

shipout/backgroundshipout/foreground



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
AUTHOR

ERIK KELEMEN

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. RADEK KOČÍ, Ph.D.

BRNO 2020

Abstrakt

Táto práca sa 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 diskretná simulácia modelu popísaná jazykom PNTalk. Výsledkom práce je plne automatický generátor, ktorý pomocou dát zo 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 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 most trustworthy result of valid sequence diagram.

Kľúčové slová

objektovo orientované petriho siete, sekvenčný diagram, simulácia.

Keywords

object oriented petri nets, sequence diagram, simulation

Citácia

KELEMEN, Erik. *Generování sekvenčních diagramů z modelů Petriho sítí*. Brno, 2020. 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. Další informace 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

5. mája 2021

Podakovanie

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

Obsah

| | | |
|----------|---|-----------|
| 1 | Úvod | 4 |
| 2 | Tvorba a analýza scenárov v softvérovom inžinierstve | 7 |
| 2.1 | Scenáre | 7 |
| 2.1.1 | Tvorba scenárov | 7 |
| 2.2 | Vývoj systému | 8 |
| 2.2.1 | Zúčastnená strana | 8 |
| 2.2.2 | Iné factory | 9 |
| 2.2.3 | Proces vývoja | 9 |
| 2.2.4 | Analýza systému | 10 |
| 2.2.5 | Návrh systému | 12 |
| 2.3 | Návrh systému | 12 |
| 2.4 | UML | 12 |
| 2.5 | Proces vývoja software | 12 |
| 2.5.1 | iteratívny | 12 |
| 2.5.2 | štrukturovaný | 13 |
| 2.5.3 | inkrementálny | 13 |
| 2.5.4 | Agilný | 13 |
| 2.6 | Tvorba scenárov | 13 |
| 3 | Petriho Siete | 15 |
| 3.1 | Obecná definícia | 15 |
| 3.2 | Objektovo orientované Petriho siete | 17 |
| 3.2.1 | Prerekvizity Petriho Sietí | 17 |
| 3.3 | Paralelizmus v Petriho sieťach | 19 |
| 3.4 | Čas v Petriho sieťach | 21 |
| 3.5 | Jazyk PNTalk | 21 |
| 3.5.1 | Termy | 21 |
| 3.5.2 | Siete objektov a metód | 22 |
| 3.5.3 | Synchrónne porty | 23 |
| 3.5.4 | Konštruktory | 23 |
| 3.5.5 | Trieda | 23 |
| 3.5.6 | Dostupné simulátory | 23 |
| 4 | Sekvenčné Diagramy | 24 |
| 4.1 | Komunikácia v sekvenčných diagramoch | 24 |
| 4.2 | Účastníci komunikácie | 25 |
| 4.3 | Stavebné Elementy sekvenčných Diagramov | 26 |

| | | |
|----------|--|-----------|
| 4.3.1 | objekt | 26 |
| 4.3.2 | čiara života | 26 |
| 4.3.3 | aktivácia | 26 |
| 4.4 | Distribuované systémy | 26 |
| 4.4.1 | Vymedzenie pojmu distribuovaný systém | 26 |
| 4.4.2 | Porovnanie s Centralizovanými systémami | 27 |
| 4.4.3 | Kedy distribuovať | 28 |
| 4.5 | Vývojové prostredie | 29 |
| 4.5.1 | Projektový pohľad | 29 |
| 4.5.2 | Editor zdrojového kódu | 29 |
| 4.5.3 | Preklad | 30 |
| 4.5.4 | Ladenie | 30 |
| 5 | Návrh Implementácie | 31 |
| 5.1 | Architektúra | 31 |
| 5.2 | Transformácia modelu OOPN na Sekvenčný diagram | 31 |
| 5.3 | Out-source simulácie | 36 |
| 5.4 | Užívateľské rozhranie | 37 |
| 5.4.1 | Rozloženie užívateľského rozhrania | 38 |
| 6 | Implementácia nástroja | 40 |
| 6.1 | Výber implementačného jazyka | 40 |
| 6.2 | Implementácie distribuovaného systému | 41 |
| 6.2.1 | virtualizácia | 41 |
| 6.2.2 | Vzdialené volanie procedúr | 43 |
| 6.3 | Užívateľské rozhranie | 44 |
| 6.3.1 | Bohaté internetové aplikácie | 44 |
| 6.3.2 | Editor Zdrojového kódu | 45 |
| 6.3.3 | Projektový pohľad | 45 |
| 6.3.4 | Diagram ako výstup aj interaktívny ladiaci nástroj | 45 |
| 7 | Testovanie | 46 |
| 8 | Záver | 47 |
| | Literatúra | 48 |
| A | Obsah přiloženého paměťového média | 50 |
| B | Manuál | 51 |

Zoznam skratiek

OOPN Objektovo orientované Petriho siete (Object-oriented Petri Nets)

PT miesto/prechod (Place/Transition)

CPU Centrální procesorová jednotka (Central processing unit)

GC Způsob automatické správy paměti (Garbage collector)

GPU Grafický procesor (Graphic processing unit)

GUI Grafické uživatelské rozhraní (Graphic User Interface)

HW Hardware

IT Informační technologie (Information Technology)

UI Uživatelské rozhraní (User Interface)

SW Software

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 sa zainteresované strany, sa spolu nezhodnú, či omylom interpretujú zadanie inak. Z pohľadu jednotlivca je ťažké udržať si prehľad o celom projekte, mať prehľad o komponentách, čo robia a aké požiadavky majú spĺňať. Dnešné systémy sú príliš rozsiahle, 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 manažmentu (anglicky Project Management Institute, skratkou PMI ¹), do ktorej prispelo 4455 praktikov projektového manažmentu z praxe. Z nich najväčšiu časť tvorili odborníci z odvetvia informačných technológií. 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%), neadekvátne vízie(29%) a slabá komunikácia(29%). [5] Ako vidno zo štatistík komunikácia stále predstavuje dosť veľký kameň úrazu.

V pracovnej komunikácii a v pochopení systému nám môžu pomôcť takzvané modelovacie jazyky. Niečo, čo nám umožní ujasniť si myšlienku o časti systému na ktorom pracuje tím z druhej polovice zemegule. Najlepšie by bolo zvoliť modelovací jazyk tak, aby mu rozumela aj menej technicky zdatní účastníci projektu. Asi pre majiteľa projektu nie je vždy prirodzené zorientovať sa v pseudo kóde a podobných technikáliach. 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 pre zkonštruovanie diagramov pre tie najzákernejšie scenáre chovania systému. Ďalšiu nevýhodou je možná chyba ľudského faktoru. Aj ten sebestačnejší odborník na vyvíjaný systém občas spraví chybu.

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. A to všetko automaticky a neomyľne. Tomuto grafickému pohľadu na časť systému by roz-

¹Project Management Institute www.pmi.org

²Respondenti na túto otázku mohli uviesť viac príčin než jednu.

umeli nielen špecialisti z oboru, ale aj technicky menej znalí účastníci projektu. Uľahčila by sa tým komunikácia s užívateľmi, či vlastníckmi projektu. Takýto generátor by otvoril dvere novým možnostiam pri špecifikácii požiadavkov systému, analýze a návrhu systému. Napríklad pri každej oprave by sme mohli ukázať chovanie zaznamenané diagramom pred našou zmenou a po nej, čo by urýchlilo validáciu. Vývojové tímy, by sa ľahšie zorientovali v komponentách, ktoré vyvíjal iný tím a podobne.

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 priekopnícka, už v minulosti boli podobné pokusy. Napríklad executive UML (xUML), sa vydal opačnou cestou a dovoľuje automaticky generovať spustiteľné a simulovateľné prototypy z diagramov [20]. Tento prístup však nepodporuje návrat k modelu. Akékoľvek zmeny v prototype sa nedajú previesť späť. Niekoľko komerčných nástrojov sa pokúša o vykresľovanie behu procesu vizuálne priamo zo zdrojového kódu. Tento prístup však vykresľuje tok procesu na úrovni kódu. Čo môže byť rovnako neprehľadné ako kód samotný.

////////////////////

V roku 1997 vznikol jazyk UML (anglicky Unified Modeling Language), ktorý definoval notáciu pre širšiu skupinu diagramov pokrývajúce radu esenciálnych pohľadov na modelovaný systém. Jazyk UML sa rýchlo stal štandardom pre diagramy používané na modelovanie systémov. Jeho najprirodzenejšie využitie je práve u objektovo orientovaných modelov.

Isteže existujú aj rozšírenia UML a metódy na ich prevod do spustiteľnej formy ako MDA methodology, Executable UML (xUML) language alebo Foundational Subset pre xUML. Všetky zo zmienených metód však trpia neduhom, pri ktorom spustiteľná forma UML modelu v priebehu validácie upravuje, je takmer nemožné vrátiť sa so zmenami k pôvodnému modelu.

Cieľom tejto práce je práve zostrojiť generátor, ktorý z modelu systému vygeneruje diagram zachytávajúci scenár chovania pri ľubovoľnej aktivite. K dosiahnutiu tohto odvážneho cieľa bude potreba prispôbiť už existujúcu implementáciu simulátoru modelu OOPN. Najdôležitejším krokom je samotná transformácia dát zo simulácie na sekvenčný diagram. Ako posledný krok by malo byť realizovanie spätného mapovania aktivít do modelu OOPN.

Sekvenčný diagram patrí do jazyka UML od roku 1997(štandard v1.1) ako jeden z diagramov na modelovanie interakcií v systéme. Na druhej strane máme Petriho siete(PT), matematický model, ktorý je schopný vyjadriť kauzalitu udalostí, asynchrónnosť, paralelizmus a synchronizáciu v modelovanom systéme. Petriho siete sa do UML dostali len ako inšpirácia pre diagram aktivít v roku 1999(v 1.3). Na prvý pohľad je zrejmé, že diagram interakcií s matematickým modelom Petriho sietí má pramálo spoločného a táto absencia relevantných informácií zrejme neumožňuje automatické generovanie z jedného modelu na druhý.

