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 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2019/2020

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 2019
Datum odevzdání: 28. května 2020
Datum schválení: 31. října 2019

Abstrakt

Zatiaľ čo petriho siete nesporne dominujú v monitorovaní zmeny stavov v modelovanom systéme, sekvenčné diagramy dokážu lepšie prezentovať externý pohľad na systém v časovom slede posielaných správ medzi komunikujúcimi objektami. I keď by sa mohlo zdať, že majú spolu pramálo spoločného, v tejto práci bude predvedený koncept transformácie ako vygenerovať sekvenčný diagram pomocou simulácie modelu OOPN zapísaného v jazyku PNTalk bez dodatočných informácií, ktoré by akokoľvek pomohli zostaviť sekvenčný diagram. Práca sa zaoberá transformáciou dát v zmysle minimálnej straty informácie z modelu OOPN a následnú prezentáciu vyťažených dát v podobe validného sekvenčného diagramu a to nad rámec triviálnych prípadov.

Abstract

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v anglickém jazyce.

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
28. júla 2020

Poďakovanie

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

Obsah

Kapitola 1

Úvod

Jeden z najzákladnejších problémov, ktoré rieši softvérový vývoj je validácia požiadavkov systému. Tieto požiadavky sú zvyčajne definované pomocou diagramu užitia z UML. Bežný postup je navrhnúť model systému podľa týchto požiadaviek za pomoci ostatných diagramov z UML a otestovať ho manuálne implementovaným prototypom. To predstavuje dosť práce jednak s diagramami UML a navyše implementovaný prototyp pravdepodobne stratí veškeré využitie po validácii modelu. Predstavme si však, že vytvoríme model za použitia objektovo orientovaných petriho sietí, ktorý prichádza s možnosťou simulácie modelu. Táto simulácia poskytuje priestor na automatické vytvorenie UML diagramov. 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 nedostatkom, keď sa spustitelná forma UML modelu v priebehu validácie upravuje, je takmer nemožné vrátiť sa so zmenamy k pôvodnému modelu.

Hlavným cieľom práce je vytvoriť plnohodnotný nástroj na validáciu modelu, ktorý vygeneruje sekvenčný diagram z jazyka UML. Jazyk UML definuje viac diagramov interakcií z ktorých by sa dalo vybrať, no narozdiel od diagramu interakcií sa dá vygenerovať zo simulácie a navyše je sekvenčný diagram druhý nanajvýš používaný z diagramov UML. Od nástroja sa očakáva, že by mal analyzovať všetky možné scenáre, rozlíšiť redundantné výskyty častí scenárov a agregovať ich, aby obmedzil zobrazované informácie. Ďalej by mal poskytovať intuitívne rozhranie a zachovať všetky informácie zo simulácie ľahko dohľadateľné.

Najzložitejšiu časť generovania sekvenčného diagramu predstavuje nahradzovanie dát potrebných na zostavenie sekvenčného diagramu, chýbajúcich v reprezentácii pomocou objektovo orientovaných petriho sietí.

V kapitole.. TODO

Kapitola 2

Úvod3

Softwarové inžinierstvo je kľúčové k úspešnému zvládnutiu dnešných rozsiahlych projektov. Rýchlosť vývoja značne ovplyvňuje komunikácia medzi jednotlivými časťami zainteresovaných osôb. V tak turbulentnej dobe, akou je tá dnešná sa zadanie navyše počas vývoja softwaru mení a býva nekompletné či zavádzajúce. Ako si dokážeme udržať prehľad, ktoré komponenty sú potreba, čo je ich prácou a ako by mali spĺňať požiadavky zákazníka? Alebo ako môžme zdieľať návrh systému s kolegami spolupracujúcich na projekte, aby sa zaistila kompatibilita častí systému? [10]. Dnešné systémy sú príliš rozsiahle, plné zákerných detailov, ktoré môžu byť zle interpretované alebo kompletne vynechané. Preto udržanie si prehľadu o systéme môže stroskotať bez patričnej pomoci.

V roku 2019 bola zverejnená správa inštitútu projektového managementu (anglicky Project Managemenet Institute, skratkou PMI), do ktorej prispelo 4455 praktikantov projektového manažmentu z praxe. Z nich najväčšiu časť tvorili odborníci z odvetvia informačných technológií. Report monitoroval obdobie projektov odštartovaných v časovom rámci 12 mesiacov, čiže rok 2018. Ako 5 najčastejšich príčin zlyhania projektov respondenti uvádzajú: Zmenu priorít organizácie(39%), Zmena projektových cieľov(37%), nepresne definované požiadavky(35%), neadekvátna vízia(29%), slabá komunikácia(29%). Ako vidno zo štatistík komunikácia stále predstavuje dosť veľký kameň úrazu.

Pomôcť pri komunikácii môžu modelovacie jazyky, ktoré nám umožnia ujasniť si naprieč celým rámcom projektu ako je systém navrhnutý. Najlepšie by bolo zvoliť modelovací jazyk tak, aby mu rozumela aj menej technicky zdatná zúčastnená strana. 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 systému. Takéto diagramy nevyžadujú žiadne vyššie vzdelanie na ich pochopenie, navyše obraz je často ľahšie uchopiteľný ako písaný text. 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 štandartom pre diagramy používané na modelovanie systémov. Jeho najprirodzenejšie využitie je práve u objektovo orientovaných systémov. V praxi kreslenie diagramov zabere určitý čas, ktorý by sa mohol využiť efektívnejšie. Vytváranie dokumentácie často nie je prioritou a tak diagramy vidíme hlavne v rannej fázy špecifikácie požiadavkov, pri analýze a navrhu, no potom už čím ďalej menej.

