a odstranění nedostatků tohoto projektu, s cílem poskytnout FIT VUT v Brně reálně využitelný nástroj pro editaci modelů v jazyce PNtalk, který bude dále rozšiřitelný.

Aplikace je vhodná zejména pro vizualizaci a demonstraci projektů, návrhů programů a simulačních modelů. Díky funkci exportu do vektorového grafického formátu SVG lze hotové sítě používat v prezentacích nebo je tisknout. Model je navíc díky propojení se simulačním serverem možné simulovat přímo v okně programu. Narozdíl od předchozí verze nový nástroj obsahuje přívětivější a přehlednější uživatelské rozhraní, opravuje také některé chyby, které se v ní vyskytují.

V této technické zprávě si nejdříve v kapitole 2 představíme formalismus Objektově orientovaných Petriho sítí a jazyk PNtalk. Kapitola 3 potom obsahuje návrh uživatelského rozhraní a architektury aplikace, kapitola 4 popisuje její implementaci. Závěrečná kapitola se věnuje praktickému použití vyvinutého nástroje a obsahuje i konkrétní příklady.

Kapitola 2

Objektově orientované Petriho sítě

V této kapitole si představíme formalismus objektových Petriho sítí a jeho vlastnosti. Důvodem pro vznik tohoto rozšíření je absence strukturovatelnosti běžných P/T Petriho sítí. Jejich hlavní výhoda, přehlednost a jednoduchost, je zároveň jejich velkou slabinou, chcemeli s jejich pomocí popsat složitější systém.

Objektově orientované Petriho sítě umožňují hierarchicky zanořovat jednotlivé prvky systému a používat je vícekrát. Jejich definice je poměrně volná a existuje mnoho variant a implementací. Tato práce se zabývá jazykem PNtalk, imlementací vyvíjenou na Faluktě informačních technologií Vysokého učení technického v Brně.

2.1 Objektová orientace

Nejprve si stručně popišme koncept objektové orientace a důvod, proč je pro nás vlastně výhodné ji v modelu zavádět.

Objektová orientace si klade za cíl co nejvěrněji reprezentovat reálný svět. Každý objekt ve vesmíru má nějaké vlastnosti a schopnosti, je také schopen interakce s jinými objekty. Všechny objekty je možné klasifikovat a třídit, například konkrétní mobilní telefon je exemplářem jisté modelové řady, kterých výrobce vyprodukoval tisíce.

Takový přístroj můžeme podle úrovně abstrakce zařadit do tříd jako mobilní telefony, telekomunikační zařízení, elektrické sbotřebiče a tak dále. Stejně tak například každá lidská bytost je exemplářem lidského druhu, jehož příslušníků jsou na světě miliardy a který můžeme zařadit mezi primáty, savce, obratlovce a živočichy.

Analogicky k tomu vytváříme při objektově orientovaném programování objekty, z nichž každý je instancí určité třídy, má nějaké atributy a metody, kterými dokáže měnit svůj stav a posílat zprávy ostatním objektům.

2.1.1 Historie

Prvním reálně použitelným objektově orientovaným programovacím jazykem byl pravděpodobně jazyk Simula 67 navržený norskými informatiky Ole-Johanem Dahlem a Kristenem Nygaardem roku (jak již název napovídá) 1967. Jedním z nejvlivnějších objektových jazyků se o několik let později stal jazyk Smalltalk (vychází z něj i jazyk PNtalk, kterému se budeme věnovat v pozdějších kapitolách), jehož vývoj začal v roce 1969.

Dnes je objektové paradigma velice populární a proniká i do dalších odvětví informatiky jako jsou databázové nebo, jak dokládá tato práce, simulační systémy. Objektově orientované jazyky jako Java, C++, C# a další jsou hojně používané, podporu objektů mají i některé skriptovací jazyky jako Python, PHP či JavaScript.

2.1.2 Třídy, objekty, dědičnost

Zakladním kamenem objektově orientovaného programování je třída (class). Třídou může být kupříkladu člověk, automobil, počítač nebo budova. Třída může být podtřídou jiné, každá třída může mít neomezené množství podtříd. Například výše uvedená třída člověk je podtřídou třídy primát, ta je podtřídou třídy savec.

Každá třída má své atributy a metody, může však použít i ty ze své nadřazené třídy. Tato vlastnost se nazývá dědičnost (inheritance). V některých jazycích je možné, aby třída dědila z více tříd (tzv. vícenásobná dědičnost, například C++, Python), v jiných nikoli (Java, C#, Smalltalk). Tímto způsobem vzniká strom tříd, přičemž všechny třídy dědí vlastnosti všech svých předků.

Objekt je instancí (konkrétním exemplářem) třídy. Mějme například člověka jménem Honza. Honza je objekt třídy člověk, a jelikož člověk je podtřídou primáta, který je podtřídou savce, můžeme prohlásit, že Honza je kromě člověka též primát a savec. Tyto nadřazené třídy ovšem mohou mít další podtřídy, které pak nepatří mezi předky třídy člověk (kupříkladu třída savec může mít podtřídu šelma, od níž Honza pochopitelně žádné vlastnosti nedědí).

Jelikož jakoukoli třídu lze dělit na konkrétnější a konkrétnější podtřídy prakticky do nekonečna, je na programátorovi, kdy rozdělit na základě určitých odlišností třídu na více tříd a kdy ji promítnout pouze jako vlastnost instancí této jedné třídy.

Metody zděděné z nadřazené třídy mohou být podtřídou přepsány. Instance podtřídy tak bude na stejné zprávy reagovat jinak než instance třídy rodičovské nebo svých sourozenců. Jelikož ale před spuštěním aplikace nemusí být zřejmé, které podtřídy daný objekt bude, není předem jasné, která varianta metody se zavolá. Tato vlastnost se nazývá polymorfismus.

2.1.3 Zapouzdření

Jedním z cílů objektové orientace je, aby objekty neměly žádné informace o vnitřní struktuře objektů ostatních. Každá třída má své rozhraní, přes které je s jejími instancemi možné komunikovat, vše ostatní je skryto a jiné objekty ani programátor, který tuto třídu použije, nemusí vůbec vědět, co se ve skutečnosti v objektu děje a jaké atributy se mění. Díky tomu nemůže dojít k situaci, kdy je narušena integrita objektu jiným objektem, a to jak úmyslně, tak v důsledku chyby.

2.2 Petriho sítě

Petriho síť je matematický modelovací jazyk sloužící k popisu distribuovaných diskrétních systémů. [7] V současné době jsou v tomto oboru de facto standardem, a to především díky své jednoduchosti a názornosti.

2.2.1 Historie a typy sítí

Petriho sítě poprvé představil v roce 1962 německý matematik Carl Adam Petri (1926–2010) jako součást své disertační práce s názvem *Kommunikation mit Automaten* na Univerzitě v Bonnu, údajně je však vymyslel již v srpnu roku 1939, ve svých 13 letech, za účelem popisu chemických procesů.

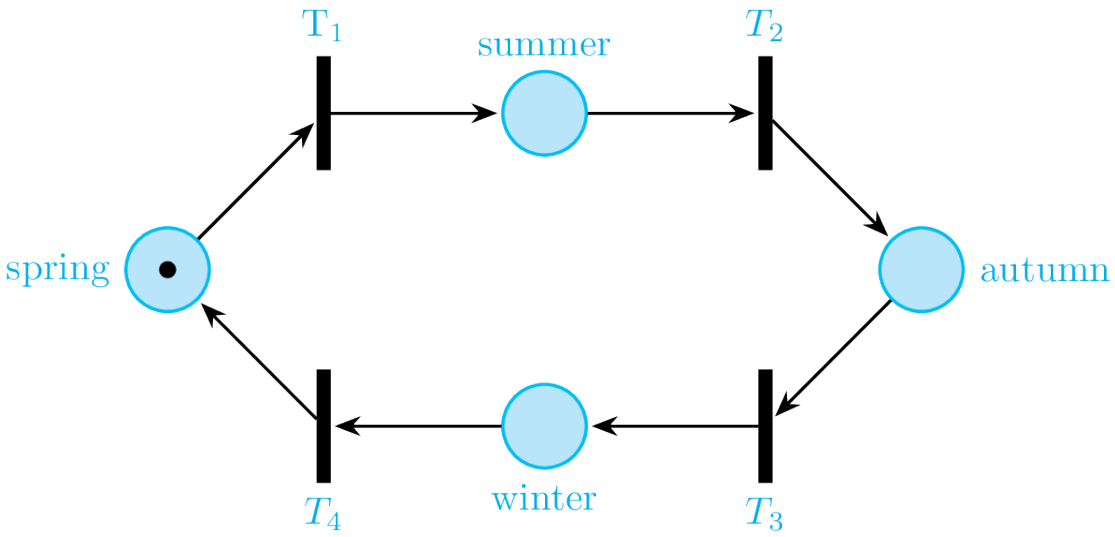
Petriho sítě prochází neustálým vývojem. Od představení původního návrhu vzniklo mnoho rozšíření, modifikací a variant. Zde jsou nejvýznamnější z nich seřazeny podle komplexnosti od těch původních až po OOPN [6]:

- **C/E Petriho sítě** – Condition/Event sítě. Podmínka (condition) drží boolovský stav (žádná nebo jedna značka) a událost (event) může tyto hodnoty měnit. Výpočetní síla tohoto typu odpovídá konečnému automatu. [2]
- **P/T Petriho sítě** – Place/Transition sítě. Oproti předchozí variantě jsou podmínky nahrazeny místy (places), která mohou obsahovat více značek. P/T sítě mohou mít mnoho různých rozšíření, jsou to například [11]:
  - **Kapacita míst** – Místům lze přidělit maximální počet značek, který mohou obsahovat. Tato změna přináší zjednodušení zápisu, ale nijak nezvyšuje výpočetní sílu.
  - **Ohodnocené hrany** – Umožňují přesouvat více tokenů provedením jediného přechodu.
  - **Testovací hrany** – Možnost existence hran (značených přerušovanou čarou), které zkontrolují přítomnost značek v místě, ale tyto značky nepřesunou. Ani tato modifikace nezvyšuje sílu sítě.
  - **Inhibiční hrany** – Inhibiční hrana směřuje výhradně od míst k přechodům a funguje opačným způsobem k ostatním hranám, tedy pokud se v místě nachází určitý počet značek, přechod není proveditelný. Tímto dostáváme výpočetní sílu Turingova stroje.
  - **Priority přechodů** – Každý přechod má svou prioritu (přirozené číslo) a pokud je proveditelných více přechodů, provede se nejdříve ten s vyšší prioritou. Síla je obdobná jako v předchozím případě.
- **Časované Petriho sítě** – Zavádí v modelu faktor času, provádění přechodů (a v některých variantách i další akce) tedy může trvat určitou dobu. Časové údaje mohou být deterministické (konstantní) nebo stochastické (vygenerované náhodně, většinou s exponenciálním rozdělením).
- **Barevné Petriho sítě** – Zavádí více typů značek. Byly vyvinuty Kurtem Jensenem v 80. letech. Přináší mnoho nových možností, především pak deklarovat proměnné a datové typy.
- **Strukturované Petriho sítě** – Umožňují hierarchické strukturování modelů. Síť je rozdělena na podsítě, které je možné vkládat jako substituce do nadřazené sítě, což výrazně zpřehledňuje celý model.
- **Objektově orientované Petriho sítě** – Spojují vlastnosti barevných a strukturovaných sítí, které vážou na objektovou orientaci. Existuje mnoho implementací. Jednou z nich, jazykem PNtalk, se budou zabývat následující kapitoly této práce.

2.2.2 Základní princip

Petriho síť je orientovaný bipartitní graf. Standardní Petriho síť obsahuje následující prvky (jejich vlastnosti se mohou lišit podle varianty):

- **Místo** – Slouží jako paměť, uchovává stavovou informaci. Značí se elipsou, může obsahovat libovolný počet značek.
- **Značka** – Někdy též token. Nachází se uvnitř míst, znázorňuje se jako tečka.
- **Přechod** – Provádí změnu stavu. Odebere daný počet značek ze vstupních míst a zároveň vytvoří určený počet značek v místech výstupních. Přechod musí být proveditelný, tedy ve vstupních místech se musí nacházet nejméně tolik značek, kolik přechod svým vykonáním odebere. Je reprezentován jako obdélník.
- **Hrana** – Propojení míst a přechodů. Jde o orientované hrany grafu, kterými lze spojit pouze místo s přechodem (proto bipartitní graf).



Obrázek 2.1: Příklad triviální Petriho sítě [5].

2.3 Jazyk PNtalk

Jazyk PNtalk je konkrétní implementací Objektově orientovaných Petriho sítí (OOPN). Jedná se o dlouhodobý projekt Fakulty informačních technologií Vysokého učení technického v Brně, jenž zahájil doc. Ing. Vladimír Janoušek, Ph.D. roku 1993 a následně jej o pět let později popsal ve své disertační práci *Modelování objektů Petriho sítěmi* [1].

Jazyk vychází ze základní definice OOPN, ale má jistá specifika založena na syntaxi jazyka Smalltalk. Konkretizuje vágní části definice OOPN a přináší syntaktická vylepšení, která umožňují snadnější zápis. Název PNtalk je zkratkou „Petri Nets & Smalltalk“.

2.3.1 Termy a zasílání zpráv

Termy jsou nejjednodušší výrazy PNtalku. Jazykově reprezentují objekty PNtalku, které nahrazují značky z klasických P/T Petriho síti. Termy mohou být:

1. **Literály** – Základní primitivní objekty. Mimo níže zmínených lze použít i další (blok a pole) z důvodu zajištění plné kompatibility s jazykem Smalltalk.
   - **Znaky** – Objekty reprezentující symboly abecedy, jsou uvozeny symbolem dolaru. Např. `$a`, `$1`.
   - **Řetězce** – Reprezentují sekvence znaků. Reagují na zprávy, které požadují přístup k jedotlivým znakům nebo porovnávají řetězec s jinými řetězci. Řetězce jsou uzavřeny v jednoduchých uvozovkách/apostrofech. Nachází-li se apostrof uvnitř řetězce, musí být zdvojený. Např. `'hello','don"t'`.
   - **Čísla** – Objekty, které reprezentují číselné hodnoty. Reagují na zprávy požadující výsledky matematických operací. Zapisují se jako prostá sekvence číslic, která může být uvozena znaménkem mínus a může obsahovat desetinnou čárku nebo písmeno e (notace s exponentem). Např. `1`, `-2`, `0.5`, `0.36e2`.
   - **Symboly** – Objekty používané (především) jako jména. Narozdíl od řetězců reprezentují stejně zapsané symboly vždy stejný objekt. Jsou uvozeny znakem hash. Např. `#name`, `#3A`.
   - **Boolovské konstanty** – Standardní binární hodnoty, jsou jim vyhrazeny identifikátory `true` a `false`.
   - **Nedefinované objekty** – Reprezentuje absenci hodnoty. Je mu vyhrazen identifikátor `nil` (obdoba hodnoty null v Javě nebo None v Pythonu).
2. **Proměnné** – Mohou v průběhu výpočtu reprezentovat různé objekty, tj. jejich hodnotu lze v programu měnit. Jsou reprezentovány jako identifikátory (sekvence alfanumerických znaků začínající malým písmenem). Např. `var`, `x1`.
3. **Pseudoproměnné** – Existují dvě, `self` (reference na objekt samotný) a `super` (reference na rodičovskou třídu). Jejich hodnoty závisí na kontextu výpočtu a nelze je programově měnit.
4. **Jména tříd** – Identifikátory s velkým počátečním písmenem. Jedná se o konstanty. Např. `Object`, `X2`.