To sa zmení pri transformácii PT Petriho sietí do funkcionálnych Petriho sietí (FPN) a následnou transformáciou do objektovo orientovaných Petriho sietí (OOPN). Týmto prechodom sa priblížia invokačné prechody z funkcionálnych Petriho sietí k volaniam správ ako ich poznáme zo sekvenčných diagramov. Triedy OOPN sa priblížia k objektom sekvenčných

diagramov. Táto analógia je základným stavebným kameňom pre vytvorenie funkčného generátoru sekvenčných diagramov z objektovo orientovaných Petriho sietí.

Kapitola 2

Tvorba a analýza scenárov v softvérovom inžinierstve

Pri modelovaní systémov uľahčuje analýzu a modelovanie vytvorenie scenáru. Takýto scenár poskytuje náhľad na určitú časť systému 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.

2.1 Scenáre

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 organizované pomocou entitno-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 4 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 datových typoch. Algoritmy môžu byť špecifikované pseudo kódom alebo špecifickou notáciou.
- Data-centrický štýl: The main aspect of the data-centric style is the structure of the data e.g. in database modeling. The structure of the data can be modeled using entity-relationship or object diagrams.

2.1.1 Tvorba scenárov

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

Doménové modelovanie

Predstavme si, že by každý v tíme hovoril rozdielnym jazykom. Povedzme niekto česky, niekto francúzsky a niekto ďalší zasa Svahilsky. Vždy keď niekto prehovorí, ostatní zachytia akýkoľvek význam týchto cudzích slov, ktorý dokázali zachytiť. Po každej porade, tak odchádzajú s naprosto milnou interpretáciou odznených slov.

Príklad s rozdielnymi jazykmi je jasne nadstretý, ale povedzme, že pracujeme na reklamnom systéme a hovoríme o pojmoch ako impresia reklamy, preklik reklamy, garantovaná reklama. Pre niekoho môžu byť tieto slová cudzím jazykom.

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.

Tvorba scenárov a prípadov využitia bez vytvorenia doménového modelu má za následky nezrozumiteľnosť. Scenár sa potom naozaj môže javiť ako napísaný v nezrozumiteľnom jazyku.

Problematicky scenárov

Pri vytváraní scenárov sa často zabúda na tzv. "rainy-days"scenáre. Voľne preložené ako scenáre za daždivých dní, ktoré opisujú chovanie systému v prípade chýb, zotavenie sa z chyby alebo nejakej drastickjšiu reakciu. Predchádzať tomuto nedostatku môžeme

2.2 Vývoj systému

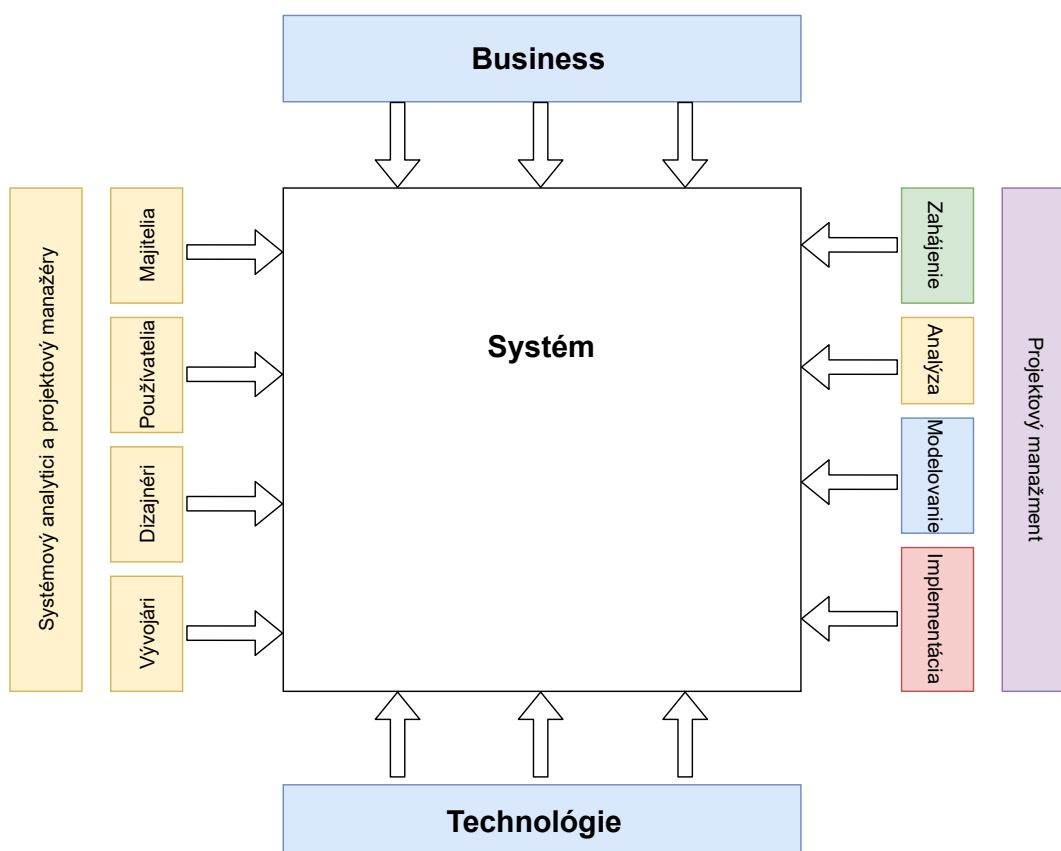
Pred ponorením sa do analýzy a modelovania softvéru, je nutno zmieniť, kde majú pri vývoji softvéru svoje miesto a aké aspekty ich ovplyvňujú.

2.2.1 Zúčastnená strana

Všetky fyzické osoby ovplyvňujúce vývoj softvéru môžeme pre akýkoľvek systém klasifikovať do 5 skupín. Zobrazené sú na ľavej strane Obr. 2.1. Podstatné je, že každá zo skupín má na systém iný uhol pohľadu.

1. **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 a projektu a vývojármi.
2. **Vývojári**, ktorí majú za úlohu celý systém zkonštruovať podľa návrhu softvérového dizajnéra riešia hlavne detaily implementácie. V menších firmách sú dizajnéry a vývojári tí istí ľudia, no vo väčších sú tieto úlohy často oddelené.
3. **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 databázový administrátori, sieťový architekti, bezpečnostný experti a mnohí ďalší.
4. **Užívatelia** systému, sú v dnešnej dobe čím ďalej technicky vyspelí a ďalšou ich nespornou výhodou je počet, ktorý väčšinou prevyšuje ostatné skupiny. Z ich pohľadu na systém je najdôležitejšia funkcionálna, intuitívnosť používania a o cenu, či profit, narázdil od majiteľov, nedať.

5. **Majitelia** projektu, ktorých môže byť viac než jeden väčšinou riešia projekt z pohľadu financií. Na koľko ich to vyjde, aký bude profit, či benefits.



Obr. 2.1: Aspekty ovplyvňujúce vývoj systému [28]

2.2.2 Iné factory

Okrem Účastníkov vývoja majú na systém vplyv ešte aspekty businessu a technológie dostupné v dobe vývoja. Business pokrýva hlavne požiadavky obchodu spojené s legislatívou. Technológie nás obmedzujú pri nedostupnosti tak pokročilých technológií aké by sme potrebovali pre svoj systém alebo naopak nové prielomy v technológii poskytujú príležitosť pozdvihnúť projekt na vyššiu úroveň.

2.2.3 Procces 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. Akiste sa budú tieto metodológie od seba diametrálne odlišovať pre jednotlivé organizácie. Avšak, všetky metódy riešenia problému sa dajú zavšeobecniť na kroky, ktoré sú spoločné ¹:

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

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 ??.
3. **Identifikovať požiadavky a očakávania riešenia**, ktoré kladú nároky obchodu či funkcionálna stránka vyžadovaná užívateľmi.
4. **Identifikovať alternatívne riešenia** a zvoliť najvhodnejšiu cestu. Pri výbere zohráva rolu rozpočet (finančný i časový), predispozície realizačného tímu a uprednostnené ciele.
5. **Navrhnuť 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 nutno objektívne zhodnotiť výsledky v zmysle splnenia cieľov. Pri nesplnení sa môžeme vrátiť ku kroku 1 a 2.

Na obrázku 2.1 je na pravej strane zobrazený pohľad procesu vývoja, ktorý bol kvôli jednoduchosť 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 za úspešným návrhom systému. Jednotlivé kroky zovšeobecneného riešenia problémov do fáz vývoja je v tabuľke 2.1.

2.2.4 Analýza systému

Analýza systému patrí k ranným fázam vývoja systému. Jedná sa o študovanie systému a jeho komponent. V sekcii 2.2.3 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 informácií o systéme, alebo aspoň dostačujúcich pre následnú fázu návrhu systému. V predchádzajúcej sekcii bola opísaná zúčastnená strana podieľajúca sa na analýze a ciele analýzy, no samotná otázka ako analyzovať systém? bola doteraz len nonšalantne opomíjaná. 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 zrozumiteľné obrázky na opis problémov, požiadavkov a riešení v systéme. Takou grafickou reprezentáciou môžu byť napríklad vývojové diagramy,

| Zjednodušený vývojový proces | Kroky zovšeobecného 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. Navrhnuť zvolené riešenie |
| Implementácia systému | 6. Implementovať zvolené riešenie 7. Vyhodnotiť výsledok |

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

štrukturované grafy a iné schémy. Tento druh analýzy študuje systém, aby zistil jeho chovanie a jeho výstupom sú grafické reprezentácie popisujúce chovanie systému. Takáto grafická reprezentácia môže odhaliť úskalía v projekte a prakticky rovno načrtnúť návrh implementácie v neskorších fázach.