Predsavme si však, že by sme dokázali z modelu systému v akomkoľvek stave a bez investície času, či námahy, automaticky vygenerovať graficky reprezentovaný scenár skúmanej aktivity v podobe sekvenčného diagramu. Tomuto grafickému pohľadu na časť systému by rozumeli nielen IT špecialisti, ale aj technicky menej znalí účastníci projektu. Uľahčila by

sa tým komunikácia s uživateľmi, či vlastníkmi 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 sekvenčným diagramom pred našou zmenou a po nej, čo by urýchlilo validáciu zo strany zákazníka. Vývojové tímy, by sa ľahšie zorientovali v komponentách, ktoré vyvýjal iný tím a podobne.

Cieľom tejto práce je práve zostrojiť generátor, ktorý z modelu Patriho sietí vygeneruje sekvenčný diagram.

Sekvenčný diagram patrí do jazyka UML od roku 1997(štandart 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, asynchrónnost, 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 informácii 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 3

Úvod2

Projektový manažment

Reportu sa účastnilo 4455 praktikantov projektového manažmentu. Z nich najväčšiu časť tvorili odborníci z odvetvia informačných technológií. Podľa reportu inštitútu projektového managementu (anglicky Project Managemenet Institute, skratkou PMI) až 36 percent projektov nie je dokončených na čas, a až 43 percent prekročí stanovený rozpočet. Tieto čisla by nemuseli byť také veľké..

Z toho ako 5 najčastejšjch respondenti uvádzajú: Zmenu priorít organizácie(39%), Zmena projektových cieľov(37%), nepresne definované požiadavky(35%), neadekvátna vízia(29%), slabá komunikácia(29%). Predsavme si, že by sme dokázali z modelu systému v akomkoľvek stave a bez investície času, či námahy, automaticky vygenerovať graficky reprezentovaný scenár skúmanej aktivity v podobe sekvenčného diagramu. Tomuto grafickému pohľadu na časť systému by rozumeli nielen naši kolegovia, ale aj technicky menej znalí účastníci projektu. Uľahčila by sa tým komunikácia s uživateľmi, či vlastníkmi projektu.

Ako si dokáže tím udržať prehľad, ktoré komponenty sú potreba, čo je ich prácou a ako by mali spĺňať požiadavky zákazníka? Ba čo viac, ako môžme zdieľať návrh systému s kolegami spolupracujúcich na projekte, aby sa zaistila kompatibilita častí systému? [10]. Dnešné systémy sú príliš rozsiahle, plné zákerných detailov, ktoré môžu byť zle interpretované alebo kompletne vynechané.

Jeden z problémov, ktoré rieši softvérové inžinierstvo v ranných fázach vývoja je špecifikácia požiadavkov. V tak turbulentnej dobe, akou je tá dnešná sa zadania počas vývoja softwaru menia a bývajú nekompletné či zavádzajúce. Navyše každá zainteresované osoby môžu mať na systém vlastné, často nezhodujúce sa nároky. Uživateľia, či vlastníci projektu vám povedia scenáre, ako sa má systém chovať, no asi vám nepovedia aké datové štruktúry, či technológie použiť. Pre technicky menej zdatné subjekty vždy pomôže grafická reprezentácia pohľadu na systém. Asi nejeden z nás sa ocitol v situácii, kedy pri vysvetľovaní chovania systému inej osobe použil nejaký ten diagram.

sa narážame na zadanie, analýza a návrh systému. K jednej zo zaužívaných variant pre modelovanie systému patrí jazyk UML (Unified Modeling Language) ku ktorému od roku 1997(štandart v1.1) patrí sekvenčný diagram ako jeden z diagramov na modelovanie interakcií v systéme. Na druhej strane máme Petriho siete(PT), matematický model, ktorý je schopný vyjadriť kauzalitu udalostí, asynchrónnost, paralelizmus a synchronizáciu. 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 informácii 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.

Cieľom práce je okolo tejto myšlienky postaviť generátor, ktorého vstupom je kód jazyka PNTalk popisujúci OOPN a výstupom sekvenčný diagram. Na záver sa správnosť vygenerovaných diagramov posúdi v porovnaní s diagramami vytvorenými ručne odborníkami z praxe.

Kapitola 4

Tvorba a analýza scenárov v modelovaní systému

4.1 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.

4.1.1 Zúčastnená strana

Všetky fyzické osoby ovplyvňujúce vývoj softvéru môžme pre akýkoľvek systém klasifikovať do 5 skupín. Zobrazené sú na ľavej strane Obr. 4.1. Podstatné je, že každá zo skupín má na systém iný uhol pohľadu.