Zpráva je výraz, jenž může být součástí stráže nebo akce přechodu (budou popsány dále). Syntaxe zaslání zprávy je *adresát zpráva*, přičemž zpráva se skládá ze selektoru a argumentů. Podle selektoru můžeme zprávy rozdělit na několik druhů:

1. **Unární zpráva** – Selektor zprávy je identifikátor, zpráva nemá žádné argumenty. Např. `Class new`, `4 sqrt`.
2. **Binární zpráva** – Selektorem je matematický nebo logický symbol, následuje jeden argument. Např. `1 + 2`, `a == b`. Matematické a logické symboly jsou v PNtalku následující:

- `+` sčítání
- `-` odčítání
- `*` násobení
- `/` dělení
- `=` rovnost
- `~=` nerovnost
- `<` menší než...
- `>` větší než...
- `<=` menší nebo rovno než...
- `>=` větší nebo rovno než...
- `&` AND (logický součin)
- `|` OR (logický součet)
- `//` celočíselné dělení
- `\\` zbytek po celočíselném dělení (modulo)
- `==` rovnost identity
- `~==` nerovnost identity
- `,` konkatenace

3. **Zpráva s klíčovými slovy** – Obsahuje nenulový počet tzv. klíčů, identifikátorů zakončených dvojtečkou. Každý z těchto klíčů má vlastní argumenty. Např. `str at: 1 put: #e`.

Jsou-li adresát i argumenty termy, jedná se o tzv. jednoduché zaslání zprávy. Jsou však přípustné i zprávy složené, tedy takové, kde jsou jako adresát či argument vnořené zprávy. Ty jsou vyhodnocovány postupně podle druhu zprávy v pořadí uvedeném výše, a to zleva doprava. Pořadí může též být ovlivněno závorkami.

Některým zprávám „rozumí“ všechny objekty PNtalku (vyplývá z definice jazyka Smalltalk). Nejdůležitějšími z nich jsou binární zprávy rovnost, rovnost identity, nerovnost a nerovnost identity.

Číselné hodnoty navíc podporují unární zprávy `negated` (opačná hodnota), `abs` (absolutní hodnota), `truncate` (zanedbání necelé části čísla) a `rounded` (zaokrouhlení), potom také (výše uvedené) binární zprávy pro početní operace a porovnávání. Pro boolovské hodnoty lze použít unární zprávu `not` (negace) a binární zprávy AND a OR.

2.3.2 Síť a její prvky

Sítě mohou být v PNtalku specifikovány graficky (anologicky k běžné P/T síti) nebo textovou formou (pro serializaci a následné strojové zpracování). Obsahují následující prvky:

- **Místo** – Reprezentováno (stejně jako v klasické P/T síti) elipsou nebo kružnicí. Může mít specifikováno počáteční značení, které syntakticky odpovídá hranovému výrazu (bude definován v popisu hran). Lze zde použít proměnné, přičemž jejich hodnoty jsou přiřazeny v počáteční akci místa. Ta je syntakticky totožná s akcí přechodu. Počáteční značení je vždy podtrženo.

- **Přechod** – Je reprezentován čtyřúhelníkem a může mít specifikovánu stráž a akci. Stráž je tvořena sekvencí výrazů, které jsou odděleny tečkou a každý z nich je zprávou (např. `o state: x. x >= 50`). Tyto výrazy jsou vzájemně v konjunkci, tedy stráž je platná, jestliže platí všechny její dílčí výrazy. Akce může (narozdíl od stráže) obsahovat výraz přiřazení, který má tvar `<proměnná> := <zpráva>`, např. `z := x + y`. Stráž přechodu je vždy podtržena.

- **Hrana** – Je reprezentována šipkou, která spojuje místo s přechodem. Každé hraně přísluší hranový výraz. Tyto výrazy reprezentují multimnožiny a mají tvar

$\mathtt{n_1}'\mathtt{c_1}, \mathtt{n_2}'\mathtt{c_2}, ..., \mathtt{n_m}'\mathtt{c_m}$, kde n_i je term a c_i je term nebo seznam (uspořádaná ntice, jejíž každý prvek je term nebo další seznam, je přípustná i prologovská notace zápisu $(\mathtt{h_1}, \mathtt{h_2}, ..., \mathtt{h_n}|\mathtt{t})$). Sémanticky n_i udává počet výskytů c_i (Je-li n_i 1, lze jej vynechat). Hranový výraz může být například $2'\#\mathtt{e}, \mathtt{x}'\mathtt{y}, 5'(\mathtt{x}, \mathtt{y}, \$\mathtt{a}), 6'7$. Jako běžnou, „bezbarvou“ značku můžeme použít symbol #e nebo `nil`.

- **Vstupní** – Směřuje z místa do přechodu, reprezentuje vstupní podmínku přechodu, jež specifikuje značky, které budou po provedení přechodu z místa odebrány.
- **Výstupní** – Směřuje z přechodu do místa, reprezentuje výstupní podmínku, která specifikuje značky, které přechod do cílového místa umístí.
- **Testovací** – Je obousměrná. Reprezentuje podmínky přechodu, které pouze testují přítomnost značek v daném místě.

Místa a přechody mají svá jména. Ta jsou platná pouze v síti, v níž jsou definována. Výjimkou jsou místa objektové sítě, jež jsou viditelná i pro přechody v sítích metod. Nejsou-li jména explicitně specifikována, jsou vygenerována automaticky.

2.3.3 Třídy a druhy sítí

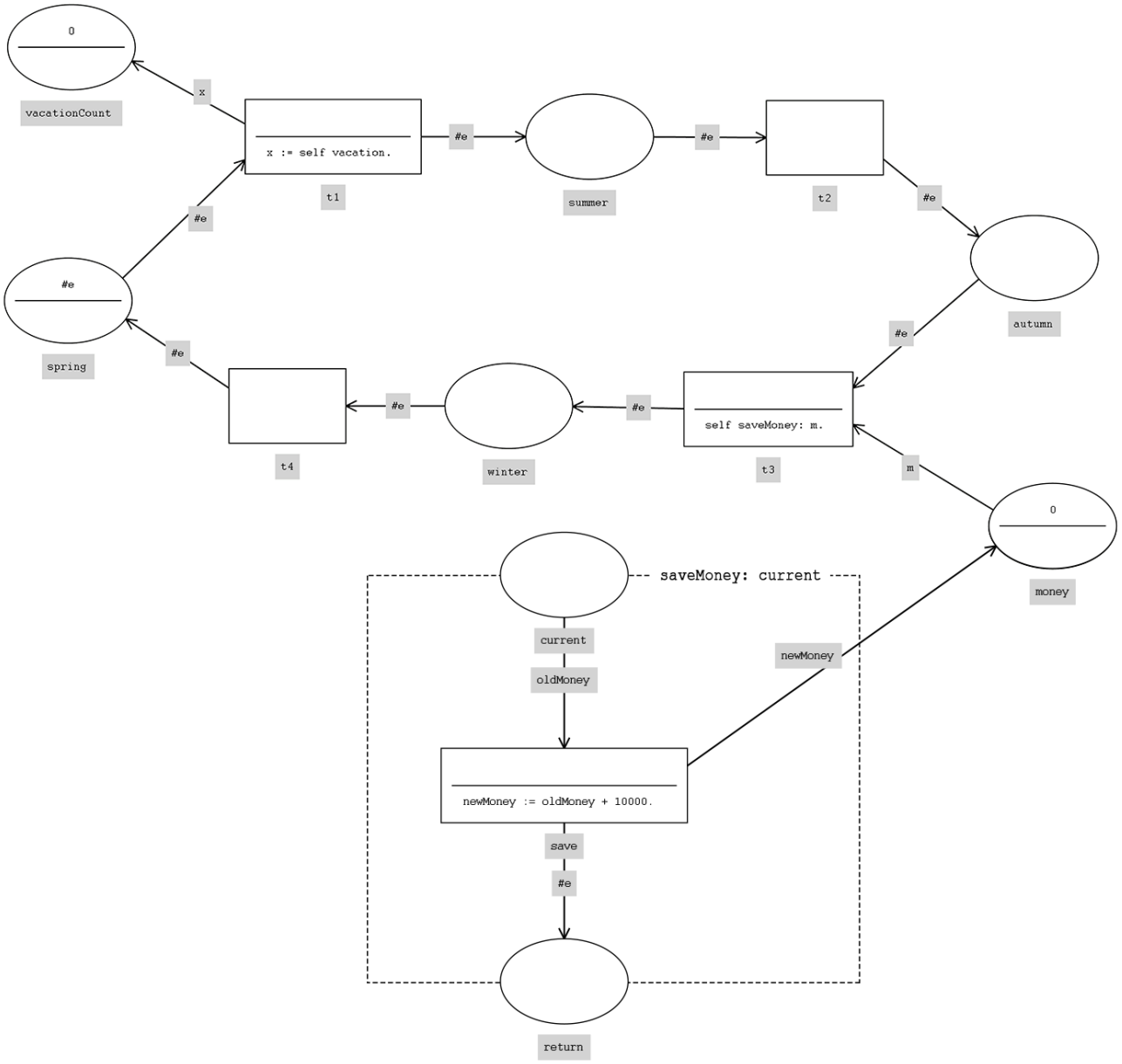
Jak bylo naznačeno výše, v PNtalku existují dva druhy sítí:

- **Síť objektu** – Reprezentuje stav samotného objektu (místa jsou zde atributy objektu) a jeho implicitní aktivitu. Každá třída má právě jednu objektovou síť.
- **Síť metody** – Reprezentuje reakci objektu na příchozí zprávu. Každá metoda má svou odpovídající síť. Obsahuje vzor zprávy, což je předpis, jehož přijetí objektem vytvoří instanci příslušné sítě. Skládá se ze selektoru zprávy a jejích parametrů. Některé z míst sítě mohou být tzv. parametrovými místy, která slouží k předání parametrů při volání metody. Jména těchto míst odpovídají jménům parametrů. Dále každá síť metody obsahuje právě jedno výstupní místo (s názvem `return`), kterým se předává výsledek metody. Sítě metod jsou s objektovou sítí propojeny prostřednictvím hran mezi místy objektové sítě a přechody metod. Metody tak mohou přistupovat ke stavu objektu.

  Speciálním případem sítě metody je konstruktor. Od ostatních metodových sítí se liší klíčovým slovem `constructor` v jejich specifikaci a tím, že vždy vrací hodnotu `self` (jméno vytvořeného objektu). Pokud třída konstruktor neobsahuje, volá se při vytváření objektu tzv. „implicitní konstruktor“, který vždy reaguje na zprávu `new` a který automaticky nastaví počáteční značení objektové sítě.

Každý model v PNtalku se skládá z množiny tříd, z nichž jedna je označena jako počáteční (main) třída. Ta je automaticky načtena při spuštění modelu. PNtalk (stejně jako Smalltalk) nepodporuje vícenásobnou dědičnost.

Vrcholem hierarchie tříd je třída `PN` (třída s prázdnou objektovou sítí), která je přímým potomkem třídy `Object`, která je (podobně jako v jiných jazycích) vrcholem stromu dědičnosti. Jejími následníky jsou též primitivní objekty PNtalku (konstanty).



Obrázek 2.2: Příklad objektové sítě třídy s jednou její metodou.

2.3.4 Synchronní porty

Synchronní porty slouží k testování a případné změně stavu objektu. Sémanticky se nacházejí na pomezí metody a přechodu. Narozdíl od sítí neobsahují místa ani přechody, zato mohou (jako přechody) obsahovat stráž a být propojeny hranami s místy objektové sítě. Oproti přechodu ale obsahuje vzor zprávy, na kterou reaguje.

Synchronní porty jsou volány ze stráže libovolného přechodu. Podobně jako přechod mohou být proveditelné nebo neproveditelné a lze je volat s předem vyhodnocenými parametry, ale i volnými proměnnými. Jsou reprezentovány (stejně jako přechody) čtyřúhelníkem, stráž však nemusí být podrtržena.

Speciálním případem synchronního portu je predikát. Ten je se všemi objektovými místy propojen testovacími hranami a tedy neovlivňuje stav objektu.

2.3.5 Textová podoba PNtalku

Mimo grafické má jazyk PNtalk i svou textovou formu. Ta se používá pro strojové zpracování vytvořených modelů a obvykle je generována automaticky nástrojem pro vytváření modelů (nástroj, o němž tato práce pojednává, není výjimkou). Autor jazyka, doc. Ing. Vladimír Janoušek, Ph.D., jej ve své disertační práci formálně definoval rozšířenou Backus-Naurovou formou.

Rozšířená Backusova–Naurova forma (ABNF) je metajazyk (jazyk sloužící k popisu jiných jazyků), který se obvykle používá pro definici komunikačních protokolů. Oproti standardní Backusově–Naurově formě má mírně odlišnou syntaxi. [8] V definici PNtalku jsou použity operátory ABNF pro nepovinný výskyt (`[ ... ]`), libovolný počet výskytů (`[ ... ]*`), výskyt alespoň jednou (`[ ... ]+`) a výběr z variant (`... | ...`).

```
classes: [class]* "main" id [class]*
class: "class" classhead [objectnet] [methodnet|constructor|sync]*
classhead: id "is_a" id

objectnet: "object" net
methodnet: "method" message net
constructor: "constructor" message net
sync: "sync" message [cond] [precond] [guard] [postcond]
message: id | binsel id | [keysel id]+
net: [place|transition]*

place: "place" id"(" [initmarking] ")" [init]
init: "init" "{"initaction "}"
initmarking: multiset
initaction: [temps] exprs

transition: "trans" id [cond] [precond] [guard] [action] [postcond]
cond: "cond" id"(" arcexpr ")" ["," id"(" arcexpr ")"]*
precond: "precond" id"(" arcexpr ")" ["," id"(" arcexpr ")"]*
postcond: "postcond" id"(" arcexpr ")" ["," id"(" arcexpr ")"]*
guard: "guard" "main{" expr3 "}"
action: "action" "{" [temps] exprs "}"
arcexpr: multiset

multiset: [n "‘"] c ["," [n "‘"] c]*
n: [dig]+ | id
c: literal | id | list
```

```
list: "("[c [","  c]* ["|" [id | list] ]] ")"

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

Výpis 2.1: Jazyk PNtalk definovaný pomocí ABNF doc Janouškem (počáteční neterminál je „classes“). [1]

```
main C1

class C1 is_a PN
 object
  place summer()
  place autumn()
  place winter()
  place spring(#e)
  place vacationCount(0)
  place money(0)
  trans t1 precond spring(#e) action {x := self vacation.}
   postcond summer(#e), vacationCount(x)
  trans t2 precond summer(#e) postcond autumn(#e) trans t3 precond
   autumn(#e), money(m) action {self saveMoney: m.} postcond winter(#e)
  trans t4 precond winter(#e) postcond spring(#e)
 method vacation
  place return()
  trans go precond vacationCount(oldCount), money(oldMoney)
   guard {oldMoney >= 5000.} action {newCount := oldCount + 1.
   newMoney := oldMoney - 5000.}postcond return(newCount), money(newMoney)
  trans wait cond money(oldMoney) precond vacationCount(oldCount)
   guard {oldMoney < 5000.} action {newCount := oldCount.}
   postcond return(newCount)
 method saveMoney: current   place return()
  place current()
  trans save precond current(oldMoney) action {newMoney := oldMoney + 10000.}
   postcond money(newMoney), return(#e)
```

Výpis 2.2: Příklad jednoduchého modelu v čistém jazyce PNtalk (odpovídá obrázku 2.2).

Kapitola 3

Analýza požadavků a návrh aplikace

Cílem projektu bylo implementovat grafický editor Objektově orientovaných Petriho sítí, a to v jazyce Java. Nástroj měl umožňovat správu tříd a dědičnosti, grafickou editaci jednotlivých sítí, ukládání modelů (a to jak v nativním formátu editoru, tak v čistém jazyce PNtalk a ve formě vektorové grafiky) a jejich opětovné načítání, úpravu výrazů (akce, stráže, hranové výrazy) přímo v grafických prvcích sítě, otevírání více sítí zároveň prostřednictvím panelů, spuštění simulace a její krokování na vzdáleném simulačním serveru, stahování modelů ze vzdáleného serveru a jejich nahrávání na něj a zobrazení aktuálního stavu simulace.

3.1 Uživatelské rozhraní

Design grafického uživatelského rozhraní navazuje na ten použitý v práci Antonína Neužila a přináší několik vylepšení, jako například:

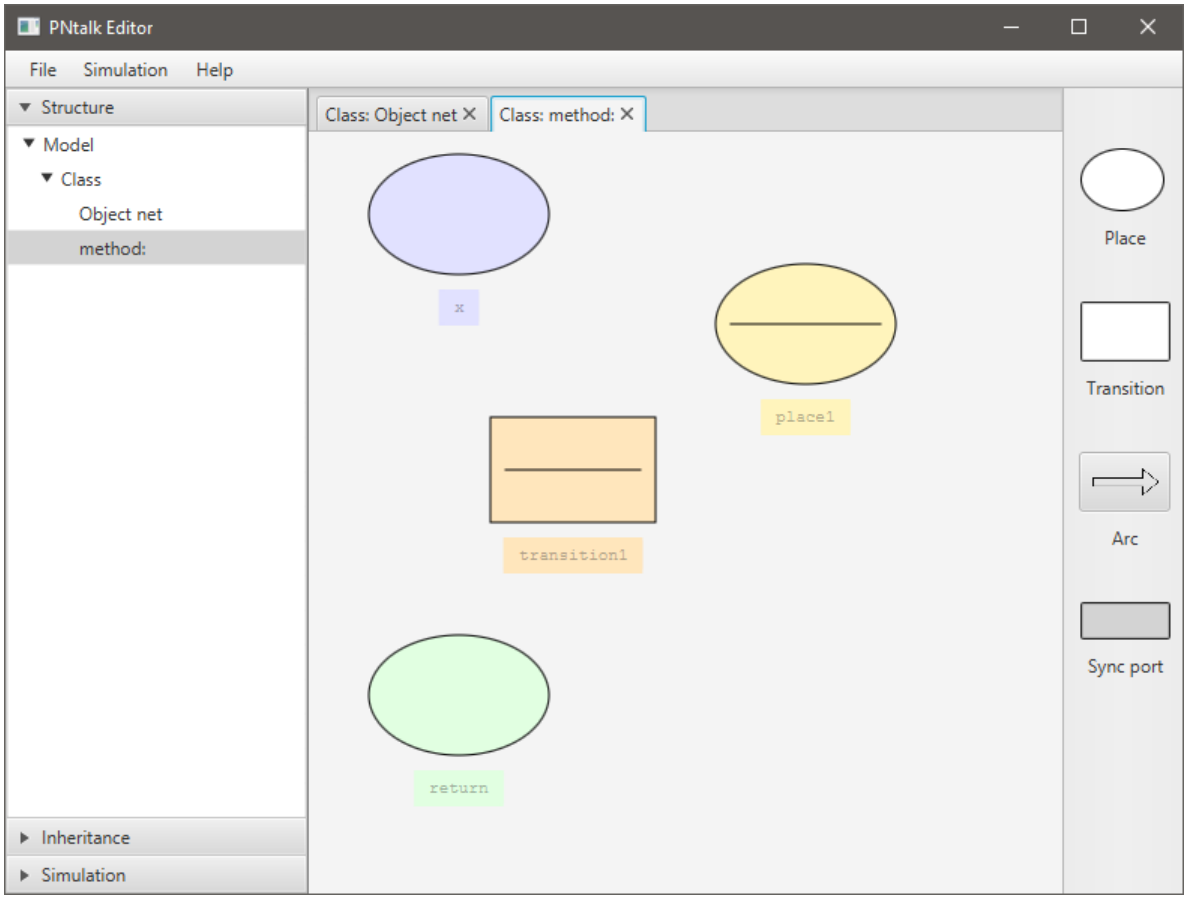
- Položky kontextové nabídky v levém podokně se zobrazují jen u objektů, ke kterým se vztahují.
- Vlastnosti prvků Petriho sítí (například akce, stráže, hranové výrazy a další) lze editovat přímo na místě.
- Šířka prvku se automaticky nastavuje podle délky textu, jenž se v něm nachází.
- Objektová síť se vytváří automaticky při vytvoření třídy.
- Při posouvání prvku sítě jsou s ním plynule posouvány i připojené hrany (a příslušné hranové výrazy).

Grafický layout je definován v souboru `opne-view.fxml` v adresáři `resources`. Je rozdělen na 4 části:

- `top` – Hlavní nabídka okna. Obsahuje element `MenuBar`, který definuje nabídky v záhlaví okna aplikace. Každá z nich má přiřazenu metodu v JavaFX kontroleru, která se vykoná při kliknutí na ni nebo provedení příslušné klávesové zkratky.
- `left` – Levé podokno se strukturou modelu, stromem dědičnosti a nabídkou simulací. Je implementováno jako JavaFX prvek `Accordion`, jednotlivé jeho části potom jako `TreeView`.

- `right` – Podokno obsahující jednotlivé komponenty, které je možné přidat do otevřené Petriho sítě. Je implementováno pomocí prvku `Toolbar`, ve kterém se jako položky nacházejí tvary a tlačítka.
- `center` – Hlavní podokno celé aplikace, v němž se v panelech otevírají sítě pro editaci nebo simulování. Jedná se o `TabPane`.

Definice vzhledu některých prvků layoutu je doplněna stylopisem v souboru `style.css` v adresáři `resources`.



Obrázek 3.1: Uživatelské rozhraní aplikace.

3.2 Architektura aplikace

Protože sítě, které uživatel edituje, a sítě, které pouze zobrazují stav simulace, mají velmi odlišné vlastnosti, byla struktura OOPN rozdělena na tři separátní části.

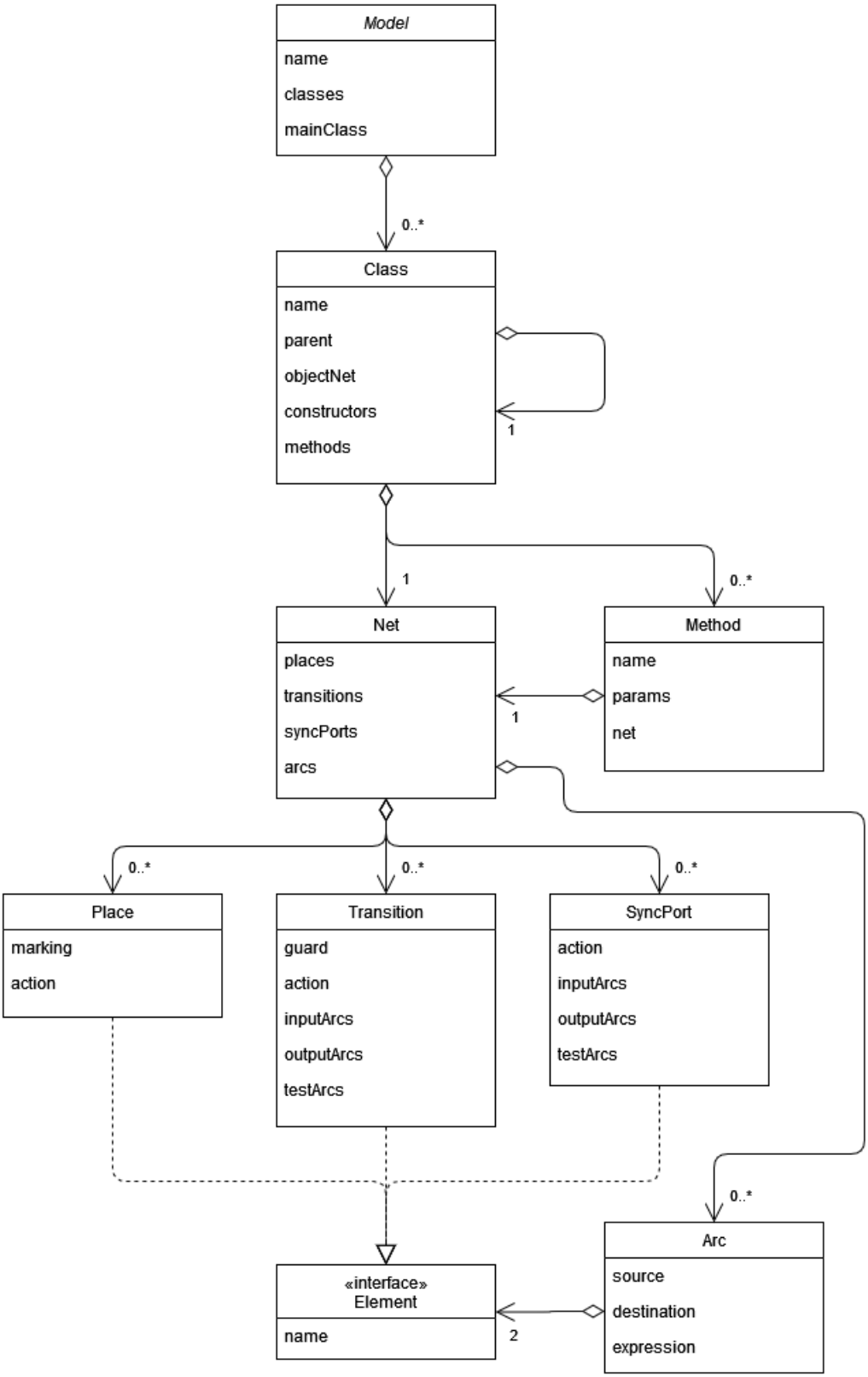
3.2.1 Prvky Objektově orientovaných Petriho sítí

Srdcem aplikace jsou třídy pro jednotlivé části OOPN modelu. Tato objektová struktura je zcela nezávislá na její grafické reprezentaci, což usnadňuje její ukládání v čistém jazyce PNtalk a odesílání na simulační server.

Jedná se o třídy `Model`, `Class`, `Method`, `Net`, `Place`, `Transition`, `SyncPort` a `Arc`. Třídy

`Place`, `Transition` a `SyncPort` všechny implementují rozhraní `Element`. Instance těchto tříd tvoří stromovou strukturu modelu (kde model obsahuje všechny třídy ve formě seznamu referencí na ně, třída potom všechny své metody a tak dále), která je za běhu aplikace uložena tzv. „in-memory“, tedy v operační paměti počítače.

Tuto strukturu je možné serializovat a uložit jako textový soubor na persistentní úložiště nebo ji odeslat na vzdálený server k odsimulování.



Obrázek 3.2: Diagram tříd pro reprezentaci OOPN.

3.2.2 Grafické reprezentace OOPN

Výše zmíněnou strukturu Objektově orientovaných Petriho sítí je v aplikaci třeba zobrazit. Jelikož má vyvíjený nástroj být schopen OOPN jak editovat, tak simulovat, a jelikož editor a simulátor mají jiné ovládací prvky a možnosti manipulace s objekty, bylo třeba vytvořit oddělenou reprezentaci editovatelných a simulovaných sítí.

Každý grafický prvek musí obsahovat reference na samotné objekty JavaFX a stavové proměnné nutné pro fungování programu. Struktura editoru kopíruje holou strukturu OOPN, navíc zde najdeme rozhraní `RectangleElementView` (implementují jej přechody a sychronní porty) a `StructureItem` (implementují jej všechny části OOPN, které se mohou nacházet v panelu struktury).

V rámci simulátoru není třeba práce s celou strukturou OOPN, ale pouze sítěmi, místy, přechody, synchronními porty a hranami. Jejich základní princip je obdobný jako v případě editoru, chybí zde ale veškeré funkce zajišťující možnosti úpravy sítí. Navíc mají některé objekty své ID přidělené simulačním serverem.

3.3 Ukládání a načítání modelů

Za účelem ukládání vytvořených modelů a jejich opětovného načítání byl v rámci této práce navržen vlastní nativní formát založený na jazyce XML. XML (Extensible Markup Language, v překladu rozšiřitelný značkovací jazyk) je jazyk sloužící k serializaci objektů, který je čitelný jak strojem, tak člověkem. Jeho hlavními přednostmi jsou přehlednost a snadná rozšiřitelnost.

Pro načítání i ukládání je použito DOM (Document Object Model) API. Při tomto přístupu je po dobu manipulace celý strom elementů uchováván v paměti a lze tak přidávat elementy a jejich atributy zcela volně. Přístupem opačným je tzv. SAX (Simple API for XML) API, které zpracovává (nebo zapisuje) jednotlivé elementy postupně.

3.3.1 Struktura formátu

Kořenovým elementem uložených modelů je `model`, který má povinný atribut `name` (název modelu) a obsahuje libovolný nenulový počet elementů `class`. Ty mimo povinného jména obsahují volitelný boolovský atribut `main`, který musí mít hodnotu `true` právě u jedné ze tříd, a rovněž volitelný atribut `parent`, jenž specifikuje rodičovskou třídu (v případě nepřítomnosti atributu se předpokládá dědičnost od kořenové třídy PN).

V elementu `class` je vždy zanořen právě jeden element `net`, který reprezentuje síť objektu, a libovolný počet elementů `method`, z nichž každý obsahuje vlastní element `net`. Metody obsahují tyto atributy:

- `name` – Jméno metody.
- `x`, `y` – Pozice návratového místa metody.
- `constructor` – Volitelný boolovský parametr, není-li uveden, je výchozí hodnota `false`. Pokud je nastaven na `true`, je metoda konstruktorem, tedy její návratová hodnota musí být vždy `self`.

Kromě elementu `net`, který reprezentuje síť metody, zahrnuje metoda libovolný počet elementů `param` specifikujících její parametry. Ty obsahují atributy `name`, jméno parametru (a zároveň parametrového místa), a `x` a `y`, což jsou souřadnice parametrového místa.

Síť samotná může obsahovat následující elementy:

- `place` – Reprezentuje místo, má atributy `name` (jméno místa), `x` a `y` (souřadnice místa), `marking` (počáteční značení) a `action` (akce). Poslední dva zmíněné jsou pouze volitelné.
- `transition` – Reprezentuje přechod, stejné atributy jako v případě místa (s výjimkou `guard` namísto `marking`).
- `sync` – Synchronní port, obdobné atributy jako u přechodu, nemůže ovšem obsahovat akci.
- `arc` – Hrana mezi místem a přechodem (případně synchronním portem). Jejími atributy jsou `src` (výchozí prvek), `dest` (cílový prvek), `exp` (hranový výraz) a `test` (volitelný boolovský atribut, značí, že je daná hrana testovací, výchozí hodnota je `false`).
- `link` – Odkaz na místo v objektové síti. Má atribut `to`, což je jméno objektového místa, na nějž je odkazováno, a dále (stejně jako v případě běžného místa) souřadnice.