1. **Štruktúrnej analýza**, ako jedna z tradičných foriem analýzy zo 70. rokov používaná do dnes je zameraná na tok dát a analyzuje systém z pohľadu procesov. :TODO: obr dataflow
2. **Informačné inžinierstvo** ako ďalší tradičný prístup narozdiel od sledovania dát v procese, sleduje štruktúru uloženia dát naprieč systémom.
3. **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 bude venovaná celá sekcia :TODO:

Prototypovanie

Okrem modelovo orientovanej analýzy môžeme 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 diagramy, ktoré by ho opísali. Namiesto toho, vďaka rýchlej prototypizácii sa pokúša systém implementovať. Na týchto pokusoch potom stojí analýza systému a odhalovanie chýb, či nezrovnalostí.

2.2.5 Návrh systému

objektovo orientovaný návrh

2.3 Návrh systému

2.4 UML

1. Diagram Použitia
2. Diagram Aktivít
3. Diagram Tried znázorňuje systémovú objektovú štruktúru. Ukazuje triedy objektov, ktoré v systéme figurujú ako aj ich väzby medzi sebou.

2.5 Proces vývoja software

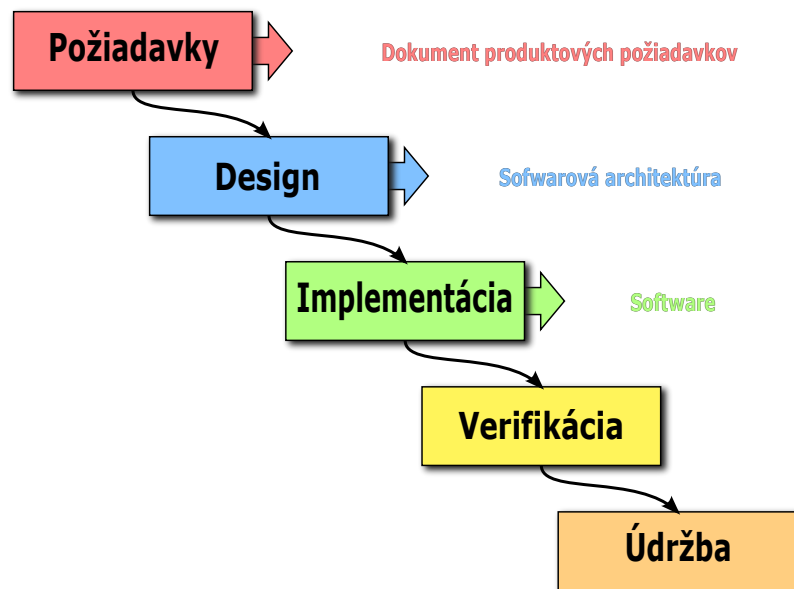
Systémy a produkty v 50. rokoch boli primárne hardvérového charakteru. Logické a matematické výpočetné procesy boli implementované pomocou elektromechanických zariadení. V 60. rokoch už integrované obvody umožnili vývojárom zvýšiť spoľahlivosť a presnosť výpočtovej techniky do istej miery aj zásahom softvéru. V tejto dobe však chyby alebo zmeny návrhu boli drahé a časovo náročné. S Príchodom mikroprocesorovej technológie, začiatkom 70. rokov, sa software stal schodnou alternatívou pre integrované obvody. Schopnosti systému, ktoré museli byť inak implementované pomocou harvéru zrazu boli implementovateľné omnoho jednoduchšie a rýchlejšie softwarom. Výsledkom bola evolúcia v návrhu systému, ktoré začali modelovať flexibilné a rekonfigurovateľné systémy.[26] To umožnilo vývojárom vytvárať špecifické aplikácie len jednoduchým prispôbením softwaru. Prechod k softvérovým systémom nastal ešte pred vznikom metód na produkovanie softwaru. Keď sa aplikácie stávali robustnejšími a komplexnejšími zvyšoval sa aj risk a pribúdali problémové oblasti. Dodržanie technických očakávaní, rozpočtu, a času odovzdania boli hlavné problémy. Vznikla potreba pre opakovanie a predvídateľnosť úspešných vývojových procesov. Začali vznikať opakovateľné metodológie procesu vývoja. software.

2.5.1 iteratívny

Iteratívne procesy akceptujú možnosť zmien a opakujú do istej miery fáze požiadavok, designu a implementácie. Jedným z iteratívnych procesov je Rational Unified Process (RUP)

2.5.2 štrukturovaný

Štrukturované modely striktne oddeľujú jednotlivé etapy vývoja, to ich robí nepraktickými pre dnešnú turbulentnú dobu, kde sa špecifikácie systému menia aj za behu vývoja software. Predstaviteľom tohto prístupu je napríklad model Vodopád (Waterfall). Vznikol ako Jeden z prvých pokusov charakterizovať softwarový vývoj ako model. Ilustrácia je na Obr. ??.



Obr. 2.2: Vodopádový model

Vo **vodopádovom** modeli sú vývojové etapy vykonávané sekvenčne s minimálnym prekrytím a bez iterácií medzi etapami. Šípky nevedú naspäť do predchádzajúcich etáp. Uživatelské potreby sú stanovené. Požiadavky sa nemenia. Celý systém je navrhnutý, implementovaný a otestovaný pre konečné odovzdanie.

2.5.3 inkrementálny

2.5.4 Agilný

Tento prístup sa vyznačuje flexibilitou. Funguje na princípe extrémne krátkych iterácií. V tomto procese máme fungujúci systém, ktorý neustále rozširujeme.

2.6 Tvorba scenárov

Scenár môže byť chápaný celou radou interpretácií, niektoré z nich sú uplatniteľné v systémovej inžinierstve. Scenár môže byť 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 sa budeme zaoberať výlučne konkrétnym scenárom systému, abstrakcia sa zo scenáru odstráni doménovým modelovaním. Viac o tejto problematike v sekcii ??.

Scenár môže byť chápaný celou radou interpretácií, niektoré z nich sú uplatniteľné v systémovej inžinierstve. Scenár môže byť 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 sa budeme zaoberať výlučne konkrétnym scenárom systému, ako súčasť požiadavkov systému definovaných pri analýze systému. Takýto konkrétny scenár bude pohľad na systém zachytávajúci prípad užitia.

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 princípmi objektovej orientácie. V neposlednej rade kapitola rozoberie možné zavedenie času do Petriho sietí.

3.1 Obecná definícia

Ako východziu Petriho sietí pre ďalšie varianty a rozširania použijeme sieť definovanú v literatúre ako PT-sieť (Place/Transition Net), [Pet81, Rei85], je zobecnením jednoduchšieho modelu CE-sietí (Condition-Event Net).

Poznámka 3.1.1. CE-sieť narozdiel od PT zobecnenia umožňuje do miest ukladať len jednu značku, miesta v tejto sieti nadobúdajú len booleovských hodnôt. Prechody CE-sietí sú provediteľné 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 zobecnenia.

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. $PI_N : P_N \rightarrow \mathbb{N}$ 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) $PRECOND_t^N : P_N \rightarrow \mathbb{N}$ sú vstupné podmienky (vstupy) prechodu
 - (b) $POSTCOND_t^N : P_N \rightarrow \mathbb{N}$ 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 počiatkové značenie siete N .

Dynamika Petriho sietí spočíva vo vykonávaní prechodov. Ich provediteľnosť závisí na značení 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 **provediteľný** 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 provediteľný v značení 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, popsaného množinou stavových strojov, je určený množinou stavov jednotlivých strojov. Stav (stavová premená) systému je distribuovaný do množiny parciálnych stavov systému. Prechody sa vykonávajú v jednotlivých strojoch je však potreba synchronizovať

Parciálne stavy systému sú modelované miestami a vzormi možných udalostí jsou de novány přechody. Místo se v grafu Petriho síti vyjadřuje jako a přechod jako . Okamžitý stav systému je de

nován umístěním značek (tokens) v místech, což v grafu Petriho siete vyjadrujeme tečkami v místech. Přítomnost značky v místě modeluje skutečnost, že daný aspekt stavu (parciální stav) je momentálně aktuální, resp. podmínka je splněna. Každý přechod má de

nována vstupní a výstupní místa, což je v grafu Petriho síti vyjádřeno orientovanými hranami mezi místy a přechody: ! a ! . Tím je deklarováno, které aspekty stavu systému podmiňují výskyt odpovídající události (provedení přechodu), a které aspekty stavu jsou výskytem této

3.2 Objektovo orientované Petriho siete

V sekcii :TODO: boli opísané niektoré princípy objektovej orientácie. Petriho sietí v obecnom zavedení sa nedá využiť pre tieto princípy. V diskusii okolo PN bola v práci [24] navrhnutá myšlienka invokačných prechodov, ktorá sa stala základom pre zavedenie OOP do Petriho sietí. Následne vzniklo viac nadväzujúcich prác na túto tému ako LOOPN++ [16], Cooperative Nets [25] 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 nezbytnnej 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 odčítanie 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 operácie s multimnožinami.

Definícia 3.2.1. Majme ľubovoľnú neprázdnu množinu E . Multimnožina nad množinou E je funkcia. $x : E \rightarrow \mathbb{N}$. 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:

$$x - y = \sum_{e \in E} (x(e) - y(e)) \cdot e$$

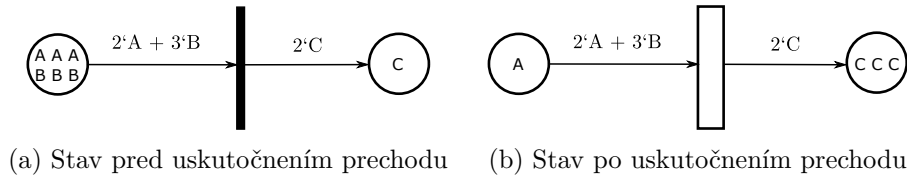
5. veľkosť:

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

Príklad 3.2.1. názorne zápis $2 \cdot A + 3 \cdot B$ predstavuje multimnožinu s tromi 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 $1 \cdot c$.

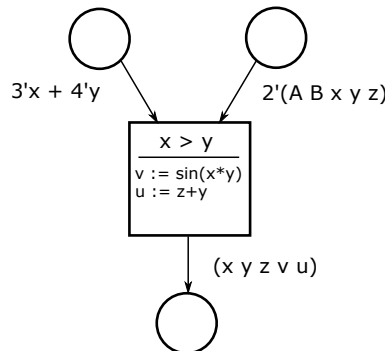
Takúto Multimnožinu môžeme konceptom **hranových výrazov** priradiť k hranám vstupným ako aj výstupným. Názorná ukážka je na Obr. 3.1.



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 strážii prechodu. Príklad strážneho výrazu „ $x > y$ “ aj s naväzovaním premenných je na Obr. 3.2.

Pre sugestívnejší zápis dovoľuje k strážii prechodu pridať **akciu prechodu**, odlišujúcu výpočty, ktoré sa realizujú pri vykonávaní prechodu, od tých, ktoré sa 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 vznikáť 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** [24].

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 inštancovanie Petriho sietí

V sekcii 3.2.1 venovanej hierarchickým PN bolo ukázané statické inštancovanie podsietí do nadradenej PN. K dynamickému inštancovaniu využijeme 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.

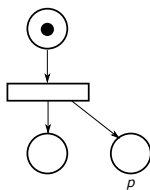
1. Vykoná sa vstupná časť invokačného prechodu, čo spočíva v odobraní značiek zo vstupných miest invokačného prechodu a vo vytvorení novej inštancie špecifikovanej stránky. Do vstupných miest stránky sú umiestnené značky, odobrané zo vstupných miest invokačného prechodu.
2. Invokovaná sieť potom beží nezávislo na ostatných inštanciách sietí v systéme pokiaľ v nje 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 invokačného 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 princípmi.

3.3 Paralelizmus v Petriho sieťach

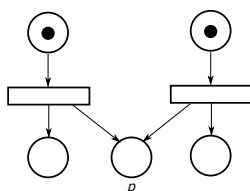
Paralelizmus môže byť prenesený do Petriho sietí viacerými spôsobmi.

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.



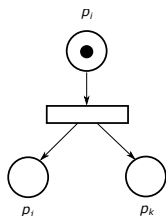
Obr. 3.3: Ukážkový proces

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

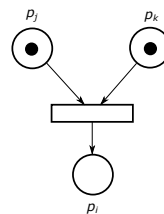


Obr. 3.4: Ukážka zloženia dvoch sietí. V praxi neúčinné.

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. Ich prevedenie do Petriho siete je nasledovné:

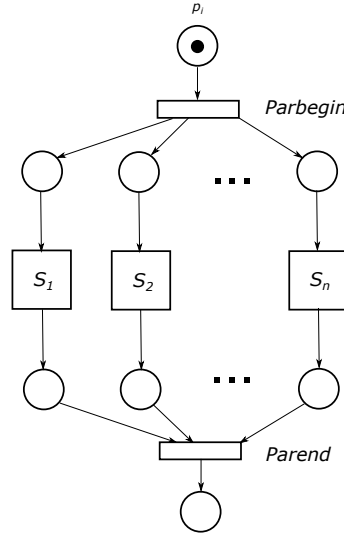


Obr. 3.5: Operácia FORK vykonaná v mieste p_i vytvorí proces v miestach p_j a p_k .



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** [Dijkstra 1968]. Koncept navrhnutý Dijkstrom má všeobecnú formu $parbegin\ S_1; S_2; \dots S_n\ parend$, kde S_i predstavuje výraz. Význam $parbegin|parend$ štruktúry je vykonať každý výraz $S_1; S_2; \dots S_n$ 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 :TODO sa čas nijako nezohľadňoval. Konceptu času sa pôvodne zámerne vyhýbalo v originálnej práci C.A.Petriho [21], 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 štruktúrou Petriho siete [18].

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 ([23], [12], [22], [19]). Spôsoby sú značne ovplyvnené tým, v akom obore boli uplatnené. Práca rozoberie v nasledujúcej sekcii výlučne Stochastickú Petriho sieť.

Stochastické Petriho siete

Generalizovaná stochastická Petriho sieť je

3.5 Jazyk PNTalk

V sekcii ?? 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 /ref).

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 číslíc, ktorej môže predchádzať znamienko mínus a môže za ňou nasledovať desatinná bodka a ďalšia sekvencia číslíc. 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) Znaký 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 parametrová 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 Synchronne porty

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

Špeciálnym prípadom synchronného 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 konstruktor.

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 synchronných portov.

Dedičnosť

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 Interpret v Jave [8].

Kapitola 4

Sekvenčné Diagramy

Jednou zo štyroch základných modelačných techník UML (Unified Modeling Language) 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 použitia a taktiež zobrazuje správy, ktoré si tieto objekty vymieňajú počas časového intervalu. Diagram je dvojdimenzionálny. Účastníci sú zoradení 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 real-time systémom a komplexným prípadom použitia.

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

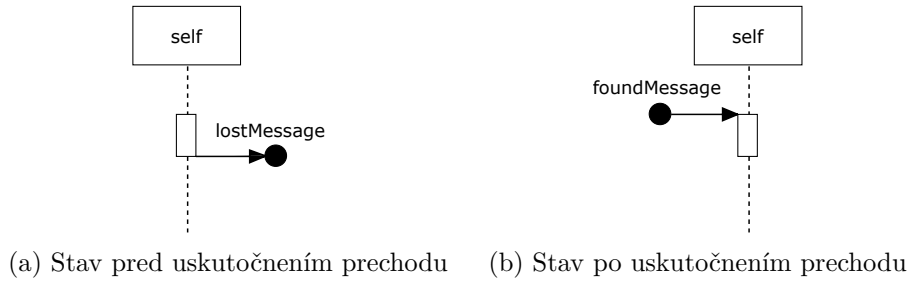
4.1 Komunikácia v sekvenčných diagramoch

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, čo opisuje sekcia :TODO

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 odosielateľa zakončená malou bodkou.

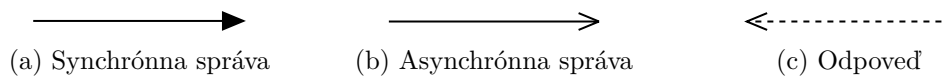


Obr. 4.1: Nekompletné správy

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

V Sekvenčných diagramoch rozlišujeme tri typy správ:

1. **Synchronná 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 paralelné procesy.
3. **Odpoveď** predstavuje spätnú správu po synchronnej správe. Nemôže vzniknúť samostatne.



Obr. 4.2: Reprezentácie troch typov správ

Nad správou sa nachádza štítok správy. Text obsiahnutý v štítku dodržiava nasledujúcu syntax:

4.2 Účastníci komunikácie

Participant komunikácie skrz správy popísané vyššie sú aktívne objekty, ktoré v sekvenčných diagramoch reprezentujeme čiarou života(anglicky *lifeline*).

Definícia 4.2.1. Pri definícii **čiaru 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.

4.3 Stavebné Elementy sekvenčných Diagramov

V nasledujúcej sekcii je popísaná syntax a sémantika sekvenčných diagramov.

4.3.1 objekt

4.3.2 čiara života

4.3.3 aktivácia

4.4 Distribuované systémy

Distribuované systémy majú veľa rozdielnych aspektov, ktoré sa ťažko zachytávajú v jednej definícii. Je omnoho jednoduchšie hovoriť o distribuovaných systémoch špecifikovaním charakteristík, symptómami, či médiami distribúcie. [] V tejto práci budeme mať pod pojmom distribuovaný systém uvažovať systém distribuovaný na počítačovej sieti.

Distribúcia prichádza ruka v ruke s vednými disciplínami ako tolerancia chýb, real-time systémy, bezpečnosť a systémový manažment.

4.4.1 Vymedzenie pojmu distribuovaný systém

Pred definovaním distribuovaného systému, je vhodné vyjasniť rozdiel s často zameňovaným pojmom počítačových sietí.

“Počítačová sieť nie je distribuovaný systém.”

Počítačová sieť je infraštruktúra slúžiaca niekoľkým počítačom pripojeným k sieti cez komunikačné prepojenie realizované rôznymi médiami a topológiami, a používajú zavedný komunikačný protokol. Zatiaľ čo **Distribuovaný systém** je systém pozostávajúci z niekoľkých počítačov, ktoré komunikujú cez počítačovú sieť, hostujú procesy, ktoré využívajú distribučné protokoly, ktoré zabezpečujú koherentné vykonanie distribuovaných aktivít.

Príklad 4.4.1. Vezmime si taký Internet, je to rozsiahla počítačová sieť, vlastne najpodstatnejšia sieť dnes. Používa TCP/IP ako komunikačný protokol. Napriek tomu, že tradične poskytuje zopár aplikačných služieb ako e-mail a telnet, nie je to distribuovaný systém.

To samozrejme nebráni distribuovaným systémom byť postavených na internete alebo používaní internetových technológií, ako distribuované súborové systémy a databázové systémy. Jeden z najpodstatnejších rozdielov je, že v prípade distribuovaných systémov procesy zdieľajú spoločný stav a spolupracujú na dosiahnutí spoločného cieľu. Na rozdiel od procesov v tomto príklade, ktoré nemusia spolupracovať, len si napríklad vymieňať správy (ako e-mail) bez spoločného cieľu.

4.4.2 Porovnanie s Centralizovanými systémami

V Tabulke 4.1 sú zaznamenané vlastnosti v porovnaní s centrálnym systémom ako protipólom k distribuovanému systému. Poznanie rozdielov, výhod a nevýhod oboch systémov je kľúčové pri návrhu systému. Na základe týchto informácií sa možno ľahšie rozhodnúť, ktorú variantu zvoliť.

| Centralizované systémy | Distribuované systémy |
|------------------------|-----------------------|
| Dostupnosť | Geografický rámec |
| Homogenita | Heterogenita |
| Spravovateľnosť | Modularita |
| | Škálovateľnosť |
| Konzistencia | Zdieľanie |
| | Pozvoľná degradácia |
| Bezpečnosť | Bezpečnosť |
| | Finančný faktor |

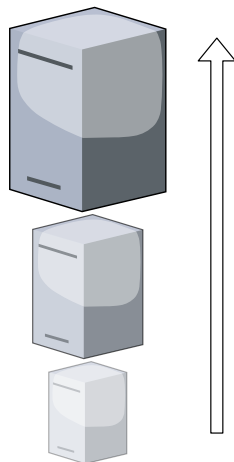
Tabuľka 4.1: Porovnanie centralizovaných a distribuovaných systémov

Centralizované systémy prirodzene prichádzajú s ľahkou **dostupnosťou** zdrojov a informácii systému, keďže sú lokálne dostupné. Na druhú stranu Distribuované systémy majú potencionálne **široký geografický rámec**, preto prístup k zdrojom je niekedy možný len cez vzdialené procedurálne volania.

Homogenita technológií a procedúr je charakteristická pre centralizované systémy, čím sa myslí jeden operačný systém pre celý systém, ťažké odklonenie sa od používaných technológií systému (programovací jazyk, aplikačný rámec). Kdežto u distribuovaných systémov je podporovaná **heterogenita**, ktorá dovoľuje mať pre každú komponentu odlišné prostredie. Homogenita zjednodušuje správu centrálnych systémov. Heterogenita činí distribuovaný systém inkrementálne rozšíriteľný, ikeď centralizované systémy môžu s dodržaním homogenity dosiahnuť rovnaké rozmery. Skutočná výhoda je až pri **škálovateľnosti**, kedy centralizované systémy môžu škálovať len **vertikálne**, 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 distribuovanom systéme máme možnosť škálovať **horizontálne**, obsluhovať dosiahnutie spoločného cieľu na viacerých inštanciách zároveň. Rozdiel medzi vertikálnym a horizontálnym škálovaním je graficky znázornený na obrázku 4.3.

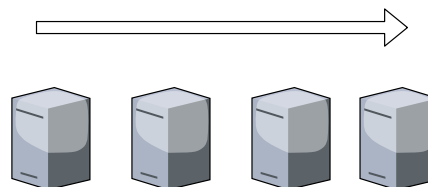
Vertikálne škálovanie

Zvyšovanie výkonu CPU, RAM etc.



Horizontálne škálovanie

Nárast počtu inštancií



Obr. 4.3: Vertikálne a horizontálne škálovanie

Konzistenciu ľahšie dosiahneme u centralizovaných systémov, u distribuovaných je obtiažnejšie zachytiť globálny stav naprieč širokým globálnym rámcom všetkých komponent. **Pozvoľná degradácia** je vlastnosť systému, ktorý beží kontinuálne spôsobom opatrujúcim možnosť zlyhania komponenty spôsobom, ktorý predíde zlyhaniu celého systému. Tu možno pozorovať silu distribučného systému, kedy pri zlyhaní menšej časti je systém stále dostupný vďaka vysporiadavaniu sa s chybami. Navyše je nepravdepodobné zlyhanie všetkých komponent v rovnaký čas kvôli geografickej separácii jednotlivých komponent.

Bezpečnosť sa dosahuje ľahšie u izolovaného systému s fyzickým prístupom. To nie je možné u distribuovaného systému, avšak vysoká miera bezpečnosti sa dá zaistiť zamieraním sa na redukovanie negatívneho efektu vniknutia do systému, než redukovaním hrozieb vzniku neoprávneného vniknutia.

Shrnutím vidíme, že výhody značne prevyšujú ak sa správne rozhodneme, kedy je potreba systém distribuovať.

4.4.3 Kedy distribuovať

Keď nepotrebujeme distribuovaný systém, tak zásadne nedistribujeme. Zbytočne by sme si tým skomplikovali život. Odpoveď pozostáva z troch esenciálnych príčin prečo distribuovať

1. Keď má riešený problém decentralizovanú podstatu

Príklad 4.4.2. Zriadujeme systém používajúci konkurenčné procesy na zdrojoch vzdialených pobočiek.

2. Keď techniky distribúcie sú vhodnou súčasťou riešenia

Príklad 4.4.3. Systém banky, ktorá potrebuje zálohovať a synchronizovať dáta v dvoch geograficky odľahlých miestach

3. Keď problém predpokladá časté zmeny a evolúciu funkcionality, či presunu geografickej polohy.

Príklad 4.4.4. Systém prepožičiavania výpočetných zdrojov medzi vzdialenými užívateľmi.

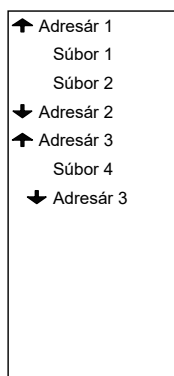
4.5 Vývojové prostredie

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 rýchleho detekovania problémov. Prostredie vedie programátora cez proces editovania, kompilácii či interpretovania kódu a odľadovania(debugging).

4.5.1 Projektový pohľad

Predtým než sa pustíme do editovania kódu, musí vývojové prostredie naviazať spojenie s operačným systémom a jeho súborovým systémom. Pri otvorení projektu sken koreňového adresára nahrá do vývojového prostredia kópie súborov a zobrazí ich graficky v projektovom pohľade. Väčšinou je na grafickom užívateľskom rozhraní zobrazovaný pomocou hierarchického stromu, kde listy sú súbory projektu a uzly sú adresáre.



Obr. 4.4: Ukážka projektového pohľadu s využitím stromovej štruktúry

4.5.2 Editor zdrojového kódu

Neoddeliteľnou súčasťou každého vývojového prostredia je editor zdrojového kódu, ktorý urýchľuje tvorenie validného kódu v danom programovacom jazyku (väčšina vývojových prostredí má len jeden) za pomoci funkcionalít ako:

1. našepkávanie kódu
2. zvýrazňovanie kľúčových slov
3. vyhľadávanie a nahradzovanie v kóde

Existuje ich omnoho viac, záleží na konkrétnej implementácii a programovacieho jazyka.

4.5.3 Preklad

Preklad vo vývojovom prostredí neprebieha na príkazovej riadke, ale odoslanie zdrojových kódov prekladaču alebo interpretu v prípade interpretovaných jazykov je schované v rozhraní za prívetivejšiu variantu tlačítka alebo klávesovej skratky.

4.5.4 Ladenie

Detekovanie chyby v kóde sa urýchly, ak nám vývojové prostredie umožní kód krokovať, zastaviť a v ktorom koľvek bode sledovať stav premenných.

Kapitola 5

Návrh Implementácie

Základná myšlienka samotného prevádzania objektovo orientovaných petriho sietí je využiť diskrétnu simuláciu tejto siete, ktorej kroky nám vytvoria časové kontinuum inak chýbajúce v objektovo orientovaných petriho sietiach.

5.1 Architektúra

Generátor sekvenčných diagramov[11] je priamo závislý na dvoch komponentách, validátorom kódu jazyka PNTalk a simulátoru objektovo orientovaných Petriho sietí. Je dôležité zvážiť napojenie týchto komponent ku generátoru. Vzhľadom k rozličným vlastnostiam jednotlivých implementácií bola motivácia navrhnúť distribuovaný systém s externými komponentami. V generátore uviesť cestu k spustiteľnému binárnemu kódu, ktorého výstup odpovedá definovanému rozhraniu. Daný scenár je uplatniteľný ak pre generátor chceme vyvíjať aj vlastný simulátor, či validátor kódu. V opačnom prípade je vhodnejšie spúšťať externé mikro služby ako webové aplikácie s znovu s vyhradeným komunikačným rozhraním. Pri tejto variante sa namiesto cesty k binárnemu kódu udá generátoru len url adresa webovej aplikácie. Tým sa odstráni nechcená závislosť na externej komponente, ktorej pamäťová náročnosť môže presiahnuť pamäťovú náročnosť samotného generátora.

Samotné dáta, prúdiace medzi komponentami, či už vo variante lokálne preloženej binárky, alebo webovej aplikácie musia dodržiavať jednotné rozhranie a musia byť serializované zo zrejmých príčin. Pri výbere serializačného formátu je nutno zvážiť viaceré faktory ako podpora v rozličných programovacích jazykoch, ľudsky čiteľnejšie textovo založené formáty alebo binárne uložené dáta, ktoré síce postrádajú ľudskú čiteľnosť no vyžadujú menej pamäte a aj ich zápis a čítanie je časovo menej náročné. Binárne serializačné formáty by zlepšili responsivitu komponent a dáta posiadané v ľudsky čiteľnom formáte by mali nespornú výhodu v odlaďovaní programu. Schodnou variantou sa preto javí podpora viacerých formátov prírnaných generátorom od ostatným komponent. Nevýhodou je vznik réžie okolo dohadovania si serializačného formátu medzi komponentami.

5.2 Transformácia modelu OOPN na Sekvenčný diagram

V tejto Kapitole budú prednesené hlavné myšlienky ako vytvoriť základné stavebné jednotky sekvenčného diagramu. Popisujú odkiaľ čerpať potrebné informácie zo simulácie, ako si poradiť z neúplnými informáciami a ako sa vysporiadať z 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 ťažšie z možností formalizmu OOPN a zároveň z vyjadrovacích schopností jazyka PNTalk na vytvorenie datovej štruktúry pre sekvenčný diagram.

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

Obr. 5.1: Štruktúra uchovávajúca dáta zo simulácie OOPN modelu

Štruktúra výsledku simulácie obsahuje 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).

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

Štruktúra krokov simulácie uchováva údaje o správach poslaných v tomto kroku, prechodoch, ktoré začali a skončili v tomto kroku. Prechody totiž môžu začať a skončiť v iných krokoch simulácie. U správ neuvažujeme žiadne spozdenie komunikácie, takže nepotrebujeme rozdeľovať správy na odoslané od odosielateľa a "doručené príjemcovi". Obe tieto veci nastanú v jednom okamihu.

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

Štruktúra inicializácií inštancií tried obsahuje meno inštancie, referenciu na svoju triedu, čas vzniku a počiatočný stav miest. 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 funguje ako východzí bod pri vypočítavaní aktuálneho stavu aplikovaním zmien spôsobených prechodmi od vzniku objektu k aktuálnemu času.

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

Štruktúra správy obsahuje jej unikátny identifikátor, obsah správy, informácie o odosielateľovi a príjemcovi správy, názov prechodu, ktorý vyvolal správu a záznam, či sa jedná o odpoveď na nejakú už existujúcu správu. 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. Ak sa jedná o odpoveď nie je nutný žiaden záznam okrem identifikátoru správy, na ktorú sa odpovedá a odpoveď samotná.

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

Teda odpoveď je akceptovaná v tomto minimálnom tvare:

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

Štruktúra začiatku prechodu uchováva svoj 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;
}
```

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;
}
```

Štruktúra hodnoty miesta obsahuje typ (hodnota alebo referencia na objekt) a hodnota. Typ referencia znamená

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

Objekt

Objekt alebo entita je kľúčová časť v scenári sekvenčného diagramu. Je to obdĺžnik so štítkom mena vo vnútri v ktorom započne čiara života (lifeline) až do deštrukcie objektu, alebo konca simulácie.

Vytvorenie objektu

Objekt môže vzniknúť za behu simulácie, alebo byť k dispozícii. Na vytvorenie objektu v sekvenčnom diagrame potrebujeme zo simulácie archivovať minimálne 3 veci:

1. čas simulácie v ktorom sa inštancia vytvorí
2. inštanciu, ktorá inicializovala vytvorenie
3. triedu vytvárajúcej inštancie

Vďaka týmto údajom sa dá vytvoriť správa v sekvenčnom diagrame, ktorá odsadí objekt vertikálne od počiatku do vzdialenosti podľa času vytvorenia.

Poznámka: dodatočne sa bude archivovať aj miesto, kam sa objekt uloží pre počítanie referencií. To sa uplatní pri deštrukcii objektu.

Deštrukcia objektu

Pre deštrukciu objektu musí zaniknúť posledná referencia na objekt. Kvôli tomu je potreba počítadlo referencií, ktoré však nebude výkonnostne náročné ako plnohodnotný garbage collector. Vďaka selektívnemu výberu prechodov, ktoré manipulujú s miestami, kde sú uložené objekty môžeme zredukovať počet opakovaní algoritmu len na vybrané prechody.

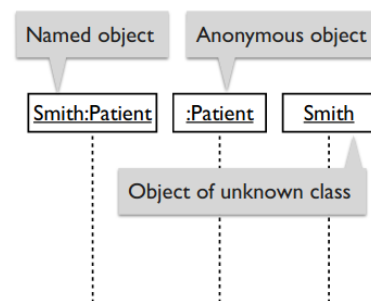
Prechod môže spôsobiť tri veci pri manipulácii s referenciou:

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 inštalácie s prechodom, ktorý poslednú referenciu vymazal.

Konvencia mena

Objekty v sekvenčných diagramoch sa pomenúvajú pomocou nasledujúcej konvencie "meno inštalácie:meno triedy"vďaka čomu môžu vzniknúť tri typy objektov:

1. Pomenovaný objekt
2. Anonymný objekt
3. objekt neznámej triedy



syntax jazyka PNTalk vytvára novú inštaláciu nasledovne:

```
var := classname new.
```

kde var je dočasná premenná alebo miesto a classname je meno triedy. Problém je zjavný a to, že chýba akákoľvek informácia o mene inštalácie. To nám hneď vylúči tretiu možnosť, pretože meno triedy je vždy známe. Varianty sú teda dve a to buď poskladať meno inštalácie pomocou známych veličín ako názov miesta, meno triedy, krok simulácie či vygenerovať identifikačné číslo. Druhá varianta je uspokojiť sa s vedomím, že budú vznikať len Anonymné objekty bez názvu inštalácie.

Čiara života

Čiara života alebo inak lifeline je vertikálna čiara reprezentujúca život objektu začína pre každý objekt v dobe vytvorenia a končí deštrukciou objektu, alebo na konci simulácie. Jej vytvorenie je triviálne pokiaľ dokážeme určiť čas vytvorenia a zániku objektu. TODO ref

Je prekrytá bielym obdĺžnikom po dobu, kedy sa metóda objektu nachádza na zásobníku.

Doba aktivácie

Doba simulácie po ktorú sa prevádza metóda objektu je viazaná s volaním metód cudzích objektov a preto je nutno archivovať prechody a inštalácie, ktoré ich vlastnia. Na tieto prechody potom namapovať prevádzané inštrukcie v chronologickom poradí.

Správa

Správa vyžaduje poznať odosielateľa, príjemcu a hlavne o aký typ správy sa jedná. Poznáme tri typy:

Synchronná Asynchronná Odpoveď

zo syntaxe volania metódy pre cudzí objekt evidentne dokážeme zo simulácie odvodiť odosielateľa aj príjemcu.

var methodname: params

kde var je premenná s premennou nesúcou informáciu o mieste s objektom príjemcu. methodname je názov volanej metódy triedy príjemcu. params sú parametre metódy.

odosielateľ je inštancia, ktorá túto akciu zapríčinila svojim prechodom.

Ak metóda vracia hodnotu v simulácii je archivovaná ako odpoveď na správu nesúca údaje o správe na ktorú odpovedá a celú odpoveď.

TODO: Sync vs Async

Cyklus

K odstráneniu redundantných scenárov značne pomôže zapúzdrenie cyklov, vždy hľadáme v prechodoch najmenší možný ohraničený celok, ktorý sa za sebou sekvenčne niekoľko krát opakuje.

Referovanie a prepájanie diagramov

Podobne ako pri cykle hľadáme rovnaké, či podobné ohraničené sekvencie prechodov opakujúce sa v simulácii.

5.3 Out-source simulácie

Pre simuláciu bude generátor využívať jeden zo simulátorov objektovo orientovaných petriho sietí z variant bližšie špecifikovaných v kapitole :TODO: . Ako najschodnejšia varianta je zvolený pre túto prácu :TODO: . Aby sme si neuzavreli definitívne dvere k iným implementáciám simulátoru jazyka PNTalk je príhodné zamyslieť sa nad napojením generátoru na simulátor.

1. Varianta pridania kódovej časti do generátoru zjavne možná nie je z dôvodu rôznych implementačných jazykov. Voľba kotlinu ako implementačného jazyka je odôvodnená v sekcii :TODO: .
2. Ponúka sa možnosť vytvoriť dynamickú knižnicu a volať funkcie simulátora z nej. Určite je táto možnosť schodné riešenie, ikeď tu doplácame na neschopnosť preložiť simulátor na všetkých platformách.

Poznámka 5.3.1. Linuxová dynamická knižnica *.so nie je ekvivalentná s windowsovou *.dll

3. Veľmi podobné riešenie je spustiť binárny kód simulátoru s argumentami cestou ku kódu v jazyku PNTalk a zachytením výstupu cout. Oproti predchádzajúcej variante, vyžaduje omnoho menej úprav.

4. Posledná a taktiež zvolená varianta je pojať generátor ako distribuovaný systém :TODO: , ktorý bude k simulácii využívať komponentu simulátora s ktorou bude komunikovať vopred známym protokolom. To, že si komponenta simulátora zavolá ďalšiu komponentu prekladača do medzikódu nebude zo strany generátoru viditeľné. Dôležitý je len pevne daný protokol medzi generátorom a simulátorom, pretože nám to dáva možnosť implementácie simulátora jednoducho meniť. Stačí aby dodržovali stanovené rozhranie.

Distribuovaný systém môže nadobnúť odlišné fyzické formy, či už ide o skupinu osobných počítačov, prepojených lokálnou sieťou, skupinu pracovných staníc zdieľajúcich nielen súborové a databázové systémy, ale navyše aj zdieľaním výpočetnej sily procesora.[]

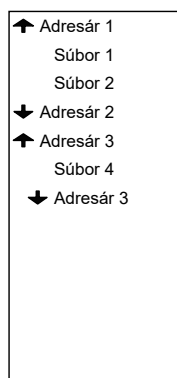
Distribuovaný systém obsahujúci sadu procesov, ktoré medzi sebou udržujú formu komunikáciu. Okrem konkurenčného behu procesov, niektoré z procesov distribuovaného systému môžu prestať pracovať, pre príklad spadnúť alebo stratiť konektivitu, zatiaľ čo ostatné zostanú bežať a pokračovať v operácii. Toto je podstata čiastočných zlyhaní charakteristických pre distribuované systémy.

5.4 Uživatelské rozhranie

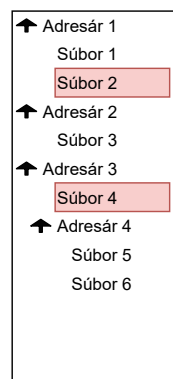
Pri návrhu grafického užívateľského rozhrania je dobré začať položením si otázky "Čo chceme zobrazovať?". Menej je však niekedy viac, pri príliš zložitom rozložení totiž strácame prehľadnosť.

1. Chceme zobraziť momentálne otvorený projekt

Reprezentáciou by mohol byť hierarchický strom, ktorý by mal v listoch uložené mená súborov a v uzloch mená adresárov. Listy, teda súbory, by mali vizuálnym efektom upozorniť ak je v súbore neuložená zmena.



Obr. 5.2: Projektový pohľad so schovaným uzlom "Adresár 2" a "Adresár 4"



Obr. 5.3: Indikácia neuložených zmien v súboroch viditeľná na rozhraní.

s

2. Chceme 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. Chceme zobrazovať vygenerovaný diagram

Potrebujeme na to minimálne rovnako veľa miesta ako na editor zdrojového kódu. Pozadie by malo byť kontrastné vôči 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 zobrazovať zmenu miest OOPN, ktoré prechody vyvolali. Označenie, ktoréhokoľvek miesta na čiare života by malo ukázať aktuálne hodnoty v miestach danej inštancie.

4. Chceme zobrazovať posledných x riadkov logov

Je fajn mať 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. Chceme 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.

5.4.1 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, užívatelia sú na ne navyknutí. Preto nie je v zájme práce sa od týchto štandardov odkláňať. 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, 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. Ničmenej 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. ??



Obr. 5.4: Návrh rozloženia grafického užívateľského rozhrania

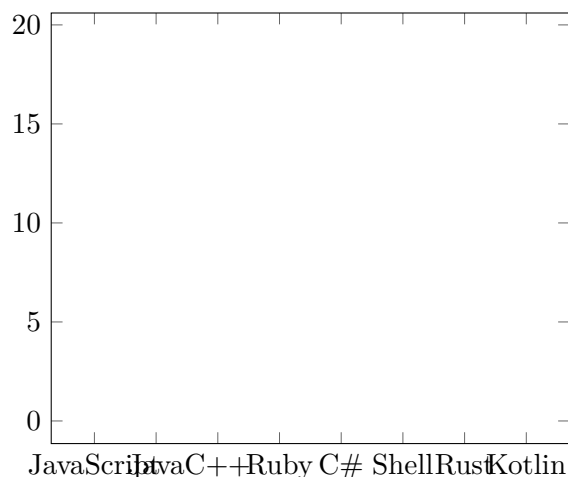
Kapitola 6

Implementácia nástroja

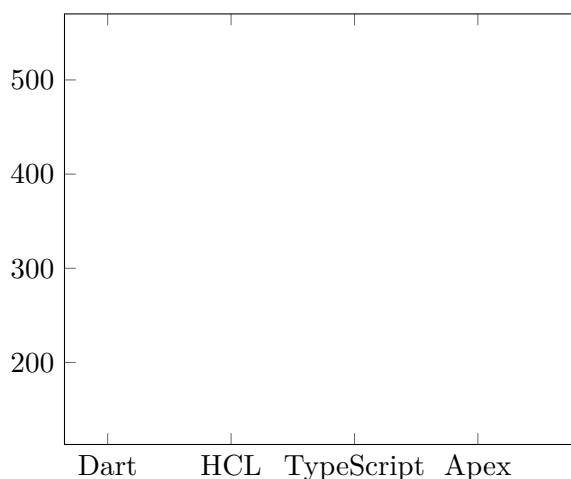
Implementoval som prácu v jazyku kotlin a prácu na ktorú som nadväzoval som upravil v jazyku C++, na ich prepojenie som využil verejne dostupné knižnice gRPC pre kotlin /cite<https://github.com/grpc/grpc-kotlin> a pre C++ []<https://github.com/grpc/grpc>. Prispôbil som implementáciu simulátoru a prekladača medzikódu pre použitie s technológiou Docker a vytvoril spustiteľné obrazy týchto dvoch komponent skrz kontajnery dockeru dostupné na :TODO:. Na grafické užívateľské rozhranie bol použitý aplikačný rámec TornadoFX nad knižnicou JavaFX.

6.1 Výber implementačného jazyka

Práca nadväzuje na simulátor napísaný v jazyku C++, čo robilo z jazyka C++ prirodzenú voľbu pre hladké naviazanie a kompatibilitu. Ničmenej ďalším požiadavkom práce, akožto zadáním z oblasti blízkej jazyku PNTalk, bola motivácia držať implementačné jazyky týchto nástrojov blízko. :TODO: príklady Keďže väčšina prác beží na Jave a je do budúcnosti zmýšľaná ich kooperácia, získali sme 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. :TODO: 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 hostuje 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 súčasnosti písania tejto práce, je celkovo na 15. priečke v obľúbenosti [2].



Obr. 6.1: Pridaný kód za rok 2020, second quarter - stats report z github.com [3]



Obr. 6.2: Najrýchlejšie rastúce jazyky - Octoverse report 2019 z github.com [3]

6.2 Implementácie distribuovaného systému

Ako bolo zmienené v návrhu, generátor sa nebude viazať na simulátor ani prekladač do medzikódu. Namiesto toho budú prepojené v distribuovanom heterogénnom systéme.

6.2.1 virtualizácia

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 bolestivý.

Nástroj Compose

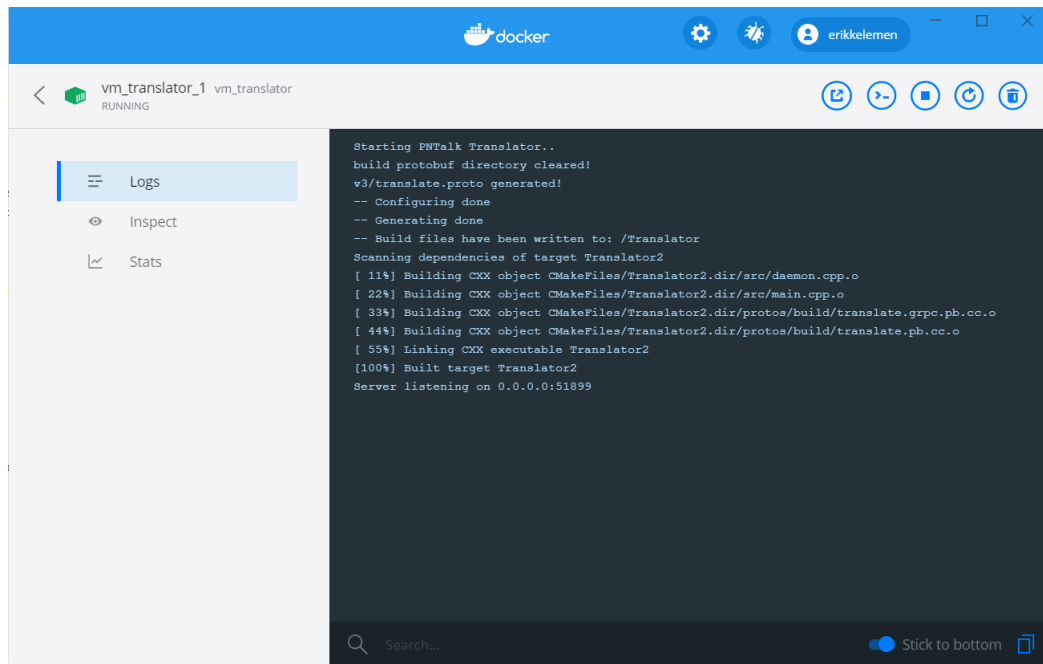
Docker Compose je nástroj pre definovanie a beh multi-kontajnerových Docker aplikácií. S nástrojom sa používa súbor na konfiguráciu služieb aplikácie. Potom jediným príkazom dokážeme vytvoriť a spustiť všetky služby z konfigurácie. [1]

`docker-compose up`

Konfiguráciu bolo nutné vytvoriť pre službu simulátora a pre prekladač do medzikódu podľa :TODO: . Pre službu simulátora sa definoval port 51898 a pre prekladač do medzikódu port 51899. V konfigurácii na Obr. 6.3 vidíme otvorené práve tieto dva porty. Služby využívajú Docker obrazy so zdrojovými kódmi a prerekvizitami k prekladu. Sledovanie spusteného kontajneru je vyobrazené na Obr. 6.4

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

Obr. 6.3: Použitá konfigurácia v docker-compose.yml



Obr. 6.4: Náhľad do kontajneru v aplikácii Docker Desktop.

Kvôli prenositeľnosti komponent sa zvolil prístup kontajnerov 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]

Nástroj Compose

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

6.2.2 Vzdialené volanie procedúr

gRPC

gRPC je technológia navrhnutá pre vzdialenú medziprocesovú komunikáciu, tak aby prekonala nedostatky konvenčných technológií vzdialených volaní procedúr. Kde ostatné technológie používajú textový formát na prenos dát ako JSON alebo XML, gRPC využíva binárneho formátu. Protokol využíva HTTP/2 [15], čo ho robí ešte rýchlejší vo vzdialenej medziprocesovej komunikácii.

```
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;
    }

```

6.3 Užívateľské rozhranie

Implementácia vychádza z dobre pripraveného návrhu v sekcii :TODO: , ktorá bola realizovaná za pomoci kotlinovského aplikačného rámcu TornadoFX nad softvérovou platformou JavaFX.

6.3.1 Bohaté internetové aplikácie

Bohaté internetové aplikácie, (Rich Internet Application), niekedy tiež v literatúre pod názvom moderné internetové aplikácie, /citepsatweb sú webové aplikácie poskytujúce responzivitu a sú "bohaté" v zmysle funkcionality a možností desktopových aplikácií. Ranné internetové aplikácie poskytovali len HTML grafické užívateľské rozhranie, ktoré dokázalo poslúžiť jednoduchým cieľom, no nemalo ani vzhľad ani responsivitu RIA hlavne kvôli pomalému internetovému spojeniu. RIA je teda výsledkom dnešnej doby poskytujúcej vyššiu responzivitu a pokročilejšie grafické užívateľské rozhrania. [10]

Na tvorbu bohatých internetových aplikácií, môžeme využiť celú radu aplikačných rámcov, tu vudú spomenuté len varianty zvažované pre túto prácu.

Ajax

Výraz Ajax (Asynchronous JavaScript and XML) vznikol vo februári roku 2005 od Jamesa Garretta. Ajax aplikácie dovoľujú čiastočné aktualizácie stránky. To znamená aktualizovať individuálne časti webu bez nutnosti obnoviť celú stránku. To vytvára viac responzívne GUI ako predtým, dovoľujúci užívateľom ďalej pracovať so stránkov, zatiaľ čo server zpracováva požiadavky.

Technológie stojace za Ajaxom - XHTML, CSS, JavaScript, the DOM, XML a XMLHttpRequest — nie sú nové. Vlastne už v 90 rokoch existujú príklady asynchrónnych aktualizácií stránky, ktoré rozultovali v JavaScript [10]. Ničmenej popularita Ajaxu dramaticky narástla až po jeho pomenovaní v roku 2005.

Ajax sa ukázal ako nevhodný pre prácu, kvôli nízkej abstrakcii.

Flex

JavaFX

JavaFX je softvérová technológia na vytváranie bohatých internetových aplikácií (Rich Internet Application) s obsahom cez širokú škálu platforiem a zariadení. Táto technológia bola zvolená k implementácii užívateľského rozhrania. Jazyk sa pôvodne nazýval F3 (Form

Follows Function) a jeho priekopníkom a tvorcom bol Chris Oliver, ktorý v tej dobe pracoval pre firmu SeeBeyond [9]. Meno bolo zmenené v roku 2007 na JavaFX. [6]

Najprv vznikol JavaFX Script ako interpretovaný jazyk, ktorý bol považovaný za prototyp :TODO: kompilovaného JavaFX Script jazyka, ktorý ho mal nahradiť. Tento deklaratívny skriptovací jazyk, tiež staticky typovaný ako Java využíval knižnice Javy a dokonca mohol volať jej metódy, či inšancovať jej triedy. [27]

V roku 2011 verzia JavaFX 2.0 prestala využívať JavaFX Script a vymenila ho za Javu a JavaFX 2.0 API. [9] Jeho hlavnou nevýhodou bola prerekvizita Javy, ktorá ho ako jediná vedela preložiť. Týmto krokom sa technológia JavaFX dostala k jazykom bežiacim na JVM ako Groovy, JRuby či Kotlin.

TornadoFX

Jedna z hlavných výhod pre kotlin spomenutá v sekcii :TODO: bola jeho 100 % interoperabilita s existujúcimi Java knižnicami, vrátane JavaFX. I keď kotlin môže využívať JavaFX priamo rovnakým spôsobom ako v Jave, niektorí verili, že Kotlin má jazykové predispozície aby mohol usmerniť a zjednodušiť vývoj JavaFX aplikácií. Eugen Kiss prototypoval vrstvu nad JavaFX ako KotlinFX, jeho nápad neskôr priviedol ku zdarnejšiemu koncu Edwin Syse v januári 2016[4], keď vydal TornadoFX.

TornadoFX usiluje o to, aby znateľným spôsobom minimalizovalo rozsah potrebného kódu na napísanie JavaFX aplikácie. Zahŕňa typovú kontrolu nad komponentmi pri skladaní komponent užívateľského rozhrania a prináša rozšírenia kotlinu ako delegované vlastnosti a injekcia závislostí. TornadoFX je ukážkový príklad ako Kotlin dokáže zúsporniť kód napísaný v Jave.

6.3.2 Editor Zdrojového kódu

V sekcii 4.5.2 boli vymenované niektoré funkcionality, ktoré nesmú chýbať v moderných editoroch zdrojového kódu. Z nich bolo implementované zvýrazňovanie kľúčových slov jazyka PNTalk a zvýrazňovanie všetkých validne definovaných názvov tried, prechodov, miest, synchrónnych portov a metód.

Zvýrazňovanie zaisťuje asynchrónna funkcia computeHighlighting volaná nad textom z editoru. Je postavená na vyhľadávaní regulárnych výrazov. Globálne v celom rámci sa zvýrazňujú kľúčové slová jazyka PNTalk a mená tried. V rámci danej triedy sa k nim pridá vyhľadávanie názvov prechodov, miest, synchrónnych portov definovaných však len v rozsahu danej triedy.

6.3.3 Projektový pohľad

6.3.4 Diagram ako výstup aj interaktívny ladiaci nástroj

Kapitola 7

Testovanie

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. Po overení dát sa vygenerované výstupy odložili a rovnakú pred rovnakú úlohu sa postavili ľudské subjekty.. Subjekty, ktoré ručne tvorili sekvenčné diagramy, taktiež nevyužívali nič iné než kód PNTalku. Jednalo sa o praktikantov softvérového inžinierstva minimálne s praxou jeden rok. Výsledok vytvorený človekom sa porovnal s výstupmi generátora. Na sekvenčný diagram sa nahliadalo zo štyroch kategórií:

Kapitola 8

Záver

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

V práci by som chcel pokračovať implementovaním filtrovania správ a hľadaním v dátach simulácie vzory cyklov či podmienok. 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 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, ale editor stále postráda niektoré vlastnosti inteligentnejších vývojových prostredí. Editor kódu by mohol skúšať dopĺňať kód podľa prvých napísaných znakov a pozícií v kóde. Implementované by to mohlo byť rozhodovacím stromom.

Práca demonštruje automatický prevod objektovo orientovaných Petriho sietí na sekvenčné diagramy, generovanie však pokrýva len podmnožinu sekvenčných diagramov. Objekty Actors vystupujúce v konvenčne vytvorených sekvenčných diagramoch sú v práci zanedbané (keďže informáciu na rozlíšenie obyčajných objektov od Actors nedokázali poskytnúť definície v kóde, ani následná simulácia) a Actors preto vystupujú len ako všeobecné objekty. Ďalší zrejmý nedostatok vyplýva z naviazania na neúplnú implementáciu simulátora, ktorá neumožňuje simuláciu všetkých validných konštrukcií jazyka PNTalk, len ich podmnožinu. Istou kompenzáciou jest architektúra navrhnutá ako distribovaný systém, ktorá robí tento problém ľahko riešiteľným v budúcnosti po implementovaní vhodnejšej varianty simulátora. Na Záver je vhodné položiť si otázku či sme boli úspešní. To nám zodpovie sada validačných testov. Jedná sa o netriviálne Petriho siete zadefinované v jazyku PNTalk, ktorých vygenerované výstupy boli porovnané s tými ručne vytvorenými. Okrem validity vzišla motivácia zaznamenať výsledky aj časovej náročnosti. Časová náročnosť sa merala pre samotný proces generácie sekvenčných diagramov ako aj celkovo beh v spolupráci externých komponent. Plán bol vytýčiť hranice, pre ktoré by bolo reálne simulovať a vykresľovať výsledok generácie ihneď pri zmene vstupného kódu. Kvôli neuspokojivým výsledkom v tomto teste (:TODO: graf) sa z pokusu o implementácie funkcie "hot-reloadüpustilo.

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/github/#/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 Cost of 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] ANDERSON, G. *Essential JavaFX*. Upper Saddle River, NJ: Prentice Hall, 2009. ISBN 978-0137042791.
- [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] 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/>.
- [9] DEA, C. *JavaFX 2.0 : introduction by example*. New York: Apress, 2011. ISBN 978-1-4302-4257-4.
- [10] DEITEL, P. *Ajax, rich Internet applications, and web development for programmers*. Upper Saddle River, N.J: Prentice Hall, 2008. ISBN 0131587382.
- [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] JANOŠEK, I. V. *Modelování objektů Petriho sítěmi*. 1998. Dizertačná práce. 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] MARSAN, M. *Modelling with generalized stochastic Petri nets*. Chichester New York: Wiley, 1995. ISBN 9780471930594.
- [19] 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.
- [20] MILICEV, D. *Model-driven development with executable UML*. Indianapolis, IN: Wrox/Wiley, 2009. ISBN 9780470481639.
- [21] 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>.
- [22] 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.
- [23] RAMCHANDANI, C. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets*. 1974. Dizertační práce. MIT, Cambridge, MA.
- [24] ROZENBERG, G. *Advances in Petri nets, 1990*. Berlin New York: Springer-Verlag, 1991. ISBN 3540538631.
- [25] 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.
- [26] WASSON, C. *System analysis, design, and development : concepts, principles, and practices*. Hoboken, N.J: Wiley-Interscience, 2006. ISBN 978-0471393337.
- [27] WEAVER, J. *JavaFX Script : dynamic Java scripting for rich Internet/client-side applications*. Berkeley Calif: Apress, 2007. ISBN 9781590599457.
- [28] WHITTEN, J. *Systems analysis and design methods*. Boston: McGraw-Hill/Irwin, 2007. ISBN 978-0073052335.

Príloha A

Obsah přiloženého paměťového média

Príloha B

Manuál