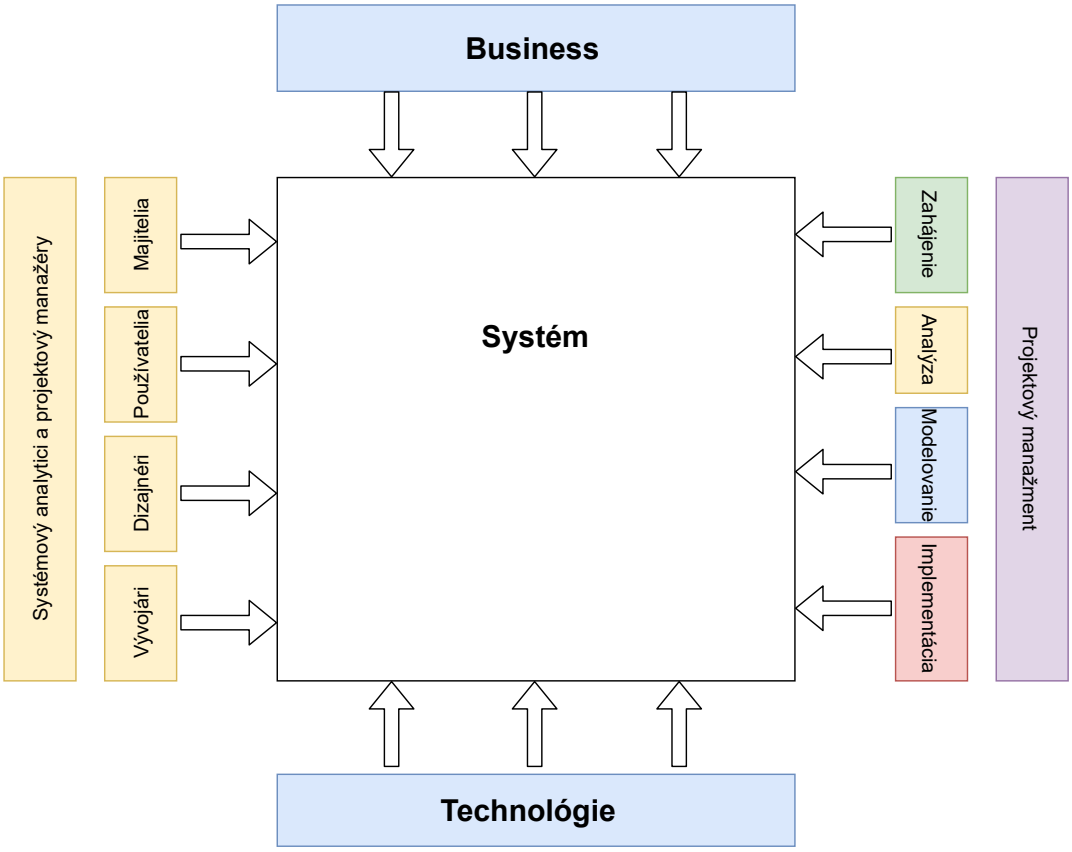
1. **Systémový analytici a projektový manažéri** sú špecialistami na analýzu a modelovanie, poskytujú ostatným skupinám poradenstvo a sú akýmsi mostom pri akomkoľvek komunikačnom šume vznikajúcom napríklad medzi menej technicky zdatnými majiteľmi a projektu a vývojármi.

2. **Vývojári**, ktorý majú za úlohu celý systém zkonštruovať podľa návrhu softvérového dizajnéra riešia hlavne detaily implementácie. V menších firmách sú dizajnéry a vývojáry 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. **Uživateľia** 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 funkcionalita, intuitívnosť používania a o cenu, či profit, narozdiel od majiteľov, nedbajú.

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



Obr. 4.1: Aspekty ovplyvňujúce vývoj systému [15]

4.1.2 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 pozdvihnnúť projekt na vyššiu úroveň.

4.1.3 Procces vývoja

Je zrejmé že väčšina organizácií bude mať vlastný formálne definovaný proces vývoja softvéru alebo sadu krokov, ktoré podľa ktorých by sa mal systém vyvíjať. Akiste sa budú tieto metodológie od seba diametrálne odlišovať pre jednotlivé organizácie. Avšak, všetky metódy riešenia problému môžeme zavšeobecniť na kroky, ktoré sú spoločné:

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 kľadú nároky obchodu či funkcionálna stránka vyžadovaná uživateľmi.

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

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 nutno objektívne zhodnotiť výsledky v zmysle splnenia cieľov. Pri nesplnení sa môžeme vrátiť ku kroku 1 a 2.

Na obrázku 4.1 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. Inizializácia 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 4.1.

4.2 Analýza systému

V sekcii 4.1.3 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. Keďže sa v literatúre nestretneme s presne vytíčenou hranicou, kde končí analýza systému a začína návrh systému, v tejto práci bude analýza pokrývať potreby majiteľov a uživateľov systému. Technické a implementačné detaily nebudeme v tejto práci uvažovať ako súčast analýzy. V predchádzajúcej sekcii bola opísaná zúčastnená strana podieľajúca sa na analýze a cieľe analýzy, no samotná otázka äko analyzovať systém"bola doteraz len nonšalantne opomíjaná. V tejto kapitole budú rozobrané vybrané metodológie a prístupy, ktoré obšírny pojem analýza systému zastrešuje.

4.2.1 Prístupy k analýze systému

Analýza je hlavne o riešení problému, a keďže riešiť problém sa dá viacerími 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, štrukturované grafy a iné schémy.

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 4.1: Namapovanie krokov zovšeobecneného postupu do jednotlivých fáz zjednodušeného vývojového procesu.

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 analýzi 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

4.3 Návrh systému

4.4 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.

4.5 Proces vývoja software

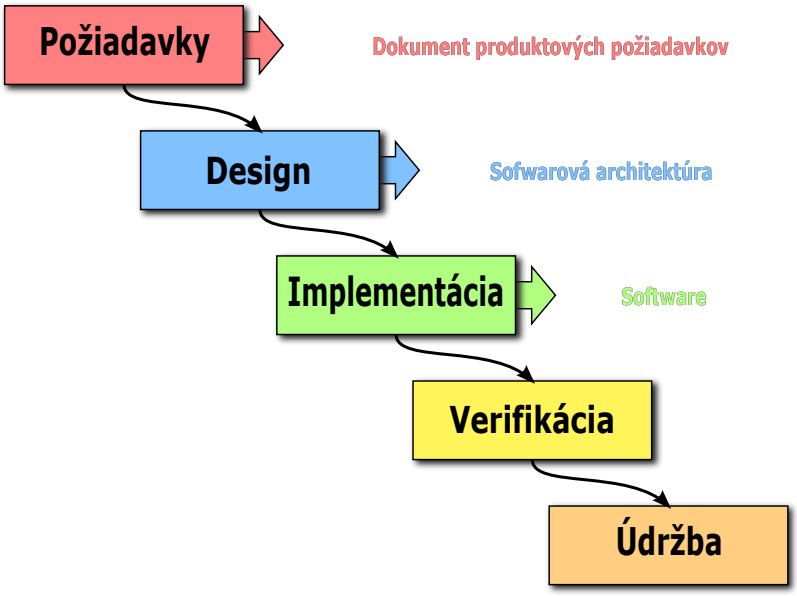
Systémy a produkty v 50. rokoch boli primárne hardvérového charakteru. Logické a matematické výpočetné procesy boli implementované pomocou elektromechanických zariadení. V 60. rokoch už integrované obvody umožnili vývojárom zvýšiť spolahlivosť a presnosť výpočtovej techniky do istej miery aj zásahom softwéru. V tejto dobe však chyby alebo zmeny návrhu boli drahé a časovo náročné. S Príchodom mikroprocesorovej technológie, začatkom 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. Výsledokom bola evolúcia v návrhu systému, ktoré začali modelovať flexibilné a rekonfigurovateľné systémy.[13] To umožnilo vývojárom vytvárať špecifické aplikácie len jednoduchým prispôsobením softwaru. Prechod k softwérovým 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.

4.5.1 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)

4.5.2 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. 4.2: Vodopádový model

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

4.5.3 inkrementálny

4.5.4 Agilný

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

4.6 Tvorba scenárov

Scenár môže byť chápaný celou radou interpretácií, niektoré z nich sú uplatniteľné v systémovom 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 5

Petriho Siete

V tejto kapitole je popísaná obecná Petriho sieť a formalizmy, ktoré vedú k jej transformácii na varianty Petriho sietí s potrebnými vlastnosťmi pre automatické generovanie sekvenčných diagramov.

5.1 Obecná definícia

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

Poznámka 5.1.1. 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 proveditelné 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 5.1.1. 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 5.1.2. 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 5.1.3. 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 5.1.4. 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 proveditelnosť závisí na značeniu siete a naopak. Tieto závislosti popisujú evolučné pravidlá.

Definícia 5.1.5. Evolučné pravidlá

Majme sieť N a jej značenie M.

1. Prechod $t \in T_N$ je **proveditelný** 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 proveditelný 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, popsaného množinou stavových strojov, je určený množinou stavov jednotlivých strojov. Stav (stavová premená) systému je distribuovaný do množiny parciálních stavov systému. Prechody sa vykonávajú v jednotlivých strojoch je však potreba synchronizovat

Parciálne stavy systému sú modelované miestamy a vzormi mo¾ných udalostí jsou de

novány pøechody. Místo se v grafu Petriho sítì vyjadøuje jako a pøechod jako . Okam¾itý stav systému je de

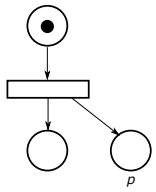
nován umístìním znaèek (tokens) v místech, co¾ v grafu Petriho siete vyjadrujeme teèkami v místech. Prítomnost značky v místì modeluje skuteènost, ¾e daný aspekt stavu (parci ální stav) je momentálnì aktuální, resp. podmínka je splnìna. Ka¾dý pøechod má de

nována vstupní a výstupní místa, co¾ je v grafu Petriho sítì vyjádøeno orientovanými hranami mezi místy a pøechody: ! a ! . Tím je deklarováno, které aspekty stavu systému podmiòují výskyt odpovídající události (provedení pøechodu), a které aspekty stavu jsou výskytem této

5.1.1 Paralelizmus v Petriho sietiach

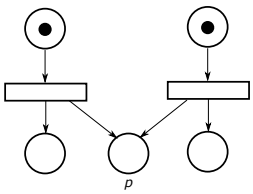
Paralelizmus môže byť prenesený do Petriho sietí viacerými spôsobmi.

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



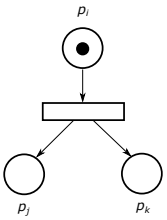
Obr. 5.1: Ukážkový proces

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

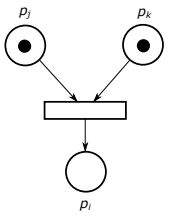


Obr. 5.2: Ukážka zloženia dvoch sietí. V praxi neužitočné.

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. Ich prevedenie do Petriho siete je nasledovné:

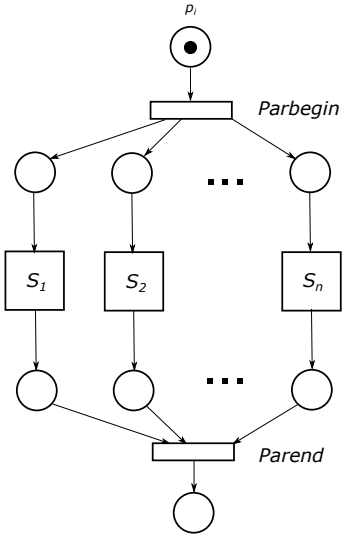


Obr. 5.3: Operácia FORK vykonaná v mieste p_i vytvorí proces v miestach p_j a p_i.



Obr. 5.4: 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** [Djikstra 1968]. 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. 5.5.



Obr. 5.5: Riadiaca štruktúra *parbegin* a *parend* v Petriho sieti

5.1.2 Čas v Petriho sietiach

5.1.3 Varianty 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 nezbytnej pre prekonanie problému spojeného s plošným statickým modelom.

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 špecifikovat 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 odjímanie 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-sítí, potrebujeme pomocný pojem multimnožina a operáciie s multimnožinami.

Definícia 5.1.6. 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:
$$x \neq y = \exists e \in E[x(e) \neq y(e)]$$
$$x \leq y = \forall e \in E[x(e) \leq y(e)]$$

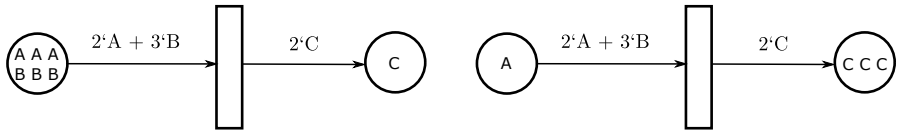
4. odčítanie:
$$x - y = \sum_{e \in E} (x(e) - y(e))`e$$

5. veľkosť:
$$|x| = \sum_{e \in E} x(e)$$

Príklad 5.1.1. názorne zápis $2`A + 3`B$ predstavuje multimnožinu s troma výskytmi prvku a a štyrmi výskytmi prvku b.

Poznámka 5.1.2. 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. 5.6.

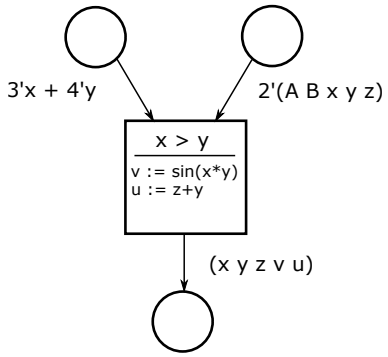


(a) Stav pred uskutočnením prechodu (b) Stav po uskutočnením prechodu

Obr. 5.6: 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. 5.7.

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



Obr. 5.7: Príklad stráže prechodu a akcie prechodu

Hierarchické Petriho siete

5.2 PNTalk

V predošlej kapitole sme sa dozvedeli akú variantu Petriho sietí budeme potrebovať, teraz je na čase predstaviť praktickú implementáciu formalizmu objektovo orientovaných petriho sietí.

5.3 Trieda a dedičnosť

5.4 Siete

5.4.1 Objektová sieť

5.4.2 Sieť metód

5.4.3 Sieť konstruktoru

5.4.4 Synchrónny port

5.5 Prechod

5.5.1 Podmienky prechodu

5.5.2 Akcia

5.5.3 Stráž

5.6 PNTalk

TODO

Kapitola 6

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 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 real-time systémom a komplexným prípadom užitia.

6.1 Scenáre

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.

6.2 Komunikácia v sekvenčných diagramoch

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

Poznámka 6.2.1. 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 6.2.1. 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.

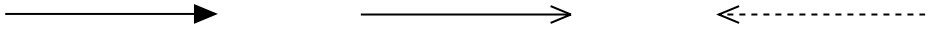
(a) Stav pred uskutočnením prechodu (b) Stav po uskutočnením prechodu

Obr. 6.1: 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 správe. Nemôže vzniknúť samostatne.



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

Obr. 6.2: Reprezentácie troch typov správ

6.3 Účastníci komunikácie

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

Definícia 6.3.1. Pri definícii **čiary života** začneme netradične notáciou, je zobrazená vertikálnou čiara (môže byť čiarkovaná) predstavujúcu čas života aktívneho objektu. Na jej počiatku sa nachádza hlavička, obdĺžnik obsahujúci **identifikačnú informáciu** vo formáte:

kde `<connectable-element-name>` referuje meno typu pripojeného elementu reprezentovaného množinou dodatočných interných datových štruktúr. 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.

6.4 Stavebné Elementy sekvenčných Diagramov

V nasledujúcej sekcii je popísaná syntax a sémantika sekvenčných diagramov.

6.4.1 Actor:TODO preklad

6.4.2 objekt

6.4.3 lifeline:TODO preklad čiara života? :D

6.4.4 focus of control:TODO preklad

6.5 Distribuované systémy

Distribuované systémy majú veľa rozdielnych aspektov, ktoré sa ťažko zachytávajú v jednej difinícii. Je omnoho jednoduchšie hovoriť o distribuovaných systémoch špecifikovaním charakteristík, symptómmi, či média 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 tolerácia chýb, real-time systémy, bezpečnosť a systémový manažment

6.5.1 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, hosťujú procesy, ktoré využívajú distribučné protokoly, ktoré zabezpečujú koherentné vykonanie distribuovaných aktivít.

Príklad 6.5.1. 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 používania internetových technológií, ako distribuované súborové systémy a databázové systémy. Jeden z najpodstatnejšjch rozdieľov je, že v prípade distribuovaných systémov procesy zdieľajú spoločný stav a spolupracujú na dosiahnutí spoločného cieľu. Narozdiel od procesov v tomto príklade, ktoré nemusia spolupracovať, len si napríklad vymieňať správy (ako e-mail) bez spoločného cieľu.

6.5.2 Porovnanie s Centralizovanými systémami

V Tabuľke 6.1 sú zaznamenané vlastnosti v porovanní s centrálnym systémom ako protipólom k distribuovanému systému. Poznanie rozdieľov, výhod a nevýhod oboch systémov je

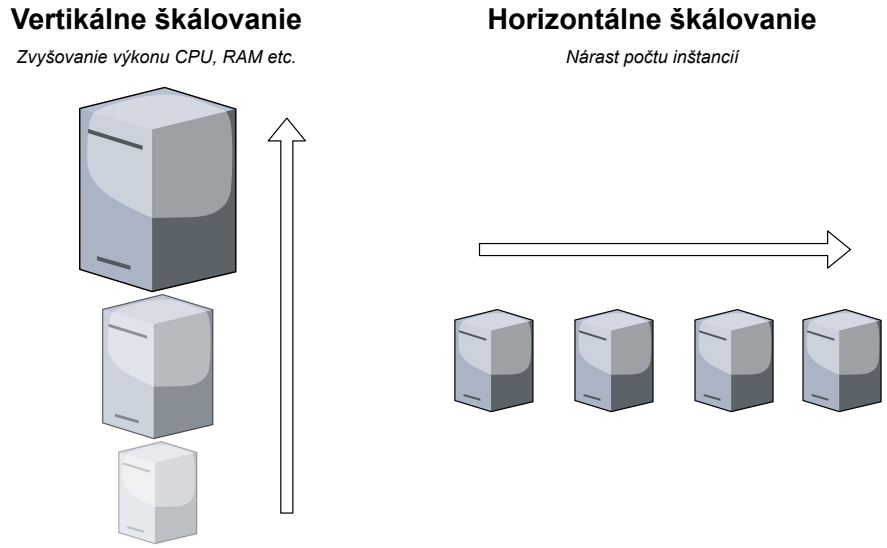
kľúčové pri návrhu systému. Na základe týchto informácií sa možno ľahšie rozhodnúť, ktorú variantu zvoliť.

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

Tabuľka 6.1: 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, aplikační 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. Rozdieľ medzi vertikálnym a horizontálnym škálovaním je graficky znázornený na obrázku 6.3.



Obr. 6.3: 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 zameraním 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.

6.5.3 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 6.5.2. Zriadujeme systém používajúci konkurenntné procesy na zdrojoch vzdialených pobočiek.

2. Keď techniky distribúcie sú vhodnou súčasťou riešenia

   Príklad 6.5.3. Systém banky, ktorá potrbuje zálohovať a synchronizovať dáta v dvoch geograficky odľahlých miestach

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

Príklad 6.5.4. Systém prepožičiavania výpočetných zdrojov medzi vzdialenými užívateľmi.

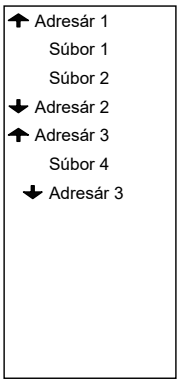
6.6 Vývojové prostredie

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

6.6.1 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 uživateľskom rozhraní zobrazovaný pomocou hierarchického stromu, kde listy sú súbory projektu a uzly sú adresáre.



Obr. 6.4: Ukážka projektového pohľadu s využitím stromovej štruktúry

6.6.2 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.

6.6.3 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.

6.6.4 Ladenie

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

Kapitola 7

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.

7.1 Architektúra

Generátor sekvenčných diagramov[9] je priamo závislý na dvoch komponentách, validátorom kódu jazyka PNTalk a simulátoru objektovo orientovaných Petriho sietí. Je dôležité zvážiť napojenie týchto komponent ku genrátoru. Vzhľadom k rozličným vlastnostiam jednotlivých implementácii 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 vyvýjať 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 aplikace. 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 dodržovať 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 posielané v ľudsky čiteľnom formáte by mali nespornú výhodu v odľaďovaní programu. Schodnou variantou sa preto javí podpora viacerých formátov prímaných generátorom od ostatným komponent. Nevýhodou je vznik réžie okolo dohadovaní si serializačného formátu medzi komponentami.

7.2 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 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 ťažiť z možností formalismu OOPN a zároveň z vyjadrovacích schopností jazyka PNTalk na vytvorenie datovej štruktúry pre sekvenčný diagram.

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

Obr. 7.1: 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 ödoslané od odosieľateľaä "doručené príjemcovi". Obe tieto veci nastanú v jednom okamihu.

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

Štruktúra inicializácií inštancí 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 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;
}
```

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 scenáry 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 dispozícií 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 vytváranej 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 môžme 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.

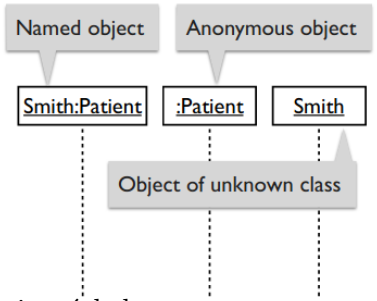
2. zduplikovať referenciu do iného miesta
   Pri zduplikovaní sa vytvorí nový záznam o referencii.

3. vymazať referenciu
   Pri vymazaní sa skontroluje, či nie je počet referencií na objekt nulový. Ak áno, objekt sa deštruuje volaním správy destruct z inštancie s prechodom, ktorý poslednú referenciu vymazal.

Konvencia mena

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

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



syntax jazyka PNTalk vytvára novú inštanciu následovne:
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 inštancie. To nám hneď vylúči tretiu možnosť, pretože meno triedy je vždy známe. Varianty sú teda dve a to buď poskladať meno inštancie 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 inštancie.

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.

Na zásobníku

Doba simulácie po ktorú sa prevádza metóda objektu je viazaná s volaním metód cudzích objektov a preto je nutno archivovať prechody a inštancie, 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:

Synchrónna Asynchrónna 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.

7.3 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 7.3.1. 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 varianty, 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 simulátoru 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 simulátoru jednoducho meniť. Stačí aby dodržovali stanovené rozhranie.

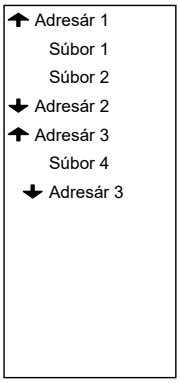
Distribuovaný systém môže nadobnúť odlišné fyzické formy, či už ide o skupinu osobných počítačov, prepojených lokálnou sieťou, skupinu pracovných staníc zdieľajúcích nielen súborové a databázové systémy, ale navyše aj zdieľaním výpočetnej sily procesora.[]

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.

7.4 Uživateľské rozhranie

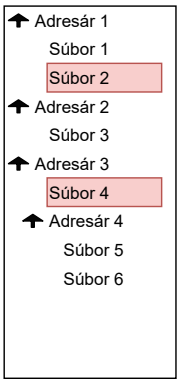
Pri návrhu grafického uživateľského rozhrania je dobré začať položením si otázky "Čo chceme zobrazovať?". Menej je však niekedy viac, pri príliš zložitom rozložení totiž strácame prehľadnosť.

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 effektom upozorniť ak je v súbore neuložená zmena.



Obr. 7.2: Projektový pohľad so schovaným uzlom “Adresár 2” a “Adresár 4”

s



Obr. 7.3: Indikácia neuložených zmien v súboroch viditeľná na rozhraní.

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 rovanko 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 zobraziť zmenu miest OOPN, ktoré prechody vyvolali. Označenie, ktoréhokoľvek miesta na čiare života by malo ukázať aktuálne hodnoty v miestach danej inštancie.

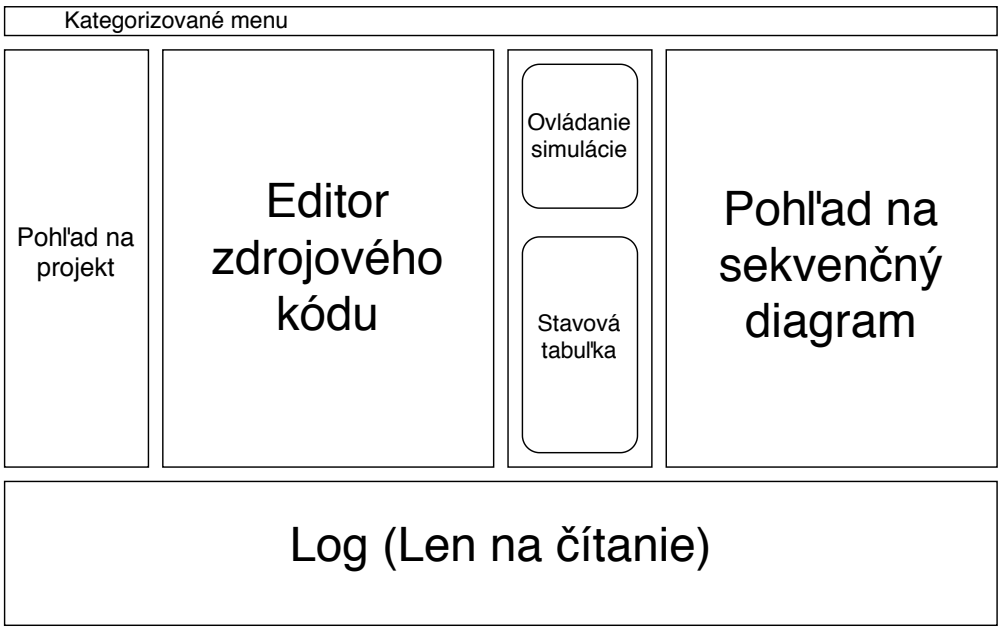
4. **Chceme zobrazovať posledných x riadkov logov** Je fajn dať uživateľovi vedieť čo sa deje formou správ, či už chybových alebo informačných. Správy sa musia dať kopírovať a musia byť viditeľné od najnovšej po najstaršiu.

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.

7.4.1 Rozloženie uživateľského rozhrania

Nie je treba znovu vynaliezať koleso. Pri návrhu rozloženia elementov uživateľského rozhrania sa preto budeme inšpirovať úspešnými vývojovými prostrediami (Visual Studio, IntelliJ IDEA). 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 veškerých nárokov na uživateľské rohranie vyšlo z procesu návrhu rozloženie na obr. :TODO:



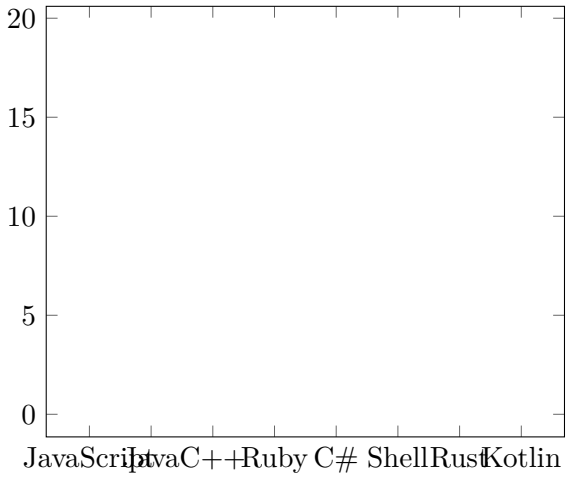
Obr. 7.4: Návrh rozloženia grafického uživateľského rozhrania

Kapitola 8

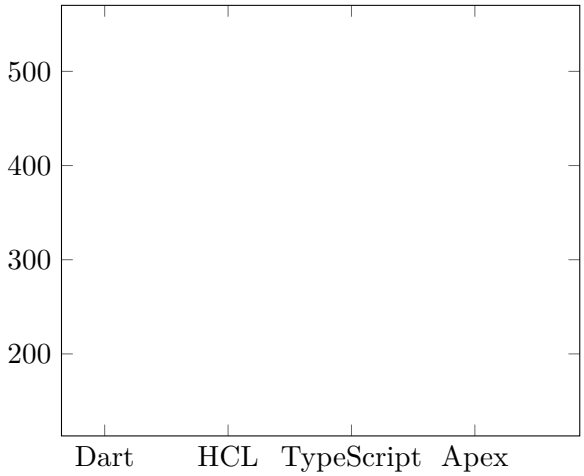
Implementácia

8.1 Výber implementačného jazyka

Práca nadväzuje na simulátor napísaný v jazyku C++, čo robilo z jazyka C++ prirodzenú voľbu pre hľadké naviazanie a kompatibilitu. Ničmenej ďalším požiadavkom práce, akožto zadaním z oblasti blízkej jazyku PNTalk, bola motivácia držať implementačné jazyky týchto nástrojov blízko. :TODO: príklady Keďže väčšina prác beží na Jave a je do budúcna zmýšlaná 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. [5] 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. [4] V prvej polovici roku 2020, teda súčastnosti písania tejto práce, je celkovo na 15. priečke v obľúbenosti [3].



Obr. 8.1: Pridaný kód za rok 2020, second quarter - stats report z github.com [4]



Obr. 8.2: Najrýchlejšie rastúce jazyky - Octoverse report 2019 z github.com [4]

8.2 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.

8.2.1 virtualizácia

Docker

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. [11]

HyperV

Nástroj Compose

Commpose 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. [2]

8.2.2 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 [12], čo ho robí ešte rýchlejší vo vzdialenej medziprocesovej komunikácií.

```
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;
}
```

8.3 Uživateľské rozhranie

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

8.3.1 Bohaté internetové aplikácie

Bohaté internetové aplikácie, (Rich Internet Aplication), 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 uživateľské rozhranie, ktoré dokázalo poslúžiť jednoduchým cieľom, no nemalo ani vzhľad ani responsivitu RIA hlavne kvôli pomalému internetovému spojeniu. RIA je teda výsledkom dnešnej doby poskytujucej vyššiu responzivitu a pokročilejšie grafické uživateľské rozhrania. [1]

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

Ajax

Výraz Ajax (Asynchronous JavaScript and XML) vznikol vo februáry 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 uživateľom ďalej pracovať so stránkov, zatiaľ čo server zpracováva požiadavky.

Technológie stojace za Ajaxom - XHTML, CSS, JavaScript, the DOM, XML a XMLHttpRequestobject — nie sú nové. Vlastne už v 90 rokoch existujú príklady asynchrónnych aktualizácií stránky, ktoré rozultovali v JavaScript [1]. 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 uživateľského 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 [8]. Meno bolo zmenené v roku 2007 na JavaFX. [6]

The first version of JavaFX Script was an interpreted language, and was considered a prototype of the compiled JavaFX Script language that was to come later

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 inštancovať jej triedy. [14]

V roku 2011 verzia JavaFX 2.0 prestala využívať JavaFX Script a vymenila ho za Javu a JavaFX 2.0 API. [8]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 Edvin Syse v januáry 2016[5], keď vydal TornadoFX.
TornadoFX usiluje o to, aby znateľným spôsobom minimalizovalo rozsah potrebného kódu na napísanie JavaFX aplikáce. Zahŕňa typovú kontrolu nad komponentmi pri skladaní komponent uživateľského rozhrania a prináša rozšírenia kotlinu ako delegované vlastnosti a injekcia závislostí. TornadoFX je ukážkový príklad ako Kotlin dokáže zúsporniť kód napísaný v Jave.

8.3.2 Editor Zdrojového kódu

V sekcii 6.6.2 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.

8.3.3 Projektový pohľad

8.3.4 Diagram ako výstup aj interaktívny ladiaci nástroj

Kapitola 9

Záver

Cieľom práce bolo implementovať nástroj pre generovanie sekvenčných diagramov. Jediným vstupom tohto generovania bol kód v jazyku PNTalk popisujúci model objektovo orientovanej petriho siete. Ak už používame akýkoľvek proces vývoja systému Výsledok práce využil na generovanie dáta zachytené simulátorom OOPN. Nástroj bol navrhnutý a implementovaný ako distribuovaný systém, ktorého jednou komponentou bol práve simulátor OOPN. Táto architektúra necháva do budúcna otvorené dvere k použitiu vhodnejšieho simulátora, či zapojeniu do systému viac implementácií simulátorov.

V práci ma prekvapilo, že som sa dostal k distribuovanému systému, čo som pôvodne od zadania neočakával. Viac komponent pracujúcich na spoločnej úlohe bol koncept, ktorý som si chcel už dlhší čas vyskúšať. Preto sa jednalo o prekvapenie značne príjemné. Dovolilo mi to pochopiť výhody mikroservís z viac praktického hľadiska.

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 vykreslovať 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 demonštroje 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 distrubovaný systém, ktorá robí tento problém ľahko riešiteľným v budúcnosti po implementovaní vhodnejšej varianty simulátora. Na Záver je vhodné položiť si otázku či sme boli úspešný. 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 vykreslovať výsledok generácie ihneď pri zmene vstupnęho kódu. Kvôli neuspokojivým výsledkom v tomto teste (:TODO: graf) sa z pokusu o implementácie funkcie "hot-reloadüpustilo.

9.1 Výsledky testovania

Literatúra

[1]

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

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

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

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

[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] Dea, C. *JavaFX 2.0 : introduction by example*. New York: Apress, 2011. ISBN 978-1-4302-4257-4.

[9] 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.

[10] Hamilton, K. *Learning UML 2.0*. Sebastopol, Calif: O'Reilly, 2006. ISBN 9780596009823.

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

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

[13] Wasson, C. *System analysis, design, and development : concepts, principles, and practices*. Hoboken, N.J: Wiley-Interscience, 2006. ISBN 978-0471393337.

[14] Weaver, J. *JavaFX Script : dynamic Java scripting for rich Internet/client-side applications*. Berkeley Calif: Apress, 2007. ISBN 9781590599457.

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

Príloha A

Obsah přiloženého paměťového média

Príloha B

Manuál