```
<?xml version="1.0" encoding="UTF-8"?>

<model name="example">

 <class main="true" name="C1">
  <net>
   <place name="summer" x="600.0" y="180.0"/>
   <place name="autumn" x="1018.0" y="294.0"/>
   <place name="winter" x="524.0" y="433.0"/>
   <place marking="#e" name="spring" x="110.0" y="334.0"/>
   <place marking="0" name="vacationCount" x="113.0" y="96.0"/>
   <place marking="0" name="money" x="1035.0" y="546.0"/>
   <transition action="x := self vacation." name="t1" x="359.0" y="180.0"/>
   <transition name="t2"x="821.0" y="181.0"/>
   <transition action="self saveMoney: m." name="t3" x="768.0" y="436.0"/>
   <transition name="t4" x="316.0" y="433.0"/>
   <arc dest="t1" exp="#e" src="spring"/>
   <arc dest="summer" exp="#e" src="t1"/>
   <arc dest="t2" exp="#e" src="summer"/>
   <arc dest="autumn" exp="#e" src="t2"/>
   <arc dest="t3" exp="#e" src="autumn"/>
   <arc dest="winter" exp="#e" src="t3"/>
   <arc dest="t4" exp="#e" src="winter"/>
   <arc dest="spring" exp="#e" src="t4"/>
   <arc dest="vacationCount" exp="x" src="t1"/>
   <arc dest="t3" exp="m" src="money"/>
```

```
  </net>
  <method name="vacation"x="99.0"y="522.0»
   <net>
    <transition action="newCount := oldCount + 1. newMoney := oldMoney - 5000."
      guard="oldMoney >= 5000." name="go" x="492.0" y="226.0"/>
    <transition action="newCount := oldCount."
      guard="oldMoney < 5000." name="wait" x="452.0" y="464.0"/>
    <link to="vacationCount" x="120.0" y="142.0"/>
    <link to="money" x="922.0" y="362.0"/>
    <arc dest="go" exp="oldCount" src="vacationCount"/>
    <arc dest="return" exp="newCount" src="go"/>
    <arc dest="money" exp="newMoney" src="go"/>
    <arc dest="go" exp="oldMoney" src="money"
      x="831.7838751807902" y="191.28480536746372"/>
    <arc dest="wait" exp="oldMoney" src="money" test="true"/>
    <arc dest="wait" exp="oldCount" src="vacationCount" x="118.0" y="334.5"/>
    <arc dest="return" exp="newCount" src="wait"/>
   </net>
  </method>
  <method name="saveMoney:" x="334.0" y="494.0">
   <param name="current" x="334.0" y="112.0"/>
   <net>
    <transition action="newMoney := oldMoney + 10000." name="save"
      x="334.0" y="310.0"/>
    <link to="money" x="793.0" y="66.0"/>
    <arc dest="save" exp="oldMoney" src="current"/>
    <arc dest="money" exp="newMoney" src="save"/>
    <arc dest="return" exp="#e" src="save"/>
   </net>
  </method>
 </class>

</model>
```

Výpis 3.1: Příklad jednoduchého modelu s jednou třídou
(odpovídá obrázku 2.2)

3.4 Simulace modelů

Aplikace umožňuje simulovat vytvořený model na vzdáleném simulačním serveru, zobrazit stav probíhající simulace a také probíhající simulaci krokovat.

3.4.1 Simulační server

Server, se kterým aplikace komunikuje, implementoval vedoucí práce, Ing. Radek Kočí, Ph.D. v prostředí Pharo. Pharo je objektově orientovaný, dynamicky typovaný jazyk založený na syntaxi jazyka Smalltalk. [4] Tento open-source projekt vznikl roku 2008 oddělením od jazyka Squeak. [9]

3.4.2 Komunikační protokol

Běžící server naslouchá standardně na portu 9999, je ovšem možné zvolit jiný port. Příkazy příjmá ve formě prostého textu, v příkazové řádce k němu lze přistoupit příkazem Telnet:

```
telnet localhost 9999
```

Takto otevřený kanál očekává jako vstup právě jeden řádek s příkazem, odpovědí může být buď text „OK“ a příslušný výstup, nebo text „Failure“, následovaný příčinou chyby. Použité příkazy jsou následující:

```
(addClass "class_definition") => ()
```

Přidá více nebo jednu nebo více tříd do databáze. `class_definition` je definice třídy nebo celého modelu (bez stanovení počáteční třídy) v čistém jazyce PNtalk.

```
(delClass "name") => ()
```

Odstraní třídu s názvem `name` z databáze.

```
(getClass "name") => (class_definition)
```

Vrátí zdrojový kód třídy `name` v čistém jazyce PNtalk.

```
(newSim "main_class_name") => (simulation_id)
```

Inicializuje novou simulaci s třídou `main_class_name` jako počáteční třídou. Vrátí ID (pořadové číslo) inicializované simulace.

```
(getState "simulation_id") => (state)
```

Vrátí aktuální stav simulace `simulation_id` ve standardizovaném formátu, který bude specifikován níže.

```
(stepSim "simulation_id") => ()
```

Provede jeden krok simulace `simulation_id`.

```
(destroySim "simulation_id") => ()
```

Zruší simulaci `simulation_id`.

3.4.3 Struktura stavu simulace

Stav simulace, který vrací příkaz `getState` má vlastní formát navržený vedoucím práce. Spacifikace byla převzata z materiálů poskytnutých autorem. Pro jeho popis budeme používat následující symboly:

- **N:** číslo, např. 10
- **S:** řetězec uvozený apostrofy, např. ‘a b c’

- **C:** symbol, např. #go
- **T:** typ prvku v místě:
  - `#obj -` jednoduchý objekt (čislo, řetězec, symbol)
  - `#ref -` reference na jiný objekt
- **V:** hodnota prvku v místě (pokud T == #obj, jde přímo o hodnotu reprezentující objekt, pokud T == #ref, jde o ID objektu, na nějž referujeme)

```
(
 ( seznam rozpracovaných zpráv
  ( zpráva
   N id zprávy
   C identifikátor metody
   N id zasílajícího objektu (sender)
   N id přijímajícího objektu (receiver)
   C typ zprávy
   ( ... seznam parametrů/hodnot ... )
  )
  ( zpráva ... )
 )

 seznam objektů
 ( objekt

  S název třídy

  N id objektu

  ( seznam procesů (objektová síť + invokované metody)
   ( proces
    S název metody nebo 'object' pro objektovou síť
    N id procesu
    ( seznam míst
     ( místo
      S název místa
      ( obsah místa, jednotlivé prvky jsou odděleny mezerou
       N->T->V prvek místa, N je četnost prvku v místě
      )
     )
    )
    ( seznam spuštených přechodů (vláken)
     ( vlákno
      S název přechodu
      N identifikátor vlákna
      N index prováděné akce v přechodu
      ( seznam proměnných a navázaných hodnot,
```

```
        jednotlivé prvky jsou odděleny mezerou
       C->T->V C = proměnná, T = typ, V = hodnota
      )
      ( obsah zásobníku mezivýsledku při vyhodnocování
      )
     )
    )
   )
  )

  ( seznam závislostí (zprávy zaslané z tohoto objektu)
   ( závislost
    N id zprávy
    N id procesu, ze kterého byla zpráva zaslána
    S název přechodu, ze kterého byla zpráva zaslána
    N id vlákna, ze kterého byla zpráva zaslána
   )
  )

  ( seznam vyvolaných procesů v reakci na zprávu, oddělené mezerou
   N->N id zprávy -> id procesu
   N->N id zprávy -> id procesu
   ...
  )

 )

 ( ... další objekt ... )
...
)
```

Výpis 3.2: Struktura výpisu stavu simulace.

```
(((8439398 #vacation 1 1 #domain ())) (‘C1’ 1 ((‘object’ 1 ((‘vacationCount’
(1->#obj->0)) (‘money’ (1->#obj->0)) (‘summer’ ()) (‘spring’ ()) (‘winter’ ())
(‘autumn’ ())) ((‘t1’ 11 1 (#self->#ref->1) ()))) (‘vacation’ 2 ((‘return’ ()))
())) ((8439398 1 ‘t1’ 11)) (8439398->2)))
```

Výpis 3.3: Příklad stavu zahrnujícího objektovou síť a jednu metodu. Objekt s ID 1 právě zaslal sám sobě zprávu, která invokovala metodu „vacation“.

3.5 Formát SVG

Scalable Vector Graphics (SVG), v překladu „škálovatelná vektorová grafika“, je formát pro ukládání vektorové grafiky založený na XML. [10]

3.5.1 Vektorová grafika

Vektorová grafika je jedním ze dvou hlavních typů počítačové grafiky. Zatímco v případě grafiky bitmapové (někdy též rastrové) se obrázek skládá z mnoha barevných bodů (tzv. „pixelů“, z anglického sousloví „picture element“), které z dostatečné vzdálenosti vytvářejí iluzi souvislého obrazu, vektorový obrázek je tvořen geometrickými tvary a obrazci, jež jsou matimaticky definovány.

Díky tomu je vektorový obrázek vždy stejně kvalitní, bez ohledu na velikost zobrazení a přiblížení. Jejich zásadní nevýhodou však je, že některé formy grafiky (nepříklad fotografie) jimi nelze reprezentovat. Jejich vykreslení je také výpočetně složitější. Jsou však neocenitelné pro ukládání jednoduchých tvarů a textu, což je i účel použití v tomto projektu. Popis SVG se navíc v mnoha ohledech podobá jazyku FXML, v němž je popsán layout aplikace.

3.5.2 Struktura SVG

SVG obrázek je textový soubor v jazyce XML. Kořenovým elementem je tag `svg`, mezi jehož atributy jsou i rozměry obrázku. Následně můžeme použít tagy reprezentující grafické obrazce jako `rect`, `circle` nebo `line`. Tyto tvary lze seskupovat pomocí tagu `g` a poté s celou skupinou operovat jako s jedním objektem. Každý tvar má svou pozici na kreslícím plátně, barvu a rozměry. Může mít i mnoho dalších parametrů.

Kapitola 4

Implementace

4.1 Jazyk Java

Java je objektově orientovaný programovací jazyk navržený v roce 1995 kanadským informatikem Jamesem Goslingem. Jeho syntaxe vychází z jazyka C. Jedná se o jeden z nejpopulárnějších programovacích jazyků na světě. Původním vývojářem jazyka byla společnost Sun Microsystems, ta ale byla roku 2010 odkoupena korporací Oracle, jež tak vývoj Javy převzala.

Java je tzv. „semi-interpreted“ jazyk, nepřekládá se přímo do strojového kódu, ale do Java byte code, což je přenositelný kód, de facto mezistupeň, který je následně interpretován pomocí virtuálního stroje. Nevýhodou tohoto přístupu je o něco nižší rychlost, zásadní výhodou je ovšem možnost spouštět programy na jakémkoli zařízení s nainstalovaným Java Runtime Environment bez nutnosti překladu pro každou platformu zvlášť.

Tento projekt byl vyvinut pomocí Java Development Kitu ve verzi 17, která vyšla v září 2021.

4.1.1 Třídy a rozhraní

Java je tzv. „class-based“ (třídní) jazyk, tedy jazyk, v němž dědičnost probíhá plnohodnotně přes třídy objektů (tak, jak bylo představeno ve druhé kapitole), nikoli objekty samotné (např. prototypování v jazyce Self nebo JavaScriptu). Navíc je striktně třídní, tedy žádný kód nemůže být napsán mimo definici nějaké třídy. Tyto definice se nacházejí ve zdrojových souborech s příponou `.java`.

Java nepodporuje vícenásobnou dědičnost, každá třída má tedy právě jednoho rodiče. Není-li nadřazená třída specifikována (klíčovým slovem `extends`), je otcovskou třídou implicitně třída `Object`, jež je v Javě kořenem stromu dědičnosti. Podtřída následně automaticky může využívat všechny atributy a metody nadřazené třídy.

Mimo klasické dědičnosti existují v jazyce Java také rozhraní (interfaces), která definují metody, jež musí obsahovat každá třída, která rozhraní implementuje. Rozhraní samo ale nikdy implementaci neobsahuje. Každá třída může implementovat libovolné množství rozhraní. Na pomezí mezi rozhraními a klasickými třídami stojí abstraktní třídy, které nelze instanciovat a které mohou obsahovat jak implementované, tak abstraktní metody (metody bez těla).

Atributy a metody mohou být instanční (inicializují se při vytváření objektu a každá instance třídy má vlastní hodnotu) nebo třídní (nesouvisejí s konkrétními objekty, jsou spjaty s třídou samotnou a volají se přes její jméno), jež se v Javě nazývají statické (a deklarují se klíčovým slovem `static`).

Atributy, metody i třídy samotné mohou být deklarovány s tzv. „modifikátory přístupu“ (access modifiers), které umožňují upravovat, odkud jsou tyto prvky přístupné. Existují modifikátory `private` (atribut nebo metoda jsou dostupné pouze v jejich vlastní třídě), `public` (dostupnost odkudkoli, pokud je takto deklarována třída, měla by se nacházet ve vlastním, stejnojmenném zdrojovém souboru) a `protected` (atributy a metody jsou dostupné v jejich třídě a všech podtřídách). Pokud není modifikátor specifikován, třídy, atributy a metody jsou viditelné v rámci jejich balíčku (package), do kterého jsou umísteny příkazem `package` na začátku zdrojového souboru.

4.1.2 Datové typy a kolekce

Java podporuje dva druhy datových typů:

- **Neprimitivní** – Deklarovaná proměnná je instancí specifikované třídy, jež je zároveň jejím typem. Typicky se vytváří klíčovým slovem `new`, existují však výjimky, kdy jeho použití není třeba (například řetězce nebo číselné typy). Jsou předávány referencí. Např. `Object o = new Object(); String text = "Sample text";`.
- **Primitivní** – Nejedná se o objekty, ale prosté hodnoty. Tyto typy tedy nemají žádné atributy ani metody a je jich celkem 8, jmenovitě boolean, byte, short, int, long, float, double a char. Jsou předávány hodnotou. Např. `int number = 10; boolean done = false;`. Všechny tyto typy mají i svůj neprimitivní ekvivalent (`Integer number = 10;`).

Významnou součástí standardní knihovny jsou kolekce. Jsou to předprogramované abstraktní datové struktury, konkrétně jde o rozhraní `List` (seznamy, tedy uspořádané ntice) a `Set` (množiny), které obě dědí z obecného rozhraní `Collection`. Samostatně potom stojí rozhraní `Map` (množina párů klíč-hodnota). Všechna tato rozhraní mají vícero konkrétních implementací (třídy jako `ArrayList`, `TreeList` nebo `HashMap`.)

4.1.3 Překlad a spouštění aplikací

Pro vývoj softwaru v jazyce Java je nutná vývojářská sada (Java Development Kit, JDK), která obsahuje překladač jazyka do byte code a další nástroje. Ke spuštění přeložených aplikací je následně třeba prostředí Java Runtime Environment (JRE), které je interpretuje pomocí virtuálnho počítače (Java Virtual Machine, JVM).

Hotové aplikace lze pro snadnjší spouštění zabalit do archivu (soubor s příponou `.jar`), jehož součástí je tzv. „manifest file“, v němž specifikujeme počáteční třídu. Následně lze takovýto archiv spustit jako běžný spustitelný soubor v jakémkoli operačním systému (za podmínky nainstalovaného JRE).

K usnadnění překladu aplikací, které využívají mnoho externích knihoven, se používají tzv. „build tools“, nástroje pro automatickou konfiguraci překladače a správu závislostí. Nejvýznamnější z nich jsou Apache Maven (používá konfiguraci ve formátu XML, byl použit i pro překlad tohoto projektu) a novější Gradle (konfigurace je psána v jazyce Groove).

4.1.4 JavaFX

Pro realizaci grafického rozhraní editoru bylo použito prostředí JavaFX. Je to platforma pro vývoj grafických desktopových aplikací v jazyce Java pro Windows, Linux a MacOS. Byla vydána v roce 2008 jako nástupce starší platformy Swing, která je však dosud poměrně široce používána. [3]

GUI v JavaFX je vyvíjeno pomocí souborů FXML (značkovací jazyk založený na XML) a metod v připojené třídě `Controller`, které jsou volány při použití některého z definovaných grafických ovládacích prvků. Soubory FXML jsou tvořeny do sebe zanořenými XML elementy, které lze z kontroleru adresovat za pomoci jejich ID.

4.2 Použité knihovny

Aplikace při svém překladu využívá prostřednictvím repozitáře Maven následující knihovny:

- **JavaFX Controls** – Ovládací prvky grafického prostředí JavaFX.
- **JavaFX FXML** – Zpracování FXML souborů, v nichž je definován design JavaFX aplikací.
- **JavaFX Graphics** – Grafické prvky JavaFX. Tato knihovna je závislá na platformě, aby byl tedy výsledný JAR archiv spustitelný v prostředích Windows, Linux i Mac OS, je nutné použít všechny tři verze knihovny zároveň. To bohužel při sestavování JAR archivu vyvolává „overlapping classes“ varování.
- **Lombok** – Knihovna umožňující automatické generování opakujících se částí kódu jako konstruktorů, getterů a setterů.
- **JetBrains Annotations** – Podpora anotací Lomboku v době kompilace.
- **Java API for XML Processing** – Knihovna pro zpracování a sestavování XML dokumentů.

4.3 Grafické prvky

Grafické objekty aplikace jsou de facto rozšířením základní struktury OOPN pro potřeby jejího zobrazení a manipulace s ní. Jsou dvojího typu a nacházejí se v balíčcích `view` a `simulation`. Obsahují reference na jednotlivé části OOPN a na tvary je reprezentující (`Ellipse`, `Rectangle`, `Line` a další)

4.3.1 Editor

Instance tříd v balíčku `view` tvoří editor OOPN. Jedná se o třídy:

- `ModelView` – Reprezentuje kořenovou úroveň in-memory struktury. Obsahuje metody `setMainClassView` (nastaví novou počáteční třídu) a `getClassByName` (vrátí `ClassView` objekt pro třídu s daným jménem).
- `ClassView` – Reprezentuje třídu.

- `MethodView` – Reprezentace metody (nebo konstruktoru). Ve svém konstruktoru vytvoří novou síť a přidá do ní návratové místo. Poté je zavolána metoda `parseMessagePattern`, která zpracuje uživatelem zadaný vzor zprávy a automaticky uloží název metody a její parametry. Tato metoda je volána též při každém přejmenování metody. Třída obsahuje také alternativní konstruktor, který metodu načítá z její XML reprezentace (element `method`).

- `NetView` – Reprezentuje síť. Obsahuje počítadla míst, přechodů a synchronních portů, metody `getPlaceByName`, `getTransitionByName` a `getSyncPortByName`, které vrací příslušný objekt s daným jménem, a `getElementByName`, jež spojuje funkci předchozích tří metod.

- `PlaceView` – Reprezentace místa obsahující hned několik mírně odlišných konstruktorů (standardní, pro načítání z XML reprezentace, pro návratová místa, pro parametrová místa a pro odkazy na místa objektové sítě). Dále třída disponuje několika metodami, které jsou volány z konstruktorů a vytváří jednotlivé části místa (`createCenterLine`, `createMarkingView`, `createActionView`, `crateNameView` a `createContextMenu`), metodou `mouseDragHandler`, jež zajišťuje souvislý přesun všech součástí místa při tažení myší po plátně, a metodou `delete`, která všechny tyto součásti odebírá z plátna a in-memory struktury.

- `TransitionView` – Obdoba předchozí třídy, líší se ovšem v několika ohledech (jen dva konstruktory, čtyřúhelník namísto elipsy, stráž namísto počátečního značení atd.).

- `SyncPortView` – Prvek, který se liší od přechodu absencí akce a několika dalšími odlišnostmi.

- `ArcView` – Reprezentuje hranu. Každá hrana se kromě základní čáry skládá i ze dvou dalších, které spolu s ní dohromady tvoří šipku. Tato šipka směřuje směrem k místu nebo přechodu/synchronnímu portu podle typu hrany. V případě testovací hrany jsou šipky na obou stranách a je tedy třeba vytvořit další dvě přídavné čáry. Každou hranu je možné jednou zalomit. V takovém případě se vytvoří kruh reprezentující bod zalomení a přídavná čára (metoda `createBreakPoint`, která, je-li zavolána se souřadnicemi jako argumenty, nejprve tyto součadnice nastaví, což je využito při načítání z XML reprezentace), která směřuje od tohoto bodu k cílovému prvku. Souřadnice středu kruhu a jednoho z konců obou čar se v tomto případě shodují. Při vytváření hrany je třeba vypočítat pozici obou jejích konců, které se musí nacházet na okraji prvku, k němuž jsou připojeny, tak, aby hrana směřovala do jeho středu. Funkce počítající pozici obou souřadnic se liší podle toho, zda se pozice počítá pro elipsu či čtyřúhelník. V případě elipsy jde o metody `relocateArcOnPlaceX` a `relocateArcOnPlaceY`, v případě čtyřúhelníku `relocateArcOnTransitionX` a `relocateArcOnTransitionY`. Ve všech zmíněných metodách je nutné zohlednit, zda v daném místě hrana začíná, nebo končí, a podle toho nastavit referenci na správný setter třídy `Line`. Dále je důležité, zda není připojená čára zmíněnou přídavnou čarou, vytvořenou při zalomení hrany. Po vytvoření hrany je třeba všechny relokalizační metody zavolat dvakrát, aby se všechny části hrany dostaly do patřičných pozic (metoda `init`). Třída dále obsahuje metody `crateExpressionView` (vytvoří plovoucí pole s hranovým výrazem), `createContextMenu` (stejně jako v předchozích případech vytvoří kontextové menu hrany a jeho položky).

V balíčku se rovněž nacházejí rozhraní `ElementView` (implementováno všemi prvky sítě, je využito v metodě `getElementByName` třídy `NetView`), `RectangleElementView` (implementováno prvky tvaru čtyřúhelníku, tedy přechody a sychronními porty, využito především při přepočtu pozice hran) a `StructureItem` (implementováno třídami `ModelView`, `ClassView` a `NetView`, využito v panelu struktury v levém podokně jako identifikátor položky stromu).

Serializací této reprezentace dostaneme XML soubor, z něhož je později možné celý model opětovně načíst, a to včetně polohy všech prvků v editoru. Struktura takového souboru bude popsána dále.

4.3.2 Simulátor

Tyto třídy jsou do značné míry podobné prvkům editoru, narozdíl od nich ovšem neobsahují kontextová menu a další ovládací prvky. Slouží pouze k zobrazení stavu simulace. Jsou to třídy v balíčku `simulation`:

- `NetSim` – Reprezentuje simulovanou síť. Obsahuje jméno a ID sítě, referenci na odpovídající instanci třídy NetView a metody `findPlaceView`, `findParamPlaceView`, `findReturnPlaceView`, `findTransitionView` a `findSyncPortView`, které v připojené NetView vyhledá patřičný objekt.
- `PlaceSim` – Reprezentace simulovaného místa obsahující konstruktory pro běžné místo (používaný i pro parametrová místa), návratové místo a odkazy na objektová místa.
- `TransitionSim` – Podobá se třídě PlaceSim, obsahuje ale pouze jeden konstruktor.
- `SyncPortSim` – Od přechodu se liší pouze absencí akce.
- `ArcSim` – Reprezentuje simulovanou hranu. Narozdíl jejího protějšku v editoru je většina koordinátů nastavovány explicitně a neprobíhají téměř žádné výpočty.

Tato reprezentace je vytvořena při stažení aktuálního stavu simulace ze serveru za použití aktuálních značení míst ze serveru. Všechna ostatní data jsou získána z editoru.

4.4 Chování aplikace

Chování všech ovládacích prvků JavaFX, které jsou definovány v souboru `opne-view.fxml`, je popsáno třídou `Controller.java`. Mapování do layoutu zajišťují anotace `@FXML`. Každá ze zde popsaných metod s touto anotací je volána jako reakce na interakci s některým z grafických prvků definovaných v FXML. Třída obsahuje metody:

- `showError` – Zobrazí chybové hlášení.
- `showAbout` – Zobrazí okno s informacemi o aplikaci.
- `selectTab` – Provede akce spjaté s přepnuntím panelu (nastavení aktivní sítě a aktivního plátna, nastavení režimu spojování hran na `READY` atd.).
- `newModel`, `newClass`, `newMethod` a `newConstructor` – Vytvoří novou položku v podokně struktury modelu a zároveň vytvoří patřičný objekt v in-memory stukrutře editoru. Před tím zobrazí dialog, jehož prostřednictvím lze změnit jméno vytvářeného objektu. Výchozími názvy jsou `Model`, `Class`, respektive `method`.

- `createMethod` – Vytváří metodu, volána metodami `newMethod` a `newConstructor`.
- `getInheritanceItemForClassView` – Rekurzivní metoda, která vrátí položku panelu dědičnosti, která odpovídá dané třídě.
- `openNet` – Otevře nový panel a nastaví jeho obsah na plátno právě vybrané sítě v panelu struktury.
- `openNetSim` – Otevře nový panel a nastaví jeho obsah na plátno právě vybrané sítě v panelu simulace.
- `setMainClass` – Nastaví počáteční třídu v in-memory struktuře a změní zobrazený název tak, že k němu přidá suffix `" (main)"`. Pokud předtím byla nastavena jiná počáteční třída, odstraní suffix v jejím názvu.
- `structureMenu` – Zneviditelní irelevantní položky kontextového menu panelu struktury v levém podokně.
- `inheritanceMenu` – Odebrání možnosti změny rodičovské třídy u kořenové třídy PN v panelu dědičnosti v levém podokně.
- `simulationMenu` – Zneviditelnění kontextového menu simulačního panelu, pokud není vybrána žádná síť.
- `renameItem` – Přejmenuje položku, jež je momentálně vybrána v panelu struktury (a zkontroluje správnost nového názvu podle toho, o který typ položky jde).
- `deleteItem` – Odstraní prvek vybraný v panelu struktury.
- `changeParent` – Změní rodičovskou třídu třídy, která je momentálně vybrána v panelu dědičnosti.
- `newPlace`, `newTransition` a `newSyncPort` – Vytvoří nové místo, nový přechod nebo nový synchronní port v místě, kde uživatel pustí levé tlačítko myši. Zároveň vytvoří prvek Petriho sítě v in-memory struktuře. Ihned po vytvoření nastaví novému objektu všechny jeho vlastnosti, jako jsou reakce na tažení myší, reakce na kliknutí při spojování míst a přechodů hranou nebo změny podoby kurzoru při přejetí myší nad ním.
- `switchArcMode` – Reakce na tlačítko pro vytváření hran. Pokud je aktuální režim vytváření hran `READY`, nastaví režim na `ACTIVE`. Od uživatele se následně očekává, že klikne levým tlačítkem na místo a přechod, které chce spojit. Po prvním kliknutí se nastaví režim na `PLACE` nebo `TRANSITION` a očekává se kliknutí na druhý typ prvku. Po druhém kliku je režim opět `READY`, dokud uživatel znovu nestiskne tlačítko vytváření hran. V případě, že uživatel stiskne tlačítko v jakémkoli jiném režimu než `READY`, dojde k přerušení všech akcí a režim je nastaven zpět na `READY`.
- `createPlaceClicked`, `createTransitionClicked` a `createSyncPortClicked` – Změna podoby kurzoru při kliknutí na ikonu prvku sítě v pravém podokně.
- `saveModel` a `saveModelAs` – Zavolá metodu `saveModel` třídy `XmlHandler`, která uloží aktuální obsah in-memory struktury jako XML dokument. V případě použití metody `saveModelAs` je uživatel vždy vyzván k výběru souboru, v opačném případě se okno pro výběr objeví, pouze pokud dosud s žádným souborem neproběhla interakce.

- `loadModel` – Zavolá metodu `loadModel` třídy `XmlHandler` a pokud ta proběhne úspěšně, provede doprovodné akce jako zpřístupnění ukládání atd.
- `exportModel` – Zavolá metodu `exportModel` třídy `PNTalkHandler`.
- `onStructureClick` – V případě dvojkliku na síť v panelu struktury tuto síť otevře.
- `renameTab` – Přejmenuje panel.
- `closeTab` – Zavře panel.
- `createSimulationTree` – Vytvoří strom simulovaných sítí v panelu simulace.
- `startSimulation` – Zavolá metodu `sendModel` třídy `SimulationHandler` a metodu `createSimulationTree`. Poté expanduje simulační panel, znepřístupní položku zahájení simulace a zpřístupní položky pro krokování a zastavení simulace.
- `nextStep` – Provede krok simulace zavoláním metody `stepSimulation` třídy `SimulationHandler`. Dále přepíše obsah otevřených simulovaných sítí novým stavem a zavře simulované sítě, které v novém stavu už neběží.
- `quitSimulation` – Zastaví simulaci zavoláním metody `quitSimulation` třídy `SimulationHandler`, zavře všechny simulované sítě a nastaví uživatelské rozhraní do stavu před zahájením simulace.
- `changeAddress` a `changePort` – Otevře dialog pro zadání adresy nebo portu serveru a změní hodnotu na zadaný vstup.

4.5 Vstup a výstup

Výměnu dat mezi aplikací a okolním světem zajišťuje balíček `io`, jehož třídy obsahují statické metody sloužící k importování a exportování všech tří zmíněných in-memory struktur. Jde o třídy:

- `XmlHandler` – Zajišťuje import z nativního XML formátu a export do něj.
- `SvgHandler` – Zajišťuje export sítě do podoby vektorové grafiky.
- `PNTalkHandler` – Zajišťuje export do textové varianty jazyka PNtalk. Takto exportovaný model může být i uploadován na simulační server.
- `SimulationHandler` – Zajišťuje komunikaci se simulačním serverem prostřednictvím síťových socketů.

4.5.1 Ukládání a načítání

Veškeré vstupní a výstupní operace týkající se nativního XML formátu zajišťuje třída `XmlHandler`, která se nachází v balíčku `io`. Tato třída se nikdy nainstancuje, všechny její metody jsou statické.

K uložení modelu slouží metoda `saveModel`. Před jejím zavoláním je třeba inicializovat třídní proměnnou `file`, která nese XML soubor otevřený pro zápis. Pokud tato proměnná

inicializována není, metoda skončí, aniž by provedla jakoukoli akci. K inicializaci slouží metoda `openFileForWriting`, jež otevře systémový dialog pro výběr souboru.

Pokud je výstupní soubor vybrán, je vytvořen nový Document Object Model a jeho kořenový element `model`. Následně jsou nad jednotlivými částmi in-memory struktury modelu postupně volány metody `exportClass`, `exportMethod`, `exportNet`, `exportPlace`, `exportTransition`, `exportSyncPort`, `exportObjectPlaceLink` a `exportArc`. Každá z těchto metod zajišťuje export stejnojmenného elementu a všech v něm zanořených (metody jsou tedy volány v kaskádě).

Po exportu celého modelu do DOM stromu je tento strom zapsán do otevřeného souboru. Uložený soubor zůstává v proměnné `file` pro případ, že uživatel bude chtít uložit další změny. Pokud se má model uložit do jiného souboru, je třeba znovu zavolat metodu `openFileForWriting`.

Načítání zajišťuje metoda `loadModel`. Ta nejdříve otevře dialog pro výběr souboru a z XML dokumentu, který uživatel zvolil, načte DOM strom. Je vytvořena nová in-memory struktura modelu, jež je následně plněna objekty specifikovanými v načteném dokumentu pomocí metod `parseClass`, `parseMethod`, `parseNet`, `parsePlace`, `parseTransition`, `parseSyncPort`, `parseObjectPlaceLink` a `parseArc`.

Tyto metody fungují inverzně k metodám zmíněným v předchozí části. Všechny části modelu jsou vytvořeny obdobným způsobem jako při jejich vytváření uživatelem, pouze se tak děje s parametry načtenými z DOM stromu.

4.5.2 Export do SVG

Manipulaci s SVG soubory zajišťuje třída `SvgHandler`. Otevírání souborů a práce s DOM stromem je obdobné jako u třídy `XmlHandler`. Není použita žádná externí knihovna, XML kód je generován (stejně jako v případě ukládání modelu) manuálně.

Jelikož je způsob definice obrazců v JavaFX a SVG je do značné míry podobný, je možné je exportovat poměrně jednoduše a přímočaře. Středobodem exportu do SVG je metoda `printNet`, která (podobně jako metoda `exportNet` ve třídě `XmlHandler`) dále využívá metody `printPlace`, `printTransition`, `printSyncPort` a `printArc`. Do každého SVG souboru je také přidán stylopis ze souboru `svg.css`, v němž je specifikována podoba všech prvků v exportovaném obrázku.

4.5.3 Simulace

Logika pro komunikaci se simulačním serverem PNtalk se nachází ve třídě `SimulationHandler`. Obsahuje metody:

- `showError` – Zobrazí chybové hlášení, které vrátil simulační server.
- `sendModel` – Odešle model aktuálně otevřený v editoru na simulační server pomocí příkazu `addClass`. Všechny třídy jsou odeslány hromadně v rámci jediné zprávy. Pokud dojde k chybě nebo se nepodaří se serverem spojit, je vypsána chyba. Pokud vše proběhne v pořádku, je automaticky invokována metoda `startSimulation`.

- `startSimulation` – Vyzve simulační server, aby zahájil novou simulaci odesláním příkazu `newSim`. V případě neúspěchu je vypsána chyba, v případě úspěchu je nastaveno číslo simulace na číslo vrácené serverem a příznak spuštění simulace na `true`. Poté je pomocí metody `getState` získán prvotní stav simulace.

- `getState` – Získá ze simulačního serveru aktuální stav simulace. Volá metodu `parseState` s řetězcem stavu vráceným serverem jako parametrem a následně metodu `drawNets`, která fyzicky vytvoří objekty reprezentující simulované sítě.

- `parseState` – Obsahuje nekonečný cyklus, který zpracovává vstupní řetězec principem podobným lexikálnímu analyzátoru. Tento automat se může nacházet v jedné z pěti fází: `MESSAGE` (zpracování aktuálně rozpracovaných zpráv), `OBJECT` (zpracování struktury simulavaného objektu), `NET` (zpracování některé ze sítí objektu), `PLACE` (zpracování konkrétního místa), `THREAD` (zpracování aktuálně běžícího vlákna). Metoda vrací seznam simulovaných sítí.

- `stepSimulation` – Odešle simulačnímu serveru požadavek na provedení jednoho kroku simulace s uloženým ID. Následně s použitím metody `getState` vrátí nový stav simulace.

- `quitSimulation` – Ukončí simulaci, která běží na simulačním serveru, a následně postupně odstraní všechny třídy, které na něj byly v rámci této simulace nahrány, pomocí metody `deleteClass`. Navíc nastaví příznak spuštění simulace na `false`. Tato metoda je automaticky vykonána při ukončení aplikace (jak zavřením okna, tak výběrem položky `exit`) v hlavní nabídce.

- `deleteClass` – Odstraní třídu s daným názvem ze simulačního serveru. V případě neúspěchu vypíše chybu a vrátí `false`, v případě úspěchu vrátí `true`.

Kapitola 5

Použití aplikace

V této kapitole si představíme implementovaný nástroj, jeho uživatelské rozhraní, funkce a příklady použití. Nástroj byl vyvíjen v operačním systému Microsoft Windows 10, vzhled a chování některých komponent JavaFX může na ostatních platformách být mírně odlišné.

5.1 Spuštění

Aplikaci lze spustit několika způsoby. Ve všech případech spuštění vyžaduje nainstalované prostředí Java Runtime Environment (JRE). Pro prostředí Windows neexistuje samostatné JRE pro Javu verze 17, je proto třeba nainstalovat kompletní Development Kit (JDK). Nejjednodušším řešením je JDK stáhnout z oficiálních webových stránek společnosti Oracle (www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html).

V linuxových systémech je možné Javu nainstalovat stejným způsobem, ale i pomocí terminálu. V Debian-based distribucích lze takto nainstalovat i samostatné JRE, a to příkazem:

```
sudo apt install openjdk-17-jre
```

Nyní můžeme aplikace spustit jednou ze čtyř možností.

5.1.1 Použití přiloženého JAR archivu

Součástí média přiloženého k této práci je soubor `opn-editor-1.0.jar`. Je-li JRE správně nainstalováno, aplikaci je možné spustit běžným dvojklikem na tento soubor. Alternativně lze v prostředí Linuxu použít následující příkaz v kořenovém adresáři projektu:

```
java -jar opn-editor-1.0.jar
```

5.1.2 Manuální sestavení JAR archivu

Pokud přiložený archiv z jakéhokoli důvodu není k dispozici, je možné jej ze zdrojových souborů sestavit manuálně. Je k tomu potřeba nainstalovaný nástroj Apache Maven ve verzi 3.8.1, který projekt využívá ke svému překladu. Verze se bohužel musí přesně shodovat, v opačném případě se sestavení nezdaří. V Debian-based systémech lze Maven standardně nainstalovat příkazem:

```
sudo apt install maven
```

V době psaní tohoto textu ovšem například Ubuntu stále instaluje verzi 3.6.0, která je nekompatibilní s JDK 17. Je tedy nutné ručně stáhnout zdrojové soubory verze 3.8.1 (ne novější) a tu nainstalovat. Pro správné fungování je navíc třeba definovat environment variables `JAVA_HOME`, `M2_HOME` a `MAVEN_HOME` a přidat adresář `M2_HOME/bin` do `PATH`.

Dalším možným řešením je změnit verzi Maven pluginu v souboru `pom.xml` v kořenovém adresáři projektu na jinou verzi, která je nainstalována na stoji, na němž se snažíme aplikaci spustit. Musí ale jít o verzi kompatibilní s Javou 17.

Se správnou verzí Maven můžeme poté k sestavení archivu použít příkaz:

```
mvn package
```

Sestavený soubor `opn-editor-1.0.jar` najdeme v adresáři `target`.

5.1.3 Manuální spuštění bez JAR archivu

Aplikaci je možné spustit i bez sestavování archivu. Stejně jako v předchozím případě je třeba nástroj Apache Maven ve stejné verzi, jakou má Maven plugin v souboru `pom.xml`. Místo sestavení JAR souboru ovšem pouze zkompilujeme zdrojové soubory příkazem:

```
mvn compile
```

Nyní můžeme aplikaci spustit pomocí příkazu `java`. Je třeba mít nainstalované knihovny pro JavaFX, které je nutné přidat jako moduly parametrem `add-modules` a lokalizovat parametrem `module-path`. Jelikož se lokace knihovny i samotných zkompilovaných tříd mohou v různých operačních systémech lišit, nebudeme si uvádět konkrétní syntaxi příkazu.

5.1.4 Spuštění z IDE

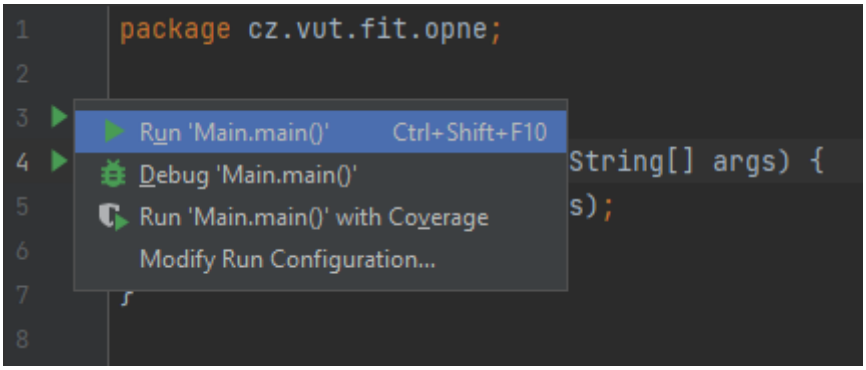
Projekt můžeme otevřít v integrovaném vývojovém prostředí (Integrated Development Environment, IDE) a aplikaci spustit přes jeho rozhraní. Aplikace byla vyvíjena v IDE *JetBrains IntelliJ IDEA Community Edition* ve verzi 2021.3.3, proto si postup spuštění popíšeme v něm.

IntelliJ IDEA lze stáhnout z oficiálních webových stránek společnosti JetBrains (www.jetbrains.com/idea/download). Program existuje ve dvou verzích: zpoplatněné verzi Ultimate a bezplatné Community Edition. Pro naše potřeby komunitní edice naprosto postačuje.

Ve spuštěném IDE otevřeme projekt kliknutím na položku `Open...` v nabídce `File` a vyhledáním souboru `pom.xml` v kořenovém adresáři projektu. IntelliJ automaticky stáhne všechny potřebné knihovny z Maven repozitáře. Pro spouštění aplikace je třeba mít nainstalován Java Development Kit (JDK) verze 17. Pokud se JDK v systému nachází, IntelliJ by jej mělo automaticky detekovat. Pokud ne, lze jej nainstalovat v okně `Project structure...` v nabídce `File`.

Nyní je třeba vytvořit novou spouštěcí konfiguraci. V levém podokně vyhledáme třídu `Main` a dvojklikem ji otevřeme. V editoru klidneme pravým tlačítkem na zelenou šipku a na řádku s definicí metody `main` a vybereme možnost `Run 'Main.main()'`. Při opakovaném

spouštění lze už použít tutéž zelenou šipku v pravé horní části okna vedle názvu spuštěcí třídy (`Main`).



Obrázek 5.1: IntelliJ IDEA: Spuštění aplikace.

V prostředí IntelliJ můžeme také provést sestavení JAR archivu, a to kliknutím na záložku `Maven` v pravém panelu nástrojů, poté kliknutím na ikonu písmene `m` (Execute Maven Goal) a v otevřeném podokně vybráním příkazu `mvn package`.

5.2 Základní ovládací prvky

Uživatelské rozhraní aplikace se skládá z následujících částí:

1. **Hlavní nabídka** – Slouží k základnímu ovládání programu. Obsahuje tři podnabídky. Nabídka `File` zajišťuje manipulaci se soubory. Zahrnuje následující položky:
   - `New...` – Vyvolá dialog pro zadání názvu nového modelu, který následně vytvoří.
   - `Open...` – Zobrazí dialog pro výběr dříve uloženého modelu ve formátu XML dokumentu.
   - `Save` – Přepíše otevřený XML soubor aktuální verzí modelu.
   - `Save as...` – Uloží aktuální verzi modelu jako nový soubor.
   - `Export to PNtalk...` – Uloží model do textového souboru ve formě čistého jazyka PNtalk.
   - `Exit` – Ukončí aplikaci.

   Nabídka `Simulation` zajišťuje ovládání komunikace se simulačním serverem. Obsahuje položky:
   - `Start simulation` – Odešle otevřený model na simulační server, spustí simulaci a získá prvotní stav simulace.
   - `Next step` – Provede jeden krok spuštěné simulace a získá její nový stav.
   - `Quit simulation` – Ukončí otevřenou simulaci a následně odstraní ze simulačního serveru všechny použité třídy.
   - `Change server address...` – Zobrazí dialog pro zadání nové adresy simulačního serveru (výchozí hodnotou je `localhost`).

- `Change port number...` – Zobrazí dialog pro zadání nového čísla portu (výchozí hodnotou je `9999`).

V nabídce `Help` se nachází jediná položka `About...`, která vyvolá okno s informacemi o aplikaci.

2. **Levé podokno** – Slouží pro manipulaci s třídami a sítěmi. Obsahuje tři panely, z nichž lze v jednu chvíli zobrazit pouze jeden:
   - `Structure` – Zobrazuje stromovou strukturu modelu a umožňuje vytvářet, odstraňovat, modifikovat a otevírat třídy, metody a konstruktory.
   - `Inheritance` – Zobrazuje strom dědičnosti tříd a dovoluje měnit, od které třídy která dědí.
   - `Simulation` – Pokud je spuštěna simulace, obsahuje seznam aktuálně otevřených sítí. Pokud není simulace spuštěna, panel je prázdný.
3. **Pravé podokno** – Obsahuje piktogramy prvků OOPN, s jejichž pomocí je možné přidávat nové prvky do otevřené sítě.
4. **Pracovní plocha s panelovým manažerem** – Zde je možné pracovat se samotnými sítěmi. Otevřené sítě mohou být dvojího typu: editovatelné a simulované. Simulované narozdíl od editovatelných pouze zobrazují hodnoty jednotlivých prvků a neumožňují uživateli žádné úpravy. Vyznačují se tmavšími barvami, než mají editovatelné sítě. Díky panelům můžeme mít otevřených několik sítí zároveň (jak editovatelných, tak simulovaných) a přepínat mezi nimi. Panel má své kontextové menu obsahující následující možnosti:
   - `Export net as SVG...` – Umožňuje exportovat otevřenou síť ve formě vektorové grafiky.
   - `Rename tab...` – Otevře dialog pro zadání nového názvu panelu. Výchozí název po otevření je vytvořen podle vzoru `class_name: method_name/"Object net"`. Jde-li o simulovanou síť, přidává se k názvu panelu suffix `" (sim)"`.
   - `Close` – Zavře vybraný panel.
   - `Close other tabs` – Zavře všechny panely mimo vybraný panel.
   - `Close all tabs` – Zavře všechny panely.

5.3 Editace modelů

Záladní funkcí aplikace je editace modelů v jazyce PNtalk. Můžeme buď otevřít již existující model (položka `Open...` v nabídce `File` nebo `CTRL + O`) nebo vytvořit nový (položka `New...` tamtéž nebo `CTRL + N`).

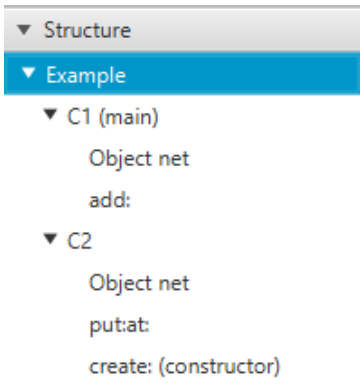
5.3.1 Panel struktury

Panel struktury má podobu stromu, jehož kořenem je název modelu. Jeho potomky jsou jednotlivé třídy (po vytvoření nového modelu tedy žádné nemá). Počáteční třída se od ostatních odlišuje suffixem `" (main)"`. Klepnutím pravým tlačítkem na model vyvoláme kontextovou nabídku, z níž můžeme přidat třídu (`Add class...`) nebo změnit název modelu (`Rename...`).

V kontextové nabídce třídy najdeme možnosti přidání nové metody (`New method...`), nového konstruktoru (`New constructor...`), nastavení příslušné třídy jako počáteční (`Set as main`), přejmenování třídy (`Rename...`) a jejího odstranění (`Delete`).

Potomky tříd jsou jejich sítě. Ihned po vytvoření třídy je vytvořena její objektová síť (má vždy název `Object net`). Dále se zde zobrazí každá přidaná metoda a každý přidaný konstruktor (od běžné metody se liší suffixem `" (constructor)"`). Názvy metod a konstruktorů jsou uvedeny bez parametrů s jednotlivými částmi jména oddělenými dvojtečkou (v případě metody bez parametrů název dvojtečku neobsahuje). Při vytváření metody uživatel zadává vzor zprávy, tedy úplný název včetně parametrů.

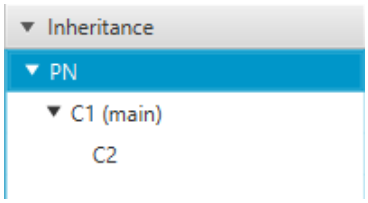
V kontextovém menu každé sítě se nachází možnost tuto síť otevřít (`Open`), čehož lze alternativně docílit i dvojklikem. Nabídka metod a konstruktorů dále obsahuje podnabídku zobrazení objektových míst (`Show object places`), z níž můžeme vybrat místo objektové sítě, které se zobrazí v síti metody jako odkaz. Dále odsud můžeme metodu přejmenovat (`Rename...`) či odstranit (`Delete`).



Obrázek 5.2: Panel struktury.

5.3.2 Panel dědičnosti

Tento panel má podobu stromu tříd, kořenem je implicitní třída PN. V kontextovém menu každé třídy (vyjma kořenové třídy) můžeme za pomoci položky `Change parent class...` změnit jejího rodiče.



Obrázek 5.3: Panel dědičnosti.

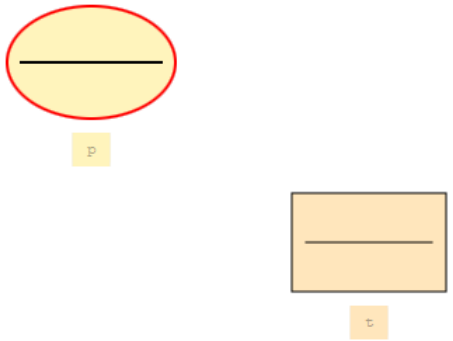
5.3.3 Editace sítě

Po otevření sítě do ní lze vkládat místa a přechody (a v případě objektových sítí navíc i synchronní porty) přetažením jejich piktogramů v pravém podokně myší na pracovní plochu.

Vložené prvky můžeme přesouvat po pracovní ploše tažením myší. Aktuálně vybraný prvek je zvýrazněn tučnými okraji a (u míst a přechodů) středovou čarou. Pomocí kontextového menu je možné upravit počáteční značení (`Edit marking`), stráž (`Edit guard`), akci (`Edit action`) nebo jméno (`Rename`), případně prvek smazat (`Delete`). Úprava jména a dalších atributů prvku je možná i dvojklikem na příslušné textové pole.

Název míst, přechodů a synchronních portů musí být v rámci dané sítě unikátní. Jmenný prostor všech prvků je společný. Žádné místo v metodě zároveň nemůže mít stejný název jako místo v objektové síti.

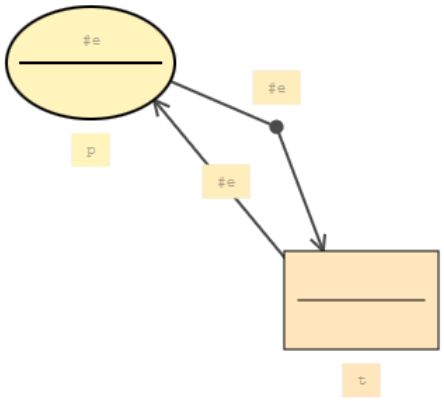
Stisknutím tlačítka `Arc` v pravém podokně se přepneme do režimu spojování prvků. Na fakt, že se nacházíme ve spojovacím režimu, nás upozorní červené podbarvení tlačítka. Nyní je možné spojit místo s přechodem nebo synchronním portem. Po kliknutí na první prvek se jeho obrys zbarví červeně a po kliknutí na druhý se mezi nimi vytvoří hrana. Vytvořená hrana vždy směřuje od prvního vybraného prvku ke druhému.



Obrázek 5.4: Spojovací režim.

Typ hrany (vstupní, výstupní, testovací) lze změnit v kontextové nabídce hrany (`Change to`). Odsud lze rovněž upravit hranový výraz (`Edit expression`), čehož docílíme i dvojklikem na něj, hranu odstranit (`Delete arc`) nebo přidat bod zalomení hrany (`Create breaking point`). V takovém případě se uprostřed hrany vytvoří bod, který můžeme přesouvat po pracovní ploše jako samostatný objekt, hrana pak vede tímto bodem a je v něm zalomena. To nám umožňuje ovlivnit, kudy hrany mezi prvky povedou a docílit tak přehlednějších sítí. Pokud již bod zalomení nepotřebujeme, možeme jej odstranit (`Remove breaking point`).

Jestliže vytvoříme mezi dvěma prvky dvě hrany, bude druhá automaticky ihned zalomena, aby se hrany nepřekrývaly. Více než dvě hrany mezi žádnými dvěma prvky vytvořit nelze. Rovněž nelze vytvořit hranu mezi navzájem nekompatibilními prvky.



Obrázek 5.5: Dvojité propojení prvků.

Sítě metod oproti sítím objektů navíc obsahují speciální typy míst. Jsou to místo `return`, parametrová místa (která jsou vygenerována automaticky podle vzoru zprávy zadaného při vytváření metody) a odkazy na místa z objektové sítě (která lze do sítě vložit pomocí zmíněné funkce `Show object places` v kontextovém menu metody). Narozdíl od objektových sítí nemohou sítě metod obsahovat sychronní porty.

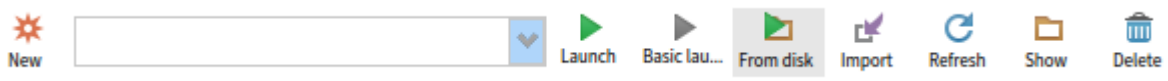
5.4 Simulace

Aplikace je připravena na komunikaci se simulačním serverem vyvinutým na FIT VUT v Brně a zobrazování stavu probíhající simulace přímo v rozhraní aplikace.

5.4.1 Spuštění simulačního serveru lokálně

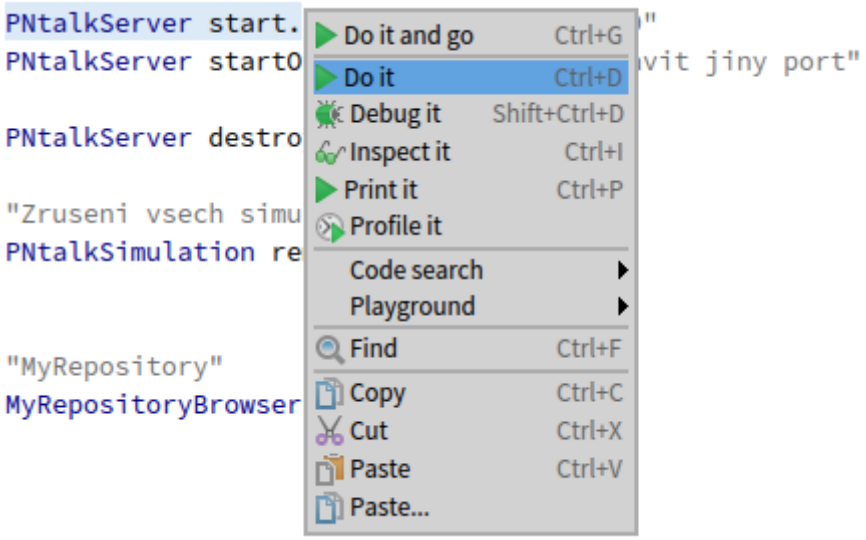
Pro testovací účely může být nutné spustit simulační server v prostředí Pharo na lokálním stroji. V takovém případě postupujeme následovně:

1. Stáhneme a nainstalujeme Pharo Launcher (www.pharo.org/download).
2. Získáme obraz PNtalk serveru, a to nejlépe kontaktováním jeho autora, Radka Kočího (www.fit.vut.cz/person/koci/contact/.cs).
3. Spustíme Pharo Launcher a na nástrojové liště klikneme na položku `From disk`. V otevřeném podokně vyhledáme obraz serveru a otevřeme tak projekt.



Obrázek 5.6: Nástrojová lišta Pharo Launcheru.

4. V okně „Basic Guide“ označíme text `PNtalkServer start.`, na označený text klepneme pravým tlačítkem a vybereme možnost `Do it`.



Obrázek 5.7: Start simulačního serveru pomocí „Do it“.

5. Nyní je PNtalk server spuštěný na výchozí adrese `localhost:9999`.

5.4.2 Zahájení simulace

Pokud simulační server běží, můžeme kliknutím na položku `Start simulation` v nabídce `Simulation` nebo klávesovou zkratkou `CTRL + P` simulaci zahájit. Adresu simulačního serveru a číslo portu lze změnit pomocí položek `Change server adress...` a `Change port number...`. Abychom to mohli provést, je třeba mít otevřený model s nastavenou počáteční třídou. Aplikace automaticky nahraje všechny třídy modelu na server, zahájí simulaci a vrátí její počátační stav.

Levé podokno se automaticky přepne na panel `Simulation`, v němž se jako kořen stromu zobrazí objektová síť počáteční třídy modelu. Síť zobrazenou v panelu simulace lze otevřít a zobrazit její obsah. Otevřená simulovaná síť má tmavší odstíny barev než editovatelná síť a nelze ji jakkoli upravovat.

5.4.3 Krokování simulace

Vybráním možnosti `Next step` v nabídce `Simulation` nebo klávesovou zkratkou `CTRL + I` posuneme spuštěnou simulaci o jeden krok vpřed. Ze serveru je získán nový stav simulace, seznam simulovaných sítí v panelu `Simulation` se aktualizuje, všechny aktuálně běžící sítě jsou zde zobrazeny jako potomci objektové sítě počáteční třídy, jejíž inicializací simulace začala.

Dojde-li ke změně značení některých z míst právě otevřené sítě, projeví se tato změna ihned v otevřeném panelu, není třeba jej otevírat znovu. Pokud došlo po provedení kroku k zavření některé z běžících sítí, je tento panel automaticky zavřen.

Simulaci můžeme kdykoli ukončit možností `Quit simulation` nebo zkratkou `CTRL + L`. Simulace je ukončena, panel simulace se zavře, spolu s ním všechny otevřené simulované

sítě. Zároveň jsou ze serveru postupně odstraněny všechny použité třídy. K tomuto úklidu dojde i v případě zavření celé aplikace bez ukončení simulace.

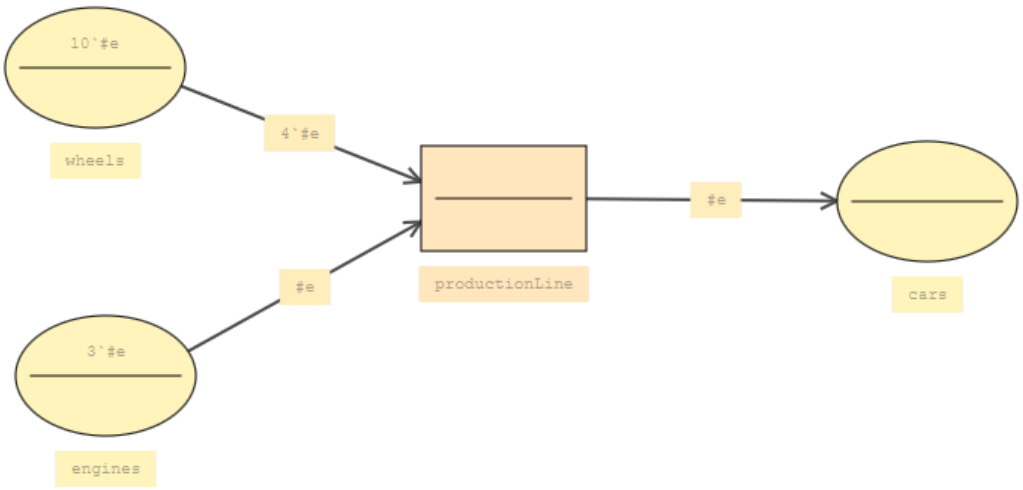
5.5 Příklady

Zde si uvedeme několik příkladů praktického využití nástroje k vykreslování a simulování Objektově orientovaných Petriho sítí.

5.5.1 Jednoduchá Petriho síť

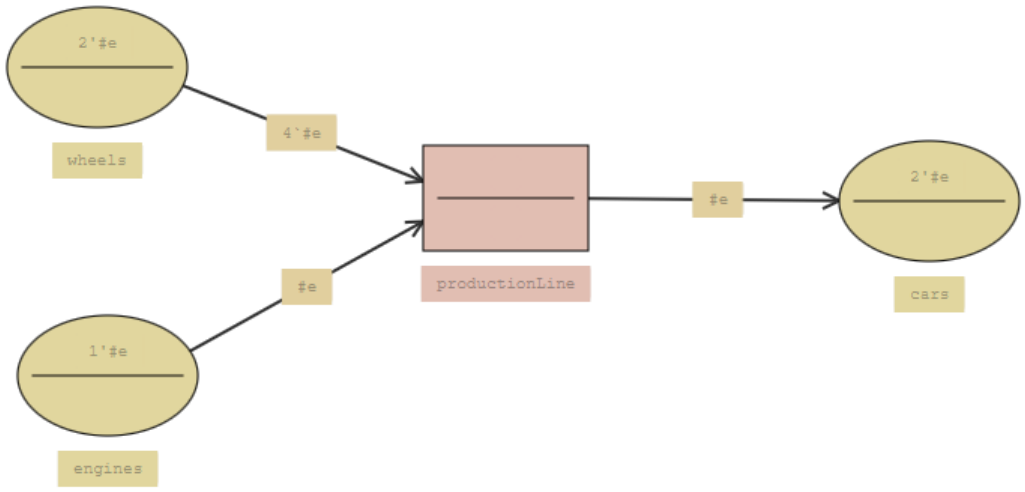
V tomto příkladu vytvoříme primitivní Petriho síť reprezentující montážní linku vyrábějící automobily. Na výrobu jednoho automobilu potřebujeme jeden motor a čtyři kola. Ve skladišti jsou k dispozici tři motory a deset kol. Symbol `#e` zde má význam tzv. „bezbarvé“ značky, tedy tečky z běžných P/T Petriho sítí.

Z pravého podokna přetáhneme na pracovní plochu tři místa a jeden přechod. Místa pojmenujeme `wheels`, `engines` a `cars`, přechod bude mít název `productionLine`. Prvky propojíme hranami podle obrázku 5.8. Vstupní hrany budou odebírat jednu značku z místa `engines` a čtyři značky z místa `wheels`. Výstupní hrana bude přidávat jednu značku v místě `cars`.



Obrázek 5.8: Příklad 1: Vytvořená síť.

Spustíme-li simulaci a budeme ji krokovat, vyrobí produkční linka postupně dva automobily (dvě značky v místě `cars`), protože právě na tolik má dostatek dílů. V místě `wheels` zbydou dvě značky, v místě `engines` jedna. Pokud budeme simulaci krokovat dále, nic se nebude dít.

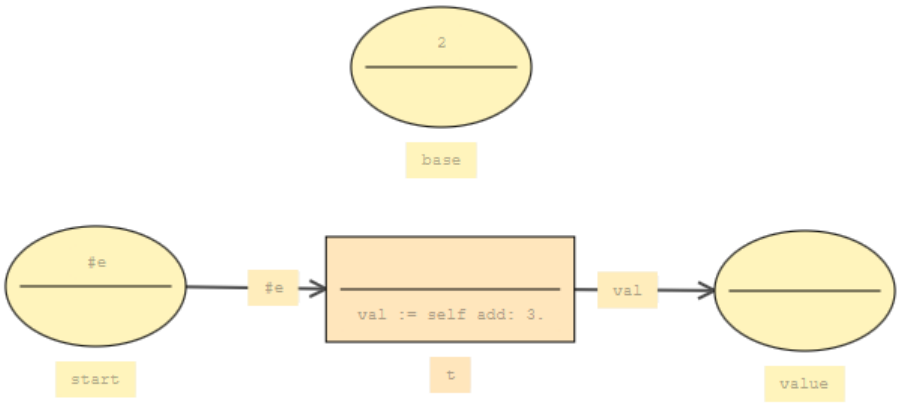


Obrázek 5.9: Příklad 1: Konec simulace.

5.5.2 Volání metody

Nyní si uvedeme příklad volání metody z přechodu. Vytvoříme novou třídu a objektovou síť podle obrázku 5.10. Místo `start` obsahuje jednu značku, která slouží jako token pro jednorázové spuštění metody, místo `base` definuje základ, ke kterému bude naše metoda přičítat. Do místa `value` bude po dokončení výpočtu vložen výsledek.

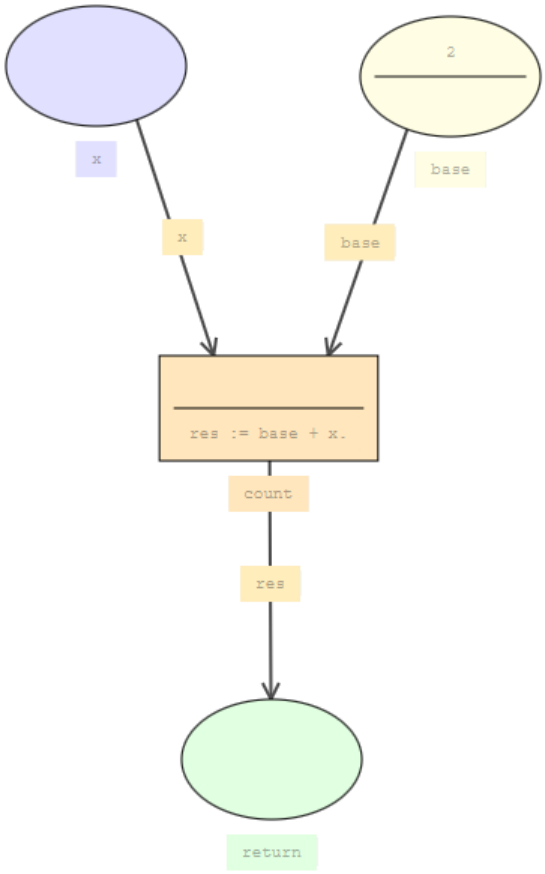
Akce přechodu `val := self add: 3.` znamená, že do proměnné `val` bude uložena hodnota vrácená metodou s názvem `add` a jedním parametrem s hodnotou 3. Pseudoproměnná `self` je referencí na objekt samotný, voláme tedy vlastní metodu běžícího objektu.



Obrázek 5.10: Příklad 2: Objektová síť.

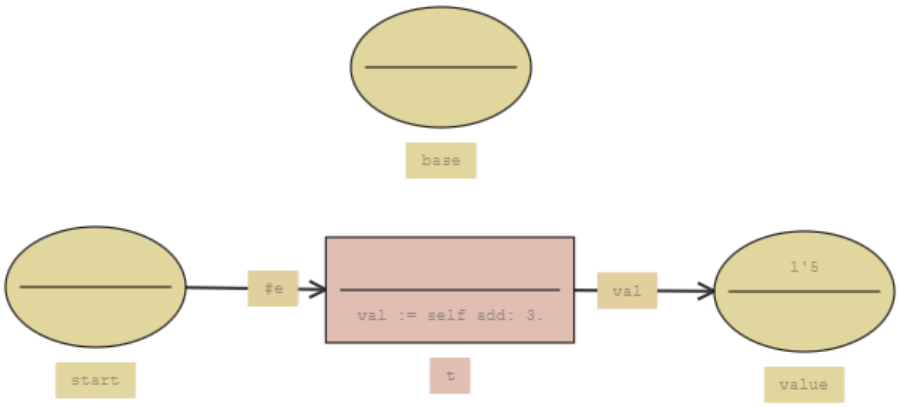
Přejděme k samotné metodě. Ta má vzor zprávy `add: x`, kde `x` je parametr, jehož hodnotu jsme při volání metody výše nastavili na hodnotu 3. Tuto hodnotu metoda bude přičítat

k základu uloženém v místě `base` v objektové síti (viz akci přechodu `res := base + x.`, která provádí tento jednoduchý výpočet).



Obrázek 5.11: Příklad 2: Metoda „add“.

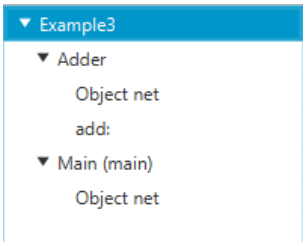
Po spuštění simulace přechod v objektové síti po odebrání značky z místa `start` zavolá metodu, která sečte základ v místě `base` (2) se zadaným parametrem (3) a výsledek vrátí přechodu, který jej uloží do místa `value`.



Obrázek 5.12: Příklad 2: Konec simulace.

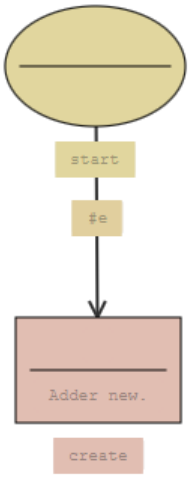
5.5.3 Vytvoření objektu

Nyní třídu z předchozího příkladu vytvoříme z jiné třídy jejím implicitním konstruktorem. Vytvoříme novou třídu a nastavíme ji jako počáteční.



Obrázek 5.13: Příklad 3: Panel struktury se dvěma třídami.

Otevřeme objektovou síť a vložíme do ní místo s jednou značkou, jež bude sloužit jako token pro vytvoření třídy, a jeden přechod. Jeho akce bude `Adder new.`, posíláme tedy třídě `Adder` (nikoli objektu) zprávu `new`. Nový objekt je vytvořen a jeho objektová síť invokuje metodu `add`, která provede sčítání stejně jako v předchozím příkladu.



Obrázek 5.14: Příklad 3: Konec simulace objektové sítě počáteční třídy.

Kapitola 6

Závěr

Cílem práce bylo vytvořit grafický editor Objektově orientovaných Petriho sítí (OOPN), který poskytne grafické rozhraní pro simulátor jazyka PNtalk vyvíjený na Fakultě informačních technologií Vysokého učení technického v Brně. Byla provedena podrobná studie formalismu OOPN, rozbor starší práce Antonína Neužila a byla navržena architektura nástroje zohledňující požadavek na jeho rozšiřitelnost.

Aplikace byla implementována v prostředí JavaFX. Způsob její obsluhy a příklady použití byly popsány v této technické zprávě. Deklarovaných cílů bylo dosaženo, aplikace je funkční a připravena k použití se simulačním serverem. Vývoj editoru pro jazyk PNtalk tímto ale zdaleka nekončí.

Celý nástroj obsahuje množství prostoru pro jeho další rozvoj, který se může stát tématem další bakalářské nebo diplomové práce. Editor v současnosti neobsahuje žádnou formu syntaktické ani sémantické kontroly akcí, stráží a hranových výrazů, stejně tak není přítomno ověřování validity modelu před odesláním na simulační server.

Vylepšit lze i uživatelské rozhraní. Bylo by vhodné umožnit v panelu dědičnosti měnit rodiče tříd prostým tažením myší. V panelu simulace je možné zavést uspořádání simulovaných tříd do víceúrovňového stromu podle toho, která síť je ze které volána. Editor by bylo možné obohatit o možnost víceřádkových hranových výrazů a vícenásobného zalomení hran, stejně tak by bylo možné oddělit jmenný prostor míst a přechodů v jedné síti. Simulátor lze rozšířit o přehled rozpracovaných zpráv a grafické znázornění aktuálně vykonávaného přechodu.

Na všechna tato rozšíření je architektura aplikace připravena a může se tedy stát základem robustního, multiplatformního a integrovaného prostředí pro editaci Objektově orientovaných Petriho sítí implementovaného v Javě.

Literatura

[1] Janoušek, V. *Modelování objektů Petriho sítěmi*. Brno, 1998. Disertační práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a informatiky. Vedoucí práce Doc. RNDr. Milan Češka, CSc.

[2] Neužil, A. *Nástroj pro práci s Objektově orientovanými Petriho sítěmi*. Brno, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Kočí, Ph.D.

[3] OpenJFX. *JavaFX*. 2022. [Online; navštíveno 08. 01. 2022]. Dostupné z: https://openjfx.io/.

[4] Pharo community. *Pharo — About*. 2022. [Online; navštíveno 20. 07. 2022]. Dostupné z: https://pharo.org/about.

[5] TikZ. *It's Time to Learn drawing Petri Nets*. 2021. [Online; navštíveno 02. 01. 2022]. Dostupné z: https://latexdraw.com/petri-nets-tikz.

[6] University of Hamburg. *Petri Nets World*. 2021. [Online; navštíveno 09. 12. 2021]. Dostupné z: https://www2.informatik.uni-hamburg.de/TGI/PetriNets.

[7] Wikipedia contributors. *Petri Net — Wikipedia*. 2021. [Online; navštíveno 05. 12. 2021]. Dostupné z: https://en.wikipedia.org/wiki/Petri_net.

[8] Wikipedia contributors. *Augmented Backus–Naur form — Wikipedia*. 2022. [Online; navštíveno 19. 07. 2022]. Dostupné z: https://en.wikipedia.org/wiki/Augmented_Backus%E2%80%93Naur_form.

[9] Wikipedia contributors. *Pharo — Wikipedia*. 2022. [Online; navštíveno 20. 07. 2022]. Dostupné z: https://en.wikipedia.org/wiki/Pharo.

[10] Wikipedia contributors. *Scalable Vector Graphics — Wikipedia*. 2022. [Online; navštíveno 09. 01. 2022]. Dostupné z: https://en.wikipedia.org/wiki/Scalable_Vector_Graphics.

[11] České vysoké učení technické. *Teorie hromadné obsluhy: Petriho sítě*. 2021. [Online; navštíveno 09. 12. 2021]. Dostupné z: https://www.fd.cvut.cz/department/k611/pedagog/K611THO_soubory/webskriptum/11_petriNet.html.

Příloha A

Plakát

Obrázek A.1: Plakát prezentující výsledky práce.

Příloha B

Obsah přiloženého média

- `doc` – Dokumentace zdrojového kódu vygenerovaná pomocí nástroje JavaDoc.
- `examples` – Příklady uložených modelů OOPN.
- `latex` – Zdrojové soubory této technické zprávy.
- `src` – Zdrojové soubory aplikace.
- `opn-editor-1.0.jar` – Spustitelný JAR archiv (multiplatformní).
- `pom.xml` – Konfigurační soubor Apache Maven.
- `README` – Informace o obsahu média.
- `xsvubd00-Graficky-editor-navrhovych-modelu.pdf` – Elektronická verze této technické zprávy.