VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

GENEROVÁNÍ SEKVENČNÍCH DIAGRAMŮ Z MODELŮ PETRIHO SÍTÍ

CODE GENERATION FROM OBJECT ORIENTED PETRI NETS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **ERIK KELEMEN**
AUTHOR

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

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Kelemen Erik**
Program: Informační technologie
Název: **Generování sekvenčních diagramů z modelů Petriho sítí**
Code Generation from Object Oriented Petri Nets
Kategorie: Softwarové inženýrství

Zadání:

1. Prostudujte problematiku tvorby a analýzy scénářů v modelování softwarových systémů.
2. Prostudujte koncept formalismu Objektově orientovaných Petriho sítí (OOPN) a dostupných simulátorů.
3. Navrhněte mechanismus generování sekvenčních diagramů ze scénářů modelů popsaných formalismem OOPN.
4. Implementujte nástroj pro generování sekvenčních diagramů. Nástroj musí umožnit mapování aktivit sekvenčních diagramů do modelů OOPN. Vytvořte sadu testovacích příkladů.
5. Analyzujte možné problémy a omezení spojená s transformacemi modelů. Pro vybrané problémy specifikujte jejich podstatu, důsledky a možná řešení.

Literatura:

- V. Janoušek: Modelování objektů Petriho sítěmi. Disertační práce. VUT v Brně, 1998.
- Krysztof Czarnecki, Ulrich Eisenecker. Generative Programming: Methods, Tools, and Applications. Addison-Wesley Professional, 2000. ISBN-13: 978-0201309775

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 2020
Datum odevzdání: 30. července 2021
Datum schválení: 11. listopadu 2020

Abstrakt

Táto práca se zaoberá transformáciou modelu systému popísaného objektovo orientovanou Petriho sieťou na sekvenčný diagram, ktorý umožňuje spätné mapovanie aktivít sekvenčného diagramu do popisu modelu. K riešeniu je použitá diskrétna simulácia modelu popísaná jazykom PNtalk. Výsledkom práce je plne automatický generátor, ktorý pomocou dát z diskrétnej simulácie vygeneruje sekvenčný diagram. Pre generátor je dôležitá minimálna strata informácie zo simulácie a prezentácia validného sekvenčného diagramu.

Abstract

This thesis focuses on transformation model described by object oriented Petri net to sequence diagram, which allows reverse mapping to original description. As a solution is used discrete simulation of model described in PNtalk language. Result of work is fully automated generator, which generates sequence diagram according to gathered data from simulation. Generator aims for minimal loss of relevant information and presenting valid sequence diagram.

Kľúčové slová

objektovo orientované Petriho siete, sekvenčný diagram, diskrétna simulácia, softvérové inžinierstvo, gRPC, JavaFX, TornadoFX, Docker

Keywords

object oriented Petri nets, sequence diagram, discrete simulation, software engineering, gRPC, JavaFX, TornadoFX, Docker

Citácia

KELEMEN, Erik. *Generování sekvenčních diagramů z modelů Petriho sítí.* Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Kočí, Ph.D.

Generování sekvenčních diagramů z modelů Petriho sítí

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Phd. Radka Kočí. Ďalšie informácie mi poskytli Tomáš Lapšanský ako konzultant práce na ktorú som priamo nadviazal. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Erik Kelemen
2. augusta 2021

Poďakovanie

Ďakujem vedúcemu práce Ing. Radkovi Kočí Phd. za vrelý prístup a čas venovaný vedeniu práce.

Obsah

Zoznam skratiek

OOPN Objektovo orientované Petriho siete (Object-oriented Petri Nets)
PT Miesto/prechod (Place/Transition)
CE Condition-Event
UML Unified Modeling Language
IT Informačné technológie (Information Technology)
JVM Java Virtual Machine
UI Užívateľské rozhranie (User Interface)
SW Software
SPN Stochastická Petriho sieť
GSPN Generalizovaná stochastická Petriho sieť

Kapitola 1

Úvod

Úspešnú realizáciu projektu na poli informačných technológií predchádza mnoho úskalí. V tak turbulentnej dobe, akou je tá dnešná sa zadania práce často menia a samotné zadanie býva nekompletné či zavádzajúce. Na projektoch pracuje viac ľudí v snahe ho urýchliť. Je však bežným javom, že zainteresované strany sa spolu nezhodnú, či omylom interpretujú zadanie rozdielne. Z pohľadu jednotlivca je ťažké udržať si prehľad o celom projekte, mať prehľad o dielčich komponentách. Dnešné systémy sú príliš rozsiahle a plné zákerných detailov, ktoré môžu byť zle interpretované alebo kompletne vynechané. Preto udržanie si prehľadu môže stroskotať bez patričnej pomoci.

V roku 2019 bola zverejnená správa inštitútu projektového menežmentu (anglicky Project Managemenet Institute, skratkou PMI [1]), do ktorej prispelo 4455 praktikantov projektového manažmentu na základe svojich praktických skúseností. Z nich najväčšiu časť tvorili práve odborníci z odvetvia IT. Report monitoroval obdobie projektov odštartovaných v časovom rámci 12 mesiacov.
Ako 5 najčastejších príčin zlyhania projektov respondenti uviedli: [2] zmenu priorít organizácie(39 %), zmena projektových cieľov(37 %), nepresne definované požiadavky(35 %), neadekvátna vízia(29 %) a slabá komunikácia(29 %) [5].

Ako vidno zo štatistík komunikácia stále predstavuje dosť veľký kameň úrazu (podieľ nepresne definovaných požiadaviek a slabej komunikácie zasiahlo viac ako 35 % projektov).

Práve pri riešení týchto dvoch problémov nám môžu pomôcť takzvané modelovacie jazyky. Je dôležité ujasniť si myšlienku a význam časti systému na ktorom pracujete naprieč celým vývojovým tímom. Najlepšie by bolo zvoliť modelovací jazyk tak, aby mu rozumeli aj menej technicky zdatní účastníci projektu. Pre všetkých ľahko pochopiteľné sú bezpochyby grafické reprezentácie pohľadov na modelovaný systém. Takéto diagramy nevyžadujú žiadne vyššie vzdelanie na ich pochopenie, navyše obraz je často ľahšie uchopiteľný ako písaný text. V praxi kreslenie diagramov zaberie určitý čas, ktorý by sa mohol využiť efektívnejšie. Preto sa tento čas investuje zvyčajne len do konštrukcie diagramov pre tie najzákernejšie scenáre chovania systému. Ďalšiou nevýhodou je možná chyba ľudského faktoru. Aj ten sebelepší odborník na vyvíjaný sytém môže pochybiť.

Predsavme si však, že by sme dokázali z modelu systému v akomkoľvek stave a bez investície času, či námahy, vygenerovať graficky reprezentovaný scenár skúmanej aktivity.

[1] Project Managemenet Institute www.pmi.org
[2] Respondenti na túto otázku mohli uviesť viac príčin než jednu.

A to všetko automaticky a neomyľne. Tomuto grafickému pohľadu na časť systému by rozumeli nielen špecialisti z oboru, ale aj technicky menej znalí účastníci projektu. Uľahčila by sa tým komunikácia s užívateľmi, či vlastníkmi projektu. Takýto generátor by otvoril dvere novým možnostiam pri špecifikácii požiadaviek systému, analýze a návrhu systému. Pri opravách nechceného chovania systému by sme mohli demonštrovať chovanie zaznamenané diagramom pred opravou a po nej, čo by urýchlilo validáciu. Vývojové tímy, by sa ľahšie zorientovali v komponentách, ktoré vyvíjal iný tím a mnoho iných výhod.

Cieľom tejto práce je práve zostrojiť generátor, ktorý z modelu systému vygeneruje diagram zachytávajúci scenár chovania pri ľubovoľnej aktivite. Práca preskúma možnosti simulácie modelu, ktorá poskytne informácie k transformácii modelu na diagram. Nasledujúci krok po transformácii bude namapovať zobrazené aktivity späť do popisu modelu.

Na tomto poli nie je práca prvá v poradí, už v minulosti boli podobné pokusy. Napríklad executive UML (xUML), sa vydal opačnou cestou a dovoľuje automaticky generovať spustiteľné a simulovateľné prototypy z diagramov [21]. Tento prístup však nepodporuje návrat k modelu. Akékoľvek zmeny v prototype sa nedajú previesť späť. Niekoľko komerčných nástrojov sa pokúša o vykreslovanie behu procesu vizuálne priamo zo zdrojového kódu. Tento prístup však vykresľuje tok procesu na úrovni kódu. Čo môže byť rovnako neprehľadné ako kód samotný. Práca je preto priekopnícka a snaží sa neduhy doterajších riešení eliminovať.

Práca je členená na úvod do tvorby a analýzy scenárov v kapitole 2. Nasleduje utvorenie znalostnej báze o Petriho sietiach i Sekvenčných diagramov pre potreby tejto práce v kapitolách 3 a 4. Najrozsiahlejšie jadro práce tvorí návrh 5 a implementácia 6 generátoru sekvenčných diagramov z modelu OOPN. Nasleduje kapitola o prostriedkoch testovania 6. Poslednou kapitolou je záver zhrňujúci dosiahnuté výsledky v kapitole 8.

Kapitola 2

Tvorba a analýza scenárov v softvérovom inžinierstve

Scenár môže byť chápaný celou radou interpretácií, niektoré z nich sú uplatniteľné v SW inžinierstve. V tejto práci bude pojmom *scenár* označovaná sekvencia aktivít alebo rozhodovací strom viacerých takýchto sekvencií. Vetvy rozhodovacieho stromu reprezentujú alternatívy, či rôzne možnosti chovania systému. Zložitosť scenára je závislá na rozvetvení rozhodovacieho stromu. Scenár môže byť konkrétny či abstraktný [7]. V tejto práci bude uvažovaný výlučne konkrétny, abstrakcia sa zo scenáru odstráni doménovým modelovaním. Viac o tejto oblasti problémov v sekcii 2.2.1 o problematikách pri tvorbe scenárov.

Pri modelovaní systémov uľahčuje analýzu a modelovanie vytvorenie takéhoto scenáru. Scenár poskytuje náhľad na určitú časť systému, väčšinou netriviálnu, a popisuje jej chovanie. Scenáre fungujú lepšie pre niektoré typy systémov, pre iné horšie. Táto kapitola je zameraná na úvod do procesu vývoja systémov modelovaných pomocou scenárov a prípadov užitia.

Rozdielne druhy systémov potrebujú rozdielne modelovacie techniky. Napríklad najdôležitejšie aspekty interaktívnych systémov sú zachytené prípadmi užitia a scenármi. Na druhú stranu rozsiahle dátovo zamerané aplikácie sú vhodnejšie modelované pomocou entitne vzťahového diagramu alebo objektového diagramu. Knižnice na komplexné výpočty sa modelujú ľahšie algoritmami popísaných psuedo-kódom. Navyše špeciálne vlastnosti ako podpora reálneho času, distribúcia či vysoká dostupnosť vyžadujú špecializované modelovacie techniky.

Signifikantné rozdiely rozdelili modelovacie techniky do 3 typov štýlov:

- **Interaktívny štýl**: hlavným aspektom je interakcia medzi entitami, napríklad interakcia medzi komponentami, grafy volaní, toky správ, toky udalostí. Interakcia môže byť modelovaná použitím diagramu prípadov užitia, spolupráce a interakcie.
- **Algoritmický štýl**: hlavný aspekt algoritmického štýlu sú algoritmy vykonávajúce komplexné výpočty na abstraktných dátových typoch. Algoritmy môžu byť špecifikované pseudokódom alebo špecifickou notáciou.
- **Data-centrický štýl**: hlavný aspekt Data-centrického štýlu je štruktúra dát ako napríklad v databázovom modely. Štruktúra môže byť popísaná entitne vzťahovým alebo objektovým diagramom.

V nasledujúcich sekciách práce budú modelovacie techniky Algoritmického a Datacentrického štýlu zámerne opomenuté. Scenáre, ktoré sú predmetom tejto práce patria do interaktívneho štýlu.

2.1 Využitie scenárov

Využitie scenárov je signifikantné práve pri interaktívnom štýle v softvérovom inžinierstve naprieč všetkými fázami vývoja softvéru. Od počiatočných prípadov užitia, môžu byť scenáre použité na definovanie užívateľských požiadaviek a definíciu funkcionality systému. V priebehu modelovania systému môžu slúžiť ako implementačné vzory a taktiež spätne pri vyhodnocovaní chovania scenáre môžu slúžiť pre testy správnosti alebo byť predmetom akceptačných požiadaviek.

2.1.1 Váha scenárov pre zúčastnené strany

Všetky fyzické osoby, ovplyvňujúce vývoj softvéru, môžme pre akýkoľvek systém klasifikovať do 5 skupín. Zobrazené sú na ľavej strane Obr. 6.2. Podstatné je, že každá zo skupín má na systém iný uhol pohľadu. Ničmenej scenáre správneho chovania systému by mali byť chápané rovnako naprieč všetkými stranami.

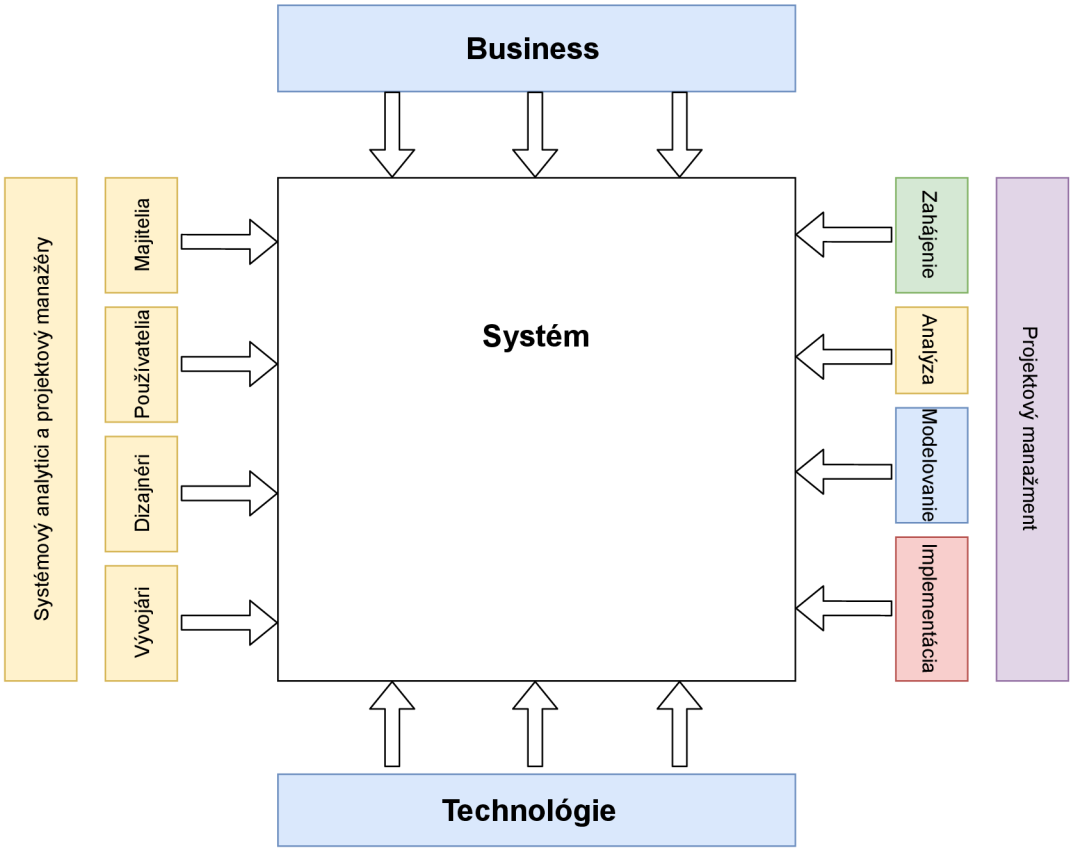
1. **Užívateľia** systému, sú v dnešnej dobe čím ďalej viac technicky vyspelí a ďalšou ich nespornou výhodou je počet, ktorý väčšinou prevyšuje ostatné skupiny. Z ich pohľadu na systém je najdôležitejšia funkcionalita, intuitívnosť používania a o cenu, či profit, na rozdiel od ostatných skupín, nedbajú. Pomocou scenárov dokáže táto početná skupina definovať chcené chovanie, alebo pri validácii poskytnúť nový scenár ako by sa systém mal zachovať na rozdiel od aktuálnej implementácie.

2. **Majitelia** projektu, ktorých môže byť viac než jeden väčšinou riešia projekt z pohľadu financií - aký bude profit, či benefity. Môžu zrušiť, či pozmeniť scenár chovania do akejkoľvek miery.

3. **Systémový analytici a projektový manažéri** sú špecialistami na analýzu a modelovanie. Poskytujú ostatným skupinám poradenstvo a sú akýmsi mostom pri akomkoľvek komunikačnom šume vznikajúcom napríklad medzi menej technicky zdatnými majiteľmi projektu a vývojármi. Distribúcia naprieč zúčastnenými stranami a dolaďovanie špecifikácie netriviálnych scenárov je zásadná.

4. **Dizajnéri** zodpovedný za modelovanie architektúry systému z ich uhlu pohľadu riešia správnu voľbu technológie pre systém. Tendenciou je mať špecializovaného návrhára pre každú časť zvlášť, preto do tejto skupiny patria databázoví administrátori, sieťoví architekti, bezpečnostní experti a mnohí ďalší. Pri tejto voľbe musia zohľadniť scenáre chovania, nielen aktuálne ale nechať systému priestor na rozvoj ak by pribudli nové scenáre. Zvolená architektúra musí tieto scenáre podporovať z technického hľadiska.

5. **Vývojári**, ktorý majú za úlohu celý systém zkonštruovať podľa návrhu softvérového dizajnéra riešia hlavne detaily implementácie. V menších firmách sú dizajnéri a vývojári tí istí ľudia, no vo väčších sú tieto úlohy často oddelené. Scenáre správneho chovania pre nich slúžia ako zadanie implementácie.



Obr. 2.1: Aspekty ovplyvňujúce vývoj systému [27].

2.1.2 Uplatnenie scenárov v procese vývoja

Je zrejmé že väčšina organizácií, pohybujúcich sa vo vodách softvérového inžinierstva, bude mať vlastný formálne definovaný proces vývoja softvéru alebo sadu krokov, podľa ktorých by sa mal systém vyvíjať. Určite sa budú tieto metodológie od seba diametrálne odlišovať pre jednotlivé organizácie. Avšak, všetky metódy riešenia problému sa dajú zovšeobecniť na kroky, ktoré sú spoločné [1]:

1. **Identifikovať problém**, akokoľvek jednoducho prvý krok môže znieť, opak je pravdou. Zadania sú často nejasné a ciele systému preto nejednoznačné. Rozsah práce môže byť podcenený, s čím ide ruka v ruke aj časový plán a rozpočet.

2. **Analyzovať a porozumieť problému**. Druhý krok poskytuje projektovému tímu hlbšie porozumenie systému, vyžaduje spoluprácu so zúčastnenou stranou 2.1.1.

3. **Identifikovať požiadavky a očakávania riešenia**, ktoré kľadú nároky obchodu či funkcionálna stránka vyžadovaná užívateľmi.

4. **Identifikovať alternatívne riešenia** a zvoliť najvhodnejšiu cestu. Pri výbere zohráva rolu rozpočet (finančný i časový), predispozície relizačného tímu a uprednostnené ciele.

[1] Pri tvrdení predpokladáme, že proces vývoja bude dodržiavať prístup techník riešenia problému

5. **Navrhnúť zvolené riešenie**, pomocou jednou z metód modelovania systémov.
6. **Implementovať zvolené riešenie** za pomoci vymodelovaného návrhu. Náročnosť implementácie je nepriamo úmerná kvalite návrhu.
7. **Vyhodnotiť výsledok.** Na záver je nutné objektívne zhodnotiť výsledky v zmysle splnenia cieľov. Pri nesplnení sa môžeme vrátiť ku kroku 1 a 2.

Na obrázku 6.2 je na pravej strane zobrazený pohľad procesu vývoja, ktorý bol kvôli jednoduchosti zredukovaný len na 4 fáze. Táto zjednodušená varianta postačuje na pokrytie problematiky analýzy a modelovania systému. Inicializácia je fáza predchádzajúca analýze a implementácia je niečo, čo prirodzene nadväzuje po úspešnom návrhu systému. Jednotlivé kroky zovšeobecneného riešenia problémov do fáz vývoja je v tabuľke 2.1.

Zjednodušený vývojový proces	**Kroky zovšeobecneného riešenia problémov**
Zahájenie	1. Identifikovať problém
Analýza systému	2. Analyzovať a porozumieť problému 3. Identifikovať požiadavky a očakávania riešenia
Modelovanie systému	4. Identifikovať alternatívne riešenia a zvoliť najschodnejšiu cestu 5. Navrhnúť zvolené riešenie
Implementácia systému	6. Implementovať zvolené riešenie 7. Vyhodnotiť výsledok

Tabuľka 2.1: Namapovanie 7 krokov zovšeobecneného postupu do jednotlivých fáz zjednodušeného vývojového procesu.

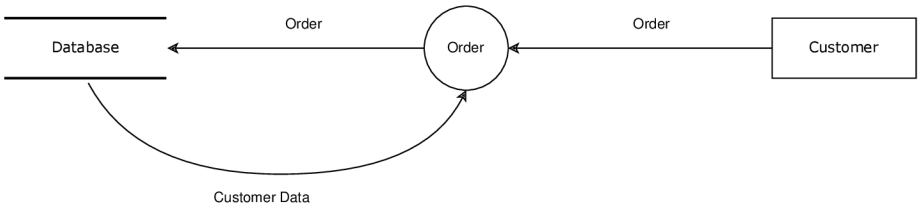
1. **Zahájenie** Identifikácia problému využíva scenáre v popise problému pre zvyšné zainteresované strany, aby sa vývoj mohol pohnúť do ďalšej fáze.

2. **Analýza systému** patrí taktiež k raným fázam vývoja systému. Jedná sa o štúdovanie systému a jeho súčastí. V sekcii 2.1.2 sme zaradili analýzu systému na svoje miesto v procese vývoja za fázu zahájenia projektu a pred fázu návrhu systému. Z toho vyplýva, že analýza je prerekvizita k úspešnému návrhu systému. Vždy je motivácia nazbierať čo najviac relevantných informácii o systéme, alebo aspoň dostačujúcich pre následnú fázu návrhu systému. V predchádzajúcej sekcii 2.1.1 bola opísaná zúčastnená strana podieľajúca sa na analýze a ciele analýzy, no samotné prístupy analýzy softvéru boli len nonšalantne opomíjané. V tejto sekcii budú rozobrané vybrané prístupy analýzy systému.

   Analýza je hlavne o riešení problému, a keďže riešiť problém sa dá viacerými prístupmi, asi nikoho neprekvapí, že aj prístupov k analýze systému bude viac.

   - **Modelom riadená analýza** – či sa jedná o štruktúrovanú analýzu, informačné inžinierstvo alebo objektovo-orientovanú analýzu, všetky tri príklady patria do skupiny modelom riadených analýz. Tento prístup používa na vyjadrovanie všetkým zrozumiteľnú formu scenárov. Sú ňou obrázky na opis problémov, požiadaviek a riešení v systéme. Takou grafickou interpretáciou môžu byť napríklad vývojové diagramy, štrukturované grafy a iné schémy. Tento druh analýzy študuje systém, aby zistil jeho chovanie a jeho výstupom sú grafické reprezentácie popisujúce chovanie systému (scenáre). Takáto grafická interpretácia môže odhaliť úskalia v projekte a prakticky rovno načrtnúť návrh implementácie v neskorších fázach.

     (a) **Štruktúrna analýza**, ako jedna z tradičných foriem analýzy zo 70. rokov používaná do dnes je zameraná na tok dát a analyzuje systém z pohľadu procesov.



Obr. 2.2: Data flow diagram.

     (b) **Informačné inžinierstvo** ako ďalší tradičný prístup na rozdiel od sledovania dát v procese, sleduje štruktúru uloženia dát naprieč systémom.

Obr. 2.3: Model informačného inžinierstva.

     (c) **Objektovo-orientovaný prístup** sa odlišuje od tradičných prístupov, ktoré sa zámerne snažili oddeliť dáta a procesy. Objektovo-orientovaný prístup zlúčil dáta a procesy do objektov, ktoré majú uložené atribúty objektov (dáta)

a metódy objektov, ktoré vykonávajú operácie nad týmito dátami (procesy nad dátami). Objektová orientácia sebou prináša celú sadu nástrojov na modelovanie tzv. jazyk UML (Unified Modeling Language). Jazyku UML je venovaná celá sekcia 2.2.2.

- **Prototypovanie** Okrem modelovo orientovanej analýzy môžme skúmať možnosti systému štýlom *„Vieme, čo chceme, keď to uvidíme“*. Tento prístup spočíva vo vytváraní funkčných, ale neúplných prototypov výsledného systému, ktoré sa postupnou iteráciou dostanú k požadovanému systému. Slovom neúplných myslíme prototyp bez funkčnej kontroly vstupov, chybovými hláškami a podobne. Prototyp by mal však obsahovať jadro funkcionalít systému. Tento prístup neštuduje model a nesnaží sa vytvoriť diagrami, ktoré by ho opísali. Namiesto toho, vďaka rýchlej prototypizácii sa pokúša systém implementovať. Na týchto pokusoch potom stojí analýza systému a odhaľovanie chýb, či nezrovnalostí. Scenáre sú v tomto prípade využité vo vágnom stave, doplňané o rozhodovacie vetvy postupom času.

3. **Modelovanie systému** Na modelovanie nie len objektovo-orientovaných modelov slúži sada nástrojov UML. Na základe analýzy sa vyhotovia diagramy popisujúce model. Viac o tvorbe scenárov pomocou UML v nadchádzajúcej sekcii 2.2.2 o behaviorálnom modelovaní.

4. **Implementácia** Tu slúžia scenáre ako predloha pre implementáciu a pri overovaní správosti sa tiež môžme obrátiť na scenáre.

2.2 Tvorba scenárov

V tejto sekcii sa popíše tvorba scenárov pre systémy u ktorých je rozumné aplikovať modelovaciu techniku z interaktívneho štýlu.

2.2.1 Doménové modelovanie

Pred zostavovaním akéhokoľvek scenáru je nutné definovať pojmy a ich skutočný význam, aby bol pre všetkých rovnaký.

Doménové modelovanie poskytuje slovník cudzích slov používaných v danej doméne (reklamný systém, knižný systém apod.). Tento glosár, potom slúži pre objasnenie pojmov použitých v scenári [10].

Tvorba scenárov a prípadov užitia bez vytvorenia doménového modelu má za následky nezrozumiteľnosť.

2.2.2 Tvorba scenárov pomocou diagramov UML

Ako široko využiteľný jazyk UML, spomínaný už v sekcii 2c, môže byť tiež okrem iného použitý na tvorbu scenárov.

Pojem	Význam
Impresia	Zobrazenie reklamy v prehliadači
Partnerský web	Web poskytujúci reklamný priestor
Inzerent	Inzerujúca právnická, či fyzická osoba a zadávateľ reklám
Preklik	Prekliknutie reklamy na stránke partnerov na web inzerenta
Výdaj reklamy	Aktívne obdobie zobrazovania reklám na stránkach
Kampaň	Časovo ohraničený interval výdaja reklám
Zostava	Časovo ohraničený interval v rámci kampane s neprázdnou množinou reklám

Tabuľka 2.2: Príklad neúplného glosára pre reklamný systém.

Čo je to UML?

UML je vizuálny jazyk pre modelovanie a komunikáciu o systémoch pri použití diagramov a doplňujúceho textu [6].

UML je skratka pre Unified Modeling Language, tj. Zjednotený modelovací jazyk. Každé z týchto troch slov hovorí o významnom aspekte UML. Nasledujúce sekcie hovoria o týchto aspektoch v obrátenom poradí.

1. **Jazyk** - UML je jazyk pre špecifikovanie, vizualizáciu, konštrukciu a dokumentáciu dielov procesu vývoja a udržovania systému.
2. **Model** je reprezentácia subjektu. *Model* zachytáva sadu vlastností subjektu. Pri modelovaní je jednoduché stratiť sa v množstve zachytenných vlastností, preto sa UML sústredí len na relevantné časti pri modelovaní systému.
3. **Zjednotenie** - UML je štandardizované organizáciou Object Management Group (OMG)[2]. UML teda prináša dohromady tie najlepšie inžinierske praktiky v priemysle.

Ciele UML:

- Pripravený na použitie
- Expresívny
- Jednoduchý
- Presný
- Rozšíriteľný
- Implementačne nezávislý

[2] https://en.wikipedia.org/wiki/Object_Management_Group

- Procesne nezávislý

Tým, že je UML pripravené na použitie, jednoduché, expresívne a napriek tomu presné je možné ho ihneď aplikovať na vývoj projektov.

Rozšíriteľnosť jazyka povoľuje definovať nové koncepty. Nezávislosť na implementačných technológiách a modelovacích procesoch zo sekcie o uplatniteľnosti scenárov 2.1.2 ho činí široko využiteľným.

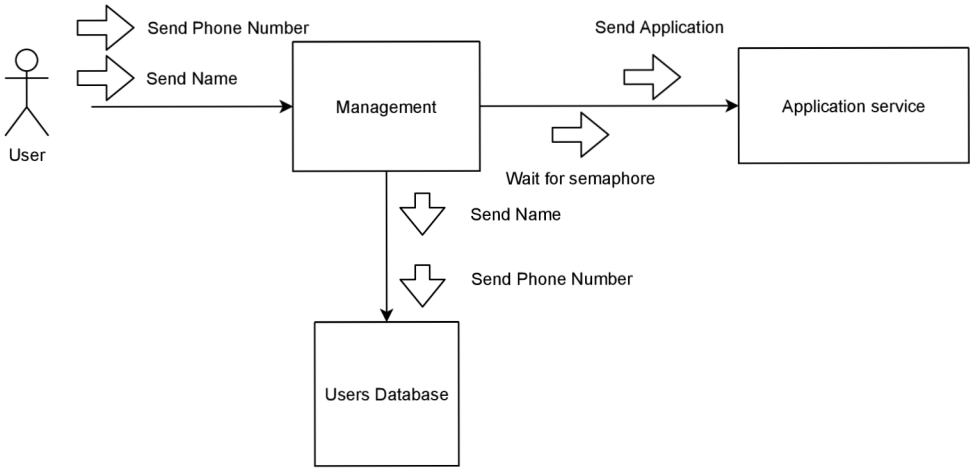
Modelovacie prístupy:

1. **Štruktúrne** modelovanie pomáha v porozumení stavebných prvkov systému a funkcionality systému.
2. **Behaviorálne** modelovanie pomáha v porozumení ako elementy spolu komunikujú a spolupracujú, aby systém fungoval správne. Do behaviorálneho modelovania môžme zaradiť práve tvorbu scenárov.

Behaviorálne modelovanie

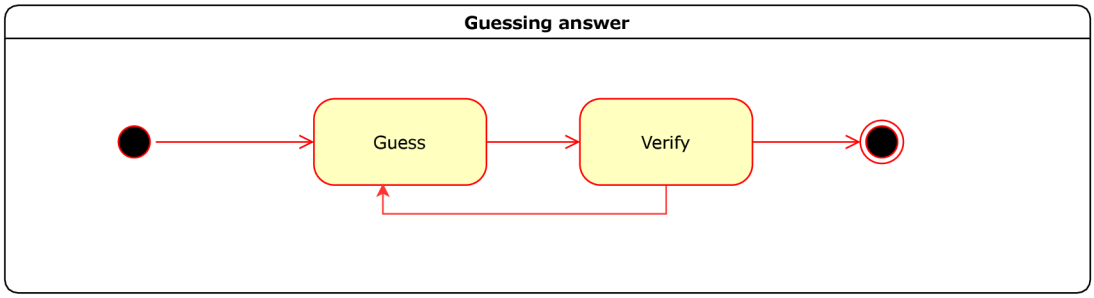
V tejto sekcii budú predstavené už konkrétne grafické reprezentácie scenárov, modelované pomocou behaviorálneho modelovania.

1. **Sekvenčný diagram**, zobrazuje interakciu elementov v priebehu času. Bude podrobne prebratý vo svojej vlastnej kapitole 4.
2. **Diagram spolupráce** zobrazuje ako elementy komunikujú v čase a aké majú väzby medzi sebou.



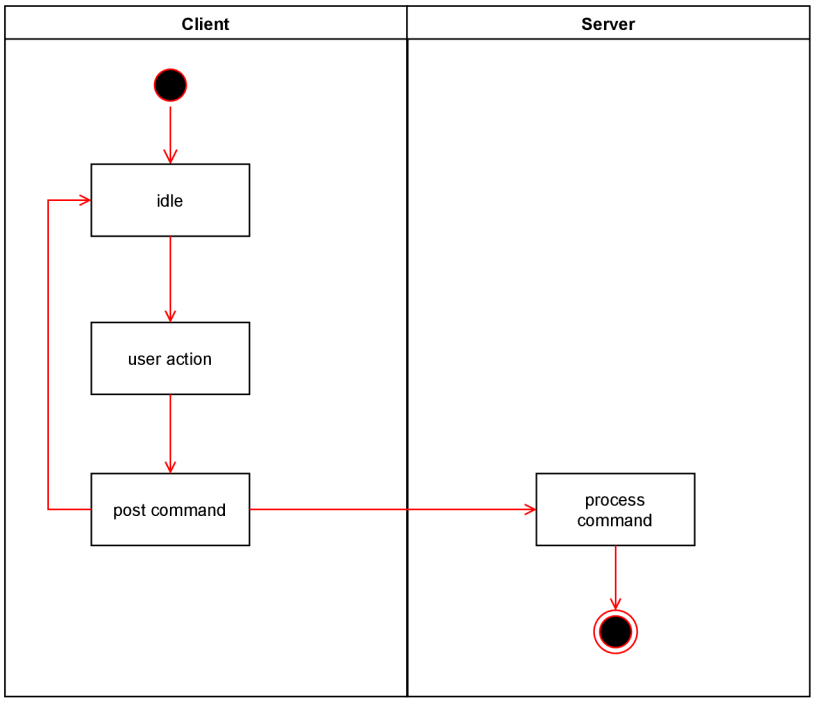
Obr. 2.4: Diagram spolupráce.

3. **Stavový diagram** zobrazuje životný cyklus elementu.



Obr. 2.5: Stavový diagram.

4. Diagram aktivít zobrazuje aktivity a funkciu elementov.



Obr. 2.6: Diagram aktivít.

Pre potreby práce bude ďalej venovaná pozornosť už len sekvenčným diagramom. Diagram aktivít, spolupráce a stavový diagram sú spomenuté len na dotvorenie obrazu o behaviorálnom modelovaní.

2.3 Problematiky scenárov

Pri tvorbe scenárov sa môže produktom stať neúplný, nejednoznačný, či dokonca nesprávny scenár. Takýto defekt u scenára sa prenesie aj na analýzu, či implementáciu v ktorej by bol scenár použitý. Existujú však techniky a metodológie ako najčastejším problematikám scenárov predchádzať.

1. **Nezrozumiteľnosť** - Scenáre by mali mať pokiaľ možno len jednu interpretáciu. To dosiahneme presne definovaným významom pojmov použitých v scenári. Problematika a jej riešenie je popísané v sekcii 2.2.1

2. **Neúplnosť** - Scenár popísaný formalismom zo sekcie 2 definoval rozhodovacie vetvy scenára. Pri absencii rozhodovacej vetvy pre validný prechod sa môžme dostať do nedefinovaného stavu. V praxi sa stretneme s pojmom *rainy-day scenáre*, tj scenáre daždivých dní. Ide o scenáre, kde sa počíta aj s katastrofickými okolnosťami. Pre jednoduchosť sa však zvyčajne tvoria scenáre pokrývajúce len najčastejšie prípady a *rainy-days* alternatívy sú často opomenuté. Tento problém vedie k voľnej interpretácii a možnými nezhodami medzi účastníkmi vývoja SW. Dá sa riešiť definovaním všetkých alternatív chovania systému ak je ich konečne mnoho.

3. **Neuskutočniteľnosť** - Na obrázku 6.2 boli popísané aspekty, ktoré ovplyvňujú vývoj softvéru. Okrem účastníkov majú na systém vplyv ešte aspekty businessu a technológie. Ďalším problémom pri tvorbe scenárov je nedostupnosť technológií, ktoré by scenár umožnili realizovať. Naopak prielomy v technológiach poskytujú príležitosť vzniku nových scenárov. Vytvorený scenár môže byť neuskutočniteľný aj kvôli aspektom businessu do ktorých je zahrnutá aj legislatíva. Scenár je napríklad neuskutočniteľný ak je v rozpore s legislatívou [27].

Kapitola 3

Petriho siete

V tejto kapitole je popísaná obecná Petriho sieť a rozšírenia PN, ktoré vedú na varianty Petriho sietí s potrebnými vlastnosťmi pre automatické generovanie sekvenčných diagramov. Od obecnej Petriho siete sa prechádza na objektovo orientovanú Petriho sieť (ďalej len OOPN). OOPN dovoľuje modelovať systémy v súlade s princípami objektovej orientácie. Je prítomná sekcia o paralelizme v Petriho sietiach a v neposlednej rade kapitola rozoberie možné zavedenie času do Petriho sietí.

3.1 Obecná definícia

Ako východziu Petriho sieť pre ďalšie varianty a rozšírania použijeme sieť definovanú ako PT-sieť (Place/Transition Net), je zovšeobecnením jednoduchšieho modelu CE-sietí (Condition-Event Net).

Poznámka 3.1.1. CE-sieť na rozdiel od PT umožňuje do miest ukladať len jednu značku, miesta v tejto sieti nadobúdajú len booleovských hodnôt. Prechody CE-sietí sú usutočniteľné len za podmienky, že sú vstupné podmienky pravdivé a výstupné nepravdivé (hodnota 0 vo všetkých výstupných miestach). Obsah práce nevyžaduje uchopenie teórie až do hĺbky CE-sietí, preto vychádzame z tohto jej zovšeobecnenia.

Definícia 3.1.1. Petriho sieť je štvorica $N = (P_N, T_N, PI_N, TI_N)$, kde

1. P_N je konečná množina miest.
2. T_N je konečná množina prechodov, $P_N . \cap T_N$
3. $PI_N : P_N \longrightarrow \mathbb{N}$ je inicializačná funkcia.
4. TI_N je popis prechodov (transition inscription function) definovaných tak, že $\forall t \in T_N : TI_N(t) = (PRECOND_t^N, POSTCOND_t^N)$, kde
   (a) $PRECOND_t^N : P_N \longrightarrow \mathbb{N}$ sú vstupné podmienky (vstupy) prechodu.
   (b) $POSTCOND_t^N : P_N \longrightarrow \mathbb{N}$ sú výstupné podmienky (výstupy) prechodu.

Pre potreby grafickej reprezentácie Petriho siete definujeme množinu hrán.

Definícia 3.1.2. Množina hrán Petriho siete A_N

$$A_N \subseteq (P_N \times T_N) \cup (T_N \times P_N)$$

pričom platí, že

$$\forall (p,t) \in (P_N \times T_N)[(p,t) \in A_N \iff PRECOND_t^N(p) > 0]$$

$$\forall (t,p) \in (T_N \times P_N)[(t,p) \in A_N \iff POSTCOND_t^N(p) > 0]$$

Definícia 3.1.3. Ohodnotenie hrán je funkcia $W_N : A_N \longrightarrow \mathbb{N}$ pre ktorú platí

$$\forall (p,t) \in A_N \cap (P_N \times T_N)[W_N(p,t) = PRECOND_t^N(p)]$$

$$\forall (t,p) \in A_N \cap (T_N \times P_N)[W_N(t,p) = POSTCOND_t^N(p)]$$

ak $(p,t) \in A_N \cap (P_N \times T_N)$ vravíme, že p je **vstupné miesto** a (p,t) je **vstupná hrana** prechodu t. ak $(t,p) \in A_N \cap (T_N \times P_N)$ vravíme, že p je **výstupné miesto** a (t,p) je **výstupná hrana** prechodu t.

Stav systému Petriho siete je určený rozmiestnením značiek v miestach.

Definícia 3.1.4. Značenie siete N je funkcia $M : P_N \longrightarrow \mathbb{N}$. Funkcia $M_0 = PI_N$ je počiatočné značenie siete N.

Dynamika Petriho sietí spočíva vo vykonávaní prechodov. Ich uskutočniteľnosť závisí na značeniu siete a naopak. Tieto závislosti popisujú evolučné pravidlá.

Definícia 3.1.5. Evolučné pravidlá

Majme sieť N a jej značenie M.

1. Prechod $t \in T_N$ je **uskutočniteľný** v značení M práve vtedy, keď

$$\forall p \in P_N[PRECOND_t^N(p) \leq M(p)]$$

2. Ak prechod $t \in T_N$ je uskutočniteľný v značeniu M, môže byť **prevedený**, čo zmení značenie M na M', definované ako:

$$\forall p \in P_N[M'(p) = M(p) - PRECOND_t^N(p) + POSTCOND_t^N(p)]$$

Stav systému, popísaného množinou stavových strojov, je určený množinou stavov jednotlivých strojov. Stav (stavová premená) systému je distribuovaný do množiny parciálnych stavov systému. Okamžitý stav systému je definovaný umiestnením značiek do miest naprieč všetkými parciálnymi stavmi.

3.2 Objektovo orientované Petriho siete

Petriho sieť vo všeobecnom zavedení sa nedá využiť pre potreby práce. V diskusii okolo PN bola v práci [25] navrhnutá myšlienka invokačných prechodov, ktorá sa stala základom pre zavedenie objektovo orientovaného prístupu do Petriho sietí. Následne vzniklo viac nadväzujúcich prác na túto tému ako LOOPN++[16], Cooperative Nets [26] či v neposlednej rade PNtalk [13].

3.2.1 Prerekvizity Petriho sietí

Petriho siete sú koncipované ako plošný (neštrukturovaný) model, kde hierarchický aspekt modelovaného systému nie je nijak vyjadrený. Varianty spomenuté v tejto sekcii sa budú zaoberať rozšírením výpočetnej a modelovacej sily nutnej pre prekonanie problému spojeného s plošným statickým modelom a povýšenie modelovacej sily k princípom objektovej orientácie.

Inhibítory

Inhibítory umožňujú testovať počet značiek v mieste a tým dávajú Petriho sietiam výpočetnú silu Turingového stroja a sú teda schopné počítať všetky vyčísliteľné funkcie. Takouto sieťou je možné špecifikovať ľubovoľný algoritmus.

Vysokoúrovňové Petriho siete

Napriek tomu, že sú siete s inhibítormi schopné vyjadriť akýkoľvek algoritmus, modelovanie čo i len prostého vyhodnocovania aritmetických výrazov je príliš zložité a neintuitívne. Dôvodom sú prostriedky, ktoré zahŕňajú len odňatie značiek zo vstupných miest a pridávanie značiek do miest výstupných. *HL-siete* riešia tento problém zavedením konceptu *hranových výrazov*, *prechodovej stráže* a *prechodovej akcie*.

K tomu, aby sme mohli vysvetliť základné koncepty *HL-sietí*, potrebujeme pomocný pojem multimnožina a operáciie s multimnožinami.

Definícia 3.2.1. Majme ľubovolnú neprázdnu množinu E. Multimnožina nad množinou E je funkcia. $x : E \longrightarrow \mathbb{N}$. Hodnota $x(e)$ je počet výskytov (koeficient) prvku e v multimnožine x. Multimnožinu zapisujeme ako formálnu sumu

$$\sum_{e \in E} x(e)'e$$

Množinu všetkých multimnožín nad E označíme E^{MS}. Pre multimnožiny x, y nad E a prirodzené číslo n definujeme:

1. sčítanie:
$$x + y = \sum_{e \in E} (x(e) + y(e))`e$$

2. skalárne násobenie:
$$n`x = \sum_{e \in E} (nx(e))`e$$

3. porovnanie:
$$x \neq y = \exists e \in E[x(e) \neq y(e)]$$
$$x \leq y = \forall e \in E[x(e) \leq y(e)]$$

4. odčítanie:
$$x - y = \sum_{e \in E} (x(e) - y(e))`e$$

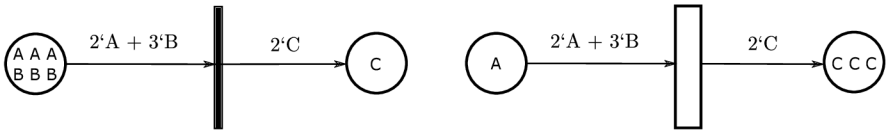
5. veľkosť:

$$|x| = \sum_{e \in E} x(e)$$

Príklad 3.2.1. názorne zápis $2`A + 3`B$ predstavuje multimnožinu s troma výskytmi prvku a a štyrmi výskytmi prvku b.

Poznámka 3.2.1. Koeficient 1 obvykle vynechávame, tj. napríklad zápis c predstavuje rovnakú multimnožinu ako zápis $1`c$.

Takúto Multimnožinu môžeme konceptom **hranových výrazov** priradiť k hranám vstupným ako aj výstupným. Názorná ukážka je na Obr. 3.1.

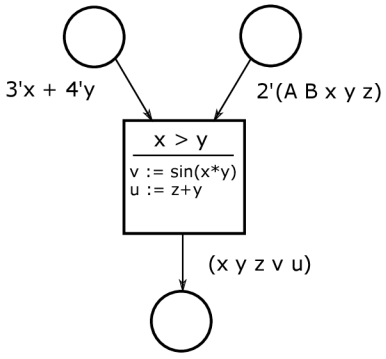


(a) Stav pred uskutočnením prechodu. (b) Stav po uskutočnení prechodu.

Obr. 3.1: Hranové výrazy na vstupnej aj výstupnej hrane.

Každému prechodu je možno priradiť **stráž prechodu**, booleovský výraz, ktorý musí byť splnený pre uskutočnenie prechodu. Je možné určité naviazanie premenných vo výrazoch na vstupných hranách a rovnako v stráži prechodu. Príklad strážneho výrazu „$x > y$“ aj s naväzovaním premenných je na Obr. 3.2.

Pre sugestívnejší zápis dovoľuje k stráži prechodu pridať **akciu prechodu**, odlišujúcu výpočty, ktoré se realizujú pri vykonávaní prechodu, od tých, ktoré se realizujú pri zisťovaní uskutočniteľnosti prechodu.



Obr. 3.2: Príklad stráže prechodu a akcie prechodu.

Hierarchické Petriho siete

Systémy, ktoré majú viac štruktúrne zhodných subsystémov je vhodné modelovať hierarchickými Petriho sieťami. Musí však byť dopredu známy ich počet. Nesmú vznikať dynamicky. Táto varianta PN zavádza množinu pomenovaných stránok obsahujúcich podsiete systému. PN v stránke obsahuje miesta, takzvané **porty**, určené na prepojenie s inou PN. Uzly v nadradenej PN, ktoré sa napoja na porty podriadenej PN v stránke sa nazývajú **sokety** [25].

Definícia 3.2.2. Substitučný prechod reprezentuje inštanciu stránky a definuje napojenie soketov na porty.

Poznámka 3.2.2. Prínos hierarchických PN nie je nijak markantný, predstavujú len akési makrá dobre známe z iných programovacích jazykov. Ich význam je zrejmý až pri dynamickej inštanciácii stránok.

Dynamické vytváranie podsietí Petriho sietí

V sekcii 3.2.1 venovanej hierarchickým PN bolo ukázané statické vytváranie podsietí do nadradenej PN. K dynamickému inštancovaniu sa využíva syntakticky podobný koncept stránok a prechodov. Stránky budú navyše obsahovať ukončovacie miesto či prechod označené kľúčovým slovom exit.

Definícia 3.2.3. Invokačný prechod bude mať rovnakú notáciu ako substitučný prechod z definície 3.2.2, jeho sémantika bude však odlišná.

1. Vykoná sa vstupná časť invokačného prechodu, čo spočíva v odobraní značiek zo vstupných miest invokačného prechodu a vo vytvorení novej inštancie špecifikovanej stránky. Do vstupných miest stránky sú umiestnené značky, odobrané zo vstupných miest invokačného prechodu.

2. Invokovaná sieť potom beží nezávislo na ostatných inštanciách sietí v systéme pokiaľ v nej nedôjde k nejakej ukončujúcej udalosti. Takou udalosťou môže byť ukončenie prechodu alebo umiestnenie značky do ukončujúceho miesta.

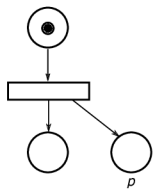
3. Dokončenie invokácie spočíva vo vykonaní výstupnej časti invokačnęho prechodu. Značky z portov, priradených výstupným miestam invokačného prechodu sú skopírované do výstupných miest invokačného prechodu. Dôjde k zrušeniu inštancie invokovanej siete.

Invokačný prechod odpovedá volaniu procedúry, čo uzatvára nároky na modelovanie pomocou PN v súlade s objektovo orientovanými princípami.

3.3 Paralelizmus v Petriho sietiach

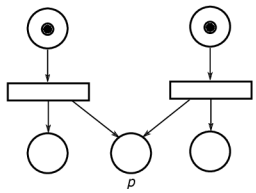
Paralelizmus môže byť prenesený do Petriho sietí viacerými spôsobmi.

1. Predstavme si príklad dvoch triviálnych konkurenčných procesov. Každý môže byť reprezentovaný Petriho sieťou, nech $p \in P_N$ a nech miesto p je zdieľané oboma procesmi.



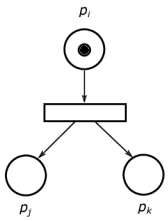
Obr. 3.3: Ukážkový proces.

Jednoducho **zložením** oboch **sietí** dostaneme jednu. Táto zložená sieť na Obr. 3.4 inicializuje dve značky, pre každý proces jednu, takáto inicializácia vo výpočetných systémoch možná nie je, preto je tento spôsob pramálo využiteľný.

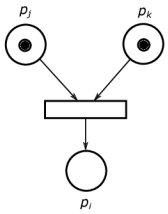


Obr. 3.4: Ukážka zloženia dvoch sietí. V praxi neužitočné.

2. Ďalší prístup je zvážiť ako sa k paralelizmu pristupuje vo výpočetných systémoch. Niekoľko návrhov je schodných. Jeden z najjednoduchších zahŕňa operácie **FORK** a **JOIN**. Operácie boli pôvodne navrhnuté Jackom Dennisom a Earlom Van Hornom v roku 1966[18]. Ich prevedenie do Petriho siete je nasledovné:

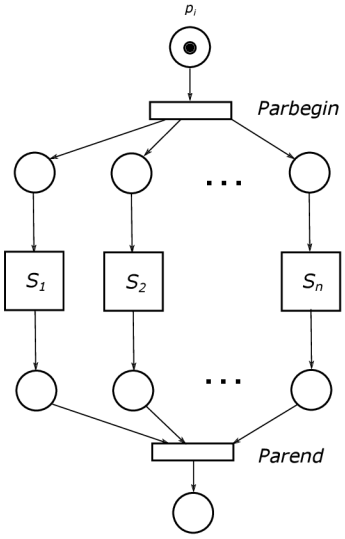


Obr. 3.5: Operácia FORK vykonaná v mieste p_i vytvorí proces v miestach p_j a p_i.



Obr. 3.6: Operácia JOIN vykonaná za koncovými miestami procesov p_j a p_k ich spojí a pokračuje v mieste p_i.

3. Iný návrh zavedenia paralelizmu je riadiaca štruktúra **parbegin** a **parend** [10]. Koncept navrhnutý Djikstrom má všeobecnú formu *parbegin* $S_1; S_2; \ldots S_n$ *parend*, kde S_i predstavuje výraz. Význam *parbegin*|*parend* štruktúry je vykonať každý výraz $S_1; S_2; \ldots S_n$ paralelne. Prevedenie v Petriho sieti je na Obr. 3.7.



Obr. 3.7: Riadiaca štruktúra *parbegin* a *parend* v Petriho sieti.

3.4 Čas v Petriho sietiach

V predchádzajúcich variantách Petriho sietí spomenutých v sekcii 3.2.1 sa čas nijako nezohľadňoval. Konceptu času sa pôvodne zámerne vyhýbalo v originálnej práci C.A.Petriho [22], kvôli efektu, ktoré malo časovanie na chovanie Petriho sietí. Asociácia časových obmedzení na aktivity reprezentované v modeloch alebo systémoch Petriho sietí mohli zabrániť istým prechodom k uskutočneniu. Zničili by tak dôležitý predpoklad, že všetky možné chovania reálnych systémov sú reprezentované štruktúrov Petriho siete [19].

Koncept času sa dostal do popredia pri popisovaní reálnych aplikácií. V skutočnosti je pravdou, že v oblastiach ako hardvér, komunikačné protokoly a softvérová systémová analýza je časovanie podstatný faktor.

V posledných tridsiatich rokoch výzkumníci definovali viacero možností ako zaviesť časovanie do Petriho sietí [24, 12, 23, 20]. Spôsoby sú značne ovplyvnené tým, v akom obore boli uplatnené. Ďalej môžu byť riešenia časového rozšírenia rozdelené na *deterministické* alebo *stochastické*. U deterministických časových prechodoch je chovanie vždy dopredu známe. Prechody sú časované konštantnými oneskoreniami kdežto u stochastických je istá forma náhody.

Stochastické Petriho siete

Stochastická Petriho sieť (SPN) je PT-sieť zo sekcie 3.1.1 rozšírená o $\Lambda = (\lambda_i, ..., \lambda_m)$, kde λ_i je exponenciálne rozložená prechodová miera. Rozloženie náhodnej premennej χ_i oneskorenia prechodu t_i je daná

$$F_{\chi_i}(x) = 1 - e^{-}\lambda_i x \tag{3.1}$$

V SPN sú všetky prechody v T časované [8].

Generalizované Stochastické Petriho siete

Generalizovaná stochastická Petriho sieť (GSPN) vracia do Petriho siete okamžité prechody a vytvára tak dve skupiny prechodov - *časované* a *okamžité*. Pri časovaných prechodoch je známy pomer exponenciálnej distribúcie časového oneskorenia prechodu vychádzajúci z rovnice 3.1 a pri okamžitých je definovaná váha prechodu na určenie priority.

GSPN je štvorica (PN, T_1, T_2, W), kde:

- PN je PT sieť $= (P, T, PI, TI, M_0)$ zo sekcie 3.1.1, ktorá slúži ako základ pre nadstavbu stochastických princípov.
- $T_1 \subseteq T$ je množina časovaných prechodov, $T_1 \neq \emptyset$
- $T_2 \subset T$ značí množinu okamžitých prechodov, pričom $T_1 \cap T_2 = \emptyset$, $T = T_1 \cup T_2$
- $W = (w_i, ..., w_{|T|})$ je množina pre každý prechod, $w_i \in \mathbb{R}^+$
  - reprezentuje pomer exponenciálnej distribúcie časového zpozdenia prechodu, ak $t_i \in T_1$.
  - reprezentuje váhu prechdu, ak $t_i \in T_2$.

Ak platí $T_2 = \emptyset$, tak GSPN odpovedá SPN z predchádzajúcej sekcie. Čiže platí $SPN \subset GSPN$.

3.5 Jazyk PNtalk

V sekcii 3.2 boli vymenované rôzne implementácie OOPN, keďže práca bude využívať PNtalk ako najschodnejšiu variantu pre účely práce, je mu v tejto sekcii venovaná dodatočná pozornosť. Medzi nevýhody použitia jazyka PNtalk patrí jednoznačne absencia zavedenia času (viz. sekcia 3.4).

3.5.1 Termy

Termy sú najjednoduchšie výrazy PNtalku. Patria sem:

1. *Literály*
   (a) Čísla sú objekty, ktoré reprezentujú číselné hodnoty a reagujú na zprávy, ktoré požadujú výsledky matematických operácií. Literál, reprezentujúci číslo je sekvencia číslic, ktorej môže predchádzať znamienko mínus a môže za ňou nasledovať desatinná bodka a ďalšia sekvencia číslic. Tiež možno použiť notáciu s exponentom, pomocou znaku *e*.
   Príklady čísiel sú `5, 456, -25, 0.005, -12.0, 1.345e5, 0.33e-20`.
   (b) Znaky sú objekty, reprezentujúce jednotlivé symboly abecedy. ich literály sa používajú so symbolom dolára, Napríklad `$a, $B, $+, $$, $7`.

(c) Reťazce sú objekty reprezentujúce sekvenciu znakov. Reťazce reagujú na správy požadujúce prístup k jednotlivým znakom a porovnanie s inými reťazcami. Literál reprezentujúci reťazec je sekvencia znakov uzavretá v apostrofoch. Apostrof vo vnútri reťazca musí byť zdvojený. Príklady reťazcových literálov sú `'abcd'`, `'can''t'`

(d) Symboly sú objekty typicky používané ako mená. Symbol je reprezentovaný sekvenciou znakov s prefixom `#`, napríklad `#abc`, `#B52`. Je zaručené, že dva rovnako zapísané symboly reprezentujú rovnaký objekt (na rozdiel od reťazcov).

(e) Booleovské konštanty sú reprezentované vyhradenými identifikátormi `true` a `false`.

(f) Nedefinovaný objekt je reprezentovaný vyhradeným identifikátorom `nil`.

2. *Premenné* môžu v priebehu výpočtu reprezentovať rôzne objekty. Ich hodnotu sa dá programovo ovplyvňovať. Sú to identifikátory s malým počiatočným písmenom s tým, že niektoré identifikátory sú rezervované (`true`, `false`, `nil`).

3. *Pseudopremenné* Existujú dve pseudopremenné, `self` a `super`. Ich hodnota závisí od kontextu a nie je programovo ovplyvniteľná.

4. *Mená tried* sú identifikátory s veľkým počiatočným písmenom. Napríklad `Object`, `PN`, `C1` sú validné mená tried. Ide o konštanty. V priebehu výpočtu sa nemenia.

3.5.2 Siete objektov a metód

Miesta a prechody, prepojené hranami tvoria sieť. Siete sú súčasťou špecifikácie tried objektov, komunikujúcich predávaním správ. PNtalk rozlišuje dva druhy sietí:

1. Sieť objektu reprezentuje atribúty objektu (v podobe miest) a jeho vlastnú aktivitu.

2. Sieť metódy špecifikuje reakciu objektu na prichádzajúcu správu. Definícia triedy môže obsahovať niekoľko sietí metód. Sieť metódy pozostáva z miest a prechodov, prepojených hranami, rovnako ako sieť objektu, ale ku každej sieti metódy je priradený vzor správy, ktorej prijatím objektu vyvolá dynamické vytvorenie inštancie siete metódy. Vzor správy je zložený zo selektoru správy a formálnych parametrov. Istá podmnožina miest sietí objektu sú parametrové miesta, ktoré slúžia k predaniu parametrov pri volaní metódy. Ich mená musia odpovedať menám formálnych parametrov vo vzore správy. Každá sieť obsahuje jedno výstupné miesto (pomenované return), ktoré slúži k predaniu výsledku na vyhodnotenie správy volajúcemu objektu.

3.5.3 Synchrónne porty

Súčasťou špecifikácie tried sú okrem siete objektov a metód aj synchrónne porty. Synchrónne porty majú charakter prechodov aj metód. Nie sú sieťou, neobsahujú miesta a prechody. Synchronny port, podobne ako prechod, môže byť prepojený hranami s miestami siete objektu a môže obsahovať stráž. K synchrónnemu portu je pripojený vzor správy na ktorú reaguje, rovnako ako u siete metódy. Môže byť volaný zaslaním správy zo stráže akéhokoľvek prechodu. Synchrónne porty slúžia k testovaniu a prípadnú zmenu stavu. Synchronny port môže byť v danom stave buď uskutočniteľný alebo neuskutočniteľný.

Špeciálnym prípadom synchrónneho portu je **predikát**, ktorý slúži čisto na testovanie a nemení stav.

3.5.4 Konštruktory

Vytvorenie objektu sa v PNtalku realizuje zaslaním správy new príslušnej triede, ktorej inštanciu chceme vytvoriť. Mená tried sú globálne dostupné. Implicitný konštruktor new je pevne zabudovaný do jazyka. Správe new rozumie každá trieda a ako reakciu na ňu vytvorí inštanciu triedy.

Inicializovanú počiatočným značením siete objektu. Pre dodatočnú a prípadne parametrizovanú inicializáciu objektu slúžia špeciálne metódy nazvané (neimplicitné) konštruktory. Syntakticky sú podobné ostatným metódam objektu, ale ako súčasť ich špecifikácie sa objavuje kľúčové slovo construktor.

3.5.5 Trieda

Model v PNtalku pozostáva z množiny tried. Jedna z nich je určená ako počiatočná. Tá je implicitne inštancovaná pri spustení (zahájení simulácie) modelu.

Trieda je zložená zo sietí objektov, množiny sietí metód, konštruktorov a synchrónnych portov.

3.5.6 Dostupné simulátory

Pre jazyk PNtalk vzniklo viac implementácií simulátorov v rôznych implementačných jazykoch. Vývoj prebieha takmer výlučne na univerzite Vysokého učenia technického v Brne. V jazyku C++ existuje implementácia využívajúca prvotný preklad do medzikódu a následnú simuláciu už syntakticky a sémanticky korektného kódu [17]. Existuje taktiež Interprét v Jave [9].

Kapitola 4

Sekvenčné diagramy

Jednou zo štyroch základných modelačných techník UML užívanou hojne pri navrhovaní programových systémov je sekvenčný diagram. Sekvenčný diagram je najbežnejší z kategórie diagramov interakcií a zobrazuje objekty, ktoré sa účastnia v prípade užitia a taktiež zobrazuje správy, ktoré si tieto objekty vymieňajú počas časového intervalu. Diagram je dvojdimenzionálny. Účastníci sú zoradený na horizontálnej ose a časový priebeh je vyjadrený na vertikálnej, kde čas plynie zhora nadol. Ich nespornou výhodou je zobrazovanie aktivity toku správ v časovej postupnosti. To je nápomocné pre porozumenie systémom plynúcim v reálnom čase a komplexným prípadom užitia.

Sekvenčné diagramy môžu byť generické, zobrazujúce všetky možné scenáre pre definovaný prípad užitia. Častejšie sa však stretneme s vypracovaním diagramov pre jednotlivé scenáre v prípade užitia samostatne.

4.1 Komunikácia v sekvenčných diagramoch

Komunikačný mechanizmus prítomný v sekvenčných diagramoch je, že aktívne entity komunikujú priamo, zasielaním správ.

Poznámka 4.1.1. Tu nachádzame konflikt s PT-sieťou v ktorej aktívne entity komunikujú nepriamo, prostredníctvom zdieľaných pasívnych objektov, miestami siete. Mechanizmy sa dajú previesť z jedného na druhý [13].

Sémantika správ je stopa jednoduchej dvojice `<sendEvent, RecieveEvent>`, kde `sendEvent` je udalosť odoslania správy a `recieveEvent` udalosť jej prijatia. Pri absencii jednej udalosti hovoríme o neúplnej správe.

Definícia 4.1.1. Stratená správa je neúplná správa, pri ktorej je známy výskyt udalosti odoslania správy `sendEvent`, ale nie je zaznamenaná udalosť prijatia správy `recieveEvent` Typická interpretácia je, že destinácia príjemcu správy je mimo popisovaného rámca. Sémantika je potom zjednodušená na tvar `<sendEvent>`. Anotácia je šípka vedená od odosielatela zakončená malou bodkou.

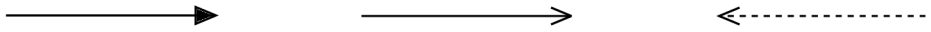
(a) Stratená správa. (b) Nájdená správa.

Obr. 4.1: Nekompletné správy.

Kompletná správa je v diagrame reprezentovaná orientovanou horizontálnou šípkou smerujúcou od aktívneho objektu odosielateľa k čiare života príjemcu správy.

V sekvenčných diagramoch rozlišujeme tri typy správ:

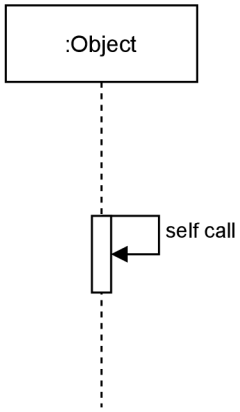
1. **Synchrónna správa** medzi objektami indikuje sémantiku *wait*, kedy odosielateľ správy čaká kým je správa spracovaná a pokračuje až po obdržaní odpovede. Správa typicky predstavuje volanie metódy.
2. **Asynchrónna správa** používa asynchrónny prístup, pri ktorom nedochádza k žiadnemu blokovaniu objektu odosielateľa. Asynchrónna správa medzi objektami indikuje *no-wait* sémantiku a objekt pokračuje bez toho, aby čakal na odpoveď. Toto dovoľuje paralélne procesy.
3. **Odpoveď** predstavuje spätnú správu po synchrónnej či asynchrónnej správe. Nemôže vzniknúť samostatne bez predchodzej správy na ktorú by odpovedala.



(a) Synchrónna správa. (b) Asynchrónna správa. (c) Odpoveď.

Obr. 4.2: Reprezentácie troch typov správ.

Existuje špeciálny typ správy kedy je odosielateľ aj príjemca správy tá istá inštancia. Ide o Reflexívnu komunikáciu:



Obr. 4.3: Reflexívna komunikácia v sekvenčnom diagrame.

4.1.1 Obsah správy

Komunikácia je v UML popísaná syntaxou:

```
[guard] *[iteration] sequence_number : return_variable := operation_name
        (argument_list)
```

Kde:

- `guard` - nepovinná podmienka, ktorá musí byť splnená na odoslanie správy.
- `iteration` - nepovinný záznam o počte správ.
- `sequence_number` - nepovinný údaj indexu sekvencie, ak je nedefinovaný odstraňuje sa aj nasledujúca dvojbodka. Diagram zobrazuje časovú osu na vertikálnej osi, preto je tento údaj nepovinný (redundantný).
- `return_variable` - je nepovinný údaj obsahujúci meno premennej uchovávajúcej odpoveď.
- `operation_name` - je povinný záznam o mene invokovanej operácie.
- `argument_list` - je nepovinný údaj obsahujúci čiarkou oddelené argumenty pre správu. Každý parameter môže byť explicitná hodnota alebo názov premennej. Zátvorky sú odstránené v prípade, že správa neobsahuje žiadne argumenty.

4.1.2 Opakovanie správ

V sekvenčných diagramoch môžme časť komunikácie uzavrieť v obdĺžniku, ktorý bude buď vľavo hore alebo vľavo dole obsahovať `iteration expression`. Tento výraz definuje počet opakovaní celého bloku. Preto počet opakovaní splňuje `iteration expression` $\in \mathbb{N}$.

4.2 Účastníci komunikácie

Participanti komunikácie skrz správy popísané vyššie sú aktívne objekty, ktoré v sekvenčných diagramoch reprezentujeme čiarou života(V literatúre *lifeline* [6]).

Definícia 4.2.1. Pri definícii **čiary života** začneme netradične notáciou, je zobrazená vertikálnou čiarou (môže byť čiarkovaná) predstavujúcu čas života aktívneho objektu. Na jej počiatku sa nachádza hlavička, obdĺžnik obsahujúci **identifikačnú informáciu** vo formáte:

```
        <lifelineident> := ([<element_name>]
                [: <element_type>] ) | 'self'

```

kde `<element-name>` referuje meno pripojeného elementu do komunikácie. `<element_type>` reprezentuje jeho typ. Napriek tomu, že to zápis dovoľuje `<lifelineident>` nemôže byť prázdny.

Ak je identifikátor `'self'` čiara života reprezentuje objekt klasifikátoru interakcie, ktorá sama vlastní čiaru života.

4.2.1 Pomenovanie objektov

Objekty v sekvenčných diagramoch sa pomenúvavajú dvojicou `<element_name>` a `<element_type>` vďaka čomu môžu vzniknúť tri typy objektov:

1. **Pomenovaný objekt**

```
<element_name> != null
<element_type> != null

<object_name> := <element_name>:<element_type>
```

   Pomenovaný objekt má meno aj typ známy.

2. **Anonymný objekt**

```
<element_name> == null
<element_type> != null

<object_name> := null:<element_type>
```

   Anonymný objekt nie je pomenovaný, no stále je identifikovateľný na základe typu

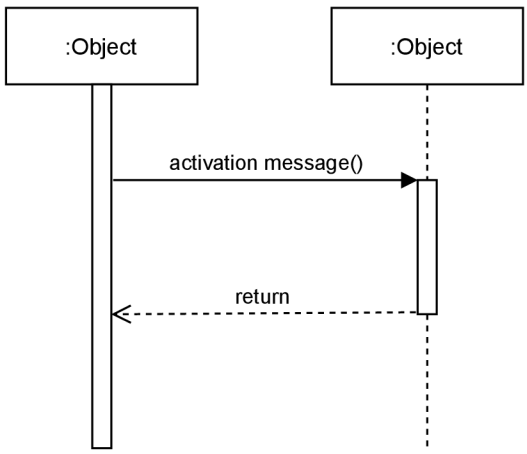
3. **Objekt neznámeho typu**

```
<element_name> == null
<element_type> == null

<object_name> := null
```

   Objekt neznámej triedy je najnáročnejší na prácu s objektami, nie je známy ani typ objektu.

4.2.2 Aktivácia objektu

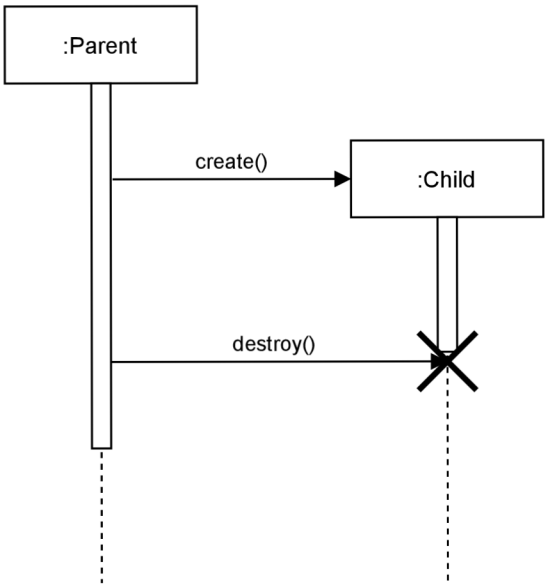
Aktivácia objektu je nepovinný údaj s notáciou obdĺžniku cez čiaru života. Reprezentuje časový údaj, kedy sa na objekte vykonávali operácie. Často neaktívnu inštanciu aktivuje správa odoslaná neaktívnej inštancii.



Obr. 4.4: Aktivácia neaktívneho objektu pomocou správy.

4.2.3 Vznik a deštrukcia objektu

Interakcia môže vyvolať vznik i zánik objektu. Komunikácia vyvolávajúca vznik má šípku smerujúcu na novo vzniknutý element. Správa deštrukcie ukazuje na miesto na čiare života objektu a ukončuje ju symbolom 'X'.



Obr. 4.5: Vznik a následná deštrukcia objektu.

Kapitola 5

Návrh implementácie

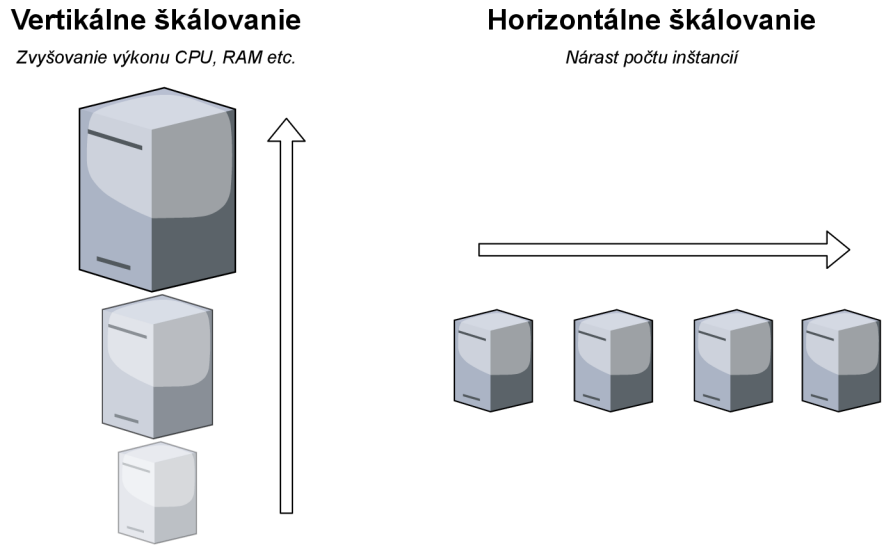
Základná myšlienka samotnej transformácie objektovo orientovaných petriho sietí je využiť diskrétnu simuláciu tejto siete, ktorej kroky nám vytvoria časové kontinuum inak chýbajúce v objektovo orientovaných petriho sietiach. V simulácii je nutné sledovať a zachytávať udalosti zmien stavu systému v závislosti na zmenu platnosti prechodových podmienok a pohyb značiek v miestach.

5.1 Architektúra

Generátor sekvenčných diagramov je priamo závislý na dvoch komponentách, validátorom kódu jazyka PNtalk a simulátoru objektovo orientovaných Petriho sietí. Je dôležité zvážiť napojenie týchto komponent ku genrátoru.

5.1.1 Počet inštancií

Pri **škálovateľnosti** hovoríme o **vertikálnej**, to jest zlepšovať výkon nahradzovaním hardvéru za výkonnejší na svojej jednej centrálnej inštancii. Takéto škálovanie je obmedzené technológiou, hardvér sa nedá zlepšovať do nekonečna. Pri komponentách, odbavujúcich diel práce máme možnosť škálovať **horizontálne**, odbavovať prácu na viacerých inštanciách zároveň. Rozdieľ medzi vertikálnym a horizontálnym škálovaním je graficky znázornený na obrázku 5.1.



Obr. 5.1: Vertikálne a horizontálne škálovanie.

Je jasné, že systém bude vyžadovať spoluprácu viacerých komponent a to minimálne troch. Generátor sekvenčných diagramov som sa rozhodol implementovať ako súčasť klientskej aplikácie. Ide totiž o vykresľovanie UI a vektorového obrazu z reportu zaznamenaného v simulácii.

Simulátor OOPN je pre účely práce nutné prerobiť z podoby terminálovej aplikácie do služby počúvajúcej na otvorenom porte a komunikujúc s klientom cez vzdialené volania. Pre podporu heterogenity je jednoduché zachovať rozhranie vzdialeného volania no mať viac implementácii simulátoru súčastne.

Prekladač do medzikódu som sa taktiež rozhodol prerobiť na službu počúvajúcu na otvorenom porte a kuminikujúcu so simulátorom cez vzdialené volania. Znovu je možné horizontálne škálovanie pre rozloženie záťaže.

5.1.2 Offline verzia

Generátor je navrhovaný tak, aby ho nebmedzovalo pripojenie k sieti a dokázal fungovať aj s lokálnym programom na simuláciu OOPN. V tomto prípade bude použitá pôvodná verzia terminálovej aplikácie. Kvôli tomu sa do komponent pridá prepínač, ktorý obalí aplikáciu do jednoduchého servera s argumentom definujúcim port, no kedykoľvek bude možné prepínač opomenúť a vrátiť sa k pôvodnej verzii terminálovej aplikácie. V nastaveniach musí byť možnosť výberu a zadania cesty vrátane portu k serveru.

5.1.3 Komunikačné rozhranie

Samotné dáta, prúdiace medzi komponentami, či už vo variante lokálnej, alebo vzdialeného prekladu a simulácie musia dodržovať jednotné rozhranie a musia byť serializované zo zrejmých príčin. Pri výbere serializačného formátu je nutné zvážiť viaceré faktory ako podpora v rozličných programovacích jazykoch, ľudsky čiteľnejšie textovo založené formáty alebo binárne uložené dáta, ktoré síce postrádajú ľudskú čitateľnosť, no vyžadujú menej pamäte a aj ich zápis a čítanie je časovo menej náročné. Binárne serializačné formáty by zlepšili responsivitu komponent a dáta posielané v ľudsky čitateľnom formáte by mali ne-

```
SimulateRequest {
code: string
settings
}

SimulateReply {
simulation_result: string
status: int
status_message: string
}
```

Obr. 5.2: Rozhranie požiadavku a odpovede na simulátor OOPN.

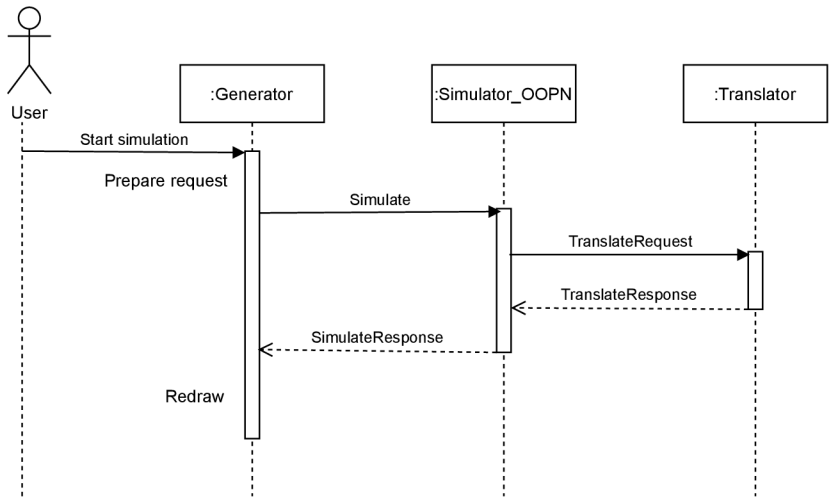
spornú výhodu v odľaďovaní programu. Schodnou variantou sa preto javí podpora viacerých formátov príjmaných generátorom od ostatným komponent. Nevýhodou je vznik réžie okolo dohadovaní si serializačného formátu medzi komponentami.

Navrhovaná varianta je zvoliť binárny formát so staticky definovanými cestami kvôli výkonu. Komunikačný protokol musí byť iniciovaný z generátoru. Poslať `SimulateRequest` obsahujúci kód (`code`) do simulátoru OOPN spolu s nastaveniami (`settings`).

Po prijatí kódu (`code`) v `SimulateRequest` musí simulátor kód preposlať na prekladač do medzikódu a paralélne počas čakania na výsledok pripraviť prostredie simulácie podľa nastavení.

```
TranslateRequest {
        code: string
}

TranslateReply {
        translated_code: string
        status: int
        status_message: string
}
```

Po obdržaní medzikódu (`translated_code`) prebehne simulácia. Výsledky zachytené v simulátore (`simulation_result`) sa potom vrátia do generátoru.



Obr. 5.3: Navrhovaný priebeh simulácie.

Obe odpovede `SimulateReply` aj `TranslateReply` vrátia status kód (`status`) a doplňujúcu správu v ľudsky čiteľnú správu (`status_message`), pretože systém musí vedieť detekovať zlyhanie oboch komponent.

Obsah správy výsledku (`simulation_result`) bude obsahovať serializované dáta výstupu zo simulácie. Túto dátovú štruktúru konkrétne rozoberie nadchádzajúca sekcia 5.2.1.

5.2 Transformácia modelu OOPN na sekvenčný diagram

V tejto Kapitole budú prednesené hlavné myšlienky ako vytvoriť základné stavebné jednotky sekvenčného diagramu. Popisujúc odkiaľ čerpať potrebné informácie zo simulácie, ako si poradiť z neúplnými informáciami a ako sa vysporiadať s absenciou potrebnej informácie zo simulácie modelu OOPN, aby bola škoda na výslednom sekvenčnom diagrame čo najnižšia. Kapitola je úzko spätá s predchádzajúcimi dvoma kapitolami, keďže bude ťažiť z možností formalismu OOPN a zároveň z vyjadrovacích schopností jazyka PNtalk na vytvorenie datovej štruktúry pre sekvenčný diagram.

5.2.1 Výstup simulácie systému

Práca nadväzuje v tomto smere na prácu Tomáša Lapšanského [17]. Tento simulátor systémov popísaných jazykom PNtalk, ale pre účely tejto práce musí byť rozšírený spôsobom, aby zachytával udalosti potrebné pre zostavenie sekvenčných diagramov a zaznamenával ich.

Štruktúra výsledku simulácie (`simulation_result`) z Obr. 5.2 musí obsahovať list krokov simulácie a inicializácií inštancií tried. Inštancie totiž môžu vzniknúť aj mimo simulované obdobie (Napríklad trieda označená ako hlavná syntaxou `main` na prvom riadku má vytvorenú inštanciu hneď na začiatku simulácie). Viac pozornosti na využitie plného potenciálu listu `initial` bude venovaných v sekcii 5.2.7 o scenároch.

```
struct SimulationResult{
    List<Step> steps;
    List<Initial> initial;
}
```

Obr. 5.4: Štruktúra uchovávajúca dáta zo simulácie OOPN modelu.

Krok simulácie

Štruktúra krokov simulácie uchováva údaje o správach poslaných v danom kroku, prechodoch, ktoré začali alebo skončili v tomto kroku. Jeden prechod môže začať a skončiť v iných krokoch simulácie.

```
struct Step{
    List<Message> messages;
    List<TransitionStart> transStarts;
    List<TransitionEnd> transEnds;
}
```

Počiatočná inicializácia

Jazyk PNtalk v sekcii definoval kľúčové slová. Jedno z nich je `main` určujúce počiatočnú inštanciu, ktorá existuje ako jediná na počiatku simulácie. I keď PNtalk definuje len jednu predvolenú triedu, `initial` v `simulation_result` je list, pretože rozšírenie scenárov zo sekcie 5.2.7 bude dovoľovať nadefinovať počiatočný stav s viacerými inštanciami.

```
struct Initial{
    string instanceName;
    string className;
    int creationTime;
    List<Place> places;
}
```

Štruktúra inicializácií inštancí tried obsahuje meno inštancie (`instanceName`), referenciu na svoju triedu (`className`), čas vzniku v intervale simulácie (`creationTime`) a počiatočný stav miest (`places`). Meno inštancie spolu s menom triedy spolu tvoria štítok objektu v sekvenčnom diagrame. Čas vzniku odsadzuje objekt na ypsilonovej ose od počiatku simulačného času. Počiatočný stav miest predáva stav miest, pomocou tohto listu sa dajú prepísať predvolené hodnoty v miestach.

Správa

Asi najdôležitejším elementom v sekvenčných diagramoch sú správy. V sekcii 4.1 boli rozlíšené jednotlivé druhy správ. Teraz bude prednesený návrh ako ich zachytiť počas simu-

lácie. Invokácia metódy bude zachytávaná na začiatku volaní a v dobe, keď sa vráti hodnota z volanej metódy do miesta invokátora metódy, tj. inštancie, ktorá ju zavolala.

V oboch prípadoch je rozhranie pre správy rovnaké:

```
struct Message{
int id;     #AUTO_INCREMENT
string messageName;
string callerInstance;
string callerClass;
string receiverInstance;
string receiverClass;
string transition;
int respondTo;
List<Value> response;
}
```

Štruktúra správy obsahuje jej unikátny identifikátor (`id`), meno správy (`messageName`), informácie o odosielateľovi (`callerClass.callerInstance`) a príjemcovi správy (`receiverClass.receiverInstance`), názov prechodu (`transition`), ktorý vyvolal správu a záznam, či sa jedná o odpoveď na nejakú už existujúcu správu (`respondTo`). Názov inštancie spolu s názvom triedy vytvára unikátny identifikátor pre odosielateľa aj príjemcu správy. Priradenie ku prechodu uľahčuje orientáciu v rámci kroku simulácie, v ktorom sa prechody vykonali simultálne.

Rozlíšenie, či sa jedná o odpoveď je triviálne:

```
if 'respond_to' not in message:
    draw_basic_message();
else:
    draw_reply_message();
```

Ak sa jedná o odpoveď, nie je nutný žiaden záznam okrem identifikátoru správy, na ktorú sa odpovedá a odpoveď samotná. Odpoveď je definovaná ako list hodnôt.

Odpoveď je akceptovaná v tomto minimálnom tvare (aj bez identifikátoru):

```
struct Message{
    int respondTo;
    List<Value> response;
}
```

Zbytok parametrov je totiž možno vyčítať z pôvodnej správy, na ktorú je odpoveďou. Jediná zmena je obrátená dvojica odosielateľa a príjemcu správy:

```
orig_message = messages.find(id=response['respond_to']);

response['message_name'] := orig_message['message_name'];
response['caller_instance'] := orig_message['receiver_instance']; #reverse
response['caller_class'] := orig_message['receiver_class']; #reverse
response['receiver_instance'] := orig_message['caller_instance']; #reverse
response['receiver_class'] := orig_message['caller_class']; #reverse
response['transition'] := orig_message['transition'];
```

V sekcii 4.1 bol definovaný špeciálny typ správy a to reflexívna, ktorá má vlastnú notáciu. Je triviálne rozlíšiteľná:

```
caller := message['caller_class'].message['caller_instance'];
receiver := message['receiver_class'].message['receiver_instance'];

if caller == receiver:
    draw_reflexive_message();
```

Prechody

Štruktúra začiatku prechodu uchováva svôj unikátny identifikátor, svoje meno a meno inštancie a triedy objektu, ktorý prechod vykonal.

```
struct TransitionStart{
    int id;     #AUTO_INCREMENT
    string transName;
    string instanceName;
    string className;
}
```

Výpis 5.1: Informácie o začiatku prechodu.

Štruktúra ukončenia prechodu si drží referenciu na začiatok prechodu, ktorý ukončuje a list zmien v miestach, ktoré sa stali od začiatku prechodu.

```
struct TransitionEnd{
    int idStart;
    List<Change> changelog;
}
```

5.2.2 Reprezentácia času

V predchádzajúcej sekcii 5.2.1 boli správy a prechody usporiadané a odosielané po krokoch simulácie.

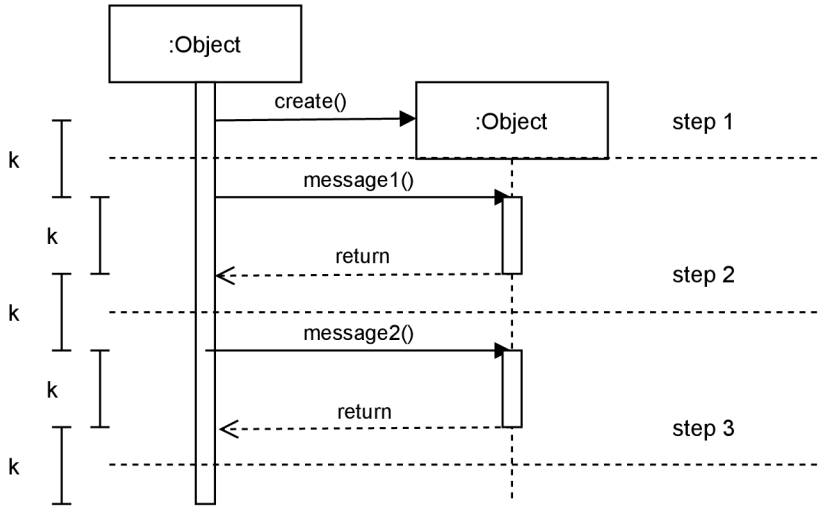
Všetky kroky simulácie sú v odpovedi vrátené ako list. Krok nemôže byť prázdny, znamenalo by to totiž, že žiadne evolučné pravidlo z definície 3.1.5 by sa nevykonalo. A stav

petriho siete zostal rovnaký. To by znamenalo, že v ďalšom kroku by sa znova vyhodnotili tie isté podmienky a prechod by nenastal ani v krokoch nasledujúcich. Preto sa v takom prípade simulácia zastaví.

Dôležitosť tohto listu sa plne prejaví v súvislosti s časom. V sekcii 3.4 bolo predstavené zavedenie času, ktoré však nemá v implenentácii OOPN PNtalk svoje miesto. Prechody sú okamžité a paralélne. Sekvencia vzniká až keď podmienka prechodu nadobúda platnosť až po uskutočnení iného prechodu. Kroky simulácie teda budú kľúčom k reprezentácii času na y-ose sekvenčného diagramu. Odsadenie na y-ose sa bude počítať pomocou konštanty k, ktorá bude oddelovať prichádzajúce správy. Vytvorí sa tým tak deterministické oneskorenie prechodov popísané v sekcii 3.4.

```
k := CONSTANT_PIXEL_LENGHT px;

time = 0;
for step in simulation_result.steps:
for message in step.messages:
time++;
message_y_position = k * time;
```

Konštantu k je nutné zvoliť tak, aby poskytovala dosť priestoru na notáciu správ a zároveň aby neboli správy príliš ďaleko od seba.



Obr. 5.5: Degradovaná časová reprezentácia.

Kroky majú rôzny počet správ. Dajú sa v rámci jedného kroku čiastočne preusporiadať, no generátor sa o to pokúšať nebude. Vykreslí sekvenčný diagram v poradí v akom boli správy odsimulované.

5.2.3 Princíp vytvárania objektu

Objekt alebo entita, ako bolo zmienené v sekcii 4.2, je kľúčová časť v scenári sekvenčného diagramu. Je to obdĺžnik so štítkom mena vo vnútri, v ktorom započne čiara života až do deštrukcie objektu, alebo konca simulácie.

Objekt môže vzniknúť za behu simulácie, alebo byť k dispozícii ako východzí objekt pri štarte simulácie.

Na vytvorenie objektu v sekvenčnom diagrame je nutné zo simulácie archivovať minimálne 2 informácie:

1. **Čas simulácie v ktorom sa inštancia vytvorí**
   Je nevyhnuteľný, bez neho by nebolo jasné kam umiestniť novo vzniknutý objekt na ypsilonovej (časovej) osi.
2. **Triedu vytváranej inštancie** Trieda definuje miesta a metódy. Bez nej by sa nedalo s inštanciou pracovať. Mala by neznáme chovanie a neznáme uložené dáta. Tým sa vylúčia objekty neznámeho typu definované v sekcii 4.2.1.

Optimálne, nie však nevyhnuteľne nutné, by bolo vedieť aj nasledujúce:

1. **inštanciu, ktorá inicializovala vytvorenie**
   Vedieť rodiča inštancie pomôže s počítaním referencií na novo vzniknutý objekt, čo vyústi v konečnom dôsledku v rozhodovaní, kedy má nastať deštrukcia objektu (počet referencií bude nulový).
2. **Meno novej inštancie**
   Pomenovanie nie je povinné ako bolo spomenuté už v sekcii 4.2.1, pre sekvenčné diagramy nie je problém vytvoriť annonymnú inštanciu bez mena. Meno však prispeje k väčšej prehľadnosti generovaného výstupu.

Existujú 2 scenáre vytvorenia objektu:

1. Objekt je definovaný ako výchozí na začiatku simulácie
   V tomto prípade stačí iterovať skrz list `Initial`. Na základe štruktúry `Initial` vieme vytvoriť takmer optimálny objekt, chýba len informácia o rodičovskej inštancii. Ide o výchozie objekty definované pre počiatok simulácie, je prirodzené, že nebudú mať rodiča a nebudú deštruované počas doby simulácie.

```
for object in initials:
    newObject.instanceName := object.instanceName;
    newObject.className := object.className;
    newObject.creationTime := object.creationTime;
    newObject.ignoreGarbageCollector := true;
    newObject.places := object.places;

```

2. Objekt sa vytvorí volaním správy `create` z inej už existujúcej inštancie. Vytvorenie inicializuje odchytená štruktúra `Message` popísaná v 5.2.1. Čas vytvorenia objektu závisí od kroku v ktorom bola správa odchytená v princípe popísanom v sekcii 5.2.2.

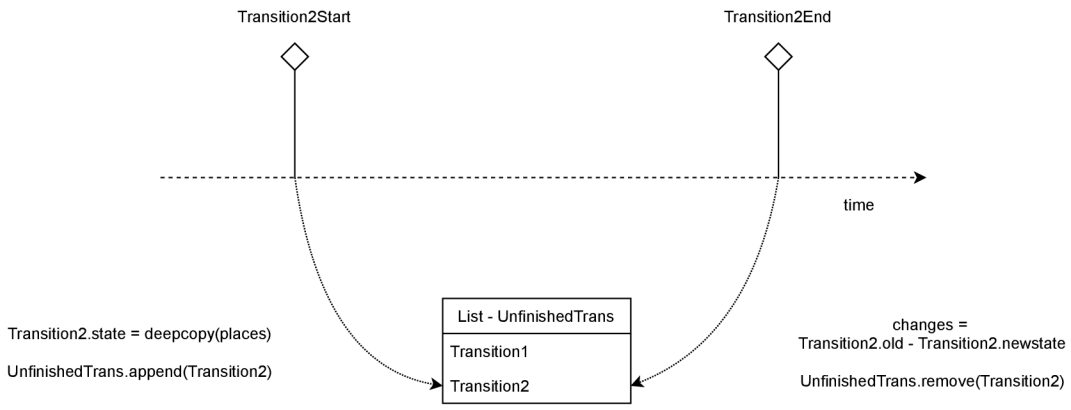
```
def entityCreation(message, time):
    if message.messageName = '<<create>>':
        newObject.instanceName := message.receiverInstance;
        newObject.className := message.receiverClass;
        newObject.creationTime := time;
        newObject.places := message.receiverClass.defaultplaces;

        message.callerInstance.addReference(newObject);
```

Vďaka týmto údajom sa dá vytvoriť správa v sekvenčnom diagrame, ktorá odsadí objekt vertikálne od počiatku do vzdialenosti podľa času vytvorenia. Prvá referencia na nový objekt sa vytvorí v inštancii odosielateľa správy. Táto referencia slúži pre deštrukciu objektov o ktorej pojedná sekcia 5.2.5.

5.2.4 Artefakty z Petriho siete

V tejto sekcii je venovaný priestor artefaktom zo simulovanej OOPN, ktoré síce priamo nie sú vykreslené v sekvenčnom diagrame, ale majú kľúčovú úlohu pri spätnom mapovaní do kódu PNtalk a interaktívnom ladení.

Celá idea je postavená na prechodoch, ktoré zachytáva a preposiela vo výsledku simulátor.



Obr. 5.6: Zachytenie začiatku a konca prechodu v simulácii.

Na začiatku invokácie prechodu sa vytvorí kópia miest danej inštancie a uloží do listu neukončených prechodov.

```
def transStart(transition):
    archive(transitionStart(
        transition.name,
        this.instanceName,
        this.className
        );

    unfinishedTrans.append(transition, deepcopy(this.places));
```

Niekedy v budúcnosti by malo byť zachytené dokončenie prechodu. Medzi tým mohli začať a skončiť aj iné prechody.

```
def transEnd(transition):
    orig_transition = unfinishedTrans.find(transition);
    change = compare(transition.places, orig_transition.state);
    archive(transitionEnd(
        orig_transition.id,
        change
    );

    unfinishedTrans.remove(orig_transition);
```

Tým vznikne **zoznam zmien** v chronologickom poradí. Štruktúra drží informáciu o miestach a hodnotách v nich, ktoré sa zmenili.

```
struct Change{
    Place place;
    List<Value> newValues;
}
```

Našťastie **miesta** v objekte sú konečná množina, s presne definovaným počtom a menami podľa predpisu triedy objektu. Za chodu systému nevznikajú ani nezanikajú. Ich deštrukcia je spoločná s deštrukciou objektu. Jedno miesto, prirodzene, nemôže patriť viacerým objektom.

```
struct Place{
    string placeName;
    List<Value> values;
}
```

Štruktúra hodnoty miesta obsahuje typ (hodnota alebo referencia na objekt) a samotnú hodnotu. Typ referencia znamená, že hodnota odkazuje na podsieť taktiež definovanú v

jazyku OOPN. V takom prípade sa cez túto hodnotu dajú volať metódy uloženej podsiete.

```
struct Value{
int type;
string value;
}
```

Mierne skrytou výhodou uchovávania neukončených prechodov je možnosť priradiť správu k prechodu. Simulátor môže pri invokovaní metódy nahliadnuť do listu neukončených prechodov a pozrieť si posledný začatý prechod. Simulátor bude predvídať, že práve tento prechod inicializoval volanie správy.

```
def invokeMethod():
    transition = unfinishedTrans.getLast();
    ...
    archive(Message(transition, etc...));
```

5.2.5 Deštrukcia objektu

Sekvenčný diagram bude deštruovať len objekty vytvorené až za behu simulácie. Východzie objekty definované syntaxou `main` v jazyku PNtalk alebo východzie objekty definované zo scenárov 5.2.7 bude tento mechanizmus ignorovať.

Pre deštrukciu objektu bude fungovať počítadlo referencií a pre zánik bude musieť zaniknúť posledná referencia na objekt.

Referenciu konkrétne definuje len dvojica inštancia a miesta, cez ktoré je inštancia referovaná. Inštancia môže mať referenciu na rovnaký objekt uloženú aj vo viacerých miestach.

```
struct Reference{
    string instanceName;
    Place place;
}
```

Prvú referenciu vždy vytvorí volanie správy `create`. V jazyku PNtalk je zápis nasledovný:

```
<place> := <className> new.
```

Referencia teda má svoje miesto, v ktorom vznikla a meno inštancie sa vygeneruje podľa kontextu.

Prechod môže spôsobiť tri veci pri manipulácii s referenciou:

1. **Presunúť referenciu do iného miesta**
   Pri presune referencie sa len pozmení záznam miesta, v ktorom sa nachádza.
2. **Zduplikovať referenciu do iného miesta**
   Pri zduplikovaní sa vytvorí nový záznam o referencii.
3. **Vymazať referenciu**
   Pri vymazaní sa skontroluje, či nie je počet referencií na objekt nulový. Ak áno, objekt sa deštruuje volaním správy `destruct` z inštancie s prechodom, ktorý poslednú referenciu vymazal.

Vďaka selektívnemu výberu prechodov, ktoré manipulujú s miestami, kde sú uložené objekty môžme zredukovať počet opakovaní algoritmu len na vybrané prechody. Optimalizácia využíva zoznam zmien zo sekcie 5.2.4.

```
for change in changelog:
    if change.place.containsRef():
        references.destroy(change.place);
    if change.newValue.type == Reference:
        refernces.add(change.place, change.newValue);
```

Bez zoznamu zmien by sa muselo iterovať cez všetky miesta a počítať referencie po každom kroku. Takto sa kontrolujú len zmenené miesta.

5.2.6 Princíp aktivácie

Čiara života môže byť prekrytá bielym obdĺžnikom značiacim aktiváciu objektu ako bolo bližšie popísané v sekcii 4.2.2 . Jeho vytvorenie je triviálne pokiaľ dokážeme určiť čas simulácie v ktorom bol objekt aktívny.

Pred hľadaním konkrétneho času aktivácie a deaktivácie je nutné presne definovať, kedy budeme objekt považovať za aktívny. Využitý na to bude list neukončených prechodov zo sekcie 5.2.4 o artefaktoch z Petriho siete.

```
def isActive(): <- Boolean
    for transition in unfinishedTrans:
        if transition.instanceName := this.instanceName:
            return true;

    return false;
```

Definícia 5.2.1. Objekt je **aktívny** ak má aspoň jeden započatý ale zároveň neukončený prechod.

Konkrétny čas bude teda odvodený od `startTrans`, kedy začne platiť podmienka definovaná vyššie. A čas ukončenia aktivácie bude zachytený v `endTrans`, kedy podmienka platiť prestane.

5.2.7 Scenáre

Jedna z prekážok mapovania OOPN na sekvenčný diagram je konceptuálne iná myšlienka využiteľnosti. Na jednej strane OOPN má popisovať chovanie celého systému na základe pravidiel, na strane druhej sekvenčný diagram sa častejšie využíva len na zachytenie menšej časti systému a konkrétneho užšie špecifikovaného scenáru. Takýto bližšie špecifikovaný scenár sa dá vytvoriť pomocou definície východzích inštancií a ich predvolenému stavu miest na začiatku simulácie.

Do simulácie budú odoslané v správe `SimulateRequest` a v odpovedi budú vrátené (doplnené o nedefinované miesta a inštanciu `main`) v liste `Initial`.

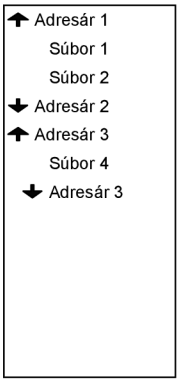
5.3 Užívateľské rozhranie

Táto kapitola sa zaoberá rozborom vývojových prostredí a ich dekompozíciou na jednotlivé editory a grafické nástroje prítomné v úspešných vývojových prostrediach.

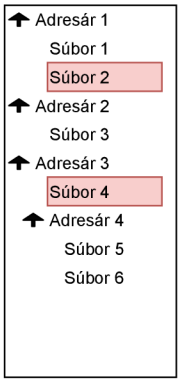
Ich význam spočíva v uľahčení práce programátora, zefektívnením kódovania a urýchleného detekovania problémov. Prostredie vedie programátora cez proces editovania, kompilácii či interpretovania kódu a ladenia.

Pri návrhu grafického užívateľského rozhrania pre nástroj generátoru práca vychádza z osvedčených a zaužívaných konceptov. Od užívateľského rozhrania sa bude vyžadovať zabezpečenie len základných potrieb:

1. **Potreba zobraziť momentálne otvorený projekt**
   Návrh predstavuje reprezentáciu pomocou hierarchického stromu, ktorý by mal v listoch uložené mená súborov a v uzloch mená adresárov. Listy, teda súbory, by mali vizuálnym effektom upozorniť ak je v súbore neuložená zmena.



Obr. 5.7: Projektový pohľad so schovaným uzlom "Adresár 2" a "Adresár 4".



Obr. 5.8: Indikácia neuložených zmien v súboroch viditeľná na rozhraní.

2. **Potreba zobraziť momentálne otvorený súbor s kódom**
   Realizujeme to ako editor zdrojového kódu s automatickým zvýrazňovaním kľúčových slov syntaxe jazyka PNtalk a mien z validných definícií. Potrebujeme zobrazovať čísla riadkov.

3. **Potreba zobrazovať vygenerovaný diagram**
   Potrebujeme na to minimálne rovnako veľa miesta ako na editor zdrojového kódu. Pozadie by malo byť kontrastné proti diagramu. Celá časť musí byť interaktívna, jednak kvôli pohybu a približovaniu diagramu v okne. Diagram by mal slúžiť ako nástroj na ladenie kódu. To znamená, že kliknutia na jednotlivé časti sekvenčného diagramu by mali vyznačiť reprezentáciu v kóde. Označenie správ by zasa malo zobraziť zmenu miest OOPN, ktoré prechody vyvolali. Označenie, ktoréhokoľvek miesta na čiare života by malo ukázať aktuálne hodnoty v miestach danej inštancie.

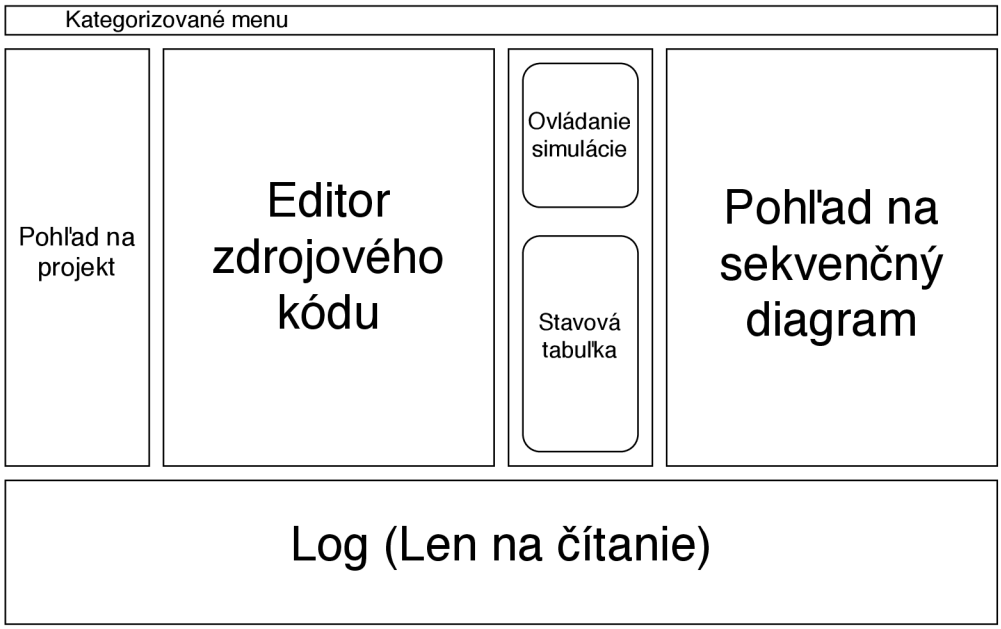
4. **Potreba zobrazovať posledných x riadkov logov** Je fajn dať užívateľovi vedieť, čo sa deje formou správ, či už chybových alebo informačných. Správy sa musia dať kopírovať a musia byť viditeľné od najnovšej po najstaršiu.

5. **Zbytok funkcionalít ukryť do hornej lišty** V hornej lište by mali byť kategoricky roztriedené funkcie s klávesovými skratkami u tých, u ktorých to dáva zmysel.

5.3.1 Rozloženie užívateľského rozhrania

Pri návrhu rozloženia elementov užívateľského rozhrania sa vychádzalo z rozložení úspešných vývojových prostredí (Visual Studio, IntelliJ IDEA). Tieto užívateľské rozhrania fungujú už dlhšiu dobu, užívateľia sú na ne navyknutí. Preto nie je v záujme práce sa od týchto štandartov odklánať. Elementy, ktoré nájdeme v každom vývojovom prostredí ako editor kódu a projektový pohľad budú umiestnené tak ako všade.
To samozrejme neplatí o netradičnom elemente vykresľujúci sekvenčný diagram. Táto časť je unikátna v tejto práci. Je to časť ktorá zobrazuje výstup a zároveň je to aj interaktívny debugger. Inšpiráciu pre tento element by sme hľadali márne, v bežných vývojových prostrediach sa nič podobné nenachádza. Je rovnako, ak nie viac, dôležitý ako editor zdrojového kódu, preto dostane rovnako veľké miesto.

Po zvážení všetkých nárokov na užívateľské rozhranie vyšlo z procesu návrhu rozloženie na Obr. 5.9



Obr. 5.9: Návrh rozloženia grafického užívateľského rozhrania.

Kapitola 6

Implementácia nástroja

Práca bola implementovaná v jazyku Kotlin a prácu na ktorú som nadväzoval som upravil v jazyku C++, na ich prepojenie som využil verejne dostupné knižnice gRPC pre Kotlin [1] a pre C++ [2]. Implementácia simulátoru a prekladača do medzikódu bola prispôsobená pre použitie s technológiou Docker a boli vytvorené spustiteľné obrazy týchto dvoch komponent skrz kontajnery dockeru. Na grafické užívatľské rozhranie bol použitý aplikačný rámec TornadoFX[3] nad knižnicou JavaFX.

6.1 Výber implementačného jazyka

Práca nadväzuje na simulátor napísaný v jazyku C++, čo robilo z jazyka C++ prirodzenú voľbu pre naviazanie a kompatibilitu. Ďalším požiadavkom práce, akožto zadaním z oblasti blízkej jazyku PNtalk, bola motivácia držať implementačné jazyky týchto nástrojov blízko [17] [9]. Keďže väčšina prác beží na Jave a je do budúcna zmýšľaná ich kooperácia, vyplýva z toho ďalší faktor ovplyvňujúci výber a to držať implementáciu blízko JVM (Java Virtual Machine). Od napojenia na simulátor priamo sa upustilo a zvolila sa varianta umožňujúca napojenie aj iných simulátorov napísaných v rôznych jazykoch [9]. Pre prácu bol zvolený ako implementačný jazyk **Kotlin**, zohľadňujúc požiadavky zmienené vyššie. Prispel k tomu rozvoj modernej doby a popularita akej sa teší Kotlin dnes. V roku 2017 ho Google učinil oficiálnym jazykom pre Android [4]. Na platforme Github, ktorá hosťuje viac ako 100 miliónov repozitárov rôznych zdrojových kódov napísaných v rôznych jazykoch, bol Kotlin za rok 2019 štvrtý najrýchlejšie rastúci programovací jazyk s nárastom o 182% oproti minulému roku [3]. V prvej polovici roku 2020, teda súčastnosti písania tejto práce, je celkovo na 15. priečke v obľúbenosti [2].

6.2 Úprava simulátoru

Ako bolo zmienené v návrhu, generátor sa nebude priamo viazať na simulátor ani prekladač do medzikódu. Namiesto toho budú prepojené v heterogénnom systéme komunikujúcim cez vzdialené volania.

[1] gRPC knižnica je verejne dostupná https://github.com/grpc/grpc-kotlin
[2] zdroj knižnice gRPC pre C++ https://github.com/grpc/grpc
[3] TornadoFX je aplikačný rámec v kotline nad JavaFX https://tornadofx.io/

6.2.1 Komunikácia

Práca implementuje komunikáciu nad HTTP2 pomocou binárnych pevne stanovených rozhraní pomocou rámca gRPC [15]. Implementácia sa drží návrhu zo sekcie 5.1.3 a definuje *proto* súbory spoločné pre všetky programovacie jazyky.

```
syntax = "proto3";

package virtualmachine;

service Simulator {
rpc simulate (SimulateRequest) returns (SimulateReply) {}
}

message SimulateRequest {
string code = 1;
string scenario = 2;
int64 steps = 3;
}

message SimulateReply {
int64 status = 1;
string result = 2;
}
```

Tak je definovaná služba *Simulator* a jej vzdialené volanie **simulate**. Úplne rovnako je definované rozhranie prekladača do medzikódu:

```
syntax = "proto3";

package translator;

service Translator {
rpc Translate (TranslateRequest) returns (TranslateReply) {}
}

message TranslateRequest {
string code = 1;
}

message TranslateReply {
int64 status = 1;
string semicode = 2;
}
```

Preklad *proto* súborov do jazyka C++ bol riešený cez nástroj **protoc**[4] a jeho plugin **grpc_cpp_plugin** cez príkazovú riadku:

```
protoc --cpp_out=build $path/$filename
protoc --grpc_out=build --plugin=protoc-gen-grpc=`which grpc_cpp_plugin`\
    $path/$filename
```

Prekladač protoc vygeneruje C++ zdrojové kódy z poskytnutého *proto* súboru.

```
simulate.grpc.pb.cc
simulate.grpc.pb.h
simulate.pb.cc
simulate.pb.h
translate.grpc.pb.cc
translate.grpc.pb.h
translate.pb.cc
translate.pb.h
```

Pre Kotlin sa na preklad v implementácii používa maven plugin **protobuf-maven-plugin**. Z rovnakých *proto* súborov vytvára zdrojové kódy v Kotline.

```
SimulateGrpcKt.kt
TranslateGrpcKt.kt
```

6.2.2 Rozšírenia funkcionality simulátoru

Rozšírenia zahŕňajú pridanie prepínačov a zmenu reakcie na chyby.

Spúšťanie komponent ako server

Simulátor aj prekladač bolo nutné rozšíriť a prerobiť, aby sa konceptuálne dali komponenty spustiť ako server počúvajúci na vybranom porte. Na tomto porte bude pripravená gRPC komunikácia zmienená vyššie. Program sa v prípade predania prepínača **-d** alebo –**deamon** nasledujúcim číslom portu zaobalí a bude sa chovať ako jednoduchý server. Prekladač do medzikódu má východzí port 51899 a virtuálny stroj na simuláciu 51898.

```
Translator2 -d 51899
VM2 -d 51898
```

Scenáre

Pre rozšírenie scenárov zo sekcie 5.2.7 bol pridaný do virtuálneho stroja parameter **-c** alebo –**scenario** nasledovaný cestou k súboru. Na syntaktickú analýzu tohto sýboru v yaml

[4]protoc je kompilačný nástroj pre protobuf buffer mechanizmus https://developers.google.com/protocol-buffers

formáte bola použitá knižnica **libyaml-cpp**.

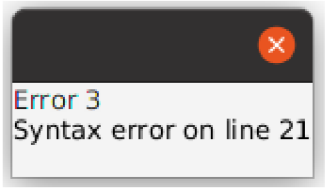
```
VM2 --scenario $path_scenario
```

Zmena bola implementovaná do funkcie `vm::setMain()`, kde sa scenár iteruje a do počiatočného stavu pridáva nové inštancie alebo ich mení. Nedefinované miesta sú vyplnené východzími hodnotami z predpisu triedy.

Návratové kódy

Do práce bolo pridané navracanie stavových kódov, akýkoľvek iný stav ako 200, (403, 500) indikuje chybu externej komponenty. Takáto správa potom dá spätnú väzbu užívateľovi v generátore.

```
reply_.set_result(error_message);
reply_.set_status(403);
```



Obr. 6.1: Chybová správa z prekladaču zobrazená v generátore.

6.2.3 Zhromažďovanie informácií pre genrátor

Na zhromažďovanie potrebných dát pre zdarné vykreslenie sekvenčného diagramu bol do simulátoru implementovaný modul **archiver** založený na knižnici **cereal**[5].

Jedná sa o globálny objekt, do ktorého sa dá archivovať dôležitá udalosť v akomkoľvek bode simulácie.

```
virtualMachine->archiver.startTrans(
    stackTransition(
        transition->name,
        name,
        reference->name,
        places
    )
);
```

[5]knižnica na serializáciu pre C++11 https://uscilab.github.io/cereal/

Všetky dátové štruktúry už byli opísané v kapitole Návrh Implementácie v sekcii 5.2.1 a taktiež princípy, kedy a ako ich zbierať sú detailne popísané v sekcii 5.2.3 Princíp vytvárania objektov, 5.2.4 Artefakty z Petriho sietí a 5.2.6 Princíp aktivácie. Preto sa nebudú znovu vypisovať v implementácii pomocou syntaxe knižnice cereal.

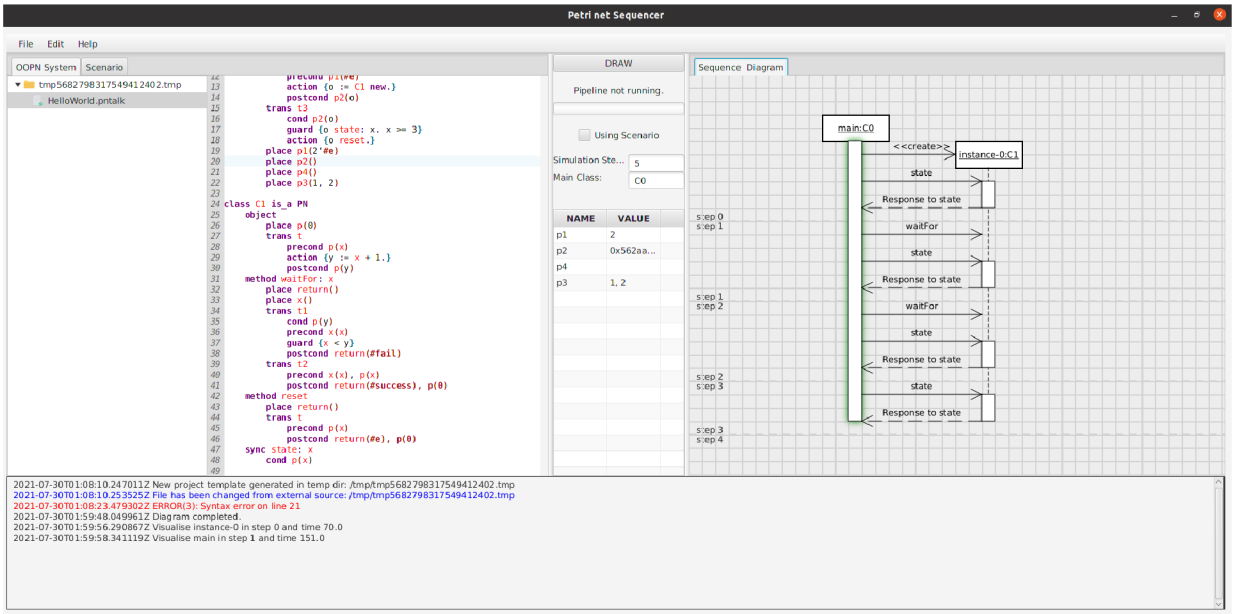
Po dokončení simulácie modul serializuje všetky nazbierané udalosti do formátu json.

```
1 virtualMachine.archiver.generate(std::cout);
```

Tento výstup je predávaný späť do generátoru v odpovedi `SimulateReply` ako pole `simulation_result`.

6.3 Generátor sekvenčných diagramov

Implementácia vychádza z dobre pripraveného návrhu z Obr. 5.9, ktorá bola realizovaná za pomoci aplikačného rámcu TornadoFX nad softvérovou platformou JavaFX. Generátor je hlavne prezentačná vrstva už nazbieraných dát. Implementácia sa príliš od navrhovaného rozloženia užívateľského rozhrania neodklonila.

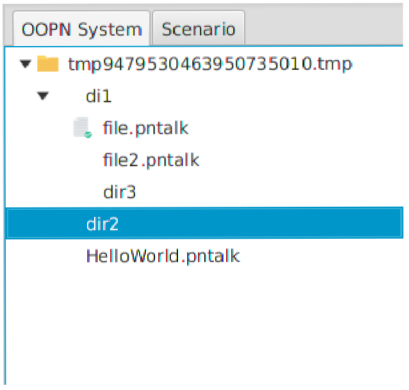


Obr. 6.2: Výsledné rozloženie užívateľského rozhrania.

V tejto sekcii sa postupne prejdú časti nástroja a popíšu sa ich hlavné implementované funkcionality.

6.3.1 Projektový pohľad

Predtým než užívateľ začne písať samotný kód, je nutné definovať prostredie, v ktorom sa bude pohybovať. Podľa návrhu zo sekcie 5.3 bol implementovaný pohľad na stromovú štruktúru súborového systému.



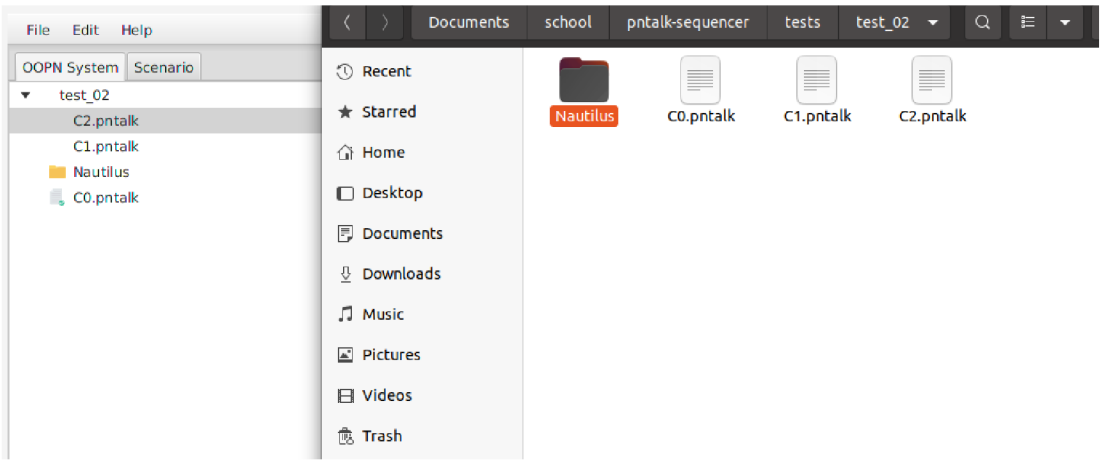
Obr. 6.3: Rozvetvená súborová štruktúra.

Reakcia na externé zmeny

Z testovania vyšla potreba reagovať na zmeny v adresári, ak sa z neho súbor vymaže, či pridá alebo nejaký súbor zmení svoj obsah.[6]

```
1 GlobalScope.async {
2     asWatchChannel().consumeEach { event ->
3         if (event.kind.kind == "created" || event.kind.kind == "deleted")
4             updateTreeView()
5     }
6 }
```

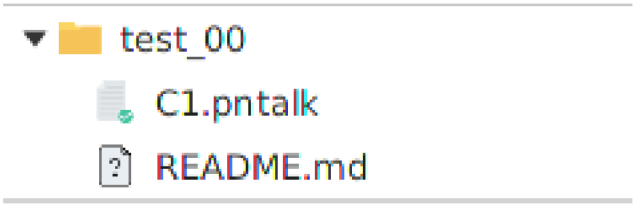
[6]Implementácia je prebratá z návodu https://proandroiddev.com/kotlin-watchservice-a-better-file-watcher-using-channels-coroutines-and-sealed-classes-7ab5c9df3ada jej originálnym autorom je Łukasz Wiśniewski



Obr. 6.4: Reakcia na zmeny z vonkajšieho prostredia.

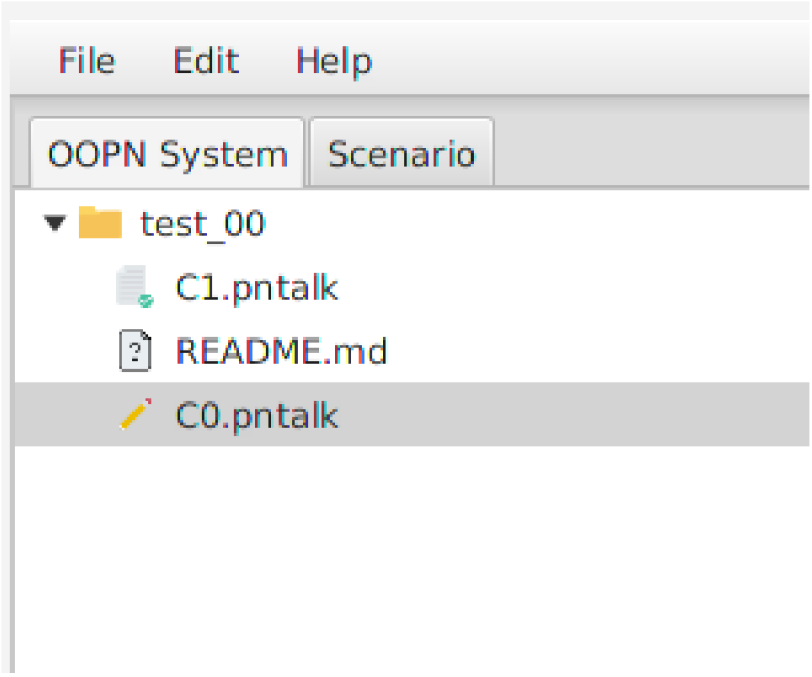
Správa súborov

Projekt posiela na preklad a simuláciu len súbory končiace príponou *.pntalk*. Ostatné súbory sú ignorované a odlíšené ikonou. Súbory *pntalk* sú konkatenované a odoslané v jednom neprerušenom toku kódu.



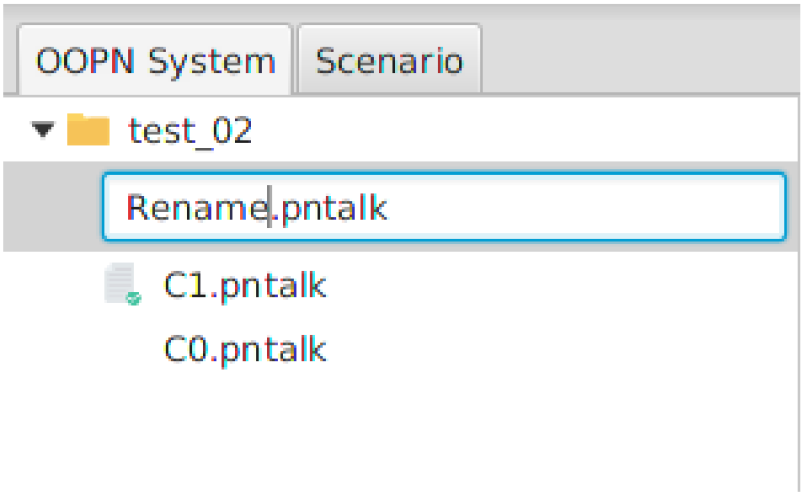
Obr. 6.5: Súbor README.md neúčastniaci sa simulácie.

Navyše ikona odlišuje aj súbory v ktorých sú neuložené zmeny, aby sa predišlo nechcenému vypnutiu nástroja pred uložením zmien.



Obr. 6.6: Neuložený súbor so zmenami je označený ceruzkou.

Jednoduché ovládanie dovoľuje premenovanie, vytváranie a mazanie súborov a zložiek.



Obr. 6.7: Premenovanie súboru.

6.3.2 Editor kódu

Editor kódu bol implementovaný pomocou balíka **richtextfx**. Editor zvláda zvýrazňovanie kľúčových slov a číslovanie riadkov.

Zvýrazňovanie kľúčových slov

V sekcii 3.5 bolo prebraná syntax jazyka PNtalk, ktorého kľúčové slová sa vyhľadávajú v kóde. Pre netriviálne prípady, kedy sú napríklad mená miest hneď za slovom *place*, sú

implementované regulárne výrazy, ktoré zvládnu vyhľadať aj názvy prechodov, tried a podobne.

```
val classNames = "class (.*) is_a PN".toRegex()
val transNames = "trans (\\w+)".toRegex()
val placeNames = "place (\\w+)\\(".toRegex()
val syncsNames = "sync (\\w+):".toRegex()
val methodNames = "method (\\w+)".toRegex()
```

```
class C0 is_a PN
        object
        trans t4
            precond p4((x, #fail))
            postcond p3(x)
        trans t2
            cond p2(o)
            precond p3(x)
            action {y := o waitFor: x}
            postcond p4((x, y))
        trans t1
            precond p1(#e)
            action {o := C1 new.}
            postcond p2(o)
        trans t3
            cond p2(o)
            guard {o state: x. x >= 3}
            action {o reset.}
        place p1(2‘#e)
        place p2()
        place p4()
        place p3(1, 2)

```

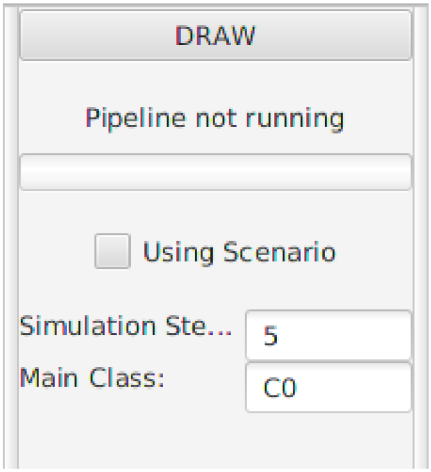
Obr. 6.8: Zvýrazňovanie kľúčových slov v editore kódu.

6.3.3 Interaktívny sekvenčný diagram

Určite najdôležitejšia komponenta, na ktorú sa v práci sústredila najväčšia pozornosť. Po úspešnej simulácii vytvorí grafické primitívy reprezentujúce notáciu sekvenčnęho diagramu.

Nastavenia simulácie

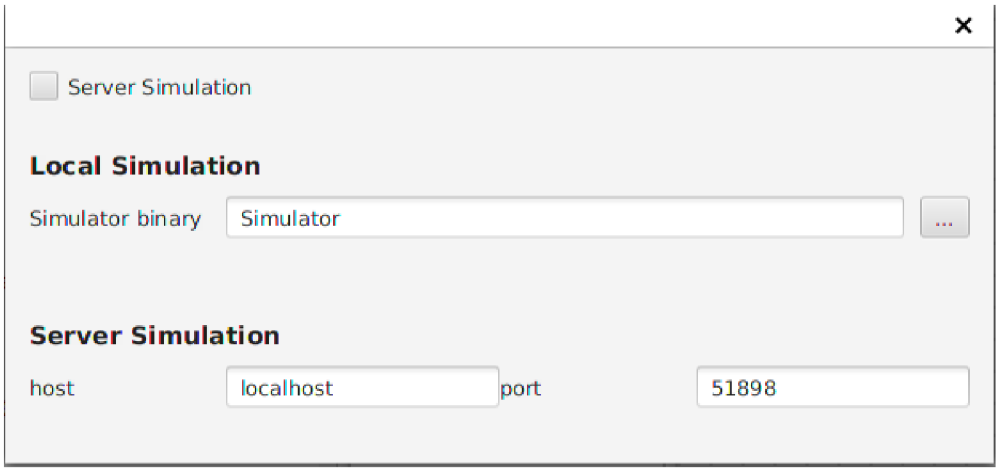
Na hlavnom paneli predeľujúcom editor zdrojového kódu a sekvenčný diagram sú navrchu základné nastavenia simulácie.



Obr. 6.9: Základné nastavenia simulácie.

- **Tlačítko *DRAW*** odštartuje simuláciu a celý proces vykreslenia diagramu.
- **Ukazateľ stavu** sa napĺňa zľava doprava pri priebehu všetkých krokov, ktoré sú potrebné pre vykreslenie diagramu. Správa nad ukazateľom slovne prezrádza v akom stave sa proces práve nachádza.
- **Použite scenáru** je nepovinné.
- **Kroky simulácie** nastavujú horný limit simulácie po ktorých zaručene skončí.
- **Východzia trieda** bude počiatočná pre simuláciu.

Zvyšok nastavení je ukrytých v hornej lište. Dá sa v nich definovať, či sa chceme spoľahnúť na simuláciu pomocou lokálneho programu alebo vzdialeného serveru.



Obr. 6.10: Nastavenia vzdialenej simulácie.

Výstup

Sekvenčný diagram je vykreslený na pohyblivé plátno. Vytvára sa pre každý vizuálne viditeľný objekt dodatočná štruktúra, väčšinou je doplnená o pomocné koordináty a údaje

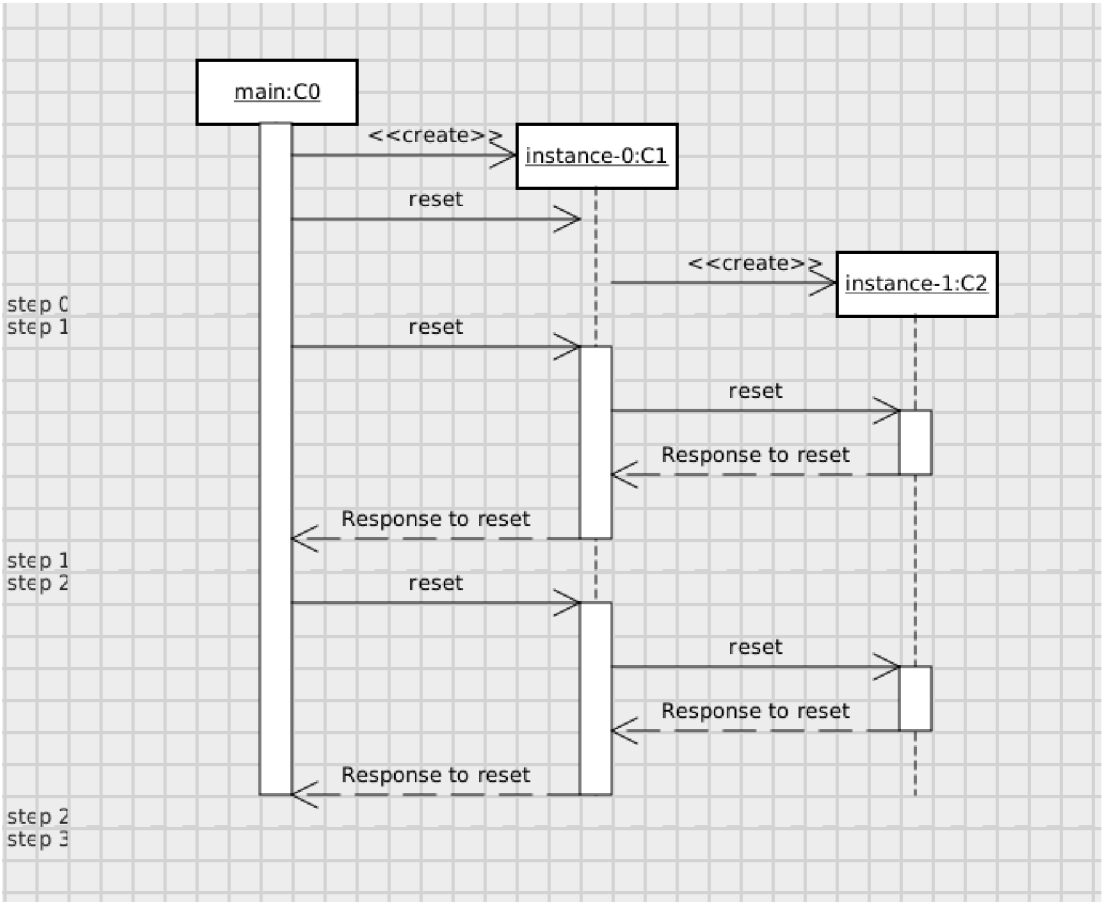
súvisiace s vizuálnym aspektom.

```
class PNObject() {
    val root : Group = Group()
    val header : StackPane = StackPane()
    val rect : Rectangle
    var lifeline : Line
    val spans : LinkedList<PNSpan> = LinkedList()
    var spansCount : Int = 0
}
```

Všetka konfigurácia konštant sa berie z objektu `PNConfiguration`. (Dĺžky, medzery,..)

```
val rect : Rectangle = Rectangle(
    PNConfiguration.instanceSize.x, PNConfiguration.instanceSize.y
)
```

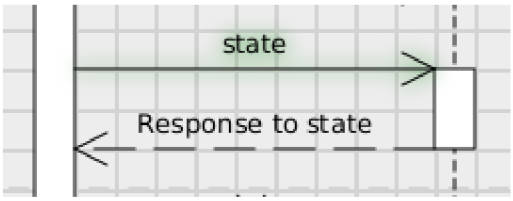
Výstupom sú vykreslené primitívy rámca *Javafx*.



Obr. 6.11: Diagram na výstupnom plátne.

Interaktívne ladenie

Spätné mapovanie na kód je realizované pomocou regulárnych výrazov zo sekcie 6.3.2 . Výstup je interaktívny, čo znamená, že užívateľ môže kliknúť na časť diagramu a zistiť stav miest pre danú inštanciu a pre daný čas (y-psilonová osa), a zároveň vyobraziť časť kódu, ktorá opisuje dané chovanie.



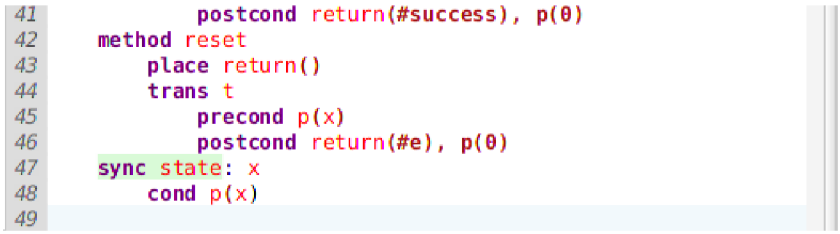
Obr. 6.12: Podsvietená časť diagramu podľa výberu.

Zobrazovanie stavu miest inštancie bolo implementované pomocou jednoduchej tabuľky. Funguje na princípe aplikovaní zoznamu zmien od začiatku simulácie po zvolený čas.

NAME	VALUE
p1	
p2	0x55fe7...
p4	
p3	1, 2

Obr. 6.13: Miesta a ich hodnoty vyobrazené v tabuľke.

V editore je vybraná časť kódu, získaná z hraníc zhody regulárneho výrazu, tiež podfarbená rovnakou farbou, aby to evokovalo zhodu.



Obr. 6.14: Podsvietená časť kódu podľa výberu.

Export do vektorového formátu

V práci je implementovaný export do formátu svg pomocou balíka **batik-svggen**. Balík však nepodporuje jednoduchý prechod Javafx grafických primitív do exportovateľného formátu.

Preto ku každej časti diagramu bola pracne implementovaná funkcia `exportSVG`, ktorá vytvorí identickú grafickú reprezentáciu, no už v exportovateľnom formáte.

Kapitola 7

Testovanie

Testoval sa generátor pre sadu systémov popísaných v jazyku PNtalk. Taktiež sa kontrolovala jeho integrácia s ostatnými komponentami.

7.1 Testy pre nástroj

K účelom testovania bola zhotovená sada validačných testov. Jedná sa o modely Petriho siete definované v jazyku PNtalk. Ich správnosť bola najprv overená validáciou nameraných hodnôt, ktoré zobrazoval v čase ladiaci nástroj v miestach. Tým sa overilo, že prechod sa zachoval podľa modelovaného systému.

7.2 Integračné testovanie

Pre potreby ladenia sa vytvorilo prostredie s testovacími servermi. Toto prostredie bolo vytvorené pomocou technológie **Docker** a **docker-compose**.

Docker

Bol zvolený prístup kontajnerov technológie Docker namiesto robustných virtuálnych strojov. Keďže virtuálne stroje obsahujú separátne jadro operačného systému ich veľkosť sa pohybuje okolo sto či tisíc Megabytov. Zatiaľ čo novo vzniknutý kontajner obsahuje len referenciu na obraz vrstvy súborového systému a nejaké meta dáta konfigurácie, čo vyjde na zopár desiatok kilobytov [14]. Vďaka tejto redukovanej pamäťovej stope sa urýchlil vývoj. Kontajnery sa spúšťali rýchlo, ani ich reštart nebol nijak časovo náročný.

Nástroj Compose

Docker Compose je nástroj pre definovanie a beh mnoho-kontajnerových Docker aplikácií. S nástrojom sa používa súbor na konfiguráciu služieb aplikácie. Potom jediným príkazom dokážeme vytvoriť a spustiť všetky služby z konfigurácie [1].

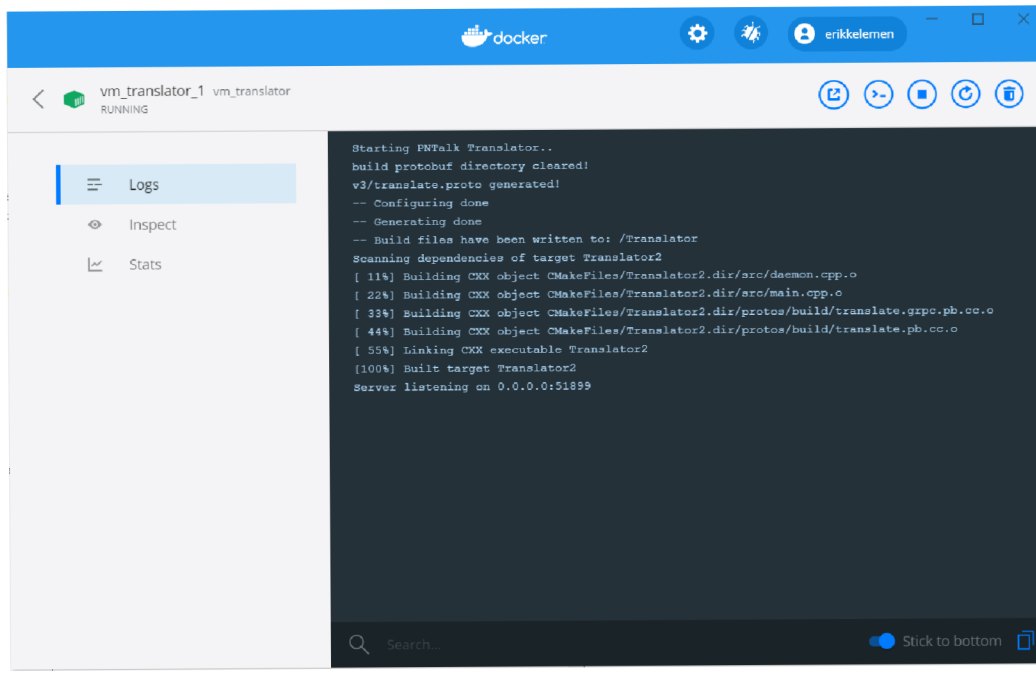
```
1 docker-compose up
```

Konfiguráciu bolo nutné vytvoriť pre službu simulátora a pre prekladač do medzikódu podľa implementácie . Pre službu simulátora sa definoval port 51898 a pre prekladač do me-

dzikódu port 51899. V konfigurácii na Obr. 7.1 vidíme otvorené práve tieto dva porty. Služby využívajú Docker obrazy so zdrojovými kódmi a prerekvizitami k prekladu. Sledovanie spusteného kontajneru je vyobrazené na Obr. 7.2 pomocou aplikácie **Docker-desktop**, ktorý bol použitý na sledovanie komunikácie so servermi.

```
services:
simulator:
build: ./VM
ports:
- "51898:51898"
links:
- "translator"
volumes:
- "./VM:/VM:rw"
translator:
build: ./Translator
ports:
- "51899:51899"
volumes:
- "./Translator:/Translator:rw"

```

Obr. 7.1: Použitá konfigurácia v docker-compose.yml.



Obr. 7.2: Náhľad do kontajneru v aplikácii Docker Desktop.

Kapitola 8

Záver

Cieľom práce bolo implementovať nástroj pre generovanie sekvenčných diagramov. Zámer práce sa podarilo splniť vo všetkých bodoch.

Práca demonštroje automatický prevod objektovo orientovaných Petriho sietí na sekvenčné diagramy, generovanie však pokrýva len podmnožinu sekvenčných diagramov. Aspekt času je v generovaných sekvenčných diagramoch značne degradovaný, nedá sa definovať rôzne časové spomalenie pre obdržanie odpovede správy. Editor kódu zvláda v terajšom stave zvýrazňovanie v kóde a mapovanie častí sekvenčného diagramu k odpovedajúcim častiam kódu, ktoré popisujú chovanie danej časti. Obe tieto funkcionality náramne uľahčujú tvorenie a ladenie kódu.

V práci by som chcel pokračovať implementovaním filtrovania správ a hľadaním v dátach simulácie vzory cyklov či podmienok. Scenáre si zaslúžia dokončiť podporu pre inicializovanie referencií. Potenciál vidím aj v zlepšení simulátoru, ktorý by mohol dosahovať lepších časov simulácie a dať tak možnosť vykresľovať sekvenčný diagram responzívne, prakticky ihneď po akejkoľvek zmene v kóde jazyka PNtalk. Za úvahu by stálo i rozšírenie vývojového prostredia. Editor kódu by mohol skúšať dopĺňať kód podľa prvých napísaných znakov a pozícií v kóde. Implementované by to mohlo byť rozhodovacím stromom.

Testovanie prebehlo uspokojivo, práca ma obohatila v implementácii užívateľských rozhraní a v budovaní systémov s rozdelením práce medzi viac súčinných zložiek. Po dokončení cítim pozitívny dopad a som vďačný za túto sebarealizáciu, ktorá ma posunula v mojom obore o kus ďalej.

Literatúra

[1] *Docker Docs*. [Online; navštívené 20.7.2020]. Dostupné z: https://docs.docker.com/compose/.

[2] *Github Language Stats 2020, second quarter* [https://madnight.github.io/githut/#/pull_requests/2020/2]. Accessed: 2020-07-30.

[3] *Github Octoverse report Over the past year* [https://octoverse.github.com/]. Accessed: 2020-07-30.

[4] *TornadoFX Guide gitbook* [https://edvin.gitbooks.io/tornadofx-guide/]. Accessed: 2020-07-30.

[5] *Success in Disruptive Times Expanding the Value Delivery Landscape to Address the High Costof Low Performance* [[online]]. 2018. [Navštívené 20.7.2020]. Dostupné z: https://www.pmi.org/-/media/pmi/documents/public/pdf/learning/thought-leadership/pulse/pulse-of-the-profession-2018.pdf.

[6] Alhir, S. S. *Learning UML*. OReilly, 2003.

[7] Arms, W. Y. *Scenarios and Use cases 2020, second quarter* [[online]]. Cornell University, Naposledy navštíveno 20. 7. 2020. Dostupné z: https://www.cs.cornell.edu/courses/cs5150/2018sp/slides/7-use-cases.pdf.

[8] Bause, F. a Kritzinger, P. S. Generalized Stochastic Petri Nets. *Stochastic Petri Nets*. 2002, s. 141–163. DOI: 10.1007/978-3-322-86501-49.

[9] Blažek, T. *Interpret Petriho sítí*. Brno, CZ, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/22744/.

[10] Czarnecki, K. a Eisenecker, U. In: *Generative programming: Methods, tools, and applications*. Addison Wesley, 2005.

[11] DENNIS, A., WIXOM, B. H. a ROTH, R. M. *Systems Analysis and Design, 5th Edition*. John Wiley & Sons, 2012. ISBN 978-1-118-05762-9.

[12] Holliday, M. A. a Vernon, M. K. A Generalized Timed Petri Net Model for Performance Analysis. *IEEE Transactions on Software Engineering*. 1987.

[13] Janoušek, I. V. *Modelování objektů Petriho sítěmi*. 1998. Dizertačná práca. Vysoké Učení Technické v Brně.

[14] Kane, S. *Docker: up & running: shipping reliable containers in production.* Sebastopol, CA: O'Reilly Media, 2018. ISBN 9781492036739.

[15] KURUPPU, D. *GRPC : up and running.* Place of publication not identified: O'REILLY MEDIA, INC, USA, 2019. ISBN 978-1492058335.

[16] Lakos, C. a Keen, C. LOOPN++: a new language for object-oriented Petri nets. 1994.

[17] Lapšanský, T. *Virtuální stroj Petriho sítí.* Brno, CZ, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/21442/.

[18] Ltiprogrammed, M., Tations, C., Dennis, J. a Horn, E. Programming Semantics For Multiprogrammed Computations. *Communications of the ACM.* August 2002, zv. 26. DOI: 10.1145/357980.357993.

[19] Marsan, M. *Modelling with generalized stochastic Petri nets.* Chichester New York: Wiley, 1995. ISBN 9780471930594.

[20] Merlin, P. a Farber, D. Recoverability of Communication Protocols - Implications of a Theoretical Study. *IEEE Transactions on Communications.* 1976, zv. 24, č. 9, s. 1036–1043.

[21] Milicev, D. *Model-driven development with executable UML.* Indianapolis, IN: Wrox/Wiley, 2009. ISBN 9780470481639.

[22] Petri, C. *Communication with Automata.* Rome Air Development Center, Research and Technology Division, 1966. AD-630. Dostupné z: https://books.google.cz/books?id=1D7FSgAACAAJ.

[23] Ramamoorthy, C. V. a Ho, G. S. Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets. *IEEE Transactions on Software Engineering.* 1980, SE-6, č. 5, s. 440–449.

[24] Ramchandani, C. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets.* 1974. Dizertačná práca. MIT, Cambridge, MA.

[25] Rozenberg, G. *Advances in Petri nets, 1990.* Berlin New York: Springer-Verlag, 1991. ISBN 3540538631.

[26] Sibertin Blanc, C. Cooperative Nets. In: Valette, R., ed. *Application and Theory of Petri Nets 1994.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, s. 471–490. ISBN 978-3-540-48462-2.

[27] Whitten, J. *Systems analysis and design methods.* Boston: McGraw-Hill/Irwin, 2007. ISBN 978-0073052335.

Príloha A

Obsah přiloženého paměťového média

Pamäťové médium priložené k práci obsahuje upravenú verziu práce Tomáša Lapšanského, Virtuální stroj Petriho sítí. Zdrojové kódy Generátoru Sekvenčných diagramov a textu práce. Všetky ukážkové príklady, testy a technickú dokumentáciu.

`/pntalk_virtual_machine` (Upravené zdrojové kódy práce T. Lapšanského)
`/pntalk_virtual_machine/VM/Dockerfile` (Predpis zostavenia Docker obrazu)
`/pntalk_virtual_machine/Translator/Dockerfile` (Predpis zostavenia Docker obrazu)
`/pntalk_virtual_machine/docker-compose.yml` (Predpis funkčného prostredia Docker obrazov)
`/pntalk_sequencer` (Generátor Sekvenčných diagramov z modelu OOPN)
`/pntalk_sequencer/tests` (Sada testov)
`/text` (Zdrojové kódy textu práce)
`/text/xkelem01.pdf` (Text práce)

Príloha B

Manuál

Preklad textu práce LaTeX:

Vyžaduje balíčky:

1. texlive-lang-czechslovak
2. texlive-fonts-extra

Preklad generátoru sekvenčných diagramov:

Vyžaduje balíčky:

1. maven

Spustenie:
`mvn clean javafx:run`

Preklad:
`make compile`

Spustenie prostredia na vzdialený preklad:
`docker-compose